Université Paris Cité

Cours de Python

Introduction à la programmation Python pour la biologie

https://python.sdv.u-paris.fr/

Patrick Fuchs et **Pierre Poulain**
prénom [point] nom [arobase] u-paris [point] fr

version du 13 novembre 2024

Université Paris Cité, France

Ce document est sous licence
Creative Commons Attribution - Partage dans les Mêmes Conditions 3.0 France
(CC BY-SA 3.0 FR)
https://creativecommons.org/licenses/by-sa/3.0/fr/

Table des matières

Avant-propos

Quelques mots sur l'origine de ce cours

Ce cours, développé par Patrick Fuchs et Pierre Poulain, a été conçu à l'origine pour les étudiants débutants en programmation Python des filières de biologie et de biochimie de l'université Paris Diderot - Paris 7, devenue Université Paris Cité [1] ; et plus spécialement pour les étudiants du master Biologie Informatique.

Si vous relevez des erreurs à la lecture de ce document, merci de nous les signaler.

Le cours est disponible en version HTML [2] et PDF [3].

Remerciements

Merci à tous les contributeurs, occasionnels ou réguliers, entre autre : Jennifer Becq, Benoist Laurent, Hubert Santuz, Virginie Martiny, Romain Laurent, Benjamin Boyer, Jonathan Barnoud, Amélie Bâcle, Thibault Tubiana, Romain Retureau, Catherine Lesourd, Philippe Label, Rémi Cuchillo, Cédric Gageat, Philibert Malbranche, Mikaël Naveau, Alexandra Moine-Franel, Dominique Tinel, et plus généralement les promotions des masters de biologie informatique et *in silico drug design*, ainsi que du diplôme universitaire en bioinformatique intégrative.

Nous remercions tout particulièrement Sander Nabuurs pour la première version de ce cours remontant à 2003, Denis Mestivier pour les idées de certains exercices et Philip Guo pour son site *Python Tutor* [4].

Enfin, merci à vous tous, les curieux de Python, qui avez été nombreux à nous envoyer des retours sur ce cours, à nous suggérer des améliorations et à nous signaler des coquilles. Cela rend le cours vivant et dynamique, continuez comme ça !

De nombreuses personnes nous ont aussi demandé les corrections des exercices. Nous ne les mettons pas sur le site afin d'éviter la tentation de les regarder trop vite, mais vous pouvez nous écrire et nous vous les enverrons.

Le livre

Ce cours est également publié aux éditions Dunod sous le titre « Programmation en Python pour les sciences de la vie [5] ». Le livre en est à sa 2e édition, vous pouvez vous le procurer dans toutes les bonnes librairies.

Afin de promouvoir le partage des connaissances et le logiciel libre, nos droits d'auteurs provenant de la vente de cet ouvrage sont reversés à deux associations : Wikimédia France [6] qui s'occupe notamment de l'encyclopédie libre Wikipédia

1. https://www.u-paris.fr/
2. https://python.sdv.u-paris.fr/index.html
3. https://python.sdv.u-paris.fr/cours-python.pdf
4. http://pythontutor.com/
5. https://www.dunod.com/sciences-techniques/programmation-en-python-pour-sciences-vie-0
6. https://www.wikimedia.fr/

FIGURE 1 – Couverture livre Dunod, 2e édition.

et NumFOCUS[7] qui soutient le développement de logiciels libres scientifiques et notamment l'écosystème scientifique autour de Python.

7. https://numfocus.org/

CHAPITRE 1

Introduction

1.1 Qu'est-ce que Python ?

Le langage de programmation Python a été créé en 1989 par Guido van Rossum, aux Pays-Bas. Le nom *Python* vient d'un hommage à la série télévisée *Monty Python's Flying Circus* dont G. van Rossum est fan. La première version publique de ce langage a été publiée en 1991.

La dernière version de Python est la version 3. Plus précisément, la version 3.11 a été publiée en octobre 2022. La version 2 de Python est obsolète et n'est plus maintenue, évitez de l'utiliser.

La *Python Software Foundation*[1] est l'association qui organise le développement de Python et anime la communauté de développeurs et d'utilisateurs.

Ce langage de programmation présente de nombreuses caractéristiques intéressantes :

- Il est multiplateforme. C'est-à-dire qu'il fonctionne sur de nombreux systèmes d'exploitation : Windows, Mac OS X, Linux, Android, iOS, depuis les mini-ordinateurs Raspberry Pi jusqu'aux supercalculateurs.
- Il est gratuit. Vous pouvez l'installer sur autant d'ordinateurs que vous voulez (même sur votre téléphone !).
- C'est un langage de haut niveau. Il demande relativement peu de connaissance sur le fonctionnement d'un ordinateur pour être utilisé.
- C'est un langage interprété. Un script Python n'a pas besoin d'être compilé pour être exécuté, contrairement à des langages comme le C ou le C++.
- Il est orienté objet. C'est-à-dire qu'il est possible de concevoir en Python des entités qui miment celles du monde réel (une molécule d'ADN, une protéine, un atome, etc.) avec un certain nombre de règles de fonctionnement et d'interactions.
- Il est relativement *simple* à prendre en main[2].
- C'est le langage de programmation le plus utilisé au monde (voir les classements TIOBE[3] et IEEE Spectrum[4]).
- Enfin, il est très utilisé en bioinformatique, chimie-informatique et plus généralement en analyse de données.

Toutes ces caractéristiques font que Python est désormais enseigné dans de nombreuses formations, du lycée à l'enseignement supérieur.

1. https://www.python.org/psf/
2. Nous sommes d'accord, cette notion est très relative.
3. https://www.tiobe.com/tiobe-index/
4. https://spectrum.ieee.org/the-top-programming-languages-2023

1.2 Conseils pour l'apprentissage de Python

Comme tout apprentissage, apprendre la programmation Python prend du temps et nécessite de pratiquer. Contrairement à d'autres activités scientifiques expérimentales (biologie moléculaire, chimie organique, électronique, etc.), programmer en Python ne nécessite pas de matériel coûteux, juste un ordinateur et éventuellement une connexion internet. Par ailleurs, Python est un programme informatique qui par définition ne se fatigue pas, est patient et toujours disponible. N'hésitez donc pas à pratiquer, pratiquer et pratiquer encore.

1.3 Conseils pour installer et configurer Python

Pour pratiquer la programmation Python, il est préférable que Python soit installé sur votre ordinateur. La bonne nouvelle est que vous pouvez installer gratuitement Python sur votre machine, que ce soit sous Windows, Mac OS X ou Linux. Nous donnons ici un résumé des points importants concernant cette installation. La marche à suivre pas-à-pas est détaillée à l'adresse `https://python.sdv.u-paris.fr/` dans la rubrique *B. Installation de Python*.

1.3.1 Python 2 ou Python 3 ?

Ce cours est basé sur la **version 3 de Python**, qui est désormais le standard.

Si, néanmoins, vous deviez un jour travailler sur un ancien programme écrit en Python 2, sachez qu'il existe quelques différences importantes entre Python 2 et Python 3. Le chapitre 26 *Remarques complémentaires* (en ligne) vous apportera plus de précisions.

1.3.2 Miniconda

Nous vous conseillons d'installer Miniconda[5], logiciel gratuit, disponible pour Windows, Mac OS X et Linux, et qui installera pour vous Python 3.

Avec le gestionnaire de paquets *conda*, fourni avec Miniconda, vous pourrez installer des modules supplémentaires qui sont très utiles en bioinformatique (*NumPy*, *scipy*, *matplotlib*, *pandas*, *Biopython*), mais également Jupyter Lab qui vous permettra d'éditer des *notebooks* Jupyter. Vous trouverez en ligne[6] une documentation pas-à-pas pour installer Miniconda, Python 3 et les modules supplémentaires qui seront utilisés dans ce cours.

1.3.3 Éditeur de texte

L'apprentissage d'un langage informatique comme Python va nécessiter d'écrire des lignes de codes à l'aide d'un éditeur de texte. Si vous êtes débutants, on vous conseille d'utiliser *notepad++* sous Windows, *BBEdit* ou *CotEditor* sous Mac OS X et *gedit* sous Linux. La configuration de ces éditeurs de texte est détaillée dans la rubrique *Installation de Python* disponible en ligne. Bien sûr, si vous préférez d'autres éditeurs comme *Visual Studio Code*, *Sublime Text*, *emacs*, *vim*, *geany*... utilisez-les !

À toute fin utile, on rappelle que les logiciels *Microsoft Word*, *WordPad* et *LibreOffice Writer* ne sont pas des éditeurs de texte, ce sont des traitements de texte qui ne peuvent pas et ne doivent pas être utilisés pour écrire du code informatique.

1.4 Notations utilisées

Dans cet ouvrage, les commandes, les instructions Python, les résultats et les contenus de fichiers sont indiqués avec `cette police` pour les éléments ponctuels ou

```
1  sous cette forme,
2  sur plusieurs lignes,
3  pour les éléments les plus longs.
```

Pour ces derniers, le numéro à gauche indique le numéro de la ligne et sera utilisé pour faire référence à une instruction particulière. Ce numéro n'est bien sûr là qu'à titre indicatif.

Par ailleurs, dans le cas de programmes, de contenus de fichiers ou de résultats trop longs pour être inclus dans leur intégralité, la notation `[...]` indique une coupure arbitraire de plusieurs caractères ou lignes.

5. https://conda.io/miniconda.html
6. https://python.sdv.u-paris.fr/livre-dunod

1.5 Introduction au *shell*

Un *shell* est un interpréteur de commandes interactif permettant d'interagir avec l'ordinateur. On utilisera le *shell* pour lancer l'interpréteur Python.

Pour approfondir la notion de *shell*, vous pouvez consulter les pages Wikipedia :

- du *shell* Unix[7] fonctionnant sous Mac OS X et Linux ;
- du *shell* PowerShell[8] fonctionnant sous Windows.

Un *shell* possède toujours une invite de commande, c'est-à-dire un message qui s'affiche avant l'endroit où on entre des commandes. Dans tout cet ouvrage, cette invite est représentée par convention par le symbole dollar `$` (qui n'a rien à avoir ici avec la monnaie), et ce quel que soit le système d'exploitation.

Par exemple, si on vous demande de lancer l'instruction suivante :

```
$ python
```

il faudra taper seulement `python` sans le `$` ni l'espace après le `$`.

1.6 Premier contact avec Python

Python est un langage interprété, c'est-à-dire que chaque ligne de code est lue puis interprétée afin d'être exécutée par l'ordinateur. Pour vous en rendre compte, ouvrez un *shell* puis lancez la commande :

```
python
```

La commande précédente va lancer l'**interpréteur Python**. Vous devriez obtenir quelque chose de ce style pour Windows :

```
PS C:\Users\pierre>python
Python 3.12.2 | packaged by Anaconda, Inc. | (main, Feb 27 2024, 17:28:07) [MSC v.1916 64 bit (AMD64)] on
    win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

pour Mac OS X :

```
iMac-de-pierre:Downloads$ python
Python 3.12.2 | packaged by Anaconda, Inc. | (main, Feb 27 2024, 12:57:28) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

ou pour Linux :

```
pierre@jeera:~$ python
Python 3.12.2 | packaged by conda-forge | (main, Feb 16 2024, 20:50:58) [GCC 12.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Les blocs

- `PS C:\Users\pierre>` pour Windows,
- `iMac-de-pierre:Downloads$` pour Mac OS X,
- `pierre@jeera:~$` pour Linux.

représentent l'invite de commande de votre *shell*. Il se peut que vous ayez aussi le mot `(base)` qui indique que vous avez un environnement conda activé. Par la suite, cette invite de commande sera représentée simplement par le caractère `$`, que vous soyez sous Windows, Mac OS X ou Linux.

Le triple chevron `>>>` est l'invite de commande (*prompt* en anglais) de l'interpréteur Python. Ici, Python attend une commande que vous devez saisir au clavier. Tapez par exemple l'instruction :

```
print("Hello world!")
```

puis, validez cette commande en appuyant sur la touche *Entrée*.

Python a exécuté la commande directement et a affiché le texte `Hello world!`. Il attend ensuite une nouvelle instruction en affichant l'invite de l'interpréteur Python (`>>>`). En résumé, voici ce qui a dû apparaître sur votre écran :

7. https://fr.wikipedia.org/wiki/Shell_Unix
8. https://fr.wikipedia.org/wiki/Windows_PowerShell

```
>>> print("Hello world!")
Hello world!
>>>
```

Vous pouvez refaire un nouvel essai en vous servant cette fois de l'interpréteur comme d'une calculatrice :

```
>>> 1+1
2
>>> 6*3
18
```

À ce stade, vous pouvez entrer une autre commande ou bien quitter l'interpréteur Python, soit en tapant la commande `exit()` puis en validant en appuyant sur la touche *Entrée*, soit en pressant simultanément les touches *Ctrl* et *D* sous Linux et Mac OS X ou *Ctrl* et *Z* puis *Entrée* sous Windows.

En résumant, l'interpréteur fonctionne sur le modèle :

```
>>> instruction python
résultat
```

où le triple chevron correspond à l'entrée (*input*) que l'utilisateur tape au clavier, et l'absence de chevron en début de ligne correspond à la sortie (*output*) générée par Python. Une exception se présente toutefois : lorsqu'on a une longue ligne de code, on peut la couper en deux avec le caractère `\` (*backslash*) pour des raisons de lisibilité :

```
>>> Voici une longue ligne de code \
... décrite sur deux lignes
résultat
```

En ligne 1 on a rentré la première partie de la ligne de code. On termine par un `\`, ainsi Python sait que la ligne de code n'est pas finie. L'interpréteur nous l'indique avec les trois points `...`. En ligne 2, on rentre la fin de la ligne de code puis on appuie sur *Entrée*. À ce moment, Python nous génère le résultat. Si la ligne de code est vraiment très longue, il est même possible de la découper en trois voire plus :

```
>>> Voici une ligne de code qui \
... est vraiment très longue car \
... elle est découpée sur trois lignes
résultat
```

L'interpréteur Python est donc un système interactif dans lequel vous pouvez entrer des commandes, que Python exécutera sous vos yeux (au moment où vous validerez la commande en appuyant sur la touche *Entrée*).

Il existe de nombreux autres langages interprétés comme Perl[9] ou R[10]. Le gros avantage de ce type de langage est qu'on peut immédiatement tester une commande à l'aide de l'interpréteur, ce qui est très utile pour débugger (c'est-à-dire trouver et corriger les éventuelles erreurs d'un programme). Gardez bien en mémoire cette propriété de Python qui pourra parfois vous faire gagner un temps précieux !

1.7 Premier programme

Bien sûr, l'interpréteur présente vite des limites dès lors que l'on veut exécuter une suite d'instructions plus complexe. Comme tout langage informatique, on peut enregistrer ces instructions dans un fichier, que l'on appelle communément un script (ou programme) Python.

Pour reprendre l'exemple précédent, ouvrez un éditeur de texte (pour choisir et configurer un éditeur de texte, reportez-vous si nécessaire à la rubrique *Installation de Python* en ligne[11]) et entrez le code suivant :

```
print("Hello world!")
```

Ensuite, enregistrez votre fichier sous le nom `test.py`, puis quittez l'éditeur de texte.

Remarque

L'extension de fichier standard des scripts Python est `.py`.

9. http://www.perl.org
10. http://www.r-project.org
11. https://python.sdv.u-paris.fr/livre-dunod

Pour exécuter votre script, ouvrez un *shell* et entrez la commande : `python test.py`

Vous devriez obtenir un résultat similaire à ceci :

```
$ python test.py
Hello world!
```

Si c'est bien le cas, bravo ! Vous avez exécuté votre premier programme Python.

1.8 Commentaires

Dans un script, tout ce qui suit le caractère `#` est ignoré par Python jusqu'à la fin de la ligne et est considéré comme un commentaire.

Les commentaires doivent expliquer votre code dans un langage humain. L'utilisation des commentaires est rediscutée dans le chapitre 16 *Bonnes pratiques en programmation Python*.

Voici un exemple :

```
1  # Votre premier commentaire en  Python.
2  print("Hello world!")
3
4  # D'autres commandes plus utiles pourraient suivre.
```

Remarque

On appelle souvent à tort le caractère `#` « dièse ». On devrait plutôt parler de « croisillon[12] ».

1.9 Notion de bloc d'instructions et d'indentation

En programmation, il est courant de répéter un certain nombre de choses (avec les boucles, voir le chapitre 5 *Boucles et comparaisons*) ou d'exécuter plusieurs instructions si une condition est vraie (avec les tests, voir le chapitre 6 *Tests*).

Par exemple, imaginons que nous souhaitions afficher chacune des bases d'une séquence d'ADN, les compter puis afficher le nombre total de bases à la fin. Nous pourrions utiliser l'algorithme présenté en pseudo-code dans la figure 1.1.

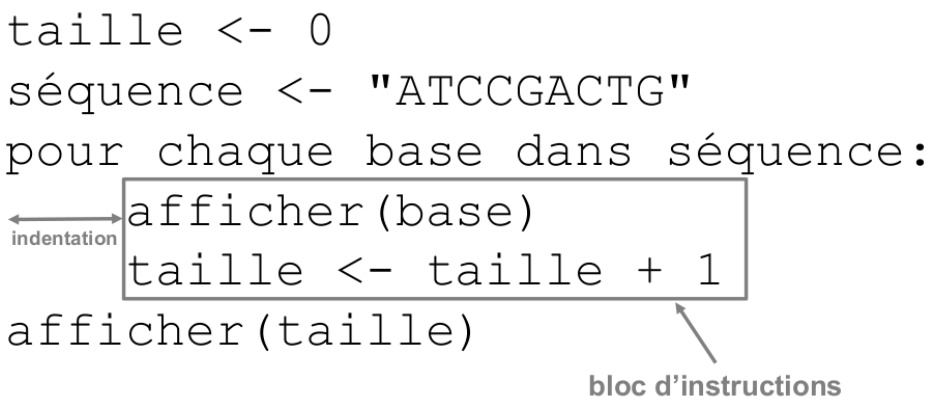


FIGURE 1.1 – Notion d'indentation et de bloc d'instructions.

Pour chaque base de la séquence ATCCGACTG, nous souhaitons effectuer deux actions : d'abord afficher la base puis compter une base de plus. Pour indiquer cela, on décalera vers la droite ces deux instructions par rapport à la ligne précédente (`pour chaque base [...]`). Ce décalage est appelé **indentation** et l'ensemble des lignes indentées constitue un **bloc d'instructions**.

Une fois qu'on aura réalisé ces deux actions sur chaque base, on pourra passer à la suite, c'est-à-dire afficher la taille de la séquence. Pour bien préciser que cet affichage se fait à la fin, donc une fois l'affichage puis le comptage de chaque base terminés, la ligne correspondante n'est pas indentée (c'est-à-dire qu'elle n'est pas décalée vers la droite).

12. https://fr.wikipedia.org/wiki/Croisillon_(signe)

Pratiquement, l'indentation en Python doit être homogène (soit des espaces, soit des tabulations, mais pas un mélange des deux). Une indentation avec 4 espaces est le style d'indentation recommandé (voir le chapitre 16 *Bonnes pratiques en programmation Python*).

Si tout cela semble un peu complexe, ne vous inquiétez pas. Vous allez comprendre tous ces détails chapitre après chapitre.

1.10 Autres ressources

Pour compléter votre apprentissage de Python, n'hésitez pas à consulter d'autres ressources complémentaires à cet ouvrage. D'autres auteurs abordent l'apprentissage de Python d'une autre manière. Nous vous conseillons les ressources suivantes en langue française :

- Le livre *Apprendre à programmer avec Python 3* de Gérard Swinnen. Cet ouvrage est téléchargeable gratuitement sur le site de Gérard Swinnen [13]. Les éditions Eyrolles proposent également la version papier de cet ouvrage.
- Le livre *Apprendre à programmer en Python avec PyZo et Jupyter Notebook* de Bob Cordeau et Laurent Pointal, publié aux éditions Dunod. Une partie de cet ouvrage est téléchargeable gratuitement sur le site de Laurent Pointal [14].
- Le livre *Apprenez à programmer en Python* de Vincent Legoff [15] que vous trouverez sur le site *Openclassroms*.

Et pour terminer, une ressource incontournable en langue anglaise :

- Le site www.python.org [16]. Il contient énormément d'informations et de liens sur Python. La page d'index des modules [17] est particulièrement utile (et traduite en français).

13. http://www.inforef.be/swi/python.htm
14. https://perso.limsi.fr/pointal/python:courspython3
15. https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python
16. http://www.python.org
17. https://docs.python.org/fr/3/py-modindex.html

CHAPITRE 2

Variables

2.1 Définition et création

Définition

Une **variable** est une zone de la mémoire de l'ordinateur dans laquelle une **valeur** est stockée. Aux yeux du programmeur, cette variable est définie par un **nom**, alors que pour l'ordinateur, il s'agit en fait d'une adresse, c'est-à-dire d'une zone particulière de la mémoire.

En Python, la **déclaration** d'une variable et son **initialisation** (c'est-à-dire la première valeur que l'on va stocker dedans) se font en même temps. Pour vous en convaincre, testez les instructions suivantes après avoir lancé l'interpréteur :

```
1 >>> x = 2
2 >>> x
3 2
```

Ligne 1. Dans cet exemple, nous avons déclaré, puis initialisé la variable `x` avec la valeur 2. Notez bien qu'en réalité, il s'est passé plusieurs choses :

- Python a « deviné » que la variable était un entier. On dit que Python est un langage au **typage dynamique**.
- Python a alloué (réservé) l'espace en mémoire pour y accueillir un entier. Chaque type de variable prend plus ou moins d'espace en mémoire. Python a aussi fait en sorte qu'on puisse retrouver la variable sous le nom `x`.
- Enfin, Python a assigné la valeur 2 à la variable `x`.

Dans d'autres langages (en C par exemple), il faut coder ces différentes étapes une par une. Python étant un langage dit de *haut niveau*, la simple instruction `x = 2` a suffi à réaliser les trois étapes en une fois !

Lignes 2 et 3. L'interpréteur nous a permis de connaître le contenu de la variable juste en tapant son nom. Retenez ceci, car c'est une **spécificité de l'interpréteur Python**, très pratique pour chasser (*debugger*) les erreurs dans un programme. En revanche, la ligne d'un script Python qui contient seulement le nom d'une variable (sans aucune autre indication) n'affichera pas la valeur de la variable à l'écran lors de l'exécution (pour autant, cette instruction reste valide et ne générera pas d'erreur).

Depuis la version 3.10, l'interpréteur Python a amélioré ses messages d'erreur. Il est ainsi capable de suggérer des noms de variables existants lorsqu'on fait une faute de frappe :

```
>>> voyelles = "aeiouy"
>>> voyelle
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'voyelle' is not defined. Did you mean: 'voyelles'?
```

Si le mot qu'on tape n'est pas très éloigné, cela fonctionne également lorsqu'on se trompe à différents endroits du mot !

```
pharmacie = "vente de médicaments"
>>> farmacia
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'farmacia' is not defined. Did you mean: 'pharmacie'?
```

Revenons sur le signe = ci-dessus.

Définition

Le symbole = est appelé **opérateur d'affectation**. Il permet d'assigner une valeur à une variable en Python. Cet opérateur s'utilise toujours de la droite vers la gauche. Par exemple, dans l'instruction `x = 2` ci-dessus, Python attribue la valeur située à droite (ici, `2`) à la variable située à gauche (ici, `x`). D'autres langages de programmation comme *R* utilisent les symboles `<-` pour rendre l'affectation d'une variable plus explicite, par exemple `x <- 2`.

Voici d'autres cas de figures que vous rencontrerez avec l'opérateur = :

```
>>> x = 2
>>> y = x
>>> y
2
>>> x = 5 - 2
>>> x
3
```

Ligne 2. Ici on a un nom de variable à gauche et à droite de l'opérateur =. Dans ce cas, on garde la règle d'aller toujours de la droite vers la gauche. C'est donc le contenu de la variable `y` qui est affecté à la variable `x`.

Ligne 5. Comme on le verra plus bas, si on a à droite de l'opérateur = une expression, ici la soustraction `4 - 2`, celle-ci est d'abord évaluée et c'est le résultat de cette opération qui sera affecté à la variable `x`. On pourra noter également que la valeur de `x` précédente (2) a été écrasée.

Attention

L'opérateur d'affectation = écrase systématiquement la valeur de la variable située à sa gauche si celle-ci existe déjà.

2.2 Les types de variables

Définition

Le **type** d'une variable correspond à la nature de celle-ci. Les trois principaux types dont nous aurons besoin dans un premier temps sont les entiers (*integer* ou *int*), les nombres décimaux que nous appellerons *floats* et les chaînes de caractères (*string* ou *str*).

Bien sûr, il existe de nombreux autres types (par exemple, les booléens, les nombres complexes, etc.). Si vous n'êtes pas effrayés, vous pouvez vous en rendre compte ici [1].

1. https://docs.python.org/fr/3.12/library/stdtypes.html

Dans l'exemple précédent, nous avons stocké un nombre entier (*int*) dans la variable `x`, mais il est tout à fait possible de stocker des *floats*, des chaînes de caractères (*string* ou *str*) ou de nombreux autres types de variables que nous verrons par la suite :

```
>>> y = 3.14
>>> y
3.14
>>> a = "bonjour"
>>> a
'bonjour'
>>> b = 'salut'
>>> b
'salut'
>>> c = """girafe"""
>>> c
'girafe'
>>> d = '''lion'''
>>> d
'lion'
```

Remarque

Python reconnaît certains types de variables automatiquement (entier, *float*). Par contre, pour une chaîne de caractères, il faut l'entourer de guillemets (doubles, simples, voire trois guillemets successifs doubles ou simples) afin d'indiquer à Python le début et la fin de la chaîne de caractères.

Dans l'interpréteur, l'affichage direct du contenu d'une chaîne de caractères se fait avec des guillemets simples, quel que soit le type de guillemets utilisé pour définir la chaîne de caractères.

En Python, comme dans la plupart des langages de programmation, c'est le point qui est utilisé comme séparateur décimal. Ainsi, `3.14` est un nombre reconnu comme un *float* en Python alors que ce n'est pas le cas de `3,14`.

Il existe également des variables de type booléen. Un booléen[2] est une variable qui ne prend que deux valeurs : Vrai ou Faux. En python, on utilise pour cela les deux mots réservés `True` et `False` :

```
>>> var = True
>>> var2 = False
>>> var
True
>>> var2
False
```

Nous verrons l'utilité des booléens dans les chapitres 5 *Boucles* et 6 *Tests*.

2.3 Nommage

Le nom des variables en Python peut être constitué de lettres minuscules (`a` à `z`), de lettres majuscules (`A` à `Z`), de nombres (`0` à `9`) ou du caractère souligné (`_`). Vous ne pouvez pas utiliser d'espace dans un nom de variable.

Par ailleurs, un nom de variable ne doit pas débuter par un chiffre et il n'est pas recommandé de le faire débuter par le caractère `_` (sauf cas très particuliers).

De plus, il faut absolument éviter d'utiliser un mot « réservé » par Python comme nom de variable (par exemple : `print`, `range`, `for`, `from`, etc.).

Dans la mesure du possible, il est conseillé de mettre des noms de variables explicites. Sauf dans de rares cas que nous expliquerons plus tard dans le cours, évitez les noms de variables à une lettre.

Enfin, Python est sensible à la casse, ce qui signifie que les variables `TesT`, `test` et `TEST` sont différentes.

2.4 Écriture scientifique

On peut écrire des nombres très grands ou très petits avec des puissances de 10 en utilisant le symbole `e` :

2. https://fr.wikipedia.org/wiki/Bool%C3%A9en

```
>>> 1e6
1000000.0
>>> 3.12e-3
0.00312
```

On appelle cela écriture ou notation scientifique. On pourra noter deux choses importantes :

- `1e6` ou `3.12e-3` n'implique pas l'utilisation du nombre exponentiel *e*, mais signifie 1×10^6 ou 3.12×10^{-3} respectivement ;
- même si on ne met que des entiers à gauche et à droite du symbole `e` (comme dans `1e6`), Python génère systématiquement un *float*.

Enfin, vous avez sans doute constaté qu'il est parfois pénible d'écrire des nombres composés de beaucoup de chiffres, par exemple le nombre d'Avogradro $6.02214076 \times 10^{23}$ ou le nombre d'humains sur Terre[3] 8094752749 au 5 mars 2024 à 19h34. Pour s'y retrouver, Python autorise l'utilisation du caractère « souligné » (ou *underscore*) _ pour séparer des groupes de chiffres. Par exemple :

```
>>> avogadro_number = 6.022_140_76e23
>>> print(avogadro_number)
6.02214076e+23
>>> humans_on_earth = 8_094_752_749
>>> print(humans_on_earth)
8094752749
```

Dans ces exemples, le caractère _ (*underscore* ou « souligné ») est utilisé pour séparer des groupes de trois chiffres, mais on peut faire ce qu'on veut :

```
>>> print(80_94_7527_49)
8094752749
```

2.5 Opérations

2.5.1 Opérations sur les types numériques

Les quatre opérations arithmétiques de base se font de manière simple sur les types numériques (nombres entiers et *floats*) :

```
>>> x = 45
>>> x + 2
47
>>> x - 2
43
>>> x * 3
135
>>> y = 2.5
>>> x - y
42.5
>>> (x * 10) + y
452.5
```

Remarquez toutefois que si vous mélangez les types entiers et *floats*, le résultat est renvoyé comme un *float* (car ce type est plus général). Par ailleurs, l'utilisation de parenthèses permet de gérer les priorités.

L'opérateur `/` effectue une division. Contrairement aux opérateurs `+`, `-` et `*`, celui-ci renvoie systématiquement un *float* :

```
>>> 3 / 4
0.75
>>> 2.5 / 2
1.25
>>> 6 / 3
2.0
>>> 10 / 2
5.0
```

3. https://thepopulationproject.org/

L'opérateur puissance utilise les symboles `**` :

```
>>> 2**3
8
>>> 2**4
16
```

Pour obtenir le quotient et le reste d'une division entière (voir ici[4] pour un petit rappel sur la division entière), on utilise respectivement les symboles `//` et modulo `%` :

```
>>> 5 // 4
1
>>> 5 % 4
1
>>> 8 // 4
2
>>> 8 % 4
0
```

Les symboles `+`, `-`, `*`, `/`, `**`, `//` et `%` sont appelés **opérateurs**, car ils réalisent des opérations sur les variables.

Enfin, il existe des opérateurs « combinés » qui effectue une opération et une affectation en une seule étape :

```
>>> i = 0
>>> i = i + 1
>>> i
1
>>> i += 1
>>> i
2
>>> i += 2
>>> i
4
```

L'opérateur `+=` effectue une addition puis affecte le résultat à la même variable. Cette opération s'appelle une « incrémentation ».

Les opérateurs `-=`, `*=` et `/=` se comportent de manière similaire pour la soustraction, la multiplication et la division.

2.5.2 Opérations sur les chaînes de caractères

Pour les chaînes de caractères, deux opérations sont possibles, l'addition et la multiplication :

```
>>> chaine = "Salut"
>>> chaine
'Salut'
>>> chaine + " Python"
'Salut Python'
>>> chaine * 3
'SalutSalutSalut'
```

L'opérateur d'addition `+` concatène (assemble) deux chaînes de caractères. On parle de concaténation.

L'opérateur de multiplication `*` entre un nombre entier et une chaîne de caractères duplique (répète) plusieurs fois une chaîne de caractères. On parle de duplication.

Attention

Vous observez que les opérateurs `+` et `*` se comportent différemment s'il s'agit d'entiers ou de chaînes de caractères. Ainsi, l'opération `2 + 2` est une addition alors que l'opération `"2" + "2"` est une concaténation. On appelle ce comportement **redéfinition des opérateurs**. Nous serons amenés à revoir cette notion dans le chapitre 24 *Avoir plus la classe avec les objets* (en ligne).

4. https://fr.wikipedia.org/wiki/Division_euclidienne

2.5.3 Opérations illicites

Attention à ne pas faire d'opération illicite, car vous obtiendriez un message d'erreur :

```
>>> "toto" * 1.3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'float'
>>> "toto" + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Notez que Python vous donne des informations dans son message d'erreur. Dans le second exemple, il indique que vous devez utiliser une variable de type *str*, c'est-à-dire une chaîne de caractères et pas un *int*, c'est-à-dire un entier.

2.6 La fonction `type()`

Si vous ne vous souvenez plus du type d'une variable, utilisez la fonction `type()` qui vous le rappellera.

```
>>> x = 2
>>> type(x)
<class 'int'>
>>> y = 2.0
>>> type(y)
<class 'float'>
>>> z = '2'
>>> type(z)
<class 'str'>
>>> type(True)
<class 'bool'>
```

Nous verrons plus tard ce que signifie le mot *class*.

Attention

Pour Python, la valeur `2` (nombre entier) est différente de `2.0` (*float*) et est aussi différente de `'2'` (chaîne de caractères).

2.7 Conversion de types

En programmation, on est souvent amené à convertir les types, c'est-à-dire passer d'un type numérique à une chaîne de caractères ou vice-versa. En Python, rien de plus simple avec les fonctions `int()`, `float()` et `str()`. Pour vous en convaincre, regardez ces exemples :

```
>>> i = 3
>>> str(i)
'3'
>>> i = '456'
>>> int(i)
456
>>> float(i)
456.0
>>> i = '3.1416'
>>> float(i)
3.1416
```

On verra au chapitre 7 *Fichiers* que ces conversions sont essentielles. En effet, lorsqu'on lit ou écrit des nombres dans un fichier, ils sont considérés comme du texte, donc des chaînes de caractères.

Toute conversion d'une variable d'un type en un autre est appelé *casting* en anglais, il se peut que vous croisiez ce terme si vous consultez d'autres ressources.

2.8 Note sur le vocabulaire et la syntaxe

Nous avons vu dans ce chapitre la notion de **variable** qui est commune à tous les langages de programmation. Toutefois, Python est un langage dit « orienté objet », il se peut que dans la suite du cours, nous employions le mot **objet** pour désigner une variable. Par exemple, « une variable de type entier » sera pour nous équivalent à « un objet de type entier ». Nous verrons dans le chapitre 23 *Avoir la classe avec les objets* (en ligne) ce que le mot « objet » signifie réellement (tout comme le mot « classe »).

Par ailleurs, nous avons rencontré plusieurs fois des **fonctions** dans ce chapitre, notamment avec `type()`, `int()`, `float()` et `str()`. Dans le chapitre 1 *Introduction*, nous avons également vu la fonction `print()`. On reconnaît qu'il s'agit d'une fonction, car son nom est suivi de parenthèses (par exemple, `type()`). En Python, la syntaxe générale est `fonction()`.

Ce qui se trouve entre les parenthèses d'une fonction est appelé **argument** et c'est ce que l'on « passe » à la fonction. Dans l'instruction `type(2)`, c'est l'entier `2` qui est l'argument passé à la fonction `type()`. Pour l'instant, on retiendra qu'une fonction est une sorte de boîte à qui on passe un (ou plusieurs) argument(s), qui effectue une action et qui peut renvoyer un résultat ou plus généralement un objet. Par exemple, la fonction `type()` renvoie le type de la variable qu'on lui a passé en argument.

Si ces notions vous semblent obscures, ne vous inquiétez pas, au fur et à mesure que vous avancerez dans le cours, tout deviendra limpide.

2.9 Minimum et maximum

Python propose les fonctions `min()` et `max()` qui renvoient respectivement le minimum et le maximum de plusieurs entiers ou *floats* :

```
>>> min(1, -2, 4)
-2
>>> pi = 3.14
>>> e = 2.71
>>> max(e, pi)
3.14
>>> max(1, 2.4, -6)
2.4
```

Par rapport à la discussion de la rubrique précédente, `min()` et `max()` sont des exemples de fonctions prenant plusieurs arguments. En Python, quand une fonction prend plusieurs arguments, on doit les séparer par une virgule. `min()` et `max()` prennent en argument autant d'entiers et de *floats* que l'on veut, mais il en faut au moins deux.

2.10 Exercices

Conseil

Pour ces exercices, utilisez l'interpréteur Python.

2.10.1 Nombres de Friedman

Les nombres de Friedman [5] sont des nombres qui peuvent s'exprimer avec tous leurs chiffres dans une expression mathématique.

Par exemple, 347 est un nombre de Friedman, car il peut s'écrire sous la forme $4+7^3$. De même pour 127 qui peut s'écrire sous la forme 2^7-1.

Déterminez si les expressions suivantes correspondent à des nombres de Friedman. Pour cela, vous les écrirez en Python puis exécuterez le code correspondant.

- $7+3^6$
- $(3+4)^3$
- 3^6-5

5. https://fr.wikipedia.org/wiki/Nombre_de_Friedman

- $(1+2^8) \times 5$
- $(2+1^8)^7$

2.10.2 Prédire le résultat : opérations

Essayez de prédire le résultat de chacune des instructions suivantes, puis vérifiez-le dans l'interpréteur Python :

- `(1+2)**3`
- `"Da" * 4`
- `"Da" + 3`
- `("Pa"+"La") * 2`
- `("Da"*4) / 2`
- `5 / 2`
- `5 // 2`
- `5 % 2`

2.10.3 Prédire le résultat : opérations et conversions de types

Essayez de prédire le résultat de chacune des instructions suivantes, puis vérifiez-le dans l'interpréteur Python :

- `str(4) * int("3")`
- `int("3") + float("3.2")`
- `str(3) * float("3.2")`
- `str(3/4) * 2`

CHAPITRE 3

Affichage

3.1 La fonction `print()`

Dans le chapitre 1 *Introduction*, nous avons rencontré la fonction `print()` qui affiche une chaîne de caractères (le fameux `"Hello world!"`). En fait, la fonction `print()` affiche l'argument qu'on lui passe entre parenthèses **et** un retour à ligne. Ce retour à ligne supplémentaire est ajouté par défaut. Si toutefois, on ne veut pas afficher ce retour à la ligne, on peut utiliser l'argument par « mot-clé » `end` :

```
1  >>> print("Hello world!")
2  Hello world!
3  >>> print("Hello world!", end="")
4  Hello world!>>>
```

Ligne 1. On a utilisé l'instruction `print()` classiquement en passant la chaîne de caractères `"Hello world!"` en argument.

Ligne 3. On a ajouté un second argument `end=""`, en précisant le mot-clé `end`. Nous aborderons les arguments par mot-clé dans le chapitre 10 *Fonctions*. Pour l'instant, dites-vous que cela modifie le comportement par défaut des fonctions.

Ligne 4. L'effet de l'argument `end=""` est que les trois chevrons `>>>` se retrouvent collés après la chaîne de caractères `"Hello world!"`.

Une autre manière de s'en rendre compte est d'utiliser deux fonctions `print()` à la suite. Dans la portion de code suivante, le caractère « `;` » sert à séparer plusieurs instructions Python sur une même ligne :

```
1  >>> print("Hello") ; print("Joe")
2  Hello
3  Joe
4  >>> print("Hello", end="") ; print("Joe")
5  HelloJoe
6  >>> print("Hello", end=" ") ; print("Joe")
7  Hello Joe
```

La fonction `print()` peut également afficher le contenu d'une variable quel que soit son type. Par exemple, pour un entier :

```
1  >>> var = 3
2  >>> print(var)
3  3
```

Il est également possible d'afficher le contenu de plusieurs variables (quel que soit leur type) en les séparant par des virgules :

```
>>> x = 32
>>> nom = "John"
>>> print(nom, "a", x, "ans")
John a 32 ans
```

Python a écrit une phrase complète en remplaçant les variables `x` et `nom` par leur contenu. Vous remarquerez que pour afficher plusieurs éléments de texte sur une seule ligne, nous avons utilisé le séparateur « , » entre les différents éléments. Python a également ajouté un espace à chaque fois que l'on utilisait le séparateur « , ». On peut modifier ce comportement en passant à la fonction `print()` l'argument par mot-clé `sep` :

```
>>> x = 32
>>> nom = "John"
>>> print(nom, "a", x, "ans", sep="")
Johna32ans
>>> print(nom, "a", x, "ans", sep="-")
John-a-32-ans
>>> print(nom, "a", x, "ans", sep="_")
John_a_32_ans
```

Pour afficher deux chaînes de caractères l'une à côté de l'autre, sans espace, on peut soit les concaténer, soit utiliser l'argument par mot-clé `sep` avec une chaîne de caractères vide :

```
>>> ani1 = "chat"
>>> ani2 = "souris"
>>> print(ani1, ani2)
chat souris
>>> print(ani1 + ani2)
chatsouris
>>> print(ani1, ani2, sep="")
chatsouris
```

3.2 Messages d'erreur

Nous avons déjà croisé des messages d'erreur dans le chapitre précédent sur les variables. Nous vous expliquons ici comment les lire.

Depuis la version 3.10 de Python, l'interpréteur renvoie des messages explicites lorsqu'on fait une erreur de syntaxe. Par exemple, on considère le script suivant (enregistré dans un fichier nommé `test.py`) qui contient plusieurs erreurs. Les voyez-vous ?

```
print("chat"
print("souris")
print(1 / 0)
print(int("deux"))
```

Vous avez sans doute repéré l'oubli d'une parenthèse fermante en **ligne 1**. Lorsqu'on lance le script, on obtient :

```
$ python test.py
  File "test.py", line 1
    print("chat"
         ^
SyntaxError: '(' was never closed
```

Comment doit-on lire ce message d'erreur ? Et bien cela se fait toujours du bas vers le haut. Le message s'appelle une *Traceback* et contient plusieurs types d'information :

- Tout en bas : On a le type d'erreur qui a été généré (on verra plus tard que cela s'appelle en réalité une exception). Ici une erreur de syntaxe appelée `SyntaxError`. Puis sur la même ligne, un indice supplémentaire (ici l'absence d'une parenthèse).
- Un peu plus haut : une description de l'erreur où on voit la parenthèse ouverte qui n'a jamais été fermée.
- Encore plus haut : le numéro de ligne dans le code où l'erreur a été détectée.

Avec cette *Traceback* il devient facile de corriger l'erreur.

Attention

Dans les vieilles versions de Python (< 3.10), la même erreur conduisait à une *Traceback* beaucoup moins claire :

```
$ python3.5 test.py
  File "test.py", line 2
    print("souris")
        ^
SyntaxError: invalid syntax
```

L'interpréteur vous indiquait une erreur de syntaxe en **ligne 2** alors que l'oubli de parenthèse était en **ligne 1** ! Pour cette raison, utilisez dans la mesure du possible une version récente de Python (3.12 ou 3.13).

Si nous corrigeons la **ligne 1** en mettant la parenthèse finale et que nous relançons le script, nous aurons cette fois-ci une erreur due à une division par zéro :

```
$ python test.py
chat
souris
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    print(1 / 0)
          ~~^~~
ZeroDivisionError: division by zero
```

Si nous corrigeons cette erreur en **ligne 3** en évitant la division par zéro (par exemple en mettant `print(1 / 1)`), l'exécution donnera un autre message d'erreur dû à la **ligne 4** où la transformation d'une chaîne de caractères en entier n'est pas possible :

```
$ python test.py
chat
souris
1
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    print(int("deux"))
          ^^^^^^^^^^
ValueError: invalid literal for int() with base 10: 'deux'
```

À nouveau dans ces deux derniers exemples de *Traceback*, vous voyez qu'on a la même construction. Tout en bas, le type d'erreur, puis en remontant une description du problème et le numéro de ligne où l'erreur a été détectée.

Conseil

Il est important de bien lire chaque message d'erreur généré par Python. En général, la clé du problème est mentionnée dans ce message vous donnant des éléments pour le corriger.

3.3 Écriture formatée et *f-strings*

3.3.1 Définitions

Définition

L'écriture formatée est un mécanisme permettant d'afficher des variables avec un format précis, par exemple justifiées à gauche ou à droite, ou encore avec un certain nombre de décimales pour les *floats*. L'écriture formatée est incontournable lorsqu'on veut créer des fichiers organisés en « belles colonnes » comme par exemple les fichiers PDB (pour en savoir plus sur ce format, reportez-vous à l'annexe A *Quelques formats de données en biologie*).

Depuis la version 3.6, Python a introduit les *f-strings* pour mettre en place l'écriture formatée que nous allons décrire en détail dans cette rubrique. Il existe d'autres manières pour formater des chaînes de caractères qui étaient utilisées

avant la version 3.6, nous expliquons cela dans le chapitre 26 *Remarques complémentaires* (en ligne). Toutefois, nous vous conseillons vivement l'utilisation des *f-strings* si vous débutez l'apprentissage de Python. Il est inutile d'apprendre les anciennes manières.

Définition

f-string est le diminutif de *formatted string literals*. Mais encore ? Dans le chapitre précédent, nous avons vu les chaînes de caractères ou encore *strings* qui étaient représentées par un texte entouré de guillemets simples ou doubles. Par exemple :

```
"Ceci est une chaîne de caractères"
```

L'équivalent en *f-string* est la même chaîne de caractères précédée du caractère `f` **sans espace** entre les deux :

```
f"Ceci est une chaîne de caractères"
```

Ce caractère `f` avant les guillemets va indiquer à Python qu'il s'agit d'une *f-string* mettant en place le mécanisme de l'écriture formatée, contrairement à une *string* normale.

Nous expliquons plus en détail dans le chapitre 11 *Plus sur les chaînes de caractères* pourquoi on doit mettre ce `f` et quel est le mécanisme sous-jacent.

3.3.2 Prise en main des *f-strings*

Les *f-strings* permettent une meilleure organisation de l'affichage des variables. Reprenons l'exemple ci-dessus à propos de notre ami John :

```
>>> x = 32
>>> nom = "John"
>>> print(f"{nom} a {x} ans")
John a 32 ans
```

Il suffit de passer un nom de variable au sein de chaque couple d'accolades et Python les remplace par leur contenu. La syntaxe apparait plus lisible que l'équivalent vu précédemment :

```
>>> print(nom, "a", x, "ans")
John a 32 ans
```

Bien sûr, il ne faut pas omettre le `f` avant le premier guillemet, sinon Python prendra cela pour une chaîne de caractères normale et ne mettra pas en place le mécanisme de remplacement entre les accolades :

```
>>> print("{nom} a {x} ans")
{nom} a {x} ans
```

Remarque

Une variable est utilisable plus d'une fois pour une *f-string* donnée :

```
>>> var = "to"
>>> print(f"{var} et {var} font {var}{var}")
to et to font toto
>>>
```

Enfin, il est possible de mettre entre les accolades des valeurs numériques ou des chaînes de caractères :

```
>>> print(f"J'affiche l'entier {10} et le float {3.14}")
J'affiche l'entier 10 et le float 3.14
>>> print(f"J'affiche la chaine {'Python'}")
J'affiche la chaine Python
```

Même si cela ne présente que peu d'intérêt pour l'instant, il s'agit d'une commande Python parfaitement valide. Nous verrons des exemples plus pertinents par la suite. Cela fonctionne avec n'importe quel type de variable (entiers, chaînes

de caractères, *floats*, etc.). Attention toutefois pour les chaînes de caractères, utilisez des guillemets simples au sein des accolades si vous définissez votre *f-string* avec des guillemets doubles.

3.3.3 Spécification de format

Les *f-strings* permettent de remplacer des variables au sein d'une chaîne de caractères. On peut également spécifier le format de leur affichage.

Prenons un exemple. Imaginez que vous vouliez calculer, puis afficher, la proportion de GC d'un génome. La proportion de GC s'obtient comme la somme des bases Guanine (G) et Cytosine (C) divisée par le nombre total de bases (A, T, C, G) du génome considéré. Si on a, par exemple, 4 500 bases G et 2 575 bases C, pour un total de 14 800 bases, vous pourriez procéder comme suit (notez bien l'utilisation des parenthèses pour gérer les priorités des opérateurs) :

```
>>> prop_GC = (4500 + 2575) / 14800
>>> print("La proportion de GC est", prop_GC)
La proportion de GC est 0.4780405405405405
```

Le résultat obtenu présente trop de décimales (seize dans le cas présent). Pour écrire le résultat plus lisiblement, vous pouvez spécifier dans les accolades `{}` le format qui vous intéresse. Dans le cas présent, vous voulez formater un *float* pour l'afficher avec deux puis trois décimales :

```
>>> print(f"La proportion de GC est {prop_GC:.2f}")
La proportion de GC est 0.48
>>> print(f"La proportion de GC est {prop_GC:.3f}")
La proportion de GC est 0.478
```

Détaillons le contenu des accolades de la première ligne (`{prop_GC:.2f}`) :

- D'abord on a le nom de la variable à formatter, `prop_GC`, c'est indispensable avec les *f-strings*.
- Ensuite on rencontre les deux-points `:`, ceux-ci indiquent que ce qui suit va spécifier le format dans lequel on veut afficher la variable `prop_GC`.
- À droite des deux-points on trouve `.2f` qui indique ce format : la lettre `f` indique qu'on souhaite afficher la variable sous forme d'un *float*, les caractères `.2` indiquent la précision voulue, soit ici deux chiffres après la virgule.

Notez enfin que le formatage avec `.xf` (`x` étant un entier positif) renvoie un résultat arrondi.

Vous pouvez aussi formater des entiers avec la lettre `d` (ici `d` veut dire *decimal integer*) :

```
>>> nb_G = 4500
>>> print(f"Ce génome contient {nb_G:d} guanines")
Ce génome contient 4500 guanines
```

ou mettre plusieurs nombres dans une même chaîne de caractères :

```
>>> nb_G = 4500
>>> nb_C = 2575
>>> print(f"Ce génome contient {nb_G:d} G et {nb_C:d} C, "
...       f"soit une proportion de {prop_GC:.2f}")
Ce génome contient 4500 G et 2575 C, soit une proportion de 0.48
>>> perc_GC = prop_GC * 100
>>> print(f"Ce génome contient {nb_G:d} G et {nb_C:d} C, "
...       f"soit un %GC de {perc_GC:.2f} %")
Ce génome contient 4500 G et 2575 C, soit un %GC de 47.80 %
```

Les instructions étant longues dans cet exemple, nous avons coupé chaque chaîne de caractères sur deux lignes. Il faut mettre à chaque fois le `f` pour préciser à Python qu'on utilise une *f-string*. Les `...` indiquent que l'interpréteur attend que l'on ferme la parenthèse du `print` entamé sur la ligne précédente. Nous reverrons cette syntaxe dans le chapitre 11 *Plus sur les chaînes de caractères*.

Enfin, il est possible de préciser sur combien de caractères vous voulez qu'un résultat soit écrit et comment se fait l'alignement (à gauche, à droite), ou si vous voulez centrer le texte. Dans la portion de code suivante, le caractère `;` sert de séparateur entre les instructions sur une même ligne :

```
>>> print(10) ; print(1000)
10
1000
>>> print(f"{10:>6d}") ; print(f"{1000:>6d}")
    10
  1000
>>> print(f"{10:<6d}") ; print(f"{1000:<6d}")
10
1000
>>> print(f"{10:^6d}") ; print(f"{1000:^6d}")
  10
 1000
>>> print(f"{10:*^6d}") ; print(f"{1000:*^6d}")
**10**
*1000*
>>> print(f"{10:0>6d}") ; print(f"{1000:0>6d}")
000010
001000
```

Notez que `>` spécifie un alignement à droite, `<` spécifie un alignement à gauche et `^` spécifie un alignement centré. Il est également possible d'indiquer le caractère qui servira de remplissage lors des alignements (l'espace est le caractère par défaut).

Ce formatage est également possible sur des chaînes de caractères avec la lettre `s` (comme *string*) :

```
>>> print("atom HN") ; print("atom HDE1")
atom HN
atom HDE1
>>> print(f"atom {'HN':>4s}") ; print(f"atom {'HDE1':>4s}")
atom   HN
atom HDE1
```

Vous voyez tout de suite l'énorme avantage de l'écriture formatée. Elle vous permet d'écrire en colonnes parfaitement alignées. Nous verrons que ceci est très pratique si l'on veut écrire les coordonnées des atomes d'une molécule au format PDB (pour en savoir plus sur ce format, reportez-vous à l'annexe A *Quelques formats de données en biologie*).

Pour les *floats*, il est possible de combiner le nombre de caractères à afficher avec le nombre de décimales :

```
>>> print(f"{perc_GC:7.3f}")
 47.804
>>> print(f"{perc_GC:10.3f}")
    47.804
```

L'instruction `7.3f` signifie que l'on souhaite écrire un *float* avec 3 décimales et formaté sur 7 caractères (par défaut justifiés à droite). L'instruction `10.3f` fait la même chose sur 10 caractères. Remarquez que le séparateur décimal `.` compte pour un caractère. De même, si on avait un nombre négatif, le signe `-` compterait aussi pour un caractère.

3.3.4 Autres détails sur les *f-strings*

Si on veut afficher des accolades littérales avec les *f-strings*, il faut les doubler pour échapper au formatage :

```
>>> print(f"Accolades littérales {{}} ou {{ ou }} "
...       f"et pour le formatage {10}")
Accolades littérales {} ou { ou } et pour le formatage 10
```

Une remarque importante, si on ne met pas de variable à formater entre les accolades dans une *f-string*, cela conduit à une erreur :

```
>>> print(f"accolades sans variable {}")
  File "<stdin>", line 1
SyntaxError: f-string: empty expression not allowed
```

Enfin, il est important de bien comprendre qu'une *f-string* est indépendante de la fonction `print()`. Si on donne une *f-string* à la fonction `print()`, Python évalue d'abord la *f-string* et c'est la chaîne de caractères qui en résulte qui est affichée à l'écran. Tout comme dans l'instruction `print(5*5)`, c'est d'abord la multiplication (`5*5`) qui est évaluée, puis son résultat qui est affiché à l'écran. On peut s'en rendre compte de la manière suivante dans l'interpréteur :

```
>>> f"{perc_GC:10.3f}"
'    47.804'
>>> type(f"{perc_GC:10.3f}")
<class 'str'>
```

Python considère le résultat de l'instruction `f"{perc_GC:10.3f}"` comme une chaîne de caractères et la fonction `type()` nous le confirme.

3.3.5 Expressions dans les *f-strings*

Une fonctionnalité extrêmement puissante des *f-strings* est de supporter des expressions Python au sein des accolades. Ainsi, il est possible d'y mettre directement une opération ou encore un appel à une fonction :

```
>>> print(f"Le résultat de 5 * 5 vaut {5 * 5}")
Le résultat de 5 * 5 vaut 25
>>> print(f"Résultat d'une opération avec des floats : {(4.1 * 6.7)}")
Résultat d'une opération avec des floats : 27.47
>>> print(f"Le minimum est {min(1, -2, 4)}")
Le minimum est -2
>>> entier = 2
>>> print(f"Le type de {entier} est {type(entier)}")
Le type de 2 est <class 'int'>
```

Nous aurons l'occasion de revenir sur cette fonctionnalité au fur et à mesure de ce cours.

Les possibilités offertes par les *f-strings* sont nombreuses. Pour vous y retrouver dans les différentes options de formatage, nous vous conseillons de consulter ce mémo [1] (en anglais).

3.4 Écriture scientifique

Pour les nombres très grands ou très petits, l'écriture formatée permet d'afficher un nombre en notation scientifique (sous forme de puissance de 10) avec la lettre `e` :

```
>>> print(f"{1_000_000_000:e}")
1.000000e+09
>>> print(f"{0.000_000_001:e}")
1.000000e-09
```

Il est également possible de définir le nombre de chiffres après la virgule. Dans l'exemple ci-dessous, on affiche un nombre avec aucun, 3 et 6 chiffres après la virgule :

```
>>> avogadro_number = 6.022_140_76e23
>>> print(f"{avogadro_number:.0e}")
6e+23
>>> print(f"{avogadro_number:.3e}")
6.022e+23
>>> print(f"{avogadro_number:.6e}")
6.022141e+23
```

3.5 Exercices

Conseil

Pour les exercices 2 à 6, utilisez l'interpréteur Python.

3.5.1 Affichage dans l'interpréteur et dans un programme

Ouvrez l'interpréteur Python et tapez l'instruction `1+1`. Que se passe-t-il ?

Écrivez la même chose dans un script `test.py` que vous allez créer avec un éditeur de texte. Exécutez ce script en tapant `python test.py` dans un *shell*. Que se passe-t-il ? Pourquoi ? Faites en sorte d'afficher le résultat de l'addition `1+1` en exécutant le script dans un *shell*.

1. https://fstring.help/cheat/

3.5.2 Poly-A

Générez une chaîne de caractères représentant un brin d'ADN poly-A (c'est-à-dire qui ne contient que des bases A) de 20 bases de longueur, sans taper littéralement toutes les bases.

3.5.3 Poly-A et poly-GC

Sur le modèle de l'exercice précédent, générez en une ligne de code un brin d'ADN poly-A (AAAA…) de 20 bases suivi d'un poly-GC régulier (GCGCGC…) de 40 bases.

3.5.4 Écriture formatée

En utilisant l'écriture formatée, affichez en une seule ligne les variables `a`, `b` et `c` dont les valeurs sont respectivement la chaîne de caractères `"salut"`, le nombre entier `102` et le *float* `10.318`. La variable `c` sera affichée avec deux décimales.

3.5.5 Écriture formatée 2

Dans un script `percGC.py`, calculez un pourcentage de GC avec l'instruction suivante :

```
perc_GC = ((4500 + 2575)/14800)*100
```

Ensuite, affichez le contenu de la variable `perc_GC` à l'écran avec 0, 1, 2 puis 3 décimales sous forme arrondie en utilisant l'écriture formatée et les *f-strings*. On souhaite que le programme affiche la sortie suivante :

```
Le pourcentage de GC est 48     %
Le pourcentage de GC est 47.8   %
Le pourcentage de GC est 47.80  %
Le pourcentage de GC est 47.804 %
```

3.5.6 Décomposition de fractions

Utilisez l'opérateur modulo (`%`) et l'opérateur division entière (`//`) pour simplifier des fractions, connaissant leur numérateur et leur dénominateur, et afficher le résultat avec des *f-strings*.

Par exemple pour la fraction $\frac{7}{3}$, le numérateur vaut 7 et le dénominateur vaut 3, et le résultat s'affichera sous la forme :

```
7/3 = 2 + 1/3
```

Ici, 2 est le résultat de la division entière du numérateur par le dénominateur et 1 est le reste de la division entière du numérateur par le dénominateur.

Faites de même pour les fractions suivantes :

$$\frac{9}{4}, \frac{23}{5}, \frac{21}{8} \text{et} \frac{7}{2}$$

CHAPITRE 4

Listes

4.1 Définition

Définition

Une **liste** est une structure de données qui contient une collection d'objets Python. Il s'agit d'un nouveau type par rapport aux entiers, *float*, booléens et chaînes de caractères que nous avons vus jusqu'à maintenant. On parle aussi d'**objet séquentiel** en ce sens qu'il contient une séquence d'autres objets.

Python autorise la construction de liste contenant des valeurs de types différents (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité. Une liste est déclarée par une série de **valeurs** (n'oubliez pas les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des **virgules**, et le tout encadré par des **crochets**. En voici quelques exemples :

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> tailles = [5, 2.5, 1.75, 0.15]
>>> mixte = ["girafe", 5, "souris", 0.15]
>>> animaux
['girafe', 'tigre', 'singe', 'souris']
>>> tailles
[5, 2.5, 1.75, 0.15]
>>> mixte
['girafe', 5, 'souris', 0.15]
```

Lorsque l'on affiche une liste, Python la restitue telle qu'elle a été saisie.

4.2 Utilisation

Un des gros avantages d'une liste est que vous accédez à ses éléments par leur position. Ce numéro est appelé **indice** (ou *index*) de la liste.

```
liste  : ["girafe", "tigre", "singe", "souris"]
indice :        0        1        2         3
```

Soyez très **attentif** au fait que les indices d'une liste de n éléments commencent à 0 et se terminent à $n-1$. Voyez l'exemple suivant :

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> animaux[0]
'girafe'
>>> animaux[1]
'tigre'
>>> animaux[3]
'souris'
```

Par conséquent, si on appelle l'élément d'indice 4 de notre liste, Python renverra un message d'erreur :

```
>>> animaux[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

N'oubliez pas ceci ou vous risquez d'obtenir des bugs inattendus !

4.3 Opération sur les listes

Tout comme les chaînes de caractères, les listes supportent l'opérateur + de concaténation, ainsi que l'opérateur * pour la duplication :

```
>>> ani1 = ["girafe", "tigre"]
>>> ani2 = ["singe", "souris"]
>>> ani1 + ani2
['girafe', 'tigre', 'singe', 'souris']
>>> ani1 * 3
['girafe', 'tigre', 'girafe', 'tigre', 'girafe', 'tigre']
```

L'opérateur + est très pratique pour concaténer deux listes.
Vous pouvez aussi utiliser la méthode `.append()` lorsque vous souhaitez ajouter un seul élément à la fin d'une liste.

Remarque

La notion de méthode est introduite dans la rubrique *Note sur le vocabulaire et la syntaxe* à la fin de ce chapitre.

Dans l'exemple suivant, nous allons créer une liste vide :

```
>>> liste1 = []
>>> liste1
[]
```

puis lui ajouter deux éléments, l'un après l'autre, d'abord avec la concaténation :

```
>>> liste1 = liste1 + [15]
>>> liste1
[15]
>>> liste1 = liste1 + [-5]
>>> liste1
[15, -5]
```

puis avec la méthode `.append()` :

```
>>> liste1.append(13)
>>> liste1
[15, -5, 13]
>>> liste1.append(-3)
>>> liste1
[15, -5, 13, -3]
```

Dans cet exemple, nous ajoutons des éléments à une liste en utilisant l'opérateur de concaténation + ou la méthode `.append()`.

Conseil

Nous vous conseillons dans ce cas précis d'utiliser la méthode `.append()`, dont la syntaxe est plus élégante.

Nous reverrons en détail la méthode `.append()` dans le chapitre 12 *Plus sur les listes*.

4.4 Indiçage négatif

La liste peut également être indexée avec des nombres négatifs selon le modèle suivant :

```
liste          : ["girafe", "tigre", "singe", "souris"]
indice positif :        0        1        2         3
indice négatif :       -4       -3       -2        -1
```

ou encore :

```
liste          : ["A", "B", "C", "D", "E", "F"]
indice positif :   0    1    2    3    4    5
indice négatif :  -6   -5   -4   -3   -2   -1
```

Les indices négatifs reviennent à compter à partir de la fin. Leur principal avantage est que vous pouvez accéder au dernier élément d'une liste à l'aide de l'indice `-1` sans pour autant connaître la longueur de cette liste. L'avant-dernier élément a lui l'indice `-2`, l'avant-avant dernier l'indice `-3`, etc. :

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> animaux[-1]
'souris'
>>> animaux[-2]
'singe'
```

Pour accéder au premier élément de la liste avec un indice négatif, il faut par contre connaître le bon indice :

```
>>> animaux[-4]
'girafe'
```

Dans ce cas, on utilise plutôt `animaux[0]`.

4.5 Tranches

Un autre avantage des listes est la possibilité de sélectionner une partie d'une liste en utilisant un indiçage construit sur le modèle `[m:n+1]` pour récupérer tous les éléments, du émième au énième (de l'élément `m` inclu à l'élément `n+1` exclu). On dit alors qu'on récupère une **tranche** de la liste, par exemple :

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> animaux[0:2]
['girafe', 'tigre']
>>> animaux[0:3]
['girafe', 'tigre', 'singe']
>>> animaux[0:]
['girafe', 'tigre', 'singe', 'souris']
>>> animaux[:]
['girafe', 'tigre', 'singe', 'souris']
>>> animaux[1:]
['tigre', 'singe', 'souris']
>>> animaux[1:-1]
['tigre', 'singe']
```

Notez que lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole deux-points `:`, Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement.

On peut aussi préciser le pas en ajoutant un symbole deux-points supplémentaire et en indiquant le pas par un entier :

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> animaux[0:3:2]
['girafe', 'singe']
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::2]
[0, 2, 4, 6, 8]
>>> x[::3]
[0, 3, 6, 9]
>>> x[1:6:3]
[1, 4]
```

Finalement, on se rend compte que l'accès au contenu d'une liste fonctionne sur le modèle `liste[début:fin:pas]`.

4.6 Fonction `len()`

L'instruction `len()` vous permet de connaître la longueur d'une liste, c'est-à-dire le nombre d'éléments que contient la liste. Voici un exemple d'utilisation :

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> len(animaux)
4
>>> len([1, 2, 3, 4, 5, 6, 7, 8])
8
```

4.7 Les fonctions `range()` et `list()`

L'instruction `range()` est une fonction spéciale en Python qui génère des nombres entiers compris dans un intervalle. Lorsqu'elle est utilisée en combinaison avec la fonction `list()`, on obtient une liste d'entiers. Par exemple :

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La commande `list(range(10))` a généré une liste contenant tous les nombres entiers de 0 inclus à 10 **exclu**. Nous verrons l'utilisation de la fonction `range()` toute seule dans le chapitre 5 *Boucles et comparaisons*.

Dans l'exemple ci-dessus, la fonction `range()` a pris un argument, mais elle peut également prendre deux ou trois arguments, voyez plutôt :

```
>>> list(range(0, 5))
[0, 1, 2, 3, 4]
>>> list(range(15, 20))
[15, 16, 17, 18, 19]
>>> list(range(0, 1000, 200))
[0, 200, 400, 600, 800]
>>> list(range(2, -2, -1))
[2, 1, 0, -1]
```

L'instruction `range()` fonctionne sur le modèle `range([début,] fin[, pas])`. Les arguments entre crochets sont optionnels. Pour obtenir une liste de nombres entiers, il faut l'utiliser systématiquement avec la fonction `list()`.

Enfin, prenez garde aux arguments optionnels par défaut (`0` pour `début` et `1` pour `pas`) :

```
>>> list(range(10,0))
[]
```

Ici la liste est vide car Python a pris la valeur du pas par défaut qui est de 1. Ainsi, si on commence à 10 et qu'on avance par pas de 1, on ne pourra jamais atteindre 0. Python génère ainsi une liste vide. Pour éviter ça, il faudrait, par exemple, préciser un pas de -1 pour obtenir une liste d'entiers décroissants :

```
>>> list(range(10,0,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

4.8 Listes de listes

Pour finir, sachez qu'il est tout à fait possible de construire des listes de listes. Cette fonctionnalité peut parfois être très pratique. Par exemple :

```
>>> prairie1 = ["girafe", 4]
>>> prairie2 = ["tigre", 2]
>>> prairie3 = ["singe", 5]
>>> savane = [prairie1, prairie2, prairie3]
>>> savane
[['girafe', 4], ['tigre', 2], ['singe', 5]]
```

Dans cet exemple, chaque sous-liste contient une catégorie d'animal et le nombre d'animaux pour chaque catégorie.

Pour accéder à un élément de la liste, on utilise l'indiçage habituel :

```
>>> savane[1]
['tigre', 2]
```

Pour accéder à un élément de la sous-liste, on utilise un double indiçage :

```
>>> savane[1][0]
'tigre'
>>> savane[1][1]
2
```

On verra un peu plus loin qu'il existe en Python des dictionnaires qui sont également très pratiques pour stocker de l'information structurée. On verra aussi qu'il existe un module nommé *NumPy* qui permet de créer des listes ou des tableaux de nombres (vecteurs et matrices) et de les manipuler.

4.9 Minimum, maximum et somme d'une liste

Les fonctions `min()`, `max()` et `sum()` renvoient respectivement le minimum, le maximum et la somme d'une liste passée en argument :

```
>>> liste1 = list(range(10))
>>> liste1
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sum(liste1)
45
>>> min(liste1)
0
>>> max(liste1)
9
```

Même si en théorie ces fonctions peuvent prendre en argument une liste de *strings*, on les utilisera la plupart du temps avec des types numériques (liste d'entiers et / ou de *floats*).

Nous avions déjà croisé `min()`, `max()` dans le chapitre 2 *Variables*. Ces deux fonctions pouvaient prendre plusieurs arguments entiers et / ou *floats*, par exemple :

```
>>> min(3, 4)
3
```

Attention toutefois à ne pas mélanger entiers et *floats* d'une part avec une liste d'autre part, car cela renvoie une erreur :

```
>>> min(liste1, 3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'list'
```

Soit on passe plusieurs entiers et / ou *floats* en argument, soit on passe une liste unique.

4.10 Problème avec les copies de listes

Nous attirons votre attention sur un comportement de Python qui peut paraitre étrange lorsqu'on copie une liste :

```
>>> liste1 = list(range(5))
>>> list(range(5))
>>> liste1
[0, 1, 2, 3, 4]
>>> liste2 = liste1
>>> liste2
[0, 1, 2, 3, 4]
>>> liste1[3] = -50
>>> liste1
[0, 1, 2, -50, 4]
>>> liste2
[0, 1, 2, -50, 4]
```

Comme vous voyez en ligne 8, la modification de `liste1` a modifié également `liste2`. Cela vient du fait que Python a effectué la copie de liste en ligne 5 par *référence*. Ainsi, les deux listes pointent vers le même objet dans la mémoire. Pour contrer ce problème et faire en sorte que `liste2` soit bien distincte de `liste1`, on peut utiliser la fonction `list()` :

```
>>> liste1 = list(range(5))
>>> liste1
[0, 1, 2, 3, 4]
>>> liste2 = list(liste1)
>>> liste2
[0, 1, 2, 3, 4]
>>> liste1[3] = -50
>>> liste1
[0, 1, 2, -50, 4]
>>> liste2
[0, 1, 2, 3, 4]
```

Attention

Cette astuce ne fonctionne que pour des listes à une dimension (c'est-à-dire pour des listes qui ne contiennent que des éléments de type simple comme des entiers, des floats, des chaînes de caractères et des booléens), mais pas pour des listes de listes. Le chapitre 12 *Plus sur les listes* explique l'origine de ce comportement et comment s'en sortir à tous les coups.

4.11 Note sur le vocabulaire et la syntaxe

Revenons quelques instants sur la notion de **méthode** abordée dans ce chapitre avec `.append()`. En Python, on peut considérer chaque variable comme un objet sur lequel on peut appliquer des méthodes. Une méthode est simplement une fonction qui utilise et/ou agit sur l'objet lui-même, les deux étant connectés par un point. La syntaxe générale est de la forme `objet.méthode()`.

Dans l'exemple suivant :

```
>>> liste1 = [1, 2]
>>> liste1.append(3)
>>> liste1
[1, 2, 3]
```

la méthode `.append()` est liée à `liste1` qui est un objet de type liste. La méthode modifie l'objet liste en lui ajoutant un élément.

Nous aurons de nombreuses occasions de revoir cette notation `objet.méthode()`.

4.12 Exercices

Conseil

Pour ces exercices, utilisez l'interpréteur Python.

4.12.1 Prédire la sortie

Soit les trois lignes de code suivantes :

```
liste1 = list(range(10, 15))
var = 0
var2 = 10
```

Prédisez le comportement de chaque instruction ci-dessous, sans les recopier dans un script ni dans l'interpréteur Python :

- `print(liste1[2])`
- `print(liste1[var])`
- `print(liste1[var2])`
- `print(liste1["var"])`

Lorsqu'une instruction produit une erreur, identifiez pourquoi.

4.12.2 Jours de la semaine

Constituez une liste `semaine` contenant les sept jours de la semaine.

1. À partir de cette liste, comment récupérez-vous seulement les cinq premiers jours de la semaine d'une part, et ceux du week-end d'autre part ? Utilisez pour cela l'indiçage.
2. Cherchez un autre moyen pour arriver au même résultat (*en utilisant un autre indiçage*).
3. Trouvez deux manières pour accéder au dernier jour de la semaine.
4. Inversez les jours de la semaine en une commande.

4.12.3 Saisons

Créez quatre listes `hiver`, `printemps`, `ete` et `automne` contenant les mois correspondants à ces saisons. Créez ensuite une liste `saisons` contenant les listes `hiver`, `printemps`, `ete` et `automne`. Prévoyez ce que renvoient les instructions suivantes, puis vérifiez-le dans l'interpréteur :

1. `saisons[2]`
2. `saisons[1][0]`
3. `saisons[1:2]`
4. `saisons[:][1]`. Comment expliquez-vous ce dernier résultat ?

4.12.4 Table de multiplication par 9

Affichez la table de multiplication par 9 en une seule commande avec les instructions `range()` et `list()`.

4.12.5 Nombres pairs

Répondez à la question suivante en une seule commande. Combien y a-t-il de nombres pairs dans l'intervalle `[2, 10000]` inclus ?

CHAPITRE 5

Boucles et comparaisons

5.1 Boucles `for`

5.1.1 Principe

En programmation, on est souvent amené à répéter plusieurs fois une instruction. Incontournables à tout langage de programmation, les boucles vont nous aider à réaliser cette tâche répétitive de manière compacte et efficace.

Imaginez par exemple que vous souhaitiez afficher les éléments d'une liste les uns après les autres. Dans l'état actuel de vos connaissances, il faudrait taper quelque chose du style :

```
1 animaux = ["girafe", "tigre", "singe", "souris"]
2 print(animaux[0])
3 print(animaux[1])
4 print(animaux[2])
5 print(animaux[3])
```

Si votre liste ne contient que 4 éléments, ceci est encore faisable mais imaginez qu'elle en contienne 100 voire 1 000 ! Pour remédier à cela, il faut utiliser les boucles . Regardez l'exemple suivant :

```
1 >>> animaux = ["girafe", "tigre", "singe", "souris"]
2 >>> for animal in animaux:
3 ...     print(animal)
4 ...
5 girafe
6 tigre
7 singe
8 souris
```

Commentons en détails ce qu'il s'est passé dans cet exemple :

La variable `animal` est appelée **variable d'itération** , elle prend successivement les différentes valeurs de la liste `animaux` à chaque **itérations** (ou tour) de boucle. On verra un peu plus loin dans ce chapitre que l'on peut choisir le nom que l'on veut pour cette variable. Celle-ci est créée par Python la première fois que la ligne contenant le `for` est exécutée (si elle existait déjà son contenu serait écrasé). Une fois la boucle terminée, cette variable d'itération `animal` n'est pas détruite et conserve la dernière valeur de la liste `animaux` (ici la chaîne de caractères `"souris"`).

Notez bien les types des variables utilisées ici :

- `animaux` est une **liste** sur laquelle on itère ;
- `animal` est une **chaîne de caractères** car chaque élément de la liste `animaux` est une chaîne de caractères.

Nous verrons plus loin que la variable d'itération peut être de n'importe quel type selon la liste parcourue. En Python, une boucle itère la plupart du temps sur un objet dit **séquentiel** (c'est-à-dire un objet constitué d'autres objets) tel qu'une liste. De tels objets sont dits **itérables** car on peut effectuer une boucle dessus. Nous verrons aussi plus tard d'autres objets séquentiels sur lesquels on peut itérer dans une boucle.

D'ores et déjà, prêtez attention au caractère **deux-points** « `:` » à la fin de la ligne débutant par `for`. Cela signifie que la boucle `for` attend un **bloc d'instructions**, en l'occurrence toutes les instructions que Python répétera à chaque itération de la boucle. On appelle ce bloc d'instructions le **corps de la boucle**. Comment indique-t-on à Python où ce bloc commence et se termine ? Cela est signalé uniquement par l'**indentation**, c'est-à-dire le décalage vers la droite de la (ou des) ligne(s) du bloc d'instructions.

Remarque

Les notions de bloc d'instruction et d'indentations ont été introduites dans le chapitre 1 *Introduction*.

Dans l'exemple suivant, le corps de la boucle contient deux instructions (ligne 2 et ligne 3) car elles sont indentées par rapport à la ligne débutant par `for` :

```
1 for animal in animaux:
2     print(animal)
3     print(animal*2)
4 print("C'est fini")
```

La ligne 4 ne fait pas partie du corps de la boucle car elle est au même niveau que le `for` (c'est-à-dire non indentée par rapport au `for`). Notez également que chaque instruction du corps de la boucle doit être indentée de la même manière (ici 4 espaces).

Remarque

Outre une meilleure lisibilité, les deux-points et l'**indentation** sont formellement requis en Python. Même si on peut indenter comme on veut (plusieurs espaces ou plusieurs tabulations, mais pas une combinaison des deux), les développeurs recommandent l'utilisation de quatre espaces. Vous pouvez consulter à ce sujet le chapitre 16 *Bonnes pratiques de programmation* en Python.

Faites en sorte de configurer votre éditeur de texte favori de façon à écrire quatre espaces lorsque vous tapez sur la touche *Tab* (tabulation).

Si on oublie l'indentation, Python renvoie un message d'erreur :

```
1 >>> for animal in animaux:
2 ... print(animal)
3   File "<stdin>", line 2
4     print(animal)
5         ^
6 IndentationError: expected an indented block
```

Dans les exemples ci-dessus, nous avons exécuté une boucle en itérant directement sur une liste. Une tranche d'une liste étant elle même une liste, on peut également itérer dessus :

```
1 >>> animaux = ["girafe", "tigre", "singe", "souris"]
2 >>> for animal in animaux[1:3]:
3 ...     print(animal)
4 ...
5 tigre
6 singe
```

On a vu que les boucles `for` pouvaient utiliser une liste contenant des chaînes de caractères, mais elles peuvent tout aussi bien utiliser des listes contenant des entiers (ou n'importe quel type de variable) :

```
>>> for i in [1, 2, 3]:
...     print(i)
...
1
2
3
```

5.1.2 Fonction range()

Python possède la fonction range() que nous avons rencontrée précédemment dans le chapitre 4 *Listes*, pratique pour faire une boucle sur une liste d'entiers de manière automatique :

```
>>> for i in range(4):
...     print(i)
...
0
1
2
3
```

Dans cet exemple, nous pouvons faire plusieurs remarques importantes :

- Contrairement à la création de liste avec list(range(4)), la fonction range() peut être utilisée telle quelle dans une boucle. Il n'est pas nécessaire de taper for i in list(range(4)): même si cela fonctionnerait également.
- Comment cela est possible ? range() est une fonction qui a été spécialement conçue pour cela [1], c'est-à-dire que l'on peut itérer directement dessus. Pour Python, il s'agit d'un nouveau type : par exemple dans l'instruction x = range(3), la variable x est de type *range* (tout comme on avait les types *int*, *float*, *str* ou *list*) à utiliser spécialement avec les boucles.
- L'instruction list(range(4)) se contente de transformer un objet de type *range* en un objet de type *list*. Si vous vous souvenez bien, il s'agit d'une fonction de *casting*, qui convertit un type en un autre (voir chapitre 2 *Variables*). Il n'y aucun intérêt à utiliser dans une boucle la construction for i in list(range(4)):. C'est même contre-productif. En effet, range() se contente de stocker l'entier actuel, le pas pour passer à l'entier suivant, et le dernier entier à parcourir, ce qui revient à stocker seulement 3 nombres entiers et ce quelle que soit la longueur de la séquence, même avec un range(1000000). Si on utilisait list(range(1000000)), Python construirait d'abord une liste de 1 million d'éléments dans la mémoire puis itérerait dessus, d'où une énorme perte de temps !

5.1.3 Nommage de la variable d'itération

Dans l'exemple précédent, nous avons choisi le nom i pour la variable d'itération. Ceci est une habitude en informatique et indique en général qu'il s'agit d'un entier (le nom i vient sans doute du mot indice ou *index* en anglais). Nous vous conseillons de suivre cette convention afin d'éviter les confusions. Si vous itérez sur les indices, vous pouvez appeler la variable d'itération i (par exemple dans for i in range(4):).

Si, par contre, vous itérez sur une liste comportant des chaînes de caractères (ou tout autre type de variable), utilisez un nom explicite pour la variable d'itération. Par exemple :

```
for prenom in ["Joe", "Bill", "John"]:
```

ou

```
for proportion in [0.12, 0.53, 0.07, 0.28]:
```

5.1.4 Itération sur les indices ou les éléments

Revenons à notre liste animaux. Nous allons maintenant parcourir cette liste, mais cette fois par une itération sur ses indices :

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> for i in range(4):
...     print(animaux[i])
...
girafe
tigre
singe
souris
```

1. https://docs.python.org/fr/3/library/stdtypes.html#typesseq-range

La variable `i` prendra les valeurs successives 0, 1, 2 et 3 et on accèdera à chaque élément de la liste `animaux` par son indice (*i.e.* `animaux[i]`). Notez à nouveau le nom `i` de la variable d'itération car on itère sur les **indices**.

Quand utiliser l'une ou l'autre des deux méthodes ? La plus efficace est celle qui **réalise les itérations directement sur les éléments** :

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> for animal in animaux:
...     print(animal)
...
girafe
tigre
singe
souris
```

Remarque

Dans le chapitre 18 *Jupyter et ses notebooks*, nous mesurerons le temps d'exécution de ces deux méthodes pour vous montrer que l'itération sur les éléments est la méthode la plus rapide.

Toutefois, il se peut qu'au cours d'une boucle vous ayez besoin des indices, auquel cas vous devrez itérer sur les indices :

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> for i in range(len(animaux)):
...     print(f"L'animal {i} est un(e) {animaux[i]}")
...
L'animal 0 est un(e) girafe
L'animal 1 est un(e) tigre
L'animal 2 est un(e) singe
L'animal 3 est un(e) souris
```

Enfin, Python possède la fonction `enumerate()` qui vous permet d'itérer sur les indices et les éléments eux-mêmes :

```
>>> animaux = ["girafe", "tigre", "singe", "souris"]
>>> for i, animal in enumerate(animaux):
...     print(f"L'animal {i} est un(e) {animal}")
...
L'animal 0 est un(e) girafe
L'animal 1 est un(e) tigre
L'animal 2 est un(e) singe
L'animal 3 est un(e) souris
```

5.2 Comparaisons

Avant de passer aux boucles `while`, abordons tout de suite les **comparaisons**. Celles-ci seront reprises dans le chapitre 6 *Tests*.

Python est capable d'effectuer toute une série de comparaisons entre le contenu de deux variables, telles que :

Opérateur de comparaison	Signification
`==`	égal à
`!=`	différent de
`>`	strictement supérieur à
`>=`	supérieur ou égal à
`<`	strictement inférieur à
`<=`	inférieur ou égal à

Observez les exemples suivants avec des nombres entiers :

```
>>> x = 5
>>> x == 5
True
>>> x > 10
False
>>> x < 10
True
```

Python renvoie la valeur `True` si la comparaison est vraie et `False` si elle est fausse. `True` et `False` sont des booléens comme nous avions vu au chapitre 2 *Variables*.

Faites bien attention à ne pas confondre l'**opérateur d'affectation** `=` qui affecte une valeur à une variable et l'**opérateur de comparaison** `==` qui compare les valeurs de deux variables.

Vous pouvez également effectuer des comparaisons sur des chaînes de caractères.

```
>>> animal = "tigre"
>>> animal == "tig"
False
>>> animal != "tig"
True
>>> animal == "tigre"
True
```

Dans le cas des chaînes de caractères, *a priori* seuls les tests `==` et `!=` ont un sens. En fait, on peut aussi utiliser les opérateurs `<`, `>`, `<=` et `>=`. Dans ce cas, l'ordre alphabétique est pris en compte, par exemple :

```
>>> "a" < "b"
True
```

`"a"` est *inférieur à* `"b"` car le caractère *a* est situé avant le caractère *b* dans l'ordre alphabétique. En fait, c'est l'ordre ASCII[2] des caractères qui est pris en compte (à chaque caractère correspond un code numérique), on peut donc aussi comparer des caractères spéciaux (comme `#` ou `~`) entre eux. Enfin, on peut comparer des chaînes de caractères de plusieurs caractères :

```
>>> "ali" < "alo"
True
>>> "abb" < "ada"
True
```

Dans ce cas, Python compare les deux chaînes de caractères, caractère par caractère, de la gauche vers la droite (le premier caractère avec le premier, le deuxième avec le deuxième, etc). Dès qu'un caractère est différent entre l'une et l'autre des deux chaînes, il considère que la chaîne la plus petite est celle qui présente le caractère ayant le plus petit code ASCII (les caractères suivants de la chaîne de caractères sont ignorés dans la comparaison), comme dans l'exemple `"abb" < "ada"` ci-dessus.

5.3 Boucles `while`

Une alternative à l'instruction `for` couramment utilisée en informatique est la boucle `while`. Avec ce type de boucle, une série d'instructions est exécutée tant qu'une condition est vraie. Par exemple :

```
>>> i = 1
>>> while i <= 4:
...     print(i)
...     i = i + 1
...
1
2
3
4
```

Remarquez qu'il est encore une fois nécessaire d'indenter le bloc d'instructions correspondant au corps de la boucle (ici, les instructions lignes 3 et 4).

2. http://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange

Une boucle `while` nécessite généralement **trois éléments** pour fonctionner correctement :

1. Initialisation de la variable d'itération avant la boucle (ligne 1).
2. Test de la variable d'itération associée à l'instruction `while` (ligne 2).
3. Mise à jour de la variable d'itération dans le corps de la boucle (ligne 4).

Faites bien attention aux tests et à l'incrémentation que vous utilisez, car une erreur mène souvent à des « boucles infinies » qui ne s'arrêtent jamais. Vous pouvez néanmoins toujours stopper l'exécution d'un script Python à l'aide de la combinaison de touches *Ctrl-C* (c'est-à-dire en pressant simultanément les touches *Ctrl* et *C*). Par exemple :

```
1 i = 0
2 while i < 10:
3     print("Le Python c'est cool !")
```

Ici, nous avons omis de mettre à jour la variable `i` dans le corps de la boucle. Par conséquent, la boucle ne s'arrêtera jamais (sauf en pressant *Ctrl-C*) puisque la condition `i < 10` sera toujours vraie.

La boucle `while` combinée à la fonction `input()` peut s'avérer commode lorsqu'on souhaite demander à l'utilisateur une valeur numérique. Par exemple :

```
1  >>> i = 0
2  >>> while i < 10:
3  ...     reponse = input("Entrez un entier supérieur à 10 : ")
4  ...     i = int(reponse)
5  ...
6  Entrez un entier supérieur à 10 : 4
7  Entrez un entier supérieur à 10 : -3
8  Entrez un entier supérieur à 10 : 15
9  >>> i
10 15
```

La fonction `input()` prend en argument un message (sous la forme d'une chaîne de caractères), demande à l'utilisateur d'entrer une valeur et renvoie celle-ci sous forme d'une chaîne de caractères, qu'il faut ensuite convertir en entier (avec la fonction `int()` ligne 4). Si on reprend les trois éléments d'une boucle while, on trouve l'initialisation de la variable d'itération en ligne 1, le test de sa valeur en ligne 2, et sa mise à jour en ligne 4.

Conseil

Comment choisir entre la boucle while et la boucle for ? La boucle while s'utilisera généralement lorsqu'on ne sait pas à l'avance le nombre d'itérations (comme dans le dernier exemple). Si on connait à l'avance le nombre d'itérations, par exemple si on veut écrire 10 fois `Le Python c'est cool`, nous vous conseillons la boucle for.

5.4 Exercices

Conseil

Pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

5.4.1 Boucles de base

Soit la liste `["vache", "souris", "levure", "bacterie"]`. Affichez l'ensemble des éléments de cette liste (un élément par ligne) de trois façons différentes (deux méthodes avec `for` et une avec `while`).

5.4.2 Boucles et jours de la semaine

Constituez une liste `semaine` contenant les 7 jours de la semaine.

Écrivez une série d'instructions affichant les jours de la semaine (en utilisant une boucle `for`), ainsi qu'une autre série d'instructions affichant les jours du week-end (en utilisant une boucle `while`).

5.4.3 Nombres de 1 à 10 sur une ligne

Avec une boucle, affichez les nombres de 1 à 10 sur une seule ligne.

Conseil

Pensez à relire le début du chapitre 3 *Affichage* qui discute de la fonction `print()`.

5.4.4 Nombres pairs et impairs

Soit `impairs` la liste de nombres `[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]`. Écrivez un programme qui, à partir de la liste `impairs`, construit une liste `pairs` dans laquelle tous les éléments de `impairs` sont incrémentés de 1.

5.4.5 Calcul de la moyenne

Voici les notes d'un étudiant `[14, 9, 6, 8, 12]`. Calculez la moyenne de ces notes. Utilisez l'écriture formatée pour afficher la valeur de la moyenne avec deux décimales.

5.4.6 Produit de nombres consécutifs

Avec les fonctions `list()` et `range()`, créez la liste `entiers` contenant les nombres entiers pairs de 2 à 20 inclus.

Calculez ensuite le produit des nombres consécutifs deux à deux de `entiers` en utilisant une boucle. Exemple pour les premières itérations :

```
8
24
48
[...]
```

5.4.7 Triangle

Créez un script qui dessine un triangle comme celui-ci :

```
*
**
***
****
*****
******
*******
********
*********
**********
```

5.4.8 Triangle inversé

Créez un script qui dessine un triangle comme celui-ci :

```
**********
*********
********
*******
******
*****
****
***
**
*
```

5.4.9 Triangle gauche

Créez un script qui dessine un triangle comme celui-ci :

```
         *
        **
       ***
      ****
     *****
    ******
   *******
  ********
 *********
**********
```

5.4.10 Pyramide

Créez un script `pyra.py` qui dessine une pyramide comme celle-ci :

```
         *
        ***
       *****
      *******
     *********
    ***********
   *************
  ***************
 *****************
*******************
```

Essayez de faire évoluer votre script pour dessiner la pyramide à partir d'un nombre arbitraire de lignes `N`. Vous pourrez demander à l'utilisateur le nombre de lignes de la pyramide avec les instructions suivantes qui utilisent la fonction `input()` :

```
1  reponse = input("Entrez un nombre de lignes (entier positif): ")
2  N = int(reponse)
```

5.4.11 Parcours de matrice

Imaginons que l'on souhaite parcourir tous les éléments d'une matrice carrée, c'est-à-dire d'une matrice qui est constituée d'autant de lignes que de colonnes.

Créez un script qui parcourt chaque élément de la matrice et qui affiche le numéro de ligne et de colonne uniquement avec des boucles `for`.

Pour une matrice de dimensions 2 × 2, le schéma de la figure 5.1 vous indique comment parcourir une telle matrice. L'affichage attendu est :

```
ligne colonne
   1     1
   1     2
   2     1
   2     2
```

Attention à bien respecter l'alignement des chiffres qui doit être justifié à droite sur 4 caractères. Testez avec une matrice de dimensions 3 × 3, puis 5 × 5, et enfin 10 × 10.

Créez une seconde version de votre script, cette fois-ci avec deux boucles `while`.

5.4.12 Parcours de demi-matrice sans la diagonale (exercice ++)

En se basant sur le script précédent, on souhaite réaliser le parcours d'une demi-matrice carrée sans la diagonale. On peut noter que cela produit tous les couples possibles une seule fois (1 et 2 est équivalent à 2 et 1), en excluant par ailleurs chaque élément avec lui même (1 et 1, 2 et 2, etc). Pour mieux comprendre ce qui est demandé, la figure 5.2 indique les cases à parcourir en gris :

Créez un script qui affiche le numéro de ligne et de colonne, puis la taille de la matrice $N \times N$ et le nombre total de cases parcourues. Par exemple pour une matrice 4 × 4 (N=4) :

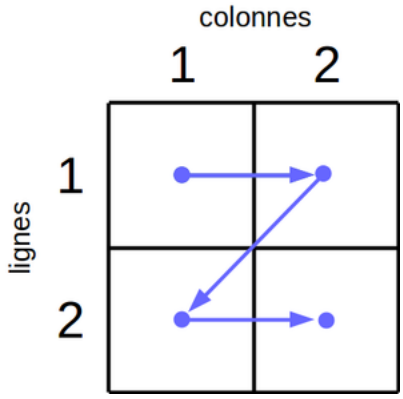


FIGURE 5.1 – Parcours d'une matrice.

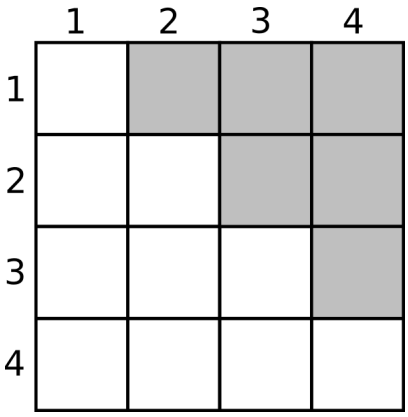


FIGURE 5.2 – Demi-matrice sans la diagonale (en gris).

```
ligne colonne
   1     2
   1     3
   1     4
   2     3
   2     4
   3     4
Pour une matrice 4x4, on a parcouru 6 cases
```

Testez votre script avec `N=3`, puis `N=4` et enfin `N=5`.

Concevez une seconde version à partir du script précédent, où cette fois on n'affiche plus tous les couples possibles, mais simplement la valeur de `N` et le nombre de cases parcourues. Affichez cela pour des valeurs de `N` allant de 2 à 10.

Pouvez-vous trouver une formule générale reliant le nombre de cases parcourues à `N` ?

5.4.13 Sauts de puce

On imagine une puce qui se déplace aléatoirement sur une ligne, en avant ou en arrière, par pas de 1 ou -1. Par exemple, si elle est à l'emplacement 0, elle peut sauter à l'emplacement 1 ou -1 ; si elle est à l'emplacement 2, elle peut sauter à l'emplacement 3 ou 1, etc.

Avec une boucle `while`, simulez le mouvement de cette puce de l'emplacement initial 0 à l'emplacement final 5 (voir le schéma de la figure 5.3). Combien de sauts sont nécessaires pour réaliser ce parcours ? Relancez plusieurs fois le programme. Trouvez-vous le même nombre de sauts à chaque exécution ?

Conseil

Utilisez l'instruction `random.choice([-1,1])` qui renvoie au hasard les valeurs -1 ou 1 avec la même probabilité.

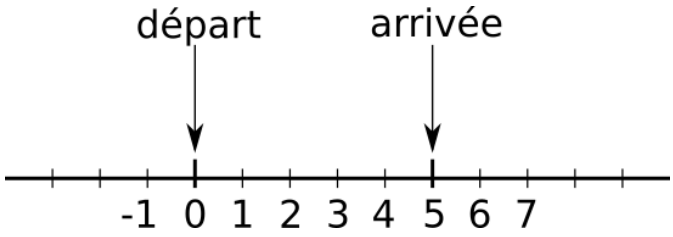


FIGURE 5.3 – Sauts de puce.

Avant d'utiliser cette instruction, mettez au tout début de votre script la ligne
`import random`
Nous verrons la signification de cette syntaxe particulière dans le chapitre 9 *Modules*.

5.4.14 Suite de Fibonacci (exercice +++)

La suite de Fibonacci [3] est une suite mathématique qui porte le nom de Leonardo Fibonacci, un mathématicien italien du XIII^{e} siècle. Initialement, cette suite a été conçue pour décrire la croissance d'une population de lapins, mais elle peut également être utilisée pour décrire certains motifs géométriques retrouvés dans la nature (coquillages, fleurs de tournesol…).

Pour la suite de Fibonacci (x_n), le terme au rang n (avec $n > 1$) est la somme des nombres aux rangs $n-1$ et $n-2$: $x_n = x_{n-1} + x_{n-2}$

Par définition, les deux premiers termes sont $x_0 = 0$ et $x_1 = 1$.

À titre d'exemple, les 10 premiers termes de la suite de Fibonacci sont donc 0, 1, 1, 2, 3, 5, 8, 13, 21 et 34.

Créez un script qui construit une liste `fibo` avec les 15 premiers termes de la suite de Fibonacci puis l'affiche.

Améliorez ce script en affichant, pour chaque élément de la liste `fibo` avec $n > 1$, le rapport entre l'élément de rang n et l'élément de rang $n-1$. Ce rapport tend-il vers une constante ? Si oui, laquelle ?

3. https://fr.wikipedia.org/wiki/Suite_de_Fibonacci

CHAPITRE 6

Tests

6.1 Définition

Les **tests** sont un élément essentiel à tout langage informatique si on veut lui donner un peu de complexité car ils permettent à l'ordinateur de prendre des décisions. Pour cela, Python utilise l'instruction `if` ainsi qu'une comparaison que nous avons abordée au chapitre précédent. Voici un premier exemple :

```
>>> x = 2
>>> if x == 2:
...     print("Le test est vrai !")
...
Le test est vrai !
```

et un second :

```
>>> x = "souris"
>>> if x == "tigre":
...     print("Le test est vrai !")
...
```

Il y a plusieurs remarques à faire concernant ces deux exemples :

- Dans le premier exemple, l'instruction `print("Le test est vrai !")` est exécutée, car le test est vrai. Dans le second exemple, le test est faux et rien n'est affiché.
- Les blocs d'instructions dans les tests doivent forcément être indentés comme pour les boucles `for` et `while`. L'indentation indique la portée des instructions à exécuter si le test est vrai.
- Comme avec les boucles `for` et `while`, la ligne qui contient l'instruction `if` se termine par le caractère deux-points « `:` ».

6.2 Tests à plusieurs cas

Parfois, il est pratique de tester si la condition est vraie ou si elle est fausse dans une même instruction `if`. Plutôt que d'utiliser deux instructions `if`, on peut se servir des instructions `if` et `else` :

```
>>> x = 2
>>> if x == 2:
...     print("Le test est vrai !")
... else:
...     print("Le test est faux !")
...
Le test est vrai !
>>> x = 3
>>> if x == 2:
...     print("Le test est vrai !")
... else:
...     print("Le test est faux !")
...
Le test est faux !
```

On peut utiliser une série de tests dans la même instruction `if`, notamment pour tester plusieurs valeurs d'une même variable.

Par exemple, on se propose de tirer au sort une base d'ADN puis d'afficher le nom de cette dernière. Dans le code suivant, nous utilisons l'instruction `random.choice(liste)` qui renvoie un élément choisi au hasard dans une liste. L'instruction `import random` sera vue plus tard dans le chapitre 9 *Modules*, admettez pour le moment qu'elle est nécessaire :

```
>>> import random
>>> base = random.choice(["a", "t", "c", "g"])
>>> if base == "a":
...     print("choix d'une adénine")
... elif base == "t":
...     print("choix d'une thymine")
... elif base == "c":
...     print("choix d'une cytosine")
... elif base == "g":
...     print("choix d'une guanine")
...
choix d'une cytosine
```

Dans cet exemple, Python teste la première condition puis, si et seulement si elle est fausse, teste la deuxième et ainsi de suite... Le code correspondant à la première condition vérifiée est exécuté puis Python sort du bloc d'instructions du `if`. Il est également possible d'ajouter une condition `else` supplémentaire qui est exécutée si aucune des conditions du `if` et des `elif` n'est vraie.

6.3 Importance de l'indentation

De nouveau, faites bien attention à l'indentation ! Vous devez être très rigoureux sur ce point. Pour vous en convaincre, exécutez ces deux exemples de code :

Code 1

```
nombres = [4, 5, 6]
for nb in nombres:
    if nb == 5:
        print("Le test est vrai")
        print(f"car la variable nb vaut {nb}")
```

Résultat :

```
Le test est vrai
car la variable nb vaut 5
```

Code 2

```
1  nombres = [4, 5, 6]
2  for nb in nombres:
3      if nb == 5:
4          print("Le test est vrai")
5      print(f"car la variable nb vaut {nb}")
```

Résultat :

```
car la variable nb vaut 4
Le test est vrai
car la variable nb vaut 5
car la variable nb vaut 6
```

Les deux codes pourtant très similaires produisent des résultats très différents. Si vous observez avec attention l'indentation des instructions sur la ligne 5, vous remarquerez que dans le code 1, l'instruction est indentée deux fois, ce qui signifie qu'elle appartient au bloc d'instructions du test `if`. Dans le code 2, l'instruction de la ligne 5 n'est indentée qu'une seule fois, ce qui fait qu'elle n'appartient plus au bloc d'instructions du test `if`, d'où l'affichage de `car la variable nb vaut xx` pour toutes les valeurs de `nb`.

6.4 Tests multiples

Les tests multiples permettent de tester plusieurs conditions en même temps en utilisant des opérateurs booléens. Les deux opérateurs les plus couramment utilisés sont **OU** et **ET**. Voici un petit rappel sur le fonctionnement de l'opérateur **OU** :

Condition 1	Opérateur	Condition 2	Résultat
Vrai	OU	Vrai	Vrai
Vrai	OU	Faux	Vrai
Faux	OU	Vrai	Vrai
Faux	OU	Faux	Faux

et de l'opérateur **ET** :

Condition 1	Opérateur	Condition 2	Résultat
Vrai	ET	Vrai	Vrai
Vrai	ET	Faux	Faux
Faux	ET	Vrai	Faux
Faux	ET	Faux	Faux

En Python, on utilise le mot réservé `and` pour l'opérateur **ET** et le mot réservé `or` pour l'opérateur **OU**. Respectez bien la casse des opérateurs `and` et `or` qui, en Python, s'écrivent en minuscule. En voici un exemple d'utilisation :

```
1  >>> x = 2
2  >>> y = 2
3  >>> if x == 2 and y == 2:
4  ...     print("le test est vrai")
5  ...
6  le test est vrai
```

Notez que le même résultat serait obtenu en utilisant deux instructions `if` imbriquées :

```
1  >>> x = 2
2  >>> y = 2
3  >>> if x == 2:
4  ...     if y == 2:
5  ...         print("le test est vrai")
6  ...
7  le test est vrai
```

Conseil

Nous vous conseillons la syntaxe avec un `and` qui est plus compacte. De manière générale, moins il y a de niveau d'indentations mieux c'est pour la lisibilité.

Vous pouvez aussi tester directement l'effet de ces opérateurs à l'aide de `True` et `False` (attention à respecter la casse) :

```
1 >>> True or False
2 True
```

Enfin, on peut utiliser l'opérateur logique de négation `not` qui inverse le résultat d'une condition :

```
1 >>> not True
2 False
3 >>> not False
4 True
5 >>> not (True and True)
6 False
```

6.5 Instructions `break` et `continue`

Ces deux instructions modifient le comportement d'une boucle (`for` ou `while`) avec un test. Ainsi, l'instruction `break` stoppe la boucle en cours :

```
1 >>> for nombre in range(4):
2 ...     if nombre > 1:
3 ...         break
4 ...     print(nombre)
5 ...
6 0
7 1
```

L'instruction `continue` saute à l'itération suivante, sans exécuter la suite du bloc d'instructions de la boucle :

```
1 >>> for nombre in range(4):
2 ...     if nombre == 2:
3 ...         continue
4 ...     print(nombre)
5 ...
6 0
7 1
8 3
```

6.6 Tests de valeur sur des *floats*

Lorsque l'on souhaite tester la valeur d'une variable de type *float*, le premier réflexe serait d'utiliser l'opérateur d'égalité comme :

```
1 >>> 1/10 == 0.1
2 True
```

Toutefois, nous vous le déconseillons formellement. Pourquoi ? Python stocke les valeurs numériques des *floats* sous forme de nombres flottants (d'où leur nom), et cela mène à certaines limitations[1]. Observez l'exemple suivant :

```
1 >>> (3 - 2.7) == 0.3
2 False
3 >>> 3 - 2.7
4 0.2999999999999998
```

Nous voyons que le résultat de l'opération `3 - 2.7` n'est pas exactement `0.3` d'où le résultat `False` en ligne 2.

En fait, ce problème ne vient pas de Python, mais plutôt de la manière dont un ordinateur traite les nombres flottants (comme un rapport de nombres binaires). Ainsi certaines valeurs de *float* ne peuvent être qu'approchées. Une manière

1. https://docs.python.org/fr/3/tutorial/floatingpoint.html

de s'en rendre compte est d'utiliser l'écriture formatée en demandant l'affichage d'un grand nombre de décimales :

```
>>> 0.3
0.3
>>> f"{0.3:.5f}"
'0.30000'
>>> f"{0.3:.60f}"
'0.299999999999999988897769753748434595763683319091796875000000'
>>> var = 3 - 2.7
>>> f"{var:.60f}"
'0.299999999999999822364316059974953532218933105468750000000000'
>>> abs(var - 0.3)
1.6653345369377348e-16
```

On observe que lorsqu'on tape `0.3`, Python affiche une valeur arrondie. En réalité, le nombre réel `0.3` ne peut être qu'approché lorsqu'on le code en nombre flottant. Il est essentiel d'avoir cela en tête lorsque l'on compare deux *floats*. Même si `0.3` et `3 - 2.7` ne donnent pas le même résultat, la différence est toutefois infinétisimale, de l'ordre de `1e-16` soit la 16^{me} décimale !

Pour ces raisons, il ne faut surtout pas utiliser l'opérateur `==` pour tester si un *float* est égal à une certaine valeur, car cet opérateur correspond à une égalité stricte. La bonne pratique est de vérifier si un *float* est compris dans un intervalle avec une certaine précision. Si on appelle cette précision *delta*, on peut procéder ainsi :

```
>>> delta = 1e-5
>>> var = 3.0 - 2.7
>>> 0.3 - delta < var < 0.3 + delta
True
>>> abs(var - 0.3) < delta
True
```

Ici on teste si `var` est compris dans l'intervalle $0.3 \pm delta$. En choisissant delta à `1e-5`, on teste jusqu'à la cinquième décimale. Les deux méthodes mènent à un résultat strictement équivalent :

- La ligne 3 est plus intuitive car elle ressemble à un encadrement mathématique.
- La ligne 5 utilise la fonction valeur absolue `abs()` et est plus compacte.

Une dernière manière pour tester la valeur d'un *float*, apparue en Python 3.5, est d'utiliser la fonction `math.isclose()` :

```
>>> import math
>>> var = 3.0 - 2.7
>>> math.isclose(var, 0.3, abs_tol=1e-5)
True
```

Cette fonction prend en argument les deux *floats* à comparer, ainsi que l'argument par mot-clé `abs_tol` correspondant à la précision souhaitée (que nous avions appelée `delta` ci-dessus). Nous vous conseillons de toujours préciser cet argument `abs_tol`. Comme vu au dessus pour tirer une base au hasard, l'instruction `import math` sera vue dans le chapitre 9 Modules, admettez pour le moment qu'elle est nécessaire.

Conseil

Sur les trois manières de procéder pour comparer un *float* à une valeur, nous vous conseillons celle avec `math.isclose()` qui nous parait la plus lisible.

6.7 Exercices

Conseil

Pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

6.7.1 Jours de la semaine

Constituez une liste `semaine` contenant le nom des sept jours de la semaine.
En utilisant une boucle, écrivez chaque jour de la semaine ainsi que les messages suivants :

- `Au travail` s'il s'agit du lundi au jeudi ;
- `Chouette c'est vendredi` s'il s'agit du vendredi ;
- `Repos ce week-end` s'il s'agit du samedi ou du dimanche.

Ces messages ne sont que des suggestions, vous pouvez laisser libre cours à votre imagination.

6.7.2 Séquence complémentaire d'un brin d'ADN

La liste ci-dessous représente la séquence d'un brin d'ADN :
`["A", "C", "G", "T", "T", "A", "G", "C", "T", "A", "A", "C", "G"]`
Créez un script qui transforme cette séquence en sa séquence complémentaire.
Rappel : la séquence complémentaire s'obtient en remplaçant A par T, T par A, C par G et G par C.

6.7.3 Minimum d'une liste

La fonction `min()` de Python renvoie l'élément le plus petit d'une liste constituée de valeurs numériques ou de chaînes de caractères. Sans utiliser cette fonction, créez un script qui détermine le plus petit élément de la liste `[8, 4, 6, 1, 5]`.

6.7.4 Fréquence des acides aminés

La liste ci-dessous représente une séquence d'acides aminés :
`["R", "A", "W", "W", "A", "W", "A", "R", "W", "W", "R", "A", "G"]`
Calculez la fréquence des acides aminés alanine (A), arginine (R), tryptophane (W) et glycine (G) dans cette séquence.

6.7.5 Notes et mention d'un étudiant

Voici les notes d'un étudiant : 14, 9, 13, 15 et 12. Créez un script qui affiche la note maximum (utilisez la fonction `max()`), la note minimum (utilisez la fonction `min()`) et qui calcule la moyenne.

Affichez la valeur de la moyenne avec deux décimales. Affichez aussi la mention obtenue sachant que la mention est « passable » si la moyenne est entre 10 inclus et 12 exclus, « assez bien » entre 12 inclus et 14 exclus et « bien » au-delà de 14.

6.7.6 Nombres pairs

Construisez une boucle qui parcourt les nombres de 0 à 20 et qui affiche les nombres pairs inférieurs ou égaux à 10 d'une part, et les nombres impairs strictement supérieurs à 10 d'autre part.

Pour cet exercice, vous pourrez utiliser l'opérateur modulo `%` qui renvoie le reste de la division entière entre deux nombres et dont voici quelques exemples d'utilisation :

```
>>> 4 % 3
1
>>> 5 % 3
2
>>> 4 % 2
0
>>> 6 % 2
0
```

Vous remarquerez qu'un nombre est pair lorsque le reste de sa division entière par 2 est nul.

6.7.7 Conjecture de Syracuse (exercice +++)

La conjecture de Syracuse[2] est une conjecture mathématique qui reste improuvée à ce jour et qui est définie de la manière suivante.

Soit un entier positif *n*. Si *n* est pair, alors le diviser par 2. S'il est impair, alors le multiplier par 3 et lui ajouter 1. En répétant cette procédure, la suite de nombres atteint la valeur 1 puis se prolonge indéfiniment par une suite de trois valeurs triviales appelée cycle trivial.

2. http://fr.wikipedia.org/wiki/Conjecture_de_Syracuse

Jusqu'à présent, la conjecture de Syracuse, selon laquelle depuis n'importe quel entier positif la suite de Syracuse atteint 1, n'a pas été mise en défaut.

Par exemple, les premiers éléments de la suite de Syracuse si on prend comme point de départ 10 sont : 10, 5, 16, 8, 4, 2, 1…

Créez un script qui, partant d'un entier positif *n* (par exemple 10 ou 20), crée une liste des nombres de la suite de Syracuse. Avec différents points de départ (c'est-à-dire avec différentes valeurs de *n*), la conjecture de Syracuse est-elle toujours vérifiée ? Quels sont les nombres qui constituent le cycle trivial ?

Conseil

1. Pour cet exercice, vous avez besoin de faire un nombre d'itérations inconnu pour que la suite de Syracuse atteigne le chiffre 1 puis entame son cycle trivial. Vous pourrez tester votre algorithme avec un nombre arbitraire d'itérations, typiquement 20 ou 100, suivant votre nombre *n* de départ.
2. Un nombre est pair lorsque le reste de sa division entière (opérateur modulo `%`) par 2 est nul.

6.7.8 Attribution de la structure secondaire des acides aminés d'une protéine (exercice +++)

Dans une protéine, les différents acides aminés sont liés entre eux par une liaison peptidique. Les angles phi et psi sont deux angles mesurés autour de cette liaison peptidique. Leurs valeurs sont utiles pour définir la conformation spatiale (appelée « structure secondaire ») adoptée par les acides aminés.

Par exemple, les angles phi et psi d'une conformation en « hélice alpha » parfaite ont une valeur de -57 degrés et -47 degrés respectivement. Bien sûr, il est très rare que l'on trouve ces valeurs parfaites dans une protéine, et il est habituel de tolérer une déviation de $\pm$ 30 degrés autour des valeurs idéales de ces angles.

Vous trouverez ci-dessous une liste de listes contenant les valeurs des angles phi et psi de 15 acides aminés de la protéine 1TFE[3] :

```
[[48.6, 53.4],[-124.9, 156.7],[-66.2, -30.8], \
[-58.8, -43.1],[-73.9, -40.6],[-53.7, -37.5], \
[-80.6, -26.0],[-68.5, 135.0],[-64.9, -23.5], \
[-66.9, -45.5],[-69.6, -41.0],[-62.7, -37.5], \
[-68.2, -38.3],[-61.2, -49.1],[-59.7, -41.1]]
```

Pour le premier acide aminé, l'angle phi vaut *48.6* et l'angle psi *53.4*. Pour le deuxième, l'angle phi vaut *-124.9* et l'angle psi *156.7*, etc.

En utilisant cette liste, créez un script qui teste, pour chaque acide aminé, s'il est ou non en hélice et affiche les valeurs des angles phi et psi et le message adapté *est en hélice* ou *n'est pas en hélice*.

Par exemple, pour les trois premiers acides aminés :

```
[48.6, 53.4] n'est pas en hélice
[-124.9, 156.7] n'est pas en hélice
[-66.2, -30.8] est en hélice
```

D'après vous, quelle est la structure secondaire majoritaire de ces 15 acides aminés ?

Remarque

Pour en savoir plus sur le monde merveilleux des protéines, n'hésitez pas à consulter la page Wikipedia sur la structure secondaire des protéines[4].

6.7.9 Détermination des nombres premiers inférieurs à 100 (exercice +++)

Voici un extrait de l'article sur les nombres premiers tiré de l'encyclopédie en ligne Wikipédia[5] :

3. https://www.rcsb.org/structure/1TFE
4. https://fr.wikipedia.org/wiki/Structure_des_prot%C3%A9ines#Angles_di%C3%A8dres_et_structure_secondaire
5. http://fr.wikipedia.org/wiki/Nombre_premier

Un nombre premier est un entier naturel qui admet exactement deux diviseurs distincts entiers et positifs (qui sont alors 1 et lui-même). Cette définition exclut 1, qui n'a qu'un seul diviseur entier positif. Par opposition, un nombre non nul produit de deux nombres entiers différents de 1 est dit composé. Par exemple $6 = 2 \times 3$ *est composé, tout comme* $21 = 3 \times 7$*, mais 11 est premier car 1 et 11 sont les seuls diviseurs de 11. Les nombres 0 et 1 ne sont ni premiers ni composés.*

Déterminez tous les nombres premiers inférieurs à 100. Combien y a-t-il de nombres premiers entre 0 et 100 ? Pour vous aider, nous vous proposons plusieurs méthodes.

Méthode 1 (peu optimale, mais assez intuitive)

Pour chaque nombre N de 2 à 100, calculez le reste de la division entière (avec l'opérateur modulo `%`) depuis 1 jusqu'à lui-même. Si N est premier, il aura exactement deux nombres pour lesquels le reste de la division entière est égal à 0 (1 et lui-même). Si N n'est pas premier, il aura plus de deux nombres pour lesquels le reste de la division entière est égal à 0.

Méthode 2 (quelques petites optimisations qui font gagner du temps)

On reprend la méthode 1 avec deux petites optimisations. On sait que tout entier N supérieur à 1 est divisible par 1 et par lui-même. Ainsi, il est inutile de tester ces deux diviseurs. On propose donc de tester tous les diviseurs de 2 à $N-1$. Si on ne trouve aucun diviseur, alors N est premier. À partir du moment où on trouve un diviseur, il est inutile de continuer à chercher d'autres diviseurs car N ne sera pas premier. On suggère ainsi de stopper la boucle (pensez à `break`). Vous pourrez aussi utiliser une variable *drapeau* comme `est_premier` qui sera à `True` si N est premier, sinon à `False`.

Méthode 3 (plus optimale et rapide, mais un peu plus compliquée)

Parcourez tous les nombres N de 2 à 100 et vérifiez si ceux-ci sont composés, c'est-à-dire s'ils sont le produit de deux nombres premiers. Pratiquement, cela consiste à vérifier que le reste de la division entière (opérateur modulo `%`) entre N et chaque nombre premier déterminé jusqu'à maintenant est nul. Le cas échéant, N n'est pas premier.

6.7.10 Recherche d'un nombre par dichotomie (exercice +++)

La recherche par dichotomie[6] est une méthode qui consiste à diviser (en général en parties égales) un problème pour en trouver la solution. À titre d'exemple, voici une discussion entre Pierre et Patrick dans laquelle Pierre essaie de deviner le nombre (compris entre 1 et 100 inclus) auquel Patrick a pensé :

- [Patrick] « C'est bon, j'ai pensé à un nombre entre 1 et 100. »
- [Pierre] « OK, je vais essayer de le deviner. Est-ce que ton nombre est plus petit ou plus grand que 50 ? »
- [Patrick] « Plus grand. »
- [Pierre] « Est-ce que ton nombre est plus petit, plus grand ou égal à 75 ? »
- [Patrick] « Plus grand. »
- [Pierre] « Est-ce que ton nombre est plus petit, plus grand ou égal à 87 ? »
- [Patrick] « Plus petit. »
- [Pierre] « Est-ce que ton nombre est plus petit, plus grand ou égal à 81 ? »
- [Patrick] « Plus petit. »
- [Pierre] « Est-ce que ton nombre est plus petit, plus grand ou égal à 78 ? »
- [Patrick] « Plus grand. »
- [Pierre] « Est-ce que ton nombre est plus petit, plus grand ou égal à 79 ? »
- [Patrick] « Égal. C'est le nombre auquel j'avais pensé. Bravo ! »

Pour arriver rapidement à deviner le nombre, l'astuce consiste à prendre à chaque fois la moitié de l'intervalle dans lequel se trouve le nombre. Voici le détail des différentes étapes :

1. Le nombre se trouve entre 1 et 100, on propose 50 (100 / 2).
2. Le nombre se trouve entre 50 et 100, on propose 75 (50 + (100-50)/2).
3. Le nombre se trouve entre 75 et 100, on propose 87 (75 + (100-75)/2).

6. https://fr.wikipedia.org/wiki/Dichotomie

4. Le nombre se trouve entre 75 et 87, on propose 81 (75 + (87-75)/2).
5. Le nombre se trouve entre 75 et 81, on propose 78 (75 + (81-75)/2).
6. Le nombre se trouve entre 78 et 81, on propose 79 (78 + (81-78)/2).

Créez un script qui reproduit ce jeu de devinettes. Vous pensez à un nombre entre 1 et 100 et l'ordinateur essaie de le deviner par dichotomie en vous posant des questions.

Votre programme utilisera la fonction `input()` pour interagir avec l'utilisateur. Voici un exemple de son fonctionnement :

```
>>> lettre = input("Entrez une lettre : ")
Entrez une lettre : P
>>> print(lettre)
P
```

Pour vous guider, voici ce que donnerait le programme avec la conversation précédente :

```
Pensez à un nombre entre 1 et 100.
Est-ce votre nombre est plus grand, plus petit ou égal à 50 ? [+/-/=] +
Est-ce votre nombre est plus grand, plus petit ou égal à 75 ? [+/-/=] +
Est-ce votre nombre est plus grand, plus petit ou égal à 87 ? [+/-/=] -
Est-ce votre nombre est plus grand, plus petit ou égal à 81 ? [+/-/=] -
Est-ce votre nombre est plus grand, plus petit ou égal à 78 ? [+/-/=] +
Est-ce votre nombre est plus grand, plus petit ou égal à 79 ? [+/-/=] =
J'ai trouvé en 6 questions !
```

Les caractères `[+/-/=]` indiquent à l'utilisateur comment il doit interagir avec l'ordinateur, c'est-à-dire entrer soit le caractère `+` si le nombre choisi est plus grand que le nombre proposé par l'ordinateur, soit le caractère `-` si le nombre choisi est plus petit que le nombre proposé par l'ordinateur, soit le caractère `=` si le nombre choisi est celui proposé par l'ordinateur (en appuyant ensuite sur la touche *Entrée*).

CHAPITRE 7

Fichiers

7.1 Lecture dans un fichier

Une grande partie de l'information en biologie est stockée sous forme de texte dans des fichiers. Pour traiter cette information, vous devez le plus souvent lire ou écrire dans un ou plusieurs fichiers. Python possède pour cela de nombreux outils qui vous simplifient la vie.

7.1.1 Méthode `.readlines()`

Avant de passer à un exemple concret, créez un fichier dans l'éditeur de texte de votre choix avec le contenu suivant :

```
girafe
tigre
singe
souris
```

Enregistrez ce fichier dans votre répertoire courant avec le nom `animaux.txt`. Puis, testez le code suivant dans l'interpréteur Python :

```
1  >>> filin = open("animaux.txt", "r")
2  >>> filin
3  <_io.TextIOWrapper name='animaux.txt' mode='r' encoding='UTF-8'>
4  >>> filin.readlines()
5  ['girafe\n', 'tigre\n', 'singe\n', 'souris\n']
6  >>> filin.close()
7  >>> filin.readlines()
8  Traceback (most recent call last):
9    File "<stdin>", line 1, in <module>
10 ValueError: I/O operation on closed file.
```

Il y a plusieurs commentaires à faire sur cet exemple :

- **Ligne 1**. La fonction `open()` ouvre le fichier `animaux.txt`. Ce fichier est ouvert en lecture seule, comme l'indique le second argument `r` (pour *read*) de `open()`. Remarquez que le fichier n'est pas encore lu, mais simplement ouvert (un peu comme lorsqu'on ouvre un livre, mais qu'on ne l'a pas encore lu). Le curseur de lecture est prêt à lire le premier caractère du fichier. L'instruction `open("animaux.txt", "r")` suppose que le fichier `animaux.txt` est dans le répertoire depuis lequel l'interpréteur Python a été lancé. Si ce n'est pas le cas, il faut préciser le **chemin d'accès** au fichier. Par exemple, `/home/pierre/animaux.txt` pour Linux ou Mac OS X ou `C:\Users\pierre\animaux.txt` pour Windows.

- **Ligne 2**. Lorsqu'on affiche le contenu de la variable `filin`, on se rend compte que Python la considère comme un objet de type fichier ouvert (ligne 3).
- **Ligne 4**. Nous utilisons à nouveau la syntaxe `objet.méthode()` (présentée dans le chapitre 3 *Affichage*). Ici la méthode `.readlines()` agit sur l'objet `filin` en déplaçant le curseur de lecture du début à la fin du fichier, puis elle renvoie une liste contenant toutes les lignes du fichier (dans notre analogie avec un livre, ceci correspondrait à lire toutes les lignes du livre).
- **Ligne 6**. Enfin, on applique la méthode `.close()` sur l'objet `filin`, ce qui, vous vous en doutez, ferme le fichier (ceci reviendrait à fermer le livre). Vous remarquerez que la méthode `.close()` ne renvoie rien, mais modifie l'état de l'objet `filin` en fichier fermé. Ainsi, si on essaie de lire à nouveau les lignes du fichier, Python renvoie une erreur, car il ne peut pas lire un fichier fermé (lignes 7 à 10).

Voici maintenant un exemple complet de lecture d'un fichier avec Python :

```
1  >>> filin = open("animaux.txt", "r")
2  >>> lignes = filin.readlines()
3  >>> lignes
4  ['girafe\n', 'tigre\n', 'singe\n', 'souris\n']
5  >>> for ligne in lignes:
6  ...     print(ligne)
7  ...
8  girafe
9
10 tigre
11
12 singe
13
14 souris
15
16 >>> filin.close()
```

Vous voyez qu'en cinq lignes de code, vous avez lu, parcouru le fichier et affiché son contenu.

Remarque

- Chaque élément de la liste `lignes` est une chaîne de caractères. C'est en effet sous forme de chaînes de caractères que Python lit le contenu d'un fichier.
- Chaque élément de la liste `lignes` se termine par le caractère `\n`. Ce caractère un peu particulier correspond au « saut de ligne[1] » qui permet de passer d'une ligne à la suivante (en anglais *line feed*). Ceci est codé par un caractère spécial que l'on représente par `\n`. Vous pourrez parfois rencontrer également la notation octale `\012`. Dans la suite de cet ouvrage, nous emploierons aussi l'expression « retour à la ligne » que nous trouvons plus intuitive.
- Par défaut, l'instruction `print()` affiche quelque chose puis revient à la ligne. Ce retour à la ligne dû à `print()` se cumule alors avec celui de la fin de ligne (`\n`) de chaque ligne du fichier et donne l'impression qu'une ligne est sautée à chaque fois.

Il existe en Python le mot-clé `with` qui permet d'ouvrir et de fermer un fichier de manière efficace. Si pour une raison ou une autre l'ouverture ou la lecture du fichier conduit à une erreur, l'utilisation de `with` garantit la bonne fermeture du fichier, ce qui n'est pas le cas dans le code précédent. Voici donc le même exemple avec `with` :

1. https://fr.wikipedia.org/wiki/Saut_de_ligne

```
>>> with open("animaux.txt", 'r') as filin:
...     lignes = filin.readlines()
...     for ligne in lignes:
...         print(ligne)
...
girafe

tigre

singe

souris

>>>
```

Remarque

- L'instruction `with` introduit un bloc d'instructions qui doit être indenté. C'est à l'intérieur de ce bloc que nous effectuons toutes les opérations sur le fichier.
- Une fois sorti du bloc d'instructions, Python fermera **automatiquement** le fichier. Vous n'avez donc plus besoin d'utiliser la méthode `.close()`.

7.1.2 Méthode `.read()`

Il existe d'autres méthodes que `.readlines()` pour lire (et manipuler) un fichier. Par exemple, la méthode `.read()` lit tout le contenu d'un fichier et renvoie une chaîne de caractères unique :

```
>>> with open("animaux.txt", "r") as filin:
...     filin.read()
...
'girafe\ntigre\nsinge\nsouris\n'
>>>
```

7.1.3 Méthode `.readline()`

La méthode `.readline()` (sans `s` à la fin) lit une ligne d'un fichier et la renvoie sous forme de chaîne de caractères. À chaque nouvel appel de `.readline()`, la ligne suivante est renvoyée. Associée à la boucle `while`, cette méthode permet de lire un fichier ligne par ligne :

```
>>> with open("animaux.txt", "r") as filin:
...     ligne = filin.readline()
...     while ligne != "":
...         print(ligne)
...         ligne = filin.readline()
...
girafe

tigre

singe

souris

>>>
```

7.1.4 Itérations directes sur le fichier

Python essaie de vous faciliter la vie au maximum. Voici un moyen à la fois simple et élégant de parcourir un fichier :

```
>>> with open("animaux.txt", "r") as filin:
...     for ligne in filin:
...         print(ligne)
...
girafe

tigre

singe

souris

>>>
```

L'objet `filin` est « itérable », ainsi la boucle `for` va demander à Python d'aller lire le fichier ligne par ligne.

Conseil

Privilégiez cette méthode par la suite.

Remarque

Les méthodes abordées précédemment permettent d'accéder au contenu d'un fichier, soit ligne par ligne (méthode `.readline()`), soit globalement en une seule chaîne de caractères (méthode `.read()`), soit globalement avec les lignes différenciées sous forme d'une liste de chaînes de caractères (méthode `.readlines()`). Il est également possible en Python de se rendre à un endroit particulier d'un fichier avec la méthode `.seek()` mais qui sort du cadre de cet ouvrage.

7.2 Écriture dans un fichier

Écrire dans un fichier est aussi simple que de le lire. Voyez l'exemple suivant :

```
1 >>> animaux2 = ["poisson", "abeille", "chat"]
2 >>> with open("animaux2.txt", "w") as filout:
3 ...     for animal in animaux2:
4 ...         filout.write(animal)
5 ...
6 7
7 7
8 4
```

Quelques commentaires sur cet exemple :

- **Ligne 1**. Création d'une liste de chaînes de caractères `animaux2`.
- **Ligne 2**. Ouverture du fichier `animaux2.txt` en mode écriture, avec le caractère `w` pour *write*. L'instruction `with` crée un bloc d'instructions qui doit être indenté.
- **Ligne 3**. Parcours de la liste `animaux2` avec une boucle `for`.
- **Ligne 4**. À chaque itération de la boucle, nous avons écrit chaque élément de la liste dans le fichier. La méthode `.write()` s'applique sur l'objet `filout`. Notez qu'à chaque utilisation de la méthode `.write()`, celle-ci nous affiche le nombre d'octets (équivalent au nombre de caractères) écrits dans le fichier (lignes 6 à 8). Ceci est valable uniquement dans l'interpréteur. Si vous créez un programme avec les mêmes lignes de code, ces valeurs ne s'afficheront pas à l'écran.

Si nous ouvrons le fichier `animaux2.txt` avec un éditeur de texte, voici ce que nous obtenons :

```
poissonabeillechat
```

Ce n'est pas exactement le résultat attendu car implicitement nous voulions le nom de chaque animal sur une ligne. Nous avons oublié d'ajouter le caractère fin de ligne après chaque nom d'animal.

Pour ce faire, nous pouvons utiliser l'écriture formatée :

```
>>> animaux2 = ["poisson", "abeille", "chat"]
>>> with open("animaux2.txt", "w") as filout:
...     for animal in animaux2:
...         filout.write(f"{animal}\n")
...
8
8
5
```

- **Ligne 4**. L'écriture formatée, vue au chapitre 3 *Affichage*, permet d'ajouter un retour à la ligne (`\n`) après le nom de chaque animal.
- **Lignes 6 à 8**. Le nombre d'octets écrits dans le fichier est augmenté de 1 par rapport à l'exemple précédent, car le caractère retour à la ligne compte pour un seul octet.

Le contenu du fichier `animaux2.txt` est alors :

```
poisson
abeille
chat
```

Vous voyez qu'il est relativement simple en Python de lire ou d'écrire dans un fichier.

7.3 Ouvrir deux fichiers avec l'instruction `with`

On peut avec l'instruction `with` ouvrir deux fichiers (ou plus) en même temps. Voyez l'exemple suivant :

```
with open("animaux.txt", "r") as fichier1, open("animaux2.txt", "w") as fichier2:
    for ligne in fichier1:
        fichier2.write("* " + ligne)
```

Si le fichier `animaux.txt` contient le texte suivant :

```
souris
girafe
lion
singe
```

alors le contenu de `animaux2.txt` sera :

```
* souris
* girafe
* lion
* singe
```

Dans cet exemple, `with` permet une notation très compacte en s'affranchissant de deux méthodes `.close()`.

Si vous souhaitez aller plus loin, sachez que l'instruction `with` est plus générale et peut être utilisée dans d'autres contextes[2].

7.4 Note sur les retours à la ligne sous Unix et sous Windows

Conseil

Si vous êtes débutant, vous pouvez sauter cette rubrique.

On a vu plus haut que le caractère spécial `\n` correspondait à un retour à la ligne. C'est le standard sous Unix (Mac OS X et Linux).

Toutefois, Windows utilise deux caractères spéciaux pour le retour à la ligne : `\r` correspondant à un retour chariot (hérité des machines à écrire) et `\n` comme sous Unix.

Si vous avez commencé à programmer en Python 2, vous aurez peut-être remarqué que, selon les versions, la lecture de fichier supprimait parfois les `\r` et d'autres fois les laissait. Heureusement, la fonction `open()` dans Python 3[3] gère

2. https://docs.python.org/fr/3/reference/compound_stmts.html#the-with-statement
3. https://docs.python.org/fr/3/library/functions.html#open

tout ça automatiquement et renvoie uniquement des sauts de ligne sous forme d'un seul `\n` (même si le fichier a été conçu sous Windows et qu'il contient initialement des `\r`).

7.5 Importance des conversions de types avec les fichiers

Vous avez sans doute remarqué que les méthodes qui lisent un fichier (par exemple `.readlines()`) vous renvoient systématiquement des chaînes de caractères. De même, pour écrire dans un fichier, il faut fournir une chaîne de caractères à la méthode `.write()`.

Pour tenir compte de ces contraintes, il faudra utiliser les fonctions de conversion de types vues au chapitre 2 *Variables* : `int()`, `float()` et `str()`. Ces fonctions de conversion sont essentielles lorsqu'on lit ou écrit des nombres dans un fichier.

En effet, les nombres dans un fichier sont considérés comme du texte, donc comme des chaînes de caractères, par la méthode `.readlines()`. Par conséquent, il faut les convertir (en entier ou en *float*) si on veut effectuer des opérations numériques avec.

7.6 Du respect des formats de données et de fichiers

Maintenant que vous savez lire et écrire des fichiers en Python, vous êtes capables de manipuler beaucoup d'information en biologie. Prenez garde cependant aux formats de fichiers, c'est-à-dire à la manière dont est stockée l'information biologique dans des fichiers. Nous vous renvoyons pour cela à l'annexe A *Quelques formats de données en biologie*.

7.7 Exercices

Conseil

Pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

7.7.1 Moyenne des notes

Le fichier `notes.txt`[4] contient les notes obtenues par des étudiants pour le cours de Python. Chaque ligne du fichier ne contient qu'une note.

Téléchargez le fichier `notes.txt` et enregistrez-le dans votre répertoire de travail. N'hésitez pas à l'ouvrir avec un éditeur de texte pour voir à quoi il ressemble.

Créez un script Python qui lit chaque ligne de ce fichier, extrait les notes sous forme de *float* et les stocke dans une liste.

Terminez le script en calculant et affichant la moyenne des notes avec deux décimales.

7.7.2 Admis ou recalé

Téléchargez le fichier `notes.txt` de l'exercice précédent et enregistrez-le dans votre répertoire de travail. N'hésitez pas l'ouvrir avec un éditeur de texte pour voir à quoi il ressemble.

Créez un script Python qui lit chaque ligne de ce fichier, extrait les notes sous forme de *float* et les stocke dans une liste.

Le script réécrira ensuite les notes dans le fichier `notes2.txt` avec une note par ligne suivie de « recalé » si la note est inférieure à 10 et « admis » si la note est supérieure ou égale à 10. Toutes les notes seront écrites avec une décimale. À titre d'exemple, voici les trois premières lignes attendues pour le fichier `notes2.txt` :

```
13.5 admis
17.0 admis
9.5 recalé
```

7.7.3 Spirale (exercice +++)

Créez un script `spirale.py` qui calcule les coordonnées cartésiennes d'une spirale à deux dimensions.

4. https://python.sdv.u-paris.fr/data-files/notes.txt

Les coordonnées cartésiennes x_A et y_A d'un point A sur un cercle de rayon r s'expriment en fonction de l'angle θ représenté sur la figure 7.1 comme :

$$x_A = \cos(\theta) \times r$$

$$y_A = \sin(\theta) \times r$$

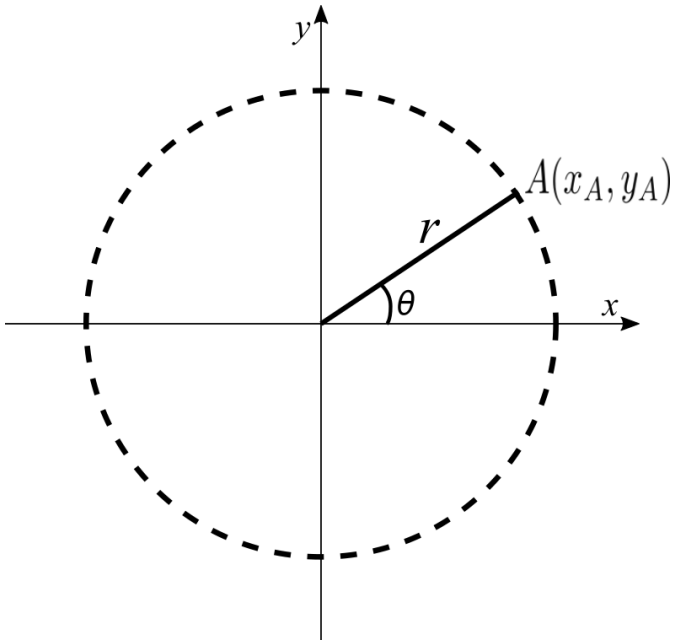


FIGURE 7.1 – Point A de coordonnées (x_A, y_A) sur le cercle de rayon r.

Pour calculer les coordonnées cartésiennes qui décrivent la spirale, vous allez faire varier deux variables en même temps :

- l'angle θ, qui va prendre des valeurs de 0 à 4π radians par pas de $0{,}1$, ce qui correspond à deux tours complets ;
- le rayon du cercle r, qui va prendre comme valeur initiale $0{,}5$ puis que vous allez incrémenter (c'est-à-dire augmenter) par pas de $0{,}1$.

Les fonctions trigonométriques sinus et cosinus sont disponibles dans le module *math* que vous découvrirez plus en détails dans le chapitre 9 *Modules*. Pour les utiliser, vous ajouterez au début de votre script l'instruction :

```
import math
```

La fonction sinus sera `math.sin()` et la fonction cosinus `math.cos()`. Ces deux fonctions prennent comme argument une valeur d'angle en radian. La constante mathématique π sera également accessible grâce à ce module via `math.pi`. Par exemple :

```
>>> math.sin(0)
0.0
>>> math.sin(math.pi/2)
1.0
>>> math.cos(math.pi)
-1.0
```

Sauvegardez ensuite les coordonnées cartésiennes dans le fichier `spirale.dat` en respectant le format suivant :

- un couple de coordonnées (x_A et y_A) par ligne ;
- au moins un espace entre les deux coordonnées x_A et y_A ;
- les coordonnées affichées sur 10 caractères avec 5 chiffres après la virgule.

Les premières lignes de `spirale.dat` devrait ressembler à :

```
  0.50000    0.00000
  0.59700    0.05990
  0.68605    0.13907
  0.76427    0.23642
  0.82895    0.35048
  0.87758    0.47943
  [...]      [...]
```

Une fois que vous avez généré le fichier `spirale.dat`, visualisez votre spirale avec le code suivant (que vous pouvez recopier dans un autre script ou à la suite de votre script `spirale.py`) :

```
1  import matplotlib.pyplot as plt
2
3  x = []
4  y = []
5  with open("spirale.dat", "r") as f_in:
6      for line in f_in:
7          coords = line.split()
8          x.append(float(coords[0]))
9          y.append(float(coords[1]))
10
11 fig, ax = plt.subplots(figsize=(8,8))
12 mini = min(x+y) - 2
13 maxi = max(x+y) + 2
14 ax.set_xlim(mini, maxi)
15 ax.set_ylim(mini, maxi)
16 ax.plot(x, y)
17 fig.savefig("spirale.png")
```

Visualisez l'image `spirale.png` ainsi créée.

Remarque

Le module *matplotlib* est utilisé ici pour la visualisation de la spirale. Son utilisation est détaillée dans le chapitre 21 *Module matplotlib*.

Essayez de jouer sur les paramètres θ et r, et leur pas d'incrémentation, pour construire de nouvelles spirales.

CHAPITRE 8

Dictionnaires et tuples

Dans ce chapitre, nous allons voir deux nouveaux types d'objet qui s'avèrent extrêmement utiles : les dictionnaires et les tuples. Comme les listes vues dans le chapitre 4, les dictionnaires et tuples contiennent une collection d'autres objets. Toutefois, nous verrons que ces trois types sont régis par des règles différentes pour accéder à leur contenu, ainsi que dans leur fonctionnement.

8.1 Dictionnaires

8.1.1 Définition et fonctionnement

Définition

Un **dictionnaire** contient une collection d'objets Python auxquels on accède à l'aide d'une **clé** de correspondance plutôt qu'un indice. Ainsi, il ne s'agit pas d'objets séquentiels comme les listes, mais plutôt d'objets dits de correspondance (*mapping objects* en anglais) ou tableaux associatifs.

Ceci étant défini, comment fonctionnent-ils exactement ? Regardons un exemple :

```
1 >>> animal1 = {}
2 >>> animal1["nom"] = "girafe"
3 >>> animal1["taille"] = 5.0
4 >>> animal1["poids"] = 1100
5 >>> animal1
6 {'nom': 'girafe', 'taille': 5.0, 'poids': 1100}
```

- **Ligne 1**. On définit un dictionnaire vide avec les accolades `{}` (tout comme on peut le faire pour les listes avec `[]`).
- **Lignes 2 à 4**. On remplit le dictionnaire avec plusieurs clés (`"nom"`, `"taille"`, `"poids"`) auxquelles on affecte des **valeurs** (`"girafe"`, `5.0`, `1100`).
- **Ligne 5**. On affiche le contenu du dictionnaire. Les accolades nous montrent qu'il s'agit bien d'un dictionnaire, et pour chaque élément séparé par une virgule on a une association du type `clé: valeur`. Ici, les clés sont des chaînes de caractères (ce qui sera souvent le cas), et les valeurs peuvent être n'importe quel objet Python.

Une fois le dictionnaire créé, on récupère la valeur associée à une clé donnée avec une syntaxe du type `dictionnaire["clé"]`. Par exemple :

```
>>> animal1["nom"]
'girafe'
>>> animal1["taille"]
5.0
```

On se souvient que pour accéder à l'élément d'une liste, il fallait utiliser un indice (par exemple, `liste[2]`). Ici, l'utilisation d'une clé (qui est souvent une chaîne de caractères) rend les choses plus explicites.

Vous pouvez mettre autant de couples clé / valeur que vous voulez dans un dictionnaire (tout comme vous pouvez ajouter autant d'éléments que vous le souhaitez dans une liste).

Remarque

Jusqu'à la version 3.6 de Python, un dictionnaire était affiché sans ordre particulier. L'ordre d'affichage des éléments n'était pas forcément le même que celui dans lequel il avait été rempli. De même, lorsqu'on itérait dessus, l'ordre n'était pas garanti. Depuis Python 3.7 (inclus), ce comportement a changé : un dictionnaire est toujours affiché dans le même ordre que celui utilisé pour le remplir. Et si on itère sur un dictionnaire, cet ordre est aussi respecté. Ce détail provient de l'implémentation interne des dictionnaires dans Python, mais cela nous concerne peu. Ce qui importe, c'est de se rappeler qu'on accède aux éléments par leur clé, et non par leur position telle que le dictionnaire est affiché. Cet ordre n'a pas d'importance, sauf dans de rares cas.

On peut aussi initialiser toutes les clés et les valeurs d'un dictionnaire en une seule opération :

```
>>> animal2 = {"nom": "singe", "poids": 70, "taille": 1.75}
```

Mais rien ne nous empêche d'ajouter une clé et une valeur supplémentaire :

```
>>> animal2["age"] = 15
>>> animal2
{'nom': 'singe', 'poids': 70, 'taille': 1.75, 'age': 15}
```

Après ce premier tour d'horizon, on perçoit l'avantage des dictionnaires : pouvoir retrouver des éléments par des noms (clés) plutôt que par des indices.

Les humains retiennent mieux les noms que les chiffres. Ainsi, les dictionnaires se révèlent très pratiques lorsque vous devez manipuler des structures complexes à décrire et que les listes présentent leurs limites. L'usage des dictionnaires rend en général le code plus lisible. Par exemple, si nous souhaitions stocker les coordonnées (x,y,z) d'un point dans l'espace, nous pourrions utiliser `coors = [0, 1, 2]` pour la version liste et `coors = {"x": 0, "y": 1, "z": 2}` pour la version dictionnaire. Quelqu'un qui lit le code comprendra tout de suite que `coors["z"]` contient la coordonnée z, ce sera moins intuitif avec `coors[2]`.

Conseil

Nous verrons dans le chapitre 14 *Conteneurs* que plusieurs types d'objets sont utilisables en tant que clé de dictionnaire. Malgré cela, nous conseillons de n'utiliser que des chaînes de caractères lorsque vous débutez.

8.1.2 Fonction `len()`

Comme pour les listes, l'instruction `len()` renvoie la longueur du dictionnaire, sauf qu'ici il s'agit du nombre de couples clé / valeur. Voici un exemple d'utilisation :

```
ani3 = {"nom": "pinson", "poids": 0.02, "taille": 0.15}
>>> len(ani3)
3
```

8.1.3 Itération sur les clés pour obtenir les valeurs

Si on souhaite voir toutes les associations clés / valeurs, on peut itérer sur un dictionnaire de la manière suivante :

```
>>> animal2 = {'nom': 'singe', 'poids': 70, 'taille': 1.75}
>>> for key in animal2:
...     print(key, animal2[key])
...
poids 70
nom singe
taille 1.75
```

Par défaut, l'itération sur un dictionnaire se fait sur les clés. Dans cet exemple, la variable d'itération `key` prend successivement la valeur de chaque clé, `animal2[key]` donne la valeur correspondant à chaque clé.

8.1.4 Méthodes `.keys()` et `.values()`

Les méthodes `.keys()` et `.values()` renvoient, comme vous vous en doutez, les clés et les valeurs d'un dictionnaire :

```
>>> animal2.keys()
dict_keys(['poids', 'nom', 'taille'])
>>> animal2.values()
dict_values([70, 'singe', 1.75])
```

Les mentions `dict_keys` et `dict_values` indiquent que nous avons à faire à des objets un peu particuliers. Ils ne sont pas indexables (on ne peut pas retrouver un élément par indice, par exemple `dico.keys()[0]` renverra une erreur). Si besoin, nous pouvons les transformer en liste avec la fonction `list()` :

```
>>> animal2.values()
dict_values(['singe', 70, 1.75])
>>> list(animal2.values())
['singe', 70, 1.75]
```

Toutefois, on peut itérer dessus dans une boucle (on dit qu'ils sont itérables) :

```
>>> for cle in animal2.keys():
...     print(cle)
...
nom
poids
taille
```

8.1.5 Méthode `.items()`

La méthode `.items()` renvoie un nouvel objet `dict_items` :

```
>>> dico = {0: "t", 1: "o", 2: "t", 3: "o"}
>>> dico.items()
dict_items([(0, 't'), (1, 'o'), (2, 't'), (3, 'o')])
```

On ne peut pas retrouver un élément par son indice dans un objet `dict_items`, toutefois on peut itérer dessus :

```
>>> dico.items()[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_items' object is not subscriptable
>>> for key, val in dico.items():
...    print(key, val)
...
0 t
1 o
2 t
3 o
```

Notez la syntaxe particulière qui ressemble à la fonction `enumerate()` vue au chapitre 5 *Boucles et comparaisons*. On itère à la fois sur `key` et sur `val`. Nous aurons l'explication de ce mécanisme dans la rubrique sur les tuples ci-après.

8.1.6 Existence d'une clé ou d'une valeur

Pour vérifier si une clé existe dans un dictionnaire, on peut utiliser le test d'appartenance avec l'opérateur `in` qui renvoie un booléen :

```
>>> animal2 = {'nom': 'singe', 'poids': 70, 'taille': 1.75}
>>> "poids" in animal2
True
>>> if "poids" in animal2:
...     print("La clé 'poids' existe pour animal2")
...
La clé 'poids' existe pour animal2
>>> "age" in animal2
False
>>> if "age" in animal2:
...     print("La clé 'age' existe pour animal2")
...
```

Dans le second test (lignes 10 à 12), le message n'est pas affiché car la clé `age` n'est pas présente dans le dictionnaire `animal2`.

Si on souhaite tester si une valeur existe dans un dictionnaire, on peut utiliser l'opérateur `in` avec l'objet renvoyé par la méthode `.values()` :

```
>>> animal2 = {'nom': 'singe', 'poids': 70, 'taille': 1.75}
>>> animal2.values()
dict_values(['singe', 70, 1.75])
>>> "singe" in animal2.values()
True
```

8.1.7 Méthode `.get()`

Par défaut, si on demande la valeur associée à une clé qui n'existe pas, Python renvoie une erreur :

```
>>> animal2 = {'nom': 'singe', 'poids': 70, 'taille': 1.75}
>>> animal2["age"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'age'
```

La méthode `.get()` s'affranchit de ce problème. Elle extrait la valeur associée à une clé mais ne renvoie pas d'erreur si la clé n'existe pas :

```
>>> animal2.get("nom")
'singe'
>>> animal2.get("age")
>>>
```

Ici, la valeur associée à la clé `nom` est `singe`, mais la clé `age` n'existe pas. On peut également indiquer à `.get()` une valeur par défaut si la clé n'existe pas :

```
>>> animal2.get("age", 42)
42
```

8.1.8 Liste de dictionnaires

En créant une liste de dictionnaires qui possèdent les mêmes clés, on obtient une structure qui ressemble à une base de données :

```
>>> animaux = [animal1, animal2]
>>> animaux
[{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}, {'nom': 'singe',
'poids': 70, 'taille': 1.75}]
>>>
>>> for ani in animaux:
...     print(ani["nom"])
...
girafe
singe
```

Vous constatez ainsi que les dictionnaires permettent de gérer des structures complexes de manière plus explicite que les listes.

8.2 Tuples

8.2.1 Définition

Définition

Les **tuples** (« n-uplets » en français) sont des **objets séquentiels** correspondant aux listes, mais ils sont toutefois **non modifiables**. On dit aussi qu'ils sont **immuables**. Vous verrez ci-dessous que nous les avons déjà croisés à plusieurs reprises !

Pratiquement, on utilise les parenthèses au lieu des crochets pour les créer :

```
1  >>> tuple1 = (1, 2, 3)
2  >>> tuple1
3  (1, 2, 3)
4  >>> type(tuple1)
5  <class 'tuple'>
6  >>> tuple1[2]
7  3
8  >>> tuple1[0:2]
9  (1, 2)
10 >>> tuple1[2] = 15
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13 TypeError: 'tuple' object does not support item assignment
```

L'affectation et l'indiçage fonctionnent comme avec les listes. Mais si on essaie de modifier un des éléments du tuple (en ligne 10), Python renvoie un message d'erreur car les tuples sont non modifiables. Si vous voulez ajouter un élément (ou le modifier), vous devez créer un nouveau tuple :

```
1  >>> tuple1 = (1, 2, 3)
2  >>> tuple1
3  (1, 2, 3)
4  >>> tuple1 = tuple1 + (2,)
5  >>> tuple1
6  (1, 2, 3, 2)
```

Conseil

Cet exemple montre que les tuples sont peu adaptés lorsqu'on a besoin d'ajouter, retirer, modifier des éléments. La création d'un nouveau tuple à chaque étape s'avère lourde et il n'y a aucune méthode pour faire cela puisque les tuples sont non modifiables. Pour ce genre de tâche, les listes sont clairement mieux adaptées.

Remarque

Pour créer un tuple d'un seul élément comme ci-dessus, utilisez une syntaxe avec une virgule `(element,)`, pour éviter une ambiguïté avec une simple expression. Par exemple, `(2)` équivaut à l'entier `2`, alors que l'expression `(2,)` est un tuple contenant l'élément `2`.

Autre particularité des tuples, il est possible de les créer sans les parenthèses, dès lors que ceci ne pose pas d'ambiguïté avec une autre expression :

```
1  >>> tuple1 = (1, 2, 3)
2  >>> tuple1
3  (1, 2, 3)
4  >>> tuple1 = 1, 2, 3
5  >>> tuple1
6  (1, 2, 3)
```

Toutefois, afin d'éviter les confusions, nous vous conseillons d'utiliser systématiquement les parenthèses lorsque vous débutez.

Les opérateurs `+` et `*` fonctionnent comme pour les listes (concaténation et duplication) :

```
>>> (1, 2) + (3, 4)
(1, 2, 3, 4)
>>> (1, 2) * 4
(1, 2, 1, 2, 1, 2, 1, 2)
```

Enfin, on peut utiliser la fonction `tuple(sequence)` qui fonctionne exactement comme la fonction `list()`, c'est-à-dire qu'elle prend en argument un objet et renvoie le tuple correspondant (opération de *casting*) :

```
>>> tuple([1,2,3])
(1, 2, 3)
>>> tuple("ATGCCGCGAT")
('A', 'T', 'G', 'C', 'C', 'G', 'C', 'G', 'A', 'T')
>>> tuple(range(5))
(0, 1, 2, 3, 4)
```

Remarque

Comme la fonction `list()`, la fonction `tuple()` prend en argument un objet contenant d'autres objets. Elle ne fonctionne pas avec les entiers, *floats* ou booléens. Par exemple, `tuple(2)` renvoie une erreur. On en verra plus sur ces questions dans le chapitre 14 *Conteneurs*.

8.2.2 Affectation multiple

Les tuples sont souvent utilisés pour l'**affectation multiple**, c'est-à-dire, affecter des valeurs à plusieurs variables en même temps :

```
>>> x, y, z = 1, 2, 3
>>> x
1
>>> y
2
>>> z
3
```

Attention, le nombre de variables et de valeurs doit être cohérents à gauche et à droite de l'opérateur `=` :

```
>>> x, y = 1, 2, 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

Il est aussi possible de faire des affectations multiples avec des listes, par exemple :
`[x, y, z] = [1, 2, 3]`.

Toutefois, cette syntaxe est alourdie par la présence des crochets. On préfèrera donc la syntaxe avec les tuples sans parenthèses.

Remarque

Nous avons appelé l'opération `x, y, z = 1, 2, 3` affectation multiple pour signifier que l'on affectait des valeurs à plusieurs variables en même temps.

Vous pourrez rencontrer aussi l'expression *tuple unpacking* que l'on pourrait traduire par « désempaquetage de tuple ». De même, il existe le *list unpacking*.

Ce terme *tuple unpacking* provient du fait que l'on peut décomposer un tuple initial de n éléments en autant de variables différentes en une seule instruction.

Par exemple, si on crée un tuple de trois éléments :

```
>>> tuple1 = (1, 2, 3)
>>> tuple1
(1, 2, 3)
```

On peut « désempaqueter » le tuple en une seule instruction :

```
>>> x, y, z = tuple1
>>> x
1
```

Cela serait possible également avec l'indiçage, mais il faudrait utiliser autant d'instruction que d'éléments :

```
>>> x = tuple1[0]
>>> y = tuple1[1]
>>> z = tuple1[2]
```

Dans les deux cas, `x` vaudra `1`, `y` vaudra `2` et `z` vaudra `3`.

Conseil

La syntaxe `x, y, z = tuple1` pour désempaqueter un tuple est plus élégante, plus lisible et plus compacte. Elle sera donc à privilégier.

L'affectation multiple est un mécanisme très puissant et important en Python. Nous verrons qu'il est particulièrement utile avec les fonctions dans les chapitres 10 *Fonctions* et 13 *Plus sur les fonctions*.

8.2.3 Itérations sur plusieurs valeurs à la fois

Nous avons déjà croisé les tuples avec la fonction `enumerate()` dans le chapitre 5 *Boucles et comparaisons*. Cette dernière permettait d'itérer en même temps sur les indices et les éléments d'une liste :

```
>>> for indice, element in enumerate([75, -75, 0]):
...     print(indice, element)
...
0 75
1 -75
2 0
>>> for bidule in enumerate([75, -75, 0]):
...     print(bidule, type(bidule))
...
(0, 75) <class 'tuple'>
(1, -75) <class 'tuple'>
(2, 0) <class 'tuple'>
```

Lignes 7 à 12. La fonction `enumerate()` itère sur une série de tuples. Pouvoir séparer `indice` et `element` dans la boucle est possible avec l'affectation multiple, par exemple : `indice, element = 0, 75` (voir rubrique précédente).

Dans le même ordre d'idée, nous avons déjà vu la méthode `.items()` qui permettait d'itérer sur des couples clé / valeur d'un dictionnaire :

```
>>> dico = {"pinson": 2, "merle": 3}
>>> for cle, valeur in dico.items():
...     print(cle, valeur)
...
pinson 2
merle 3
>>> for bidule in dico.items():
...     print(bidule, type(bidule))
...
('pinson', 2) <class 'tuple'>
('merle', 3) <class 'tuple'>
```

La méthode `.items()` itère, comme `enumerate()`, sur une série de tuples.

Enfin, on peut itérer sur trois valeurs en même temps à partir d'une liste de tuples de trois éléments :

```
>>> liste = [(5, 6, 7), (6, 7, 8), (7, 8, 9)]
>>> for x, y, z in liste:
...     print(x, y, z)
...
5 6 7
6 7 8
7 8 9
```

On pourrait concevoir la même chose sur quatre ou cinq éléments, voire plus. La seule contrainte est d'avoir une correspondance systématique entre le nombre de variables d'itération (par exemple trois variables dans l'exemple ci-dessus avec `x, y, z`) et la longueur de chaque sous-*tuple* de la liste sur laquelle on itère (dans l'exemple ci-dessus, chaque sous-*tuple* a trois éléments).

8.2.4 Fonction `divmod()`

Dans le chapitre 2 *Variables*, on a vu les opérateurs `//` et `%` qui renvoient respectivement le quotient et le reste d'une division entière. La fonction `divmod()` prend en argument deux valeurs, le numérateur et le dénominateur d'une division, et renvoie le quotient et le reste de la division entière correspondante :

```
>>> 3 / 4
0.75
>>> 3 // 4
0
>>> 3 % 4
3
>>> divmod(3, 4)
(0, 3)
```

En utilisant l'affectation multiple, on peut ainsi récupérer à la volée le quotient et le reste en une seule ligne :

```
>>> quotient, reste = divmod(3, 4)
>>> quotient
0
>>> reste
3
```

Cette fonction est très pratique, notamment quand on souhaite convertir des secondes en minutes et secondes résiduelles. Par exemple, si on veut convertir 754 secondes en minutes :

```
>>> 754 / 60
12.566666666666666
>>> divmod(754, 60)
(12, 34)
```

La division normale nous donne un *float* en minutes qui n'est pas très pratique, il faut encore convertir `0.566666666666666` minute en secondes et gérer les problèmes d'arrondi. La fonction `divmod()` renvoie le résultat directement : 12 min et 34 s. On pourrait raisonner de manière similaire pour convertir des minutes en heures, des heures en jours, etc.

8.2.5 Remarque finale

Les listes, les dictionnaires et les tuples sont tous des objets contenant une collection d'autres objets. En Python, on peut construire des listes qui contiennent des dictionnaires, des tuples ou d'autres listes, mais aussi des dictionnaires contenant des tuples, des listes, etc. Les combinaisons sont infinies !

8.3 Exercices

Conseil

Pour le premier exercice, utilisez l'interpréteur Python. Pour les suivants, créez des scripts puis exécutez-les dans un *shell*.

8.3.1 Prédire la sortie

Soit les deux lignes de code suivantes :

```
dico = {"nom": "Joe", "age": 24, "taille": 181}
var = "nom"
```

Prédisez le comportement de chaque instruction ci-dessous, sans les recopier dans un script ni dans l'interpréteur Python :

- `print(dico["age"])`
- `print(dico[var])`
- `print(dico[24])`
- `print(dico["var"])`
- `print(dico["taille"])`

Lorsqu'une instruction produit une erreur, identifiez pourquoi. Vérifiez ensuite vos prédictions en recopiant les instructions dans l'interpréteur Python.

8.3.2 Moyennes des notes

Soit le dictionnaire suivant donnant les notes d'un étudiant :

```
dico_notes = {
  "math": 14, "programmation": 12,
  "anglais": 16, "biologie": 10,
  "sport": 19
}
```

Calculez la moyenne de ses notes de deux manières différentes. Calculez à nouveau la moyenne sans la note de biologie.

8.3.3 Composition en acides aminés

En utilisant un dictionnaire, déterminez le nombre d'occurrences de chaque acide aminé dans la séquence `AGWPSGGASAGLAILWGASAIN` . Le dictionnaire ne doit contenir que les acides aminés présents dans la séquence.

Vous ne pouvez pas utiliser autant d'instructions `if` que d'acides aminés différents. Pensez au test d'appartenance.

8.3.4 Convertisseur de secondes

Un athlète court un marathon, malheureusement sa montre ne mesure son temps qu'en secondes. Celle-ci affiche 11 905. Aidez-le à convertir son temps en heures, minutes et secondes avec la fonction `divmod()`.

8.3.5 Convertisseur de jours

L'âge de Camille et Céline en jours est respectivement de 8 331 jours et 8 660 jours. Quel est leur âge en années, mois et jours, en supposant qu'une année compte 365 jours et qu'un mois compte 30 jours ? La fonction `divmod()` vous aidera à nouveau.

8.3.6 Propriétés des acides aminés

Les acides aminés peuvent être séparés en quatre grandes catégories : apolaires (a), polaires (p), chargés positivement ($+$) et chargés négativement (-). Le dictionnaire suivant implémente cette classification :

```
aa2prop = {'A': 'a', 'V': 'a', 'L': 'a', 'G': 'a', 'I': 'a', 'M': 'a',
           'W': 'a', 'F': 'a', 'P': 'a',
           'S': 'p', 'C': 'p', 'N': 'p', 'Q': 'p', 'T': 'p', 'Y': 'p',
           'D': '-', 'E': '-',
           'K': '+', 'R': '+', 'H': '+' }
```

On souhaite convertir la séquence en acide aminé du domaine transmembranaire d'une intégrine humaine `SNADVVYEKQMLYLYVLSGIG` en une série de signes indiquant la nature des acides aminés (a, p, $+$ et -). Affichez tout d'abord la séquence sur une ligne, puis la nature des acides aminés sur une seconde ligne.

La séquence contient une hélice transmembranaire, donc une succession de résidus apolaires, essayez de la retrouver visuellement.

Pour cet exercice, nous vous conseillons d'itérer sur la chaîne de caractères contenant la séquence. Nous reverrons cela dans le chapitre 11 *Plus sur les chaînes de caractères*.

8.3.7 Boucle sur plusieurs éléments simultanément

À partir de la liste de tuples suivante :

```
[("chien", 3), ("chat", 4), ("souris", 16)]
```

affichez chaque animal et son nombre en utilisant qu'une seule boucle `for`. Attention, pour cet exercice, il est interdit d'utiliser l'indiçage des listes.

CHAPITRE 9

Modules

9.1 Définition

Les modules sont des programmes Python qui contiennent des fonctions que l'on est amené à souvent réutiliser (on les appelle aussi bibliothèques, ou *libraries* en anglais). Ce sont des « boîtes à outils » qui vous seront très utiles.

Les développeurs de Python ont mis au point de nombreux modules qui effectuent différentes tâches. Pour cette raison, prenez toujours le réflexe de vérifier si une partie du code que vous souhaitez écrire n'existe pas déjà sous forme de module.

La plupart de ces modules sont déjà installés dans les versions standards de Python. Vous pouvez accéder à une documentation exhaustive[1] sur le site de Python. N'hésitez pas à explorer un peu ce site, la quantité de modules disponibles est impressionnante (plus de 300 modules).

9.2 Importation de modules

Dans les chapitres précédents, nous avons rencontré la notion de module plusieurs fois, notamment lorsque nous avons voulu tirer un nombre aléatoire :

```
1  >>> import random
2  >>> random.randint(0, 10)
3  4
```

Regardons de plus près cet exemple :

- **Ligne 1**. L'instruction `import` donne accès à toutes les fonctions du module *random*[2].
- **Ligne 2**. Nous utilisons la fonction `randint(0, 10)` du module *random*. Cette fonction renvoie un nombre entier tiré aléatoirement entre `0` inclus et `10` inclus.

Nous avons également croisé le module *math* lors de l'exercice sur la spirale (voir le chapitre 7 *Fichiers*). Ce module nous a donné accès aux fonctions trigonométriques sinus et cosinus, et à la constante π :

```
1  >>> import math
2  >>> math.cos(math.pi / 2)
3  6.123233995736766e-17
4  >>> math.sin(math.pi / 2)
5  1.0
```

1. https://docs.python.org/fr/3/py-modindex.html
2. https://docs.python.org/fr/3/library/random.html#module-random

En résumé, l'utilisation de la syntaxe `import module` permet d'importer tout une série de fonctions organisées par « thèmes ». Par exemple, les fonctions gérant les nombres aléatoires avec *random* et les fonctions mathématiques avec *math*. Python possède de nombreux autres modules internes (c'est-à-dire présent de base lorsqu'on installe Python).

Remarque

Dans le chapitre 3 *Affichage*, nous avons introduit la syntaxe `truc.bidule()` avec `truc` étant un objet et `.bidule()` une méthode. Nous vous avions expliqué qu'une *méthode* était une fonction un peu particulière :

- elle était liée à un objet par un point ;
- en général, elle agissait sur ou utilisait l'objet auquel elle était liée.

Par exemple, la méthode `.append()` vue dans le chapitre 4 *Listes*. Dans l'instruction `liste1.append(3)`, la méthode `.append()` ajoute l'entier `3` à l'objet `liste1` auquel elle est liée.

Avec les modules, nous rencontrons une syntaxe similaire. Par exemple, dans l'instruction `math.cos()`, on pourrait penser que `.cos()` est aussi une méthode. En fait la documentation officielle de Python[3] précise bien que dans ce cas `.cos()` est une fonction. Dans cet ouvrage, nous utiliserons ainsi le mot **fonction** lorsqu'on évoquera des fonctions issues de modules.

Si cela vous parait encore ardu, ne vous inquiétez pas : c'est à force de pratiquer et de lire que vous vous approprierez le vocabulaire. La syntaxe `module.fonction()` est là pour rappeler de quel module provient la fonction en un coup d'œil.

Il existe un autre moyen d'importer une ou plusieurs fonctions d'un module :

```
>>> from random import randint
>>> randint(0,10)
7
```

À l'aide du mot-clé `from`, on peut importer une fonction spécifique d'un module donné. Remarquez bien qu'il est inutile de répéter le nom du module dans ce cas : seul le nom de la fonction en question est requis.

On peut également importer toutes les fonctions d'un module :

```
>>> from random import *
>>> randint(0,50)
46
>>> uniform(0,2.5)
0.64943174760727951
```

L'instruction `from random import *` importe toutes les fonctions du module *random*. On peut utiliser toutes ses fonctions directement, comme par exemple `randint()` et `uniform()` qui renvoient des nombres aléatoires entiers et *floats*.

Dans la pratique, plutôt que de charger toutes les fonctions d'un module en une seule fois :

```
from random import *
```

Nous vous conseillons de charger le module seul de la manière suivante :

```
import random
```

puis d'appeler explicitement les fonctions voulues, par exemple :

```
>>> import random
>>> random.randint(1, 10)
4
>>> random.uniform(1, 3)
1.8645753676306085
```

Il est également possible de définir un alias (un nom plus court) pour un module :

3. https://docs.python.org/fr/3/tutorial/modules.html

```
>>> import random as rand
>>> rand.randint(1, 10)
6
>>> rand.uniform(1, 3)
2.643472616544236
```

Dans cet exemple, les fonctions du module *random* sont accessibles via l'alias `rand`.
Enfin, pour vider de la mémoire un module déjà chargé, on peut utiliser l'instruction `del` :

```
1  >>> import random
2  >>> random.randint(0,10)
3  2
4  >>> random.uniform(1, 3)
5  2.825594756352219
6  >>> del random
7  >>> random.randint(0,10)
8  Traceback (most recent call last):
9    File "<stdin>", line 1, in ?
10 NameError: name 'random' is not defined. Did you forget to import 'random'?
```

On constate alors qu'un rappel (ligne 7) d'une fonction du module *random*, après l'avoir vidé de la mémoire (ligne 6), retourne un message d'erreur (lignes 8-10). Dans le cas présent, le message d'erreur est explicite et demande à l'utilisateur s'il n'a pas oublié d'importer le module *random*.

9.3 Obtenir de l'aide sur les modules importés

Pour obtenir de l'aide sur un module, rien de plus simple : il suffit d'utiliser la commande `help()` :

```
>>> import random
>>> help(random)
[...]
```

Ce qui renvoie :

```
Help on module random:

NAME
    random - Random variable generators.

MODULE REFERENCE
    https://docs.python.org/3.7/library/random

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    integers
    --------
           uniform within range

    sequences
    ---------
           pick random element
           pick random sample
```

Remarque

- Pour vous déplacer dans l'aide, utilisez les flèches du haut et du bas pour le parcourir ligne par ligne, ou les touches *Page-up* et *Page-down* pour faire défiler l'aide page par page.
- Pour quitter l'aide, appuyez sur la touche *Q*.
- Pour chercher du texte, tapez le caractère / puis le texte que vous cherchez, puis la touche *Entrée*. Par exemple, pour chercher l'aide sur la fonction `randint()`, tapez `/randint` puis *Entrée*.

- Vous pouvez également obtenir de l'aide sur une fonction particulière d'un module avec : `help(random.randint)`

La commande `help()` est en fait une commande plus générale, permettant d'avoir de l'aide sur n'importe quel objet chargé en mémoire :

```
>>> t = [1, 2, 3]
>>> help(t)
Help on list object:

class list(object)
 |  list() -> new list
 |  list(sequence) -> new list initialized from sequence's items
 |
 |  Methods defined here:
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
 |
...
```

Enfin, pour connaître d'un seul coup d'œil toutes les méthodes ou variables associées à un objet, utilisez la fonction `dir()` :

```
>>> import random
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
'SystemRandom', 'TWOPI', 'WichmannHill', '_BuiltinMethodType', '_MethodT
ype', '__all__', '__builtins__', '__doc__', '__file__', '__name__', '_ac
os', '_ceil', '_cos', '_e', '_exp', '_hexlify', '_inst', '_log', '_pi',
'_random', '_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_wa
rn', 'betavariate', 'choice', 'expovariate', 'gammavariate', 'gauss', 'g
etrandbits', 'getstate', 'jumpahead', 'lognormvariate', 'normalvariate',
 'paretovariate', 'randint', 'random', 'randrange', 'sample', 'seed', 's
 etstate', 'shuffle', 'uniform', 'vonmisesvariate', 'weibullvariate']
>>>
```

9.4 Quelques modules courants

Il existe une série de modules que vous serez probablement amenés à utiliser si vous programmez en Python. En voici une liste non exhaustive (pour la liste complète, reportez-vous à la page des modules[4] sur le site de Python) :

- *math*[5] : fonctions et constantes mathématiques de base (sin, cos, exp, pi…).
- *sys*[6] : interaction avec l'interpréteur Python, notamment pour le passage d'arguments (voir plus bas).
- *pathlib*[7] : gestion des fichiers et des répertoires (voir plus bas).
- *random*[8] : génération de nombres aléatoires.
- *time*[9] : accès à l'heure de l'ordinateur et aux fonctions gérant le temps.
- *urllib*[10] : récupération de données sur internet depuis Python.
- *Tkinter*[11] : interface python avec Tk. Création d'objets graphiques (voir chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne)).
- *re*[12] : gestion des expressions régulières (voir chapitre 17 *Expressions régulières et parsing*).

Nous vous conseillons d'aller explorer les pages de ces modules pour découvrir toutes leurs potentialités.

4. https://docs.python.org/fr/3/py-modindex.html
5. https://docs.python.org/fr/3/library/math.html#module-math
6. https://docs.python.org/fr/3/library/sys.html#module-sys
7. https://docs.python.org/fr/3/library/os.html#module-os
8. https://docs.python.org/fr/3/library/random.html#module-random
9. https://docs.python.org/fr/3/library/time.html#module-time
10. https://docs.python.org/fr/3/library/urllib.html#module-urllib
11. https://docs.python.org/fr/3/library/tkinter.html#module-tkinter
12. https://docs.python.org/fr/3/library/re.html#module-re

Vous verrez dans le chapitre 15 *Création de modules* comment créer votre propre module lorsque vous souhaitez réutiliser souvent vos propres fonctions.

Enfin, notez qu'il existe de nombreux autres modules externes qui ne sont pas installés de base dans Python, mais qui sont très utilisés en bioinformatique et en analyse de données. Par exemple : *NumPy* (manipulations de vecteurs et de matrices), *Biopython* (manipulation de séquences ou de structures de biomolécules), *matplotlib* (représentations graphiques), *pandas* (analyse de données tabulées), etc. Ces modules vous serons présentés dans les chapitres 19 à 22.

9.5 Module *random* : génération de nombres aléatoires

Comme indiqué précédemment, le module *random*[13] contient des fonctions pour la génération de nombres aléatoires :

```
>>> import random
>>> random.randint(0, 10)
4
>>> random.randint(0, 10)
10
>>> random.uniform(0, 10)
6.574743184892878
>>> random.uniform(0, 10)
1.1655547702189106
```

Le module *random* permet aussi de permuter aléatoirement des listes :

```
>>> x = [1, 2, 3, 4]
>>> random.shuffle(x)
>>> x
[2, 3, 1, 4]
>>> random.shuffle(x)
>>> x
[4, 2, 1, 3]
```

Mais aussi de tirer aléatoirement un élément dans une liste donnée :

```
>>> bases = ["A", "T", "C", "G"]
>>> random.choice(bases)
'A'
>>> random.choice(bases)
'G'
```

La fonction `choices()` (avec un *s* à la fin) réalise plusieurs tirages aléatoires (avec remise, c'est-à-dire qu'on peut piocher plusieurs fois le même élément) dans une liste donnée. Le nombre de tirages est précisé par le paramètre `k` :

```
>>> random.choices(bases, k=5)
['G', 'A', 'A', 'T', 'G']
>>> random.choices(bases, k=5)
['A', 'T', 'A', 'A', 'C']
>>> random.choices(bases, k=10)
['C', 'T', 'T', 'T', 'G', 'A', 'C', 'A', 'G', 'G']
```

Si vous exécutez vous-même les exemples précédents, vous devriez obtenir des résultats légèrement différents de ceux indiqués.

Pour des besoins de reproductibilité des analyses en science, on a souvent besoin de retrouver les mêmes résultats même si on utilise des nombres aléatoires. Pour cela, on peut définir ce qu'on appelle la « graine aléatoire ».

Définition

En informatique, la génération de nombres aléatoires est un problème complexe. On utilise plutôt des « générateurs de nombres pseudo-aléatoires[14] ». Pour cela, une graine aléatoire[15] doit être définie. Cette graine est la plupart du temps un nombre entier qu'on passe au générateur : celui-ci va alors produire une **série donnée** de nombres pseudo-aléatoires qui dépendent de cette graine. Si on change la graine, la série de nombres change.

13. https://docs.python.org/fr/3/library/random.html#module-random
14. https://fr.wikipedia.org/wiki/G%C3%A9n%C3%A9rateur_de_nombres_pseudo-al%C3%A9atoires
15. https://fr.wikipedia.org/wiki/Graine_al%C3%A9atoire

En Python, la graine aléatoire se définit avec la fonction `seed()` :

```
>>> random.seed(42)
>>> random.randint(0, 10)
1
>>> random.randint(0, 10)
0
>>> random.randint(0, 10)
4
```

Ici, la graine aléatoire est fixée à 42. Si on ne précise pas la graine, par défaut Python utilise la date (plus précisément, il s'agit du nombre de secondes écoulées depuis une date fixe passée). Ainsi, à chaque fois qu'on relance Python, la graine sera différente, car ce nombre de secondes sera différent.

Si vous exécutez ces mêmes lignes de code (depuis l'instruction `random.seed(42)`), il se peut que vous ayez des résultats différents selon la version de Python. Néanmoins, vous devriez systématiquement obtenir les mêmes résultats si vous relancez plusieurs fois de suite ces instructions sur une même machine.

Remarque

Quand on utilise des nombres aléatoires, il est fondamental de connaitre la distribution de probablités utilisée par la fonction.

Par exemple, la fonction de base du module *random* est `random.random()`, elle renvoie un *float* aléatoire entre 0 et 1 tiré dans une **distribution uniforme**. Si on tire beaucoup de nombres, on aura la même probabilité d'obtenir tous les nombres possibles entre 0 et 1. La fonction `random.randint()` tire aussi un entier dans une distribution uniforme. La fonction `random.gauss()` tire quant à elle un *float* aléatoire dans une **distribution gaussienne**.

9.6 Module *sys* : passage d'arguments

Le module *sys*[16] contient des fonctions et des variables spécifiques à l'interpréteur Python lui-même.

Ce module est particulièrement intéressant pour récupérer les arguments passés à un script Python lorsque celui-ci est appelé en ligne de commande.

Dans cet exemple, créons le script suivant que l'on enregistrera sous le nom `test.py` :

```
import sys
print(sys.argv)
```

Ensuite, dans un *shell*, exécutons le script `test.py` suivi de plusieurs arguments :

```
$ python test.py salut girafe 42
['test.py', 'salut', 'girafe', '42']
```

- **Ligne 1**. Le caractère `$` représente l'invite du *shell*, `test.py` est le nom du script Python, `salut`, `girafe` et `42` sont les arguments passés au script (tous séparés par un espace).
- **Ligne 2**. Le script affiche le contenu de la variable `sys.argv`. Cette variable est une liste qui contient tous les arguments de la ligne de commande, y compris le nom du script Python lui-même qu'on retrouve comme premier élément de cette liste dans `sys.argv[0]`. On peut donc accéder à chacun des différents arguments du script avec `sys.argv[1]`, `sys.argv[2]`, etc.

Toujours dans le module *sys*, la fonction `sys.exit()` est utile pour quitter un script Python. On peut donner un argument à cette fonction (en général une chaîne de caractères) qui sera renvoyé au moment où Python quittera le script. Par exemple, si vous attendez au moins un argument en ligne de commande, vous pouvez renvoyer un message pour indiquer à l'utilisateur ce que le script attend comme argument :

```
import sys

if len(sys.argv) != 2:
    sys.exit("ERREUR : il faut exactement un argument.")

print(f"Argument vaut : {sys.argv[1]}")
```

16. https://docs.python.org/fr/3/library/sys.html#module-sys

Puis on l'exécute sans argument :

```
$ python test.py
ERREUR : il faut exactement un argument.
```

avec un seul argument :

```
$ python test.py 42
Argument vaut : 42
```

puis avec plusieurs :

```
$ python test.py 42 salut
ERREUR : il faut exactement un argument.
```

Remarque

On vérifie dans cet exemple que le script possède deux arguments, car le nom du script lui-même compte pour un argument (le tout premier).

L'intérêt de récupérer des arguments passés dans la ligne de commande à l'appel du script est de pouvoir ensuite les utiliser dans le script.

Voici comme nouvel exemple le script `compte_lignes.py`, qui prend comme argument le nom d'un fichier puis affiche le nombre de lignes qu'il contient :

```
1  import sys
2
3  if len(sys.argv) != 2:
4      sys.exit("ERREUR : il faut exactement un argument.")
5
6  nom_fichier = sys.argv[1]
7  taille = 0
8  with open(nom_fichier, "r") as f_in:
9      taille = len(f_in.readlines())
10
11 print(f"{nom_fichier} contient {taille} lignes.")
```

Supposons que dans le même répertoire, nous ayons le fichier `animaux1.txt` dont voici le contenu :

```
girafe
tigre
singe
souris
```

et le fichier `animaux2.txt` qui contient :

```
poisson
abeille
chat
```

Utilisons maintenant le script `compte_lignes.py` :

```
$ python compte_lignes.py
ERREUR : il faut exactement un argument.
$ python compte_lignes.py animaux1.txt
animaux1.txt contient 4 lignes.
$ python compte_lignes.py animaux2.txt
animaux2.txt contient 3 lignes.
$ python compte_lignes.py animaux1.txt animaux2.txt
ERREUR : il faut exactement un argument.
```

Notre script est donc capable de :

- vérifier si un argument lui est donné et si ce n'est pas le cas d'afficher un message d'erreur ;
- d'ouvrir le fichier dont le nom est fourni en argument, de compter puis d'afficher le nombre de lignes.

Par contre, le script ne vérifie pas si le fichier fourni en argument existe bien :

```
$ python compte_lignes.py animaux3.txt
Traceback (most recent call last):
  File "compte_lignes.py", line 8, in <module>
    with open(nom_fichier, "r") as f_in:
         ^^^^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'animaux3.txt'
```

La lecture de la partie suivante va nous permettre d'améliorer notre script `compte_lignes.py`.

9.7 Module *pathlib* : gestion des fichiers et des répertoires

Le module *pathlib*[17] permet de manipuler les fichiers et les répertoires.

Le plus souvent, on utilise uniquement la classe `Path` du module `pathlib`, qu'on charge de cette manière :

```
>>> from pathlib import Path
```

La méthode `.exists()` vérifie la présence d'un fichier sur le disque dur :

```
>>> import sys
>>> from pathlib import Path
>>> if Path("toto.pdb").exists():
...     print("le fichier est présent")
... else:
...     sys.exit("le fichier est absent")
...
le fichier est absent
```

Dans cet exemple, si le fichier n'existe pas sur le disque dur, on quitte le programme avec la fonction `exit()` du module *sys* que nous venons de voir.

La méthode `.cwd()` renvoie le chemin complet du répertoire depuis lequel est lancé Python (*cwd* signifiant *current working directory*) :

```
>>> from pathlib import Path
>>> Path().cwd()
PosixPath('/home/pierre')
```

On obtient un objet de type `PosixPath` qu'il est possible de transformer si besoin en chaîne de caractères avec la fonction `str()`, que nous avons vu dans le chapitre 2 *Variables* :

```
>>> str(Path().cwd())
'/home/pierre'
```

Mais l'intérêt de récupérer un objet de type `PosixPath` est qu'on peut ensuite utiliser les méthodes `.name` et `.parent` pour obtenir respectivement le nom du répertoire (sans son chemin complet) et le répertoire parent :

```
>>> Path().cwd()
PosixPath('/home/pierre')
>>> Path().cwd().name
'pierre'
>>> Path().cwd().parent
PosixPath('/home')
```

Enfin, la méthode `.iterdir()` liste le contenu du répertoire depuis lequel est lancé Python :

```
>>> list(Path().iterdir())
[PosixPath('demo.py'), PosixPath('tests'), PosixPath('1BTA.pdb')]
```

Tout comme la fonction `range()` (voir le chapitre 4 *Listes*), la méthode `.iterdir()` est un itérateur. La fonction `list()` permet d'obtenir une liste.

Toutefois, il est possible d'itérer très facilement sur le contenu d'un répertoire et de savoir s'il contient des fichiers ou des sous-répertoires :

17. https://docs.python.org/fr/3/library/pathlib.html

```
>>> for nom in Path().iterdir():
...     if nom.is_file():
...         print(f"{nom} est un fichier")
...     else:
...         print(f"{nom} n'est pas un fichier")
...
demo.py est un fichier
tests n'est pas un fichier
1BTA.pdb est un fichier
```

La méthode `.is_file()` renvoie `True` si l'objet est un fichier, et `False` si ce n'est pas le cas.

La méthode `.iterdir()` parcourt le contenu d'un répertoire, sans en explorer les éventuels sous-répertoires. Si on souhaite parcourir récursivement un répertoire, on utilise la méthode `.glob()`. Prenons l'arborescence suivante comme exemple :

```
1BTA.pdb
demo.py
tests
├── results.csv
├── script1.py
└── script2.py
```

Le répertoire courant contient les fichiers `1BTA.pdb` et `demo.py`, ainsi que le répertoire `tests`. Ce dernier contient lui-même les fichiers `results.csv`, `script1.py` et `script2.py`.

On souhaite maintenant lister tous les scripts Python (dont l'extension est `.py`) présents dans le répertoire courant et dans ses sous-répertoires :

```
>>> for nom in Path().glob("**/*.py"):
...     print(f"{nom}")
...
demo.py
tests/script1.py
tests/script2.py
```

Dans la chaîne de caractères `"**/*.py"`, `**` recherche tous les sous-répertoires récursivement et `*.py` signifie n'importe quel nom de fichier qui se termine par l'extension `.py`.

Il existe de nombreuse autres méthodes associées à la classe `Path` du module *pathlib*, n'hésitez pas à consulter la documentation [18].

9.8 Exercices

Conseil

Pour les trois premiers exercices, utilisez l'interpréteur Python. Pour les exercices suivants, créez des scripts puis exécutez-les dans un *shell*.

9.8.1 Racine carrée

Affichez sur la même ligne les nombres de 10 à 20 (inclus) ainsi que leur racine carrée avec trois décimales. Utilisez pour cela le module *math* avec la fonction `sqrt()`. Exemple :

```
10 3.162
11 3.317
12 3.464
13 3.606
[...]
```

Consultez pour cela la documentation [19] de la fonction `math.sqrt()`.

18. https://docs.python.org/3/library/pathlib.html
19. https://docs.python.org/fr/3/library/math.html#math.sqrt

9.8.2 Cosinus

Calculez le cosinus de $\pi/2$ en utilisant le module *math* avec la fonction `cos()` et la constante `pi`.
Consultez pour cela la documentation [20] de la fonction `math.cos()` et la documentation [21] de la constante `math.pi`.

9.8.3 Comparaison de *floats*

Montrez que

$$\sqrt{5+\sqrt{4+\sqrt{3}}}$$

est égale à e à 0,001 près.
Montrez ensuite que

$$\sqrt{7+\sqrt{6+\sqrt{5}}}$$

est égale à π à 0,0001 près.

Conseil

- Jetez un oeil à la rubrique sur la comparaison de *floats* abordée dans le chapitre 6 *Tests*.
- Les constantes π et e sont obtenues par `math.pi` et `math.e`.

9.8.4 Chemin et contenu du répertoire courant

Affichez le chemin et le contenu du répertoire courant (celui depuis lequel vous avez lancé l'interpréteur Python).
Déterminez également le nombre total de fichiers et de répertoires présents dans le répertoire courant.

9.8.5 Affichage temporisé

Affichez les nombres de 1 à 10 avec 1 seconde d'intervalle. Utilisez pour cela le module *time* et sa fonction `sleep()`.
Consultez pour cela la documentation [22] de la fonction `time.sleep()`.

9.8.6 Séquences aléatoires de chiffres

Générez une séquence aléatoire de six chiffres, ceux-ci étant des entiers tirés entre 1 et 4. Utilisez le module *random* avec la fonction `randint()`.
Consultez pour cela la documentation [23] de la fonction `random.randint()`.

9.8.7 Compteur de points de jeu de belote

On considère un jeu de belote avec la variante sans-atout, où chaque carte vaut un certain nombre de points quelle que soit sa couleur (trèfle, carreau, coeur, pique). Un dictionnaire permet de mettre la correspondance entre chaque carte et son nombre de points :

```
1  # Nombre de points de chaque carte.
2  # (V = valet, D = dame, R = roi, # d = 10, A = as).
3  dico_points_sans_atouts = {"7": 0, "8": 0, "9": 0, "V": 2, "D": 3,
4                             "R": 4, "d": 10, "A": 11}
```

Par ailleurs, on peut représenter un jeu de 32 cartes par une liste :
`jeu_cartes = ["7", "8", "9", "d", "V", "D", "R", "A"] * 4`
Créez un programme `belote.py` qui tire huit cartes au hasard sans remise et qui affiche le nombre de points correspondant. Pour cela, vous pouvez utiliser la fonction `random.sample()` et son argument par mot-clé `k`. N'hésitez pas à consulter la documentation [24]. On souhaite une sortie de ce style :

20. https://docs.python.org/fr/3/library/math.html#math.cos
21. https://docs.python.org/fr/3/library/math.html#math.pi
22. https://docs.python.org/fr/3/library/time.html#time.sleep
23. https://docs.python.org/fr/3/library/random.html#random.randint
24. https://docs.python.org/fr/3/library/random.html#random.sample

```
$ python belote.py
La main est ['7', 'A', '9', 'A', 'R', 'd', 'V', 'D'].
7 -->  0 points
A --> 11 points
9 -->  0 points
A --> 11 points
R -->  4 points
d --> 10 points
V -->  2 points
D -->  3 points
Le nombre total de points de la main est 41.
```

9.8.8 Séquences aléatoires d'ADN

Générez une séquence aléatoire d'ADN de 20 bases de deux manières différentes. Utilisez le module *random* avec la fonction `choice()` ou `choices()`.

9.8.9 Séquences aléatoires d'ADN avec argument

Créez un script `dna_random.py` qui prend comme argument un nombre de bases, construit une séquence aléatoire d'ADN dont la longueur est le nombre de bases fourni en argument, puis affiche cette séquence.

Le script devra vérifier qu'un argument est bien fourni et renvoyer un message d'erreur si ce n'est pas le cas.

Conseil

Pour générer la séquence d'ADN, utilisez la fonction `random.choice()` abordée dans l'exercice précédent.

9.8.10 Compteur de lignes

Améliorez le script `compte_lignes.py`, dont le code a été donné précédemment, de façon à ce qu'il renvoie un message d'erreur si le fichier n'existe pas.

Par exemple, si les fichiers `animaux1.txt` et `animaux2.txt` sont bien dans le répertoire courant, mais pas `animaux3.txt` :

```
$ python compte_lignes.py animaux1.txt
animaux1.txt contient 4 lignes.
$ python compte_lignes.py animaux2.txt
animaux2.txt contient 3 lignes.
$ python compte_lignes.py animaux3.txt
ERREUR : animaux3.txt n'existe pas.
```

9.8.11 Détermination du nombre pi par la méthode Monte Carlo (exercice +++)

Soit un cercle de rayon 1 (en trait plein sur la figure 9.1) inscrit dans un carré de côté 2 (en trait pointillé).

Avec $R = 1$, l'aire du carré vaut $(2R)^2$ soit 4 et l'aire du disque délimité par le cercle vaut πR^2 soit π.

En choisissant N points aléatoires (à l'aide d'une distribution uniforme) à l'intérieur du carré, la probabilité que ces points se trouvent aussi dans le cercle est :

$$p = \frac{\text{aire du cercle}}{\text{aire du carré}} = \frac{\pi}{4}$$

Soit n, le nombre de points effectivement dans le cercle, il vient alors

$$p = \frac{n}{N} = \frac{\pi}{4},$$

d'où

$$\pi = 4 \times \frac{n}{N}.$$

Déterminez une approximation de π par cette méthode. Pour cela, pour N itérations :

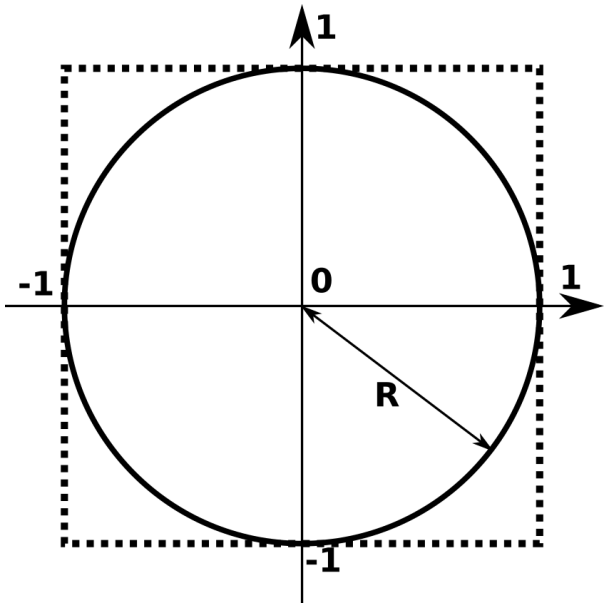


FIGURE 9.1 – Cercle de rayon 1 inscrit dans un carré de côté 2.

1. Choisissez aléatoirement les coordonnées *x* et *y* d'un point entre -1 et 1. Utilisez la fonction `uniform()` du module *random*.
2. Calculez la distance entre le centre du cercle et ce point.
3. Déterminez si cette distance est inférieure au rayon du cercle, c'est-à-dire si le point est dans le cercle ou pas.
4. Si le point est effectivement dans le cercle, incrémentez le compteur *n*.

Finalement calculez le rapport entre *n* et *N* et proposez une estimation de π. Quelle valeur de π obtenez-vous pour 100 itérations ? 1000 itérations ? 10 000 itérations ? Comparez les valeurs obtenues à la valeur de π fournie par le module *math*.

On rappelle que la distance *d* entre deux points A et B de coordonnées respectives (x_A, y_A) et (x_B, y_B) se calcule comme :

$$d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

Pour vous aider, consultez la documentation [25] de la fonction `random.uniform()`.

25. https://docs.python.org/fr/3/library/random.html#random.uniform

CHAPITRE 10

Fonctions

10.1 Principe et généralités

En programmation, les **fonctions** sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme. Elles rendent également le code plus lisible et plus clair en le fractionnant en blocs logiques.

Vous connaissez déjà certaines fonctions Python. Par exemple `math.cos(angle)` du module `math` renvoie le cosinus de la variable `angle` exprimé en radian. Vous connaissez aussi des fonctions internes à Python comme `range()` ou `len()`. Pour l'instant, une fonction est à vos yeux une sorte de « boîte noire » :

1. À laquelle vous passez aucune, une ou plusieurs variable(s) entre parenthèses. Ces variables sont appelées **arguments**. Il peut s'agir de n'importe quel type d'objet Python.
2. Qui effectue une action.
3. Qui renvoie un objet Python ou rien du tout.

Tout cela est illustré schématiquement dans la figure ci-dessous.

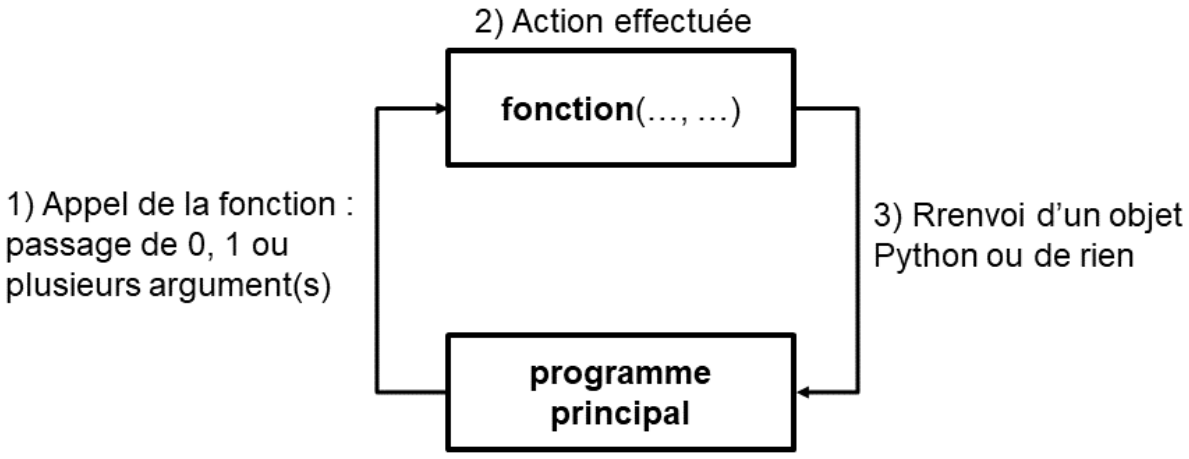


FIGURE 10.1 – Fonctionnement schématique d'une fonction.

Par exemple, si vous appelez la fonction `len()` de la manière suivante :

```
>>> len([0, 1, 2])
3
```

voici ce qu'il se passe :

1. vous appelez `len()` en lui passant une liste en argument (ici la liste `[0, 1, 2]`) ;
2. la fonction calcule la longueur de cette liste ;
3. elle vous renvoie un entier égal à cette longueur.

Autre exemple, si vous appelez la méthode `ma_liste.append()` (n'oubliez pas, une **méthode** est une **fonction** qui agit sur l'objet auquel elle est attachée par un point) :

```
>>> ma_liste.append(5)
```

1. Vous passez l'entier `5` en argument ;
2. la méthode `append()` ajoute l'entier `5` à l'objet `ma_liste` ;
3. et elle ne renvoie rien.

Aux yeux du programmeur, au contraire, une fonction est une portion de code effectuant une suite d'instructions bien particulière. Mais avant de vous présenter la syntaxe et la manière de construire une fonction, revenons une dernière fois sur cette notion de « boîte noire » :

- Une fonction effectue une tâche. Pour cela, elle reçoit éventuellement des arguments et renvoie éventuellement quelque chose. L'algorithme utilisé au sein de la fonction n'intéresse pas directement l'utilisateur. Par exemple, il est inutile de savoir comment la fonction `math.cos()` calcule un cosinus. On a juste besoin de savoir qu'il faut lui passer en argument un angle en radian, et qu'elle renvoie le cosinus de cet angle. Ce qui se passe à l'intérieur de la fonction ne regarde que le programmeur.
- Chaque fonction effectue en général une tâche **unique et précise**. Si cela se complique, il est plus judicieux d'écrire plusieurs fonctions (qui peuvent éventuellement s'appeler les unes les autres). Cette **modularité** améliore la qualité générale et la lisibilité du code. Vous verrez qu'en Python, les fonctions présentent une grande flexibilité.

Pour finir sur les généralités, nous avons utilisé dans la Figure ci-dessus le terme **programme principal** (*main* en anglais), pour désigner l'endroit depuis lequel on appelle une fonction (on verra plus tard que l'on peut en fait appeler une fonction de n'importe où). Le programme principal désigne le code qui est exécuté lorsqu'on lance le script Python, c'est-à-dire toute la suite d'instructions en dehors des fonctions. En général, dans un script Python, on écrit d'abord les fonctions, puis le programme principal. Nous aurons l'occasion de revenir sur cette notion de programme principal plus tard dans ce chapitre, ainsi que dans le chapitre 13 *Plus sur les fonctions*.

10.2 Définition

Pour définir une fonction, Python utilise le mot-clé `def`. Si on souhaite que la fonction renvoie quelque chose, il faut utiliser le mot-clé `return`. Par exemple :

```
>>> def carre(x):
...     return x**2
...
>>> print(carre(2))
4
```

Notez que la syntaxe de `def` utilise les deux-points comme les boucles `for` et `while` ainsi que les tests `if` : un bloc d'instructions est donc attendu. De même que pour les boucles et les tests, l'**indentation** de ce bloc d'instructions (qu'on appelle le corps de la fonction) est **obligatoire**.

Dans l'exemple précédent, nous avons passé un argument à la fonction `carre()`, qui nous a renvoyé (ou retourné) une valeur que nous avons immédiatement affichée à l'écran avec l'instruction `print()`. Que veut dire valeur renvoyée ? Et bien cela signifie que cette dernière est récupérable dans une variable :

```
>>> res = carre(2)
>>> print(res)
4
```

Ici, le résultat renvoyé par la fonction est stocké dans la variable `res`. Notez qu'une fonction ne prend pas forcément un argument et ne renvoie pas forcément une valeur, par exemple :

```
>>> def hello():
...     print("bonjour")
...
>>> hello()
bonjour
```

Dans ce cas, la fonction `hello()` se contente d'afficher la chaîne de caractères `"bonjour"` à l'écran. Elle ne prend aucun argument et ne renvoie rien. Par conséquent, cela n'a pas de sens de vouloir récupérer dans une variable le résultat renvoyé par une telle fonction. Si on essaie tout de même, Python affecte la valeur `None` qui signifie *rien* en anglais :

```
>>> var = hello()
bonjour
>>> print(var)
None
```

Ceci n'est pas une faute car Python n'émet pas d'erreur, toutefois cela ne présente, la plupart du temps, guère d'intérêt.

10.3 Passage d'arguments

Le nombre d'arguments que l'on peut passer à une fonction est variable. Nous avons vu ci-dessus des fonctions auxquelles on passait zero ou un argument. Dans les chapitres précédents, vous avez rencontré des fonctions internes à Python qui prenaient au moins deux arguments. Souvenez-vous par exemple de `range(1, 10)` ou encore `range(1, 10, 2)`. Le nombre d'arguments est donc laissé libre à l'initiative du programmeur qui développe une nouvelle fonction.

Une particularité des fonctions en Python est que vous n'êtes pas obligé de préciser le type des arguments que vous lui passez, dès lors que les opérations que vous effectuez avec ces arguments sont valides. Python est en effet connu comme étant un langage au « typage dynamique », c'est-à-dire qu'il reconnaît pour vous le type des variables au moment de l'exécution. Par exemple :

```
>>> def fois(x, y):
...     return x*y
...
>>> fois(2, 3)
6
>>> fois(3.1415, 5.23)
16.430045000000003
>>> fois("to", 2)
'toto'
>>> fois([1,3], 2)
[1, 3, 1, 3]
```

L'opérateur `*` reconnaît plusieurs types (entiers, *floats*, chaînes de caractères, listes). Notre fonction `fois()` est donc capable d'effectuer des tâches différentes ! Même si Python autorise cela, méfiez-vous tout de même de cette grande flexibilité qui pourrait conduire à des surprises dans vos futurs programmes. En général, il est plus judicieux que chaque argument ait un type précis (entiers, *floats*, chaînes de caractères, etc.) et pas l'un ou l'autre.

10.4 Renvoi de résultats

Un énorme avantage en Python est que les fonctions sont capables de renvoyer plusieurs objets à la fois, comme dans cette fraction de code :

```
>>> def carre_cube(x):
...     return x**2, x**3
...
>>> carre_cube(2)
(4, 8)
```

En réalité Python ne renvoie qu'un seul objet, mais celui-ci peut être séquentiel, c'est-à-dire contenir lui-même d'autres objets. Dans notre exemple, Python renvoie un objet de type `tuple`, type que nous avons vu dans le chapitre 8 *Dictionnaires et tuples* (souvenez-vous, il s'agit d'une sorte de liste avec des propriétés différentes). Notre fonction pourrait tout autant renvoyer une liste :

```
>>> def carre_cube2(x):
...     return [x**2, x**3]
...
>>> carre_cube2(3)
[9, 27]
```

Renvoyer un *tuple* ou une liste de deux éléments (ou plus) est très pratique en conjonction avec l'**affectation multiple**, par exemple :

```
>>> z1, z2 = carre_cube2(3)
>>> z1
9
>>> z2
27
```

Cela permet de récupérer plusieurs valeurs renvoyées par une fonction et de les affecter à la volée à des variables différentes.

Une fonction peut aussi renvoyer un booléen :

```
def est_pair(x):
    if x % 2 == 0:
        return True
    else:
        return False

# Programme principal.
for chiffre in range(1, 5):
    if est_pair(chiffre):
        print(f"{chiffre} est pair")
```

Comme la fonction renvoie un booléen, on peut utiliser la notation `if est_pair(chiffre):` qui équivaut à `if est_pair(chiffre) == True:`. Il est courant d'appeler une fonction qui renvoie un booléen `est_quelquechose()` car on comprend que ça pose la question si c'est vrai ou faux. En anglais, on trouvera la notation `is_even()`. Nous reverrons ces notions dans le chapitre 13 *Plus sur les fonctions*.

10.5 Arguments positionnels et arguments par mot-clé

Jusqu'à maintenant, nous avons systématiquement passé le nombre d'arguments que la fonction attendait. Que se passe-t-il si une fonction attend deux arguments et que nous ne lui en passons qu'un seul ?

```
>>> def fois(x, y):
...     return x*y
...
>>> fois(2, 3)
6
>>> fois(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fois() missing 1 required positional argument: 'y'
```

On constate que passer un seul argument à une fonction qui en attend deux conduit à une erreur.

Définition

Lorsqu'on définit une fonction `def fct(x, y):` les arguments `x` et `y` sont appelés **arguments positionnels** (en anglais, *positional arguments*). Il est strictement obligatoire de les préciser lors de l'appel de la fonction. De plus, il est nécessaire de respecter le même ordre lors de l'appel que dans la définition de la fonction. Dans l'exemple ci-dessus, `2` correspondra à `x` et `3` correspondra à `y`. Finalement, tout dépendra de leur position, d'où leur qualification de positionnel.

Mais il est aussi possible de passer un ou plusieurs argument(s) de manière facultative et de leur attribuer une valeur par défaut :

```
>>> def fct(x=1):
...     return x
...
>>> fct()
1
>>> fct(10)
10
```

Définition

Un argument défini avec une syntaxe `def fct(arg=val):` est appelé **argument par mot-clé** (en anglais, *keyword argument*). Le passage d'un tel argument lors de l'appel de la fonction est facultatif. Ce type d'argument ne doit pas être confondu avec les arguments positionnels présentés ci-dessus, dont la syntaxe est `def fct(arg):`.

Il est bien sûr possible de passer plusieurs arguments par mot-clé :

```
>>> def fct(x=0, y=0, z=0):
...     return x, y, z
...
>>> fct()
(0, 0, 0)
>>> fct(10)
(10, 0, 0)
>>> fct(10, 8)
(10, 8, 0)
>>> fct(10, 8, 3)
(10, 8, 3)
```

On observe que pour l'instant, les arguments par mot-clé sont pris dans l'ordre dans lesquels on les passe lors de l'appel. Comment faire si l'on souhaitait préciser l'argument par mot-clé `z` et garder les valeurs de `x` et `y` par défaut ? Simplement en précisant le nom de l'argument lors de l'appel :

```
>>> fct(z=10)
(0, 0, 10)
```

Python permet même de rentrer les arguments par mot-clé dans un ordre arbitraire :

```
>>> fct(z=10, x=3, y=80)
(3, 80, 10)
>>> fct(z=10, y=80)
(0, 80, 10)
```

Que se passe-t-il lorsque nous avons un mélange d'arguments positionnels et par mot-clé ? Et bien les arguments positionnels doivent toujours être placés avant les arguments par mot-clé :

```
>>> def fct(a, b, x=0, y=0, z=0):
...     return a, b, x, y, z
...
>>> fct(1, 1)
(1, 1, 0, 0, 0)
>>> fct(1, 1, z=5)
(1, 1, 0, 0, 5)
>>> fct(1, 1, z=5, y=32)
(1, 1, 0, 32, 5)
```

On peut toujours passer les arguments par mot-clé dans un ordre arbitraire à partir du moment où on précise leur nom. Par contre, si les deux arguments positionnels `a` et `b` ne sont pas passés à la fonction, Python renvoie une erreur.

```
>>> fct(z=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fct() missing 2 required positional arguments: 'a' and 'b'
```

Conseil

Préciser le nom des arguments par mot-clé lors de l'appel d'une fonction est une pratique que nous vous recommandons. Cela les distingue clairement des arguments positionnels.

L'utilisation d'arguments par mot-clé est habituelle en Python. Elle permet de modifier le comportement par défaut de nombreuses fonctions. Par exemple, si on souhaite que la fonction `print()` n'affiche pas un retour à la ligne, on peut utiliser l'argument `end` :

```
>>> print("Message ", end="")
Message >>>
```

Nous verrons, dans le chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne), que l'utilisation d'arguments par mot-clé est systématique lorsqu'on crée un objet graphique (une fenêtre, un bouton, etc.).

10.6 Variables locales et variables globales

Lorsqu'on manipule des fonctions, il est essentiel de bien comprendre comment se comportent les variables. Une variable est dite **locale** lorsqu'elle est créée dans une fonction. Elle n'existera et ne sera visible que lors de l'exécution de ladite fonction.

Une variable est dite **globale** lorsqu'elle est créée dans le programme principal. Elle sera visible partout dans le programme.

Ceci ne vous paraît pas clair ? Nous allons prendre un exemple simple qui vous aidera à mieux saisir ces concepts. Observez le code suivant :

```
# Définition d'une fonction carre().
def carre(x):
    y = x**2
    return y

# Programme principal.
var = 5
resultat = carre(var)
print(resultat)
```

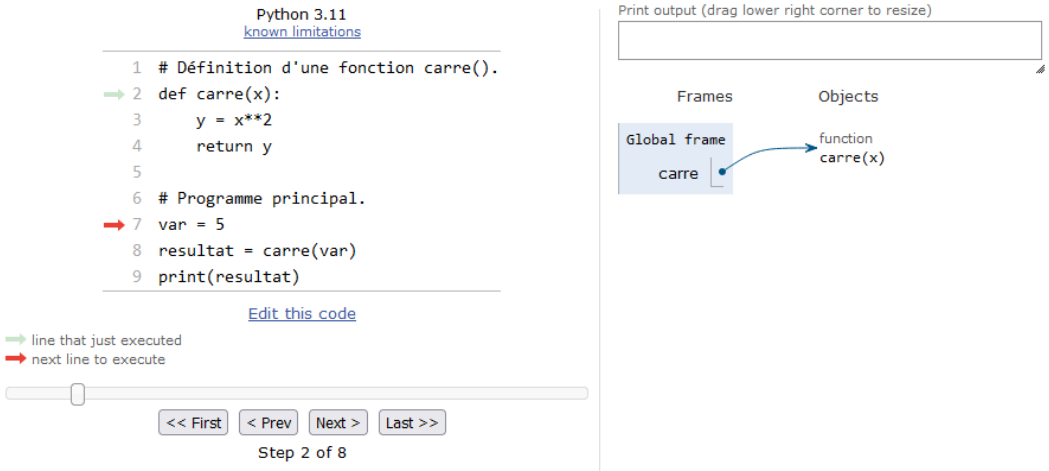
Pour la suite des explications, nous allons utiliser l'excellent site *Python Tutor*[1] qui permet de visualiser l'état des variables au fur et à mesure de l'exécution d'un code Python. Avant de poursuivre, nous vous conseillons de prendre 5 minutes pour tester ce site.

Regardons maintenant ce qui se passe dans le code ci-dessus, étape par étape :

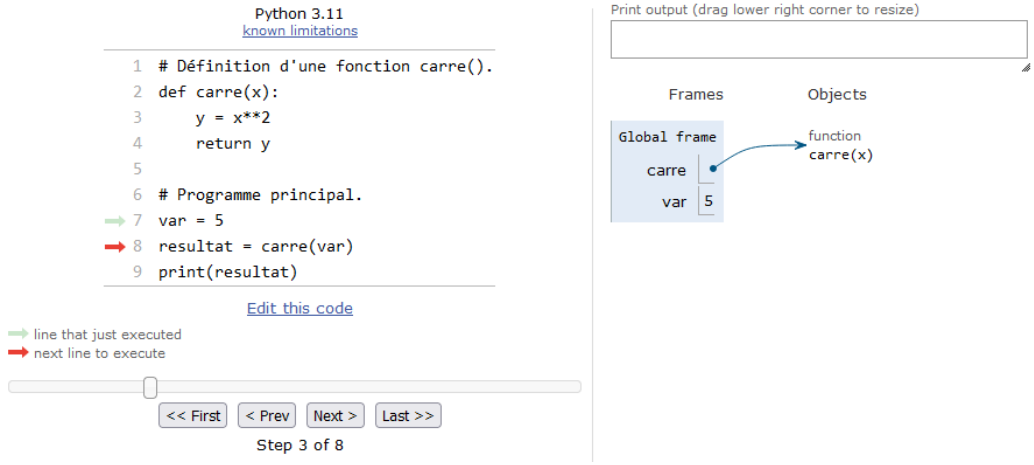
- **Étape 1** : Python est prêt à lire la première ligne de code.

1. http://www.pythontutor.com

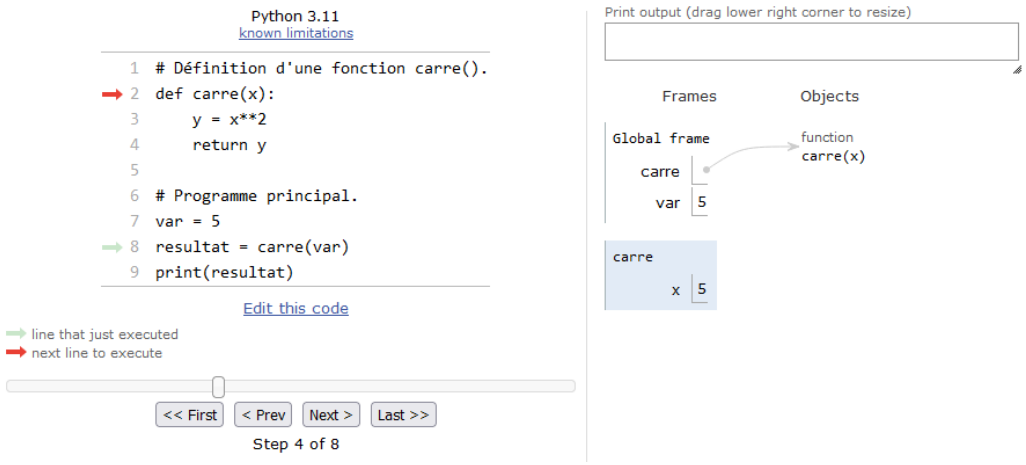
- **Étape 2** : Python met en mémoire la fonction `carre()`. Notez qu'il ne l'exécute pas ! La fonction est mise dans un espace de la mémoire nommé *Global frame*, il s'agit de l'espace du programme principal. Dans cet espace seront stockées toutes les variables *globales* créées dans le programme. Python est maintenant prêt à exécuter le programme principal.



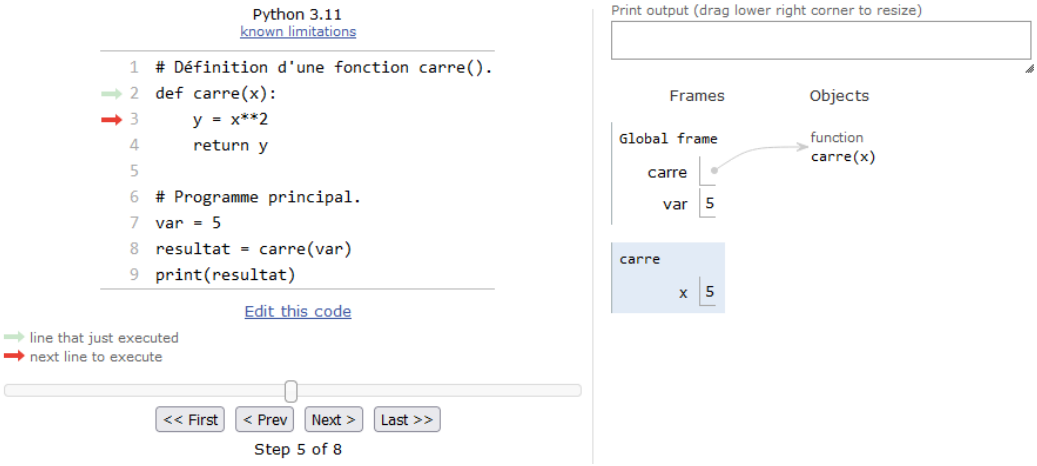
- **Étape 3** : Python lit et met en mémoire la variable `var`. Celle-ci étant créée dans le programme principal, il s'agira d'une variable *globale*. Ainsi, elle sera également stockée dans le *Global frame*.



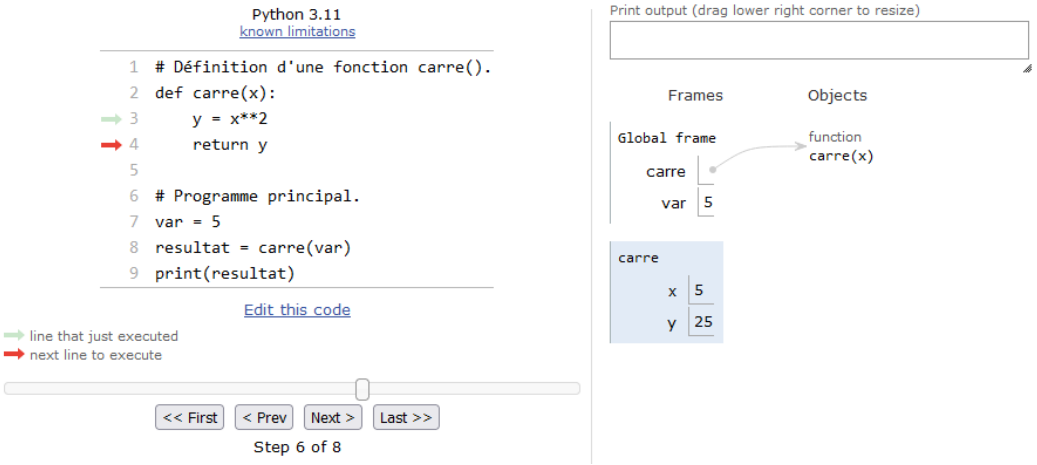
- **Étape 4** : La fonction `carre()` est appelée et on lui passe en argument l'entier `var`. La fonction s'exécute et un nouveau cadre est créé dans lequel *Python Tutor* va indiquer toutes les variables *locales* à la fonction. Notez bien que la variable passée en argument, qui s'appelle `x` dans la fonction, est créée en tant que variable *locale*. On remarquera aussi que les variables *globales* situées dans le *Global frame* sont toujours là.



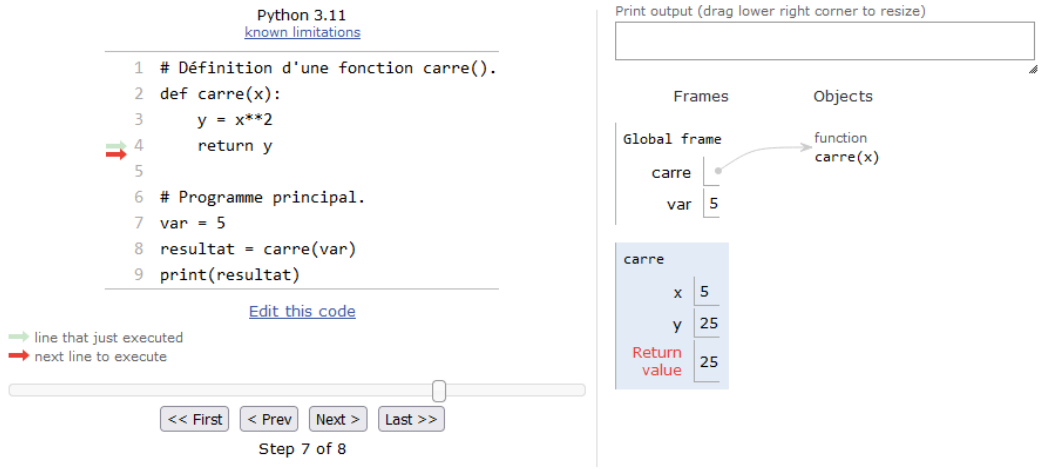
- **Étape 5** : Python est maintenant prêt à exécuter chaque ligne de code de la fonction.



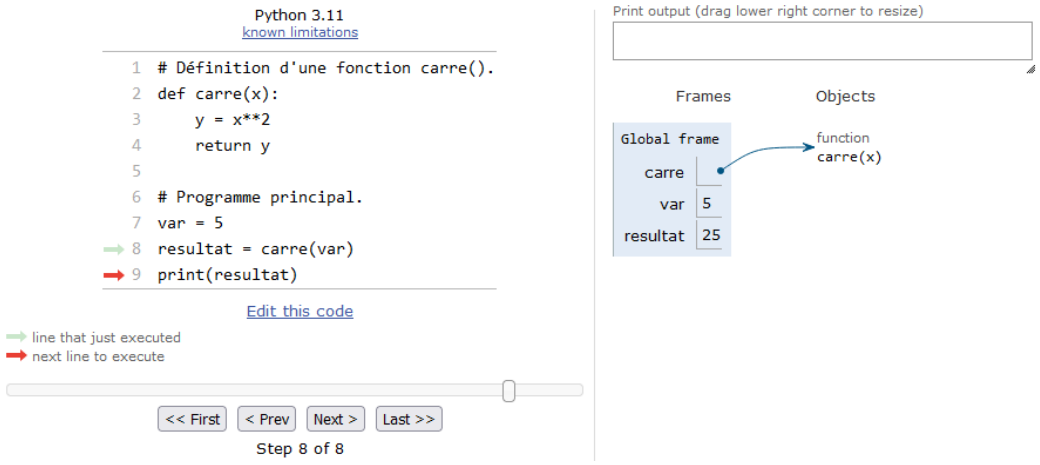
- **Étape 6** : La variable y est créée dans la fonction. Celle-ci est donc stockée en tant que variable *locale* à la fonction.



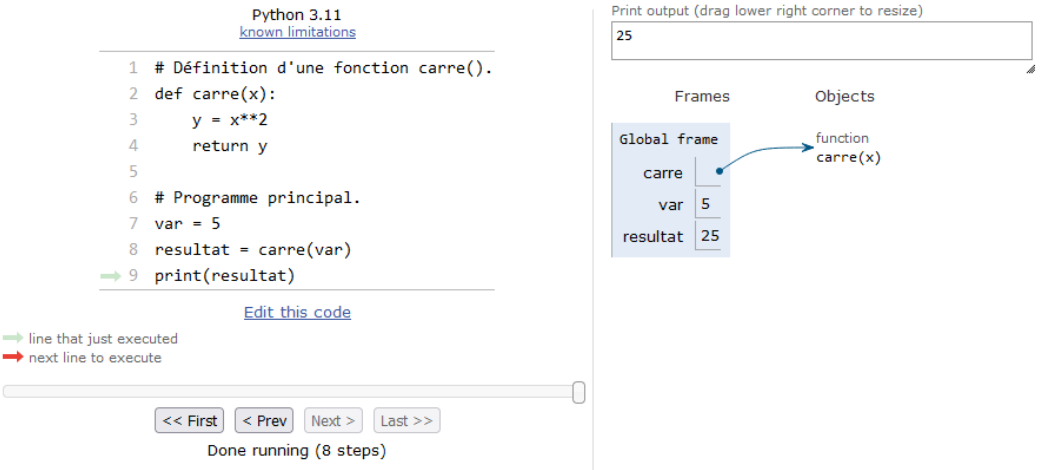
- **Étape 7** : Python s'apprête à renvoyer la variable *locale* y au programme principal. *Python Tutor* nous indique le contenu de la valeur renvoyée.



- **Étape 8** : Python quitte la fonction et la valeur renvoyée par celle-ci est affectée à la variable *globale* `resultat`. Notez bien que lorsque Python quitte la fonction, **l'espace des variables alloué à la fonction est détruit**. Ainsi, toutes les variables créées dans la fonction n'existent plus. On comprend pourquoi elles portent le nom de *locales* puisqu'elles n'existent que lorsque la fonction est exécutée.



- **Étape 9** : Python affiche le contenu de la variable `resultat` et l'exécution est terminée.



Nous espérons que cet exemple guidé facilitera la compréhension des concepts de variables locales et globales. Cela viendra aussi avec la pratique. Nous irons un peu plus loin sur les fonctions dans le chapitre 13 *Plus sur les fonctions*.

D'ici là, essayez de vous entraîner au maximum avec les fonctions. C'est un concept ardu, mais il est impératif de le maîtriser.

Enfin, comme vous avez pu le constater, *Python Tutor* nous a grandement aidé à comprendre ce qui se passait. N'hésitez pas à l'utiliser sur des exemples ponctuels, ce site vous aidera à visualiser ce qui se passe lorsqu'un code ne fait pas ce que vous attendez.

10.7 Principe DRY

L'acronyme DRY[2] signifie *Don't Repeat Yourself*. Les fonctions permettent de satisfaire ce principe en évitant la duplication de code. En effet, plus un code est dupliqué plusieurs fois dans un programme, plus il sera source d'erreurs, notamment lorsqu'il faudra le faire évoluer.

Considérons par exemple le code suivant qui convertit plusieurs températures des degrés Fahrenheit en degrés Celsius :

```
1  >>> temp_in_fahrenheit = 60
2  >>> (temp_in_fahrenheit - 32) * (5/8)
3  17.5
4  >>> temp_in_fahrenheit = 80
5  >>> (temp_in_fahrenheit - 32) * (5/8)
6  30.0
7  >>> temp_in_fahrenheit = 100
8  >>> (temp_in_fahrenheit - 32) * (5/8)
9  42.5
```

Malheureusement, il y a une erreur dans la formule de conversion. En effet, la formule exacte est :

$$\text{temp\_celsius} = (\text{temp\_fahrenheit} - 32) \times \frac{5}{9}$$

Il faut alors reprendre les lignes 2, 5 et 8 précédentes et les corriger. Cela n'est pas efficace, surtout si le même code est utilisé à différents endroits dans le programme.

En écrivant qu'une seule fois la formule de conversion dans une fonction, on applique le principe DRY :

```
1  >>> def convert_fahrenheit_to_celsius(temperature):
2  ...     return (temperature - 32) * (5/9)
3  ...
4  >>> temp_in_fahrenheit = 60
5  >>> convert_fahrenheit_to_celsius(temp_in_fahrenheit)
6  15.555555555555557
7  >>> temp_in_fahrenheit = 80
8  >>> convert_fahrenheit_to_celsius(temp_in_fahrenheit)
9  26.666666666666668
10 >>> temp_in_fahrenheit = 100
11 >>> convert_fahrenheit_to_celsius(temp_in_fahrenheit)
12 37.77777777777778
```

Et s'il y a une erreur dans la formule, il suffira de ne la corriger qu'une seule fois, dans la fonction `convert_fahrenheit_to_celsius()`.

10.8 Exercices

Conseil

Pour le premier exercice, utilisez *Python Tutor*. Pour les exercices suivants, créez des scripts puis exécutez-les dans un *shell*.

10.8.1 Carré et factorielle

Reprenez l'exemple précédent à l'aide du site *Python Tutor*[3] :

2. https://www.earthdatascience.org/courses/intro-to-earth-data-science/write-efficient-python-code/intro-to-clean-code/dry-modular-code/
3. http://www.pythontutor.com

```python
# Définition d'une fonction carre().
def carre(x):
    y = x**2
    return y

# Programme principal.
z = 5
resultat = carre(z)
print(resultat)
```

Analysez ensuite le code suivant et tentez de prédire sa sortie :

```python
def calc_factorielle(n):
    fact = 1
    for i in range(2, n+1):
        fact = fact * i
    return fact

# Programme principal.
nb = 4
factorielle_nb = calc_factorielle(nb)
print(f"{nb}! = {factorielle_nb}")
nb2 = 10
print(f"{nb2}! = {calc_factorielle(nb2)}")
```

Testez ensuite cette portion de code avec *Python Tutor*, en cherchant à bien comprendre chaque étape. Avez-vous réussi à prédire la sortie correctement ?

Remarque

Une remarque concernant l'utilisation des *f-strings* que nous avions abordées dans le chapitre 3 *Affichage*. On découvre ici une autre possibilité des *f-strings* dans l'instruction `f"{nb2}! = {calc_factorielle(nb2)}"` : il est en effet possible d'appeler entre les accolades une fonction (ici `{calc_factorielle(nb2)}`) ! Ainsi, il n'est pas nécessaire de créer une variable intermédiaire dans laquelle on stocke ce que retourne la fonction.

10.8.2 Puissance

Créez une fonction `calc_puissance(x, y)` qui renvoie x^y en utilisant l'opérateur `**`. Pour rappel :

```
>>> 2**2
4
>>> 2**3
8
>>> 2**4
16
```

Dans le programme principal, calculez et affichez à l'écran 2^i avec i variant de 0 à 20 inclus. On souhaite que le résultat soit présenté avec le formatage suivant :

```
2^ 0 =       1
2^ 1 =       2
2^ 2 =       4
[...]
2^20 = 1048576
```

10.8.3 Pyramide

Reprenez l'exercice du chapitre 5 *Boucles et comparaisons* qui dessine une pyramide.

Dans un script `pyra.py`, créez une fonction `gen_pyramide()` à laquelle vous passez un nombre entier `N` et qui renvoie une pyramide de N lignes sous forme de chaîne de caractères. Le programme principal demandera à l'utilisateur le nombre de lignes souhaitées (utilisez pour cela la fonction `input()`) et affichera la pyramide à l'écran.

10.8.4 Nombres premiers

Reprenez l'exercice du chapitre 6 *Tests* sur les nombres premiers.

Créez une fonction `est_premier()` qui prend comme argument un nombre entier positif *n* (supérieur à 2), et qui renvoie le booléen `True` si *n* est premier et `False` si *n* n'est pas premier. Déterminez tous les nombres premiers de 2 à 100. On souhaite avoir une sortie similaire à celle-ci :

```
  2 est premier
  3 est premier
  4 n'est pas premier
[...]
100 n'est pas premier
```

10.8.5 Séquence complémentaire

Créez une fonction `seq_comp()` qui prend comme argument une liste de bases et qui renvoie la séquence complémentaire d'une séquence d'ADN sous forme de liste.

Dans le programme principal, à partir de la séquence d'ADN

`seq = ["A", "T", "C", "G", "A", "T", "C", "G", "A", "T", "C"]`

affichez `seq` et sa séquence complémentaire (en utilisant votre fonction `seq_comp()`).

Rappel : la séquence complémentaire s'obtient en remplaçant A par T, T par A, C par G et G par C.

10.8.6 Distance 3D

Créez une fonction `calc_distance_3D()` qui calcule la distance euclidienne en trois dimensions entre deux atomes. Testez votre fonction sur les 2 points `A(0,0,0)` et `B(1,1,1)`. Trouvez-vous bien $\sqrt{3}$?

On rappelle que la distance euclidienne *d* entre deux points A et B de coordonnées cartésiennes respectives (x_A, y_A, z_A) et (x_B, y_B, z_B) se calcule comme suit :

$$d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2 + (z_B - z_A)^2}$$

10.8.7 Distribution et statistiques

Créez une fonction `gen_distrib()` qui prend comme argument trois entiers : *debut*, *fin* et *n*. La fonction renverra une liste de *n floats* aléatoires entre *debut* et *fin*. Pour générer un nombre aléatoire dans un intervalle donné, utilisez la fonction `uniform()` du module *random*, dont voici quelques exemples d'utilisation :

```
>>> import random
>>> random.uniform(1, 10)
8.199672607202174
>>> random.uniform(1, 10)
2.607528561528022
>>> random.uniform(1, 10)
9.000404025130946
```

Avec la fonction `random.uniform()`, les bornes passées en argument sont incluses, c'est-à-dire qu'ici, le nombre aléatoire renvoyé est dans l'intervalle [1, 10].

Créez une autre fonction `calc_stat()` qui prend en argument une liste de *floats* et qui renvoie une liste de trois éléments contenant respectivement le minimum, le maximum et la moyenne de la liste.

Dans le programme principal, générez 20 listes aléatoires de 100 *floats* compris entre 0 et 100 et affichez le minimum (`min()`), le maximum (`max()`) et la moyenne pour chacune d'entre elles. La moyenne pourra être calculée avec les fonctions `sum()` et `len()`.

Pour chacune des 20 listes, affichez les statistiques (valeur minimale, valeur maximale et moyenne) avec deux chiffres après la virgule :

```
Liste  1 : min = 0.17 ; max = 99.72 ; moyenne = 57.38
Liste  2 : min = 1.25 ; max = 99.99 ; moyenne = 47.41
[...]
Liste 19 : min = 1.05 ; max = 99.36 ; moyenne = 49.43
Liste 20 : min = 1.33 ; max = 97.63 ; moyenne = 46.53
```

Les écarts sur les statistiques entre les différentes listes sont-ils importants ? Relancez votre script avec des listes de 1000 éléments, puis 10 000 éléments. Les écarts changent-ils quand le nombre d'éléments par liste augmente ?

10.8.8 Distance à l'origine (exercice +++)

En reprenant votre fonction de calcul de distance euclidienne en trois dimensions `calc_distance_3D()`, faites-en une version pour deux dimensions que vous appellerez `calc_distance_2D()`.

Créez une autre fonction `calc_dist2ori()`, à laquelle vous passez en argument deux listes de *floats* `list_x` et `list_y` représentant les coordonnées d'une fonction mathématique (par exemple x et $sin(x)$). Cette fonction renverra une liste de *floats* représentant la distance entre chaque point de la fonction et l'origine (de coordonnées $(0,0)$).

La figure 10.2 montre un exemple sur quelques points de la fonction $sin(x)$ (courbe en trait épais). Chaque trait pointillé représente la distance que l'on cherche à calculer entre les points de la courbe et l'origine du repère de coordonnées $(0,0)$.

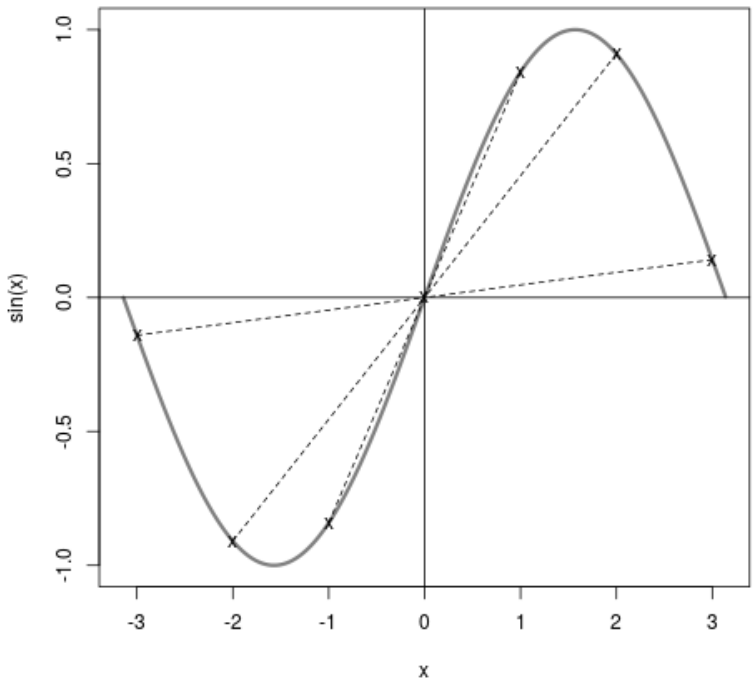


FIGURE 10.2 – Illustration de la distance à l'origine.

Votre programme générera un fichier `sin2ori.dat` qui contiendra deux colonnes : la première représente les x, la seconde la distance entre chaque point de la fonction $sin(x)$ à l'origine.

Enfin, pour visualiser votre résultat, ajoutez le code suivant tout à la fin de votre script :

```
# Création d'une image pour la visualisation du résultat.
import matplotlib.pyplot as plt

x = []
y = []
with open("sin2ori.dat", "r") as f_in:
    for line in f_in:
        coords = line.split()
        x.append(float(coords[0]))
        y.append(float(coords[1]))
fig, ax = plt.subplots(figsize=(6, 6))
ax.plot(x, y)
ax.set_xlabel("x")
ax.set_ylabel("Distance de sin(x) à l'origine")
fig.savefig("sin2ori.png")
```

Ouvrez l'image `sin2ori.png`.

Remarque

Le module *matplotlib* sera expliqué en détail dans le chapitre 21 *Module matplotlib*.

CHAPITRE 11

Plus sur les chaînes de caractères

11.1 Préambule

Nous avons déjà abordé les chaînes de caractères dans les chapitres 2 *Variables* et 3 *Affichage*. Ici nous allons un peu plus loin, notamment avec les méthodes associées aux chaînes de caractères[1].

11.2 Chaînes de caractères et listes

Les chaînes de caractères peuvent être considérées comme des listes (de caractères) un peu particulières :

```
>>> animaux = "girafe tigre"
>>> animaux
'girafe tigre'
>>> len(animaux)
12
>>> animaux[3]
'a'
```

Nous pouvons donc utiliser certaines propriétés des listes comme les tranches :

```
>>> animaux = "girafe tigre"
>>> animaux[0:4]
'gira'
>>> animaux[9:]
'gre'
>>> animaux[:-2]
'girafe tig'
>>> animaux[1:-2:2]
'iaetg'
```

Mais *a contrario* des listes, les chaînes de caractères présentent toutefois une différence notable, ce sont **des listes non modifiables**. Une fois une chaîne de caractères définie, vous ne pouvez plus modifier un de ses éléments. Le cas échéant, Python renvoie un message d'erreur :

1. https://docs.python.org/fr/3/library/string.html

```
>>> animaux = "girafe tigre"
>>> animaux[4]
'f'
>>> animaux[4] = "F"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Par conséquent, si vous voulez modifier une chaîne de caractères, vous devez en construire une nouvelle. Pour cela, n'oubliez pas que les opérateurs de concaténation (`+`) et de duplication (`*`) (introduits dans le chapitre 2 *Variables*) peuvent vous aider. Vous pouvez également générer une liste, qui elle est modifiable, puis revenir à une chaîne de caractères (voir plus bas).

11.3 Caractères spéciaux

Il existe certains caractères spéciaux comme `\n` que nous avons déjà vu (pour le retour à la ligne). Le caractère `\t` produit une tabulation. Si vous voulez écrire des guillemets simples ou doubles et que ceux-ci ne soient pas confondus avec les guillemets de déclaration de la chaîne de caractères, vous pouvez utiliser `\'` ou `\"` :

```
>>> print("Un backslash n\npuis un backslash t\t puis un guillemet\"")
Un backslash n
puis un backslash t     puis un guillemet"
>>> print('J\'affiche un guillemet simple')
J'affiche un guillemet simple
```

Vous pouvez aussi utiliser astucieusement des guillemets doubles ou simples pour déclarer votre chaîne de caractères :

```
>>> print("Un brin d'ADN")
Un brin d'ADN
>>> print('Python est un "super" langage de programmation')
Python est un "super" langage de programmation
```

Quand on souhaite écrire un texte sur plusieurs lignes, il est très commode d'utiliser les guillemets triples qui conservent le formatage (notamment les retours à la ligne) :

```
>>> x = """souris
... chat
... abeille"""
>>> x
'souris\nchat\nabeille'
>>> print(x)
souris
chat
abeille
```

Attention, les caractères spéciaux n'apparaissent intérprétés que lorsqu'ils sont utilisés avec la fonction `print()`. Par exemple, le `\n` n'apparait comme un retour à la ligne que lorsqu'il est dans une chaîne de caractères passée à la fonction `print()` :

```
>>> "bla\nbla"
'bla\nbla'
>>> print("bla\nbla")
bla
bla
```

11.4 Préfixe de chaîne de caractères

Nous avons vu au chapitre 3 *Affichage* la notion de *f-string*. Il s'agit d'un mécanisme pour formater du texte au sein d'une chaîne de caractères. Par exemple :

```
>>> var = "f-string"
>>> f"voici une belle {var}"
'voici une belle f-string'
```

Que signifie le `f` que l'on accole aux guillemets de la chaîne de caractères ? Celui-ci est appelé « préfixe de chaîne de caractères » ou *stringprefix*.

Remarque

Un *stringprefix* modifie la manière dont Python va interpréter ladite *string*. Celui-ci doit être systématiquement « collé » à la chaîne de caractères, c'est-à-dire sans espace entre les deux.

Il existe différents *stringprefixes* en Python, nous vous montrons ici les deux qui nous apparaissent les plus importants.

- Le préfixe `r` mis pour *raw string*, qui force la non-interprétation des caractères spéciaux :

```
>>> s = "Voici un retour à la ligne\nEt là une autre ligne"
>>> s
'Voici un retour à la ligne\nEt là une autre ligne'
>>> print(s)
Voici un retour à la ligne
Et là une autre ligne
>>> s = r"Voici un retour à la ligne\nEt là une autre ligne"
>>> s
'Voici un retour à la ligne\\nEt là une autre ligne'
>>> print(s)
Voici un retour à la ligne\nEt là une autre ligne
```

L'ajout du `r` va forcer Python à ne pas interpréter le `\n` comme un retour à la ligne, mais comme un *backslash* littéral suivi d'un *n*. Quand on demande à l'interpréteur d'afficher cette chaîne de caractères, celui-ci met deux *backslashes* pour signifier qu'il s'agit d'un *backslash* littéral (le premier échappe le second). Finalement, l'utilisation de la syntaxe `r"Voici un retour à la ligne\nEt là une autre ligne"` renvoie une chaîne de caractères normale, puisqu'on voit ensuite que le `r` a disparu lorsqu'on demande à Python d'afficher le contenu de la variable `s`. Comme dans `var = 2 + 2`, d'abord Python évalue `2 + 2`. Puis ce résultat est affecté à la variable `var`. Enfin, on notera que seule l'utilisation du `print()` mène à l'interprétation des caractères spéciaux comme `\n`, comme expliqué dans la rubrique précédente.

Les caractères spéciaux non interprétés dans les *raw strings* sont de manière générale tout ce dont le *backslash* modifie la signification, par exemple un `\n`, un `\t`, etc.

- Le préfixe `f` mis pour *formatted string*, qui met en place l'écriture formatée comme vue au chapitre 3 *Affichage* :

```
>>> animal = "renard"
>>> animal2 = "poulain"
>>> s = f"Le {animal} est un animal gentil\nLe {animal2} aussi"
>>> s
'Le renard est un animal gentil\nLe poulain aussi'
>>> print(s)
Le renard est un animal gentil
Le poulain aussi
>>> s = "Le {animal} est un animal gentil\nLe {animal2} aussi"
>>> s
'Le {animal} est un animal gentil\nLe {animal2} aussi'
>>> print(s)
Le {animal} est un animal gentil
Le {animal2} aussi
```

La *f-string* remplace le contenu des variables situées entre les accolades et interprète le `\n` comme un retour à la ligne. Pour rappel, consultez le chapitre 3 si vous souhaitez plus de détails sur le fonctionnement des *f-strings*.

Conseil

Il existe de nombreux autres détails concernant les préfixes qui vont au-delà de ce cours. Pour en savoir plus, vous pouvez consulter la documentations officielle[2].

2. https://docs.python.org/fr/3/reference/lexical_analysis.html#grammar-token-stringprefix

11.5 Méthodes associées aux chaînes de caractères

Voici quelques méthodes[3] spécifiques aux objets de type `str` :

```
>>> x = "girafe"
>>> x.upper()
'GIRAFE'
>>> x
'girafe'
>>> 'TIGRE'.lower()
'tigre'
```

Les méthodes `.lower()` et `.upper()` renvoient un texte en minuscule et en majuscule respectivement. On remarque que l'utilisation de ces méthodes n'altère pas la chaîne de caractères de départ, mais renvoie une chaîne de caractères transformée.

Pour mettre en majuscule la première lettre seulement, vous pouvez faire :

```
>>> x[0].upper() + x[1:]
'Girafe'
```

ou plus simplement utiliser la méthode adéquate :

```
>>> x.capitalize()
'Girafe'
```

Il existe une méthode associée aux chaînes de caractères qui est particulièrement pratique, la méthode `.split()` :

```
>>> animaux = "girafe tigre singe souris"
>>> animaux.split()
['girafe', 'tigre', 'singe', 'souris']
>>> for animal in animaux.split():
...     print(animal)
...
girafe
tigre
singe
souris
```

La méthode `.split()` découpe une chaîne de caractères en plusieurs éléments appelés *champs*, en utilisant comme séparateur n'importe quelle combinaison « d'espace(s) blanc(s) ».

Définition

Un espace blanc[4] (*whitespace* en anglais) correspond aux caractères qui sont invisibles à l'œil, mais qui occupent de l'espace dans un texte. Les espaces blancs les plus classiques sont l'espace, la tabulation et le retour à la ligne.

Il est possible de modifier le séparateur de champs, par exemple :

```
>>> animaux = "girafe:tigre:singe::souris"
>>> animaux.split(":")
['girafe', 'tigre', 'singe', '', 'souris']
```

Attention, dans cet exemple, le séparateur est un seul caractères « `:` » (et non pas une combinaison de un ou plusieurs `:`) conduisant ainsi à une chaîne vide entre `singe` et `souris`.

Il est également intéressant d'indiquer à `.split()` le nombre de fois qu'on souhaite découper la chaîne de caractères avec l'argument `maxsplit` :

```
>>> animaux = "girafe tigre singe souris"
>>> animaux.split(maxsplit=1)
['girafe', 'tigre singe souris']
>>> animaux.split(maxsplit=2)
['girafe', 'tigre', 'singe souris']
```

3. https://docs.python.org/fr/3/library/string.html
4. https://en.wikipedia.org/wiki/Whitespace_character

La méthode `.find()`, quant à elle, recherche une chaîne de caractères passée en argument :

```
>>> animal = "girafe"
>>> animal.find("i")
1
>>> animal.find("afe")
3
>>> animal.find("z")
-1
>>> animal.find("tig")
-1
```

Si l'élément recherché est trouvé, alors l'indice du début de l'élément dans la chaîne de caractères est renvoyé. Si l'élément n'est pas trouvé, alors la valeur `-1` est renvoyée.

Si l'élément recherché est trouvé plusieurs fois, seul l'indice de la première occurrence est renvoyé :

```
>>> animaux = "girafe tigre"
>>> animaux.find("i")
1
```

On trouve aussi la méthode `.replace()` qui substitue une chaîne de caractères par une autre :

```
>>> animaux = "girafe tigre"
>>> animaux.replace("tigre", "singe")
'girafe singe'
>>> animaux.replace("i", "o")
'gorafe togre'
```

La méthode `.count()` compte le nombre d'occurrences d'une chaîne de caractères passée en argument :

```
>>> animaux = "girafe tigre"
>>> animaux.count("i")
2
>>> animaux.count("z")
0
>>> animaux.count("tigre")
1
```

La méthode `.startswith()` vérifie si une chaîne de caractères commence par une autre chaîne de caractères :

```
>>> chaine = "Bonjour monsieur le capitaine !"
>>> chaine.startswith("Bonjour")
True
>>> chaine.startswith("Au revoir")
False
```

Cette méthode est particulièrement utile lorsqu'on lit un fichier et que l'on veut récupérer certaines lignes commençant par un mot-clé. Par exemple dans un fichier PDB, les lignes contenant les coordonnées des atomes commencent par le mot-clé `ATOM`.

Enfin, la méthode `.strip()` permet de « nettoyer les bords » d'une chaîne de caractères :

```
>>> chaine = "  Comment enlever les espaces au début et à la fin ?       "
>>> chaine.strip()
'Comment enlever les espaces au début et à la fin ?'
```

La méthode `.strip()` enlève les espaces situés sur les bords de la chaîne de caractère mais pas ceux situés entre des caractères visibles. En réalité, cette méthode enlève n'importe quel combinaison « d'espace(s) blanc(s) » sur les bords, par exemple :

```
>>> chaine = "  \tfonctionne avec les tabulations et les retours à la ligne\n"
>>> chaine.strip()
'fonctionne avec les tabulations et les retours à la ligne'
```

Cette méthode est utile pour se débarrasser des retours à la ligne quand on lit un fichier.

11.6 Extraction de valeurs numériques d'une chaîne de caractères

Une tâche courante en Python est de lire une chaîne de caractères (provenant par exemple d'un fichier), d'en extraire des valeurs pour ensuite les manipuler.

On considère par exemple la chaîne de caractères `chaine1` :

```
>>> chaine1 = "3.4 17.2 atom"
```

On souhaite extraire les valeurs `3.4` et `17.2` pour ensuite les additionner.

D'abord, on découpe la chaîne de caractères avec la méthode `.split()` :

```
>>> liste1 = chaine1.split()
>>> liste1
['3.4', '17.2', 'atom']
>>> nb1, nb2, nom = liste1
>>> nb1
'3.4'
>>> nb2
'17.2'
```

On obtient alors une liste de chaînes de caractères `liste1`. Avec l'affectation multiple, on récupère les nombres souhaités dans `nb1` et `nb2`, mais ils sont toujours sous forme de chaîne de caractères. Il faut ensuite les convertir en *floats* pour pouvoir les additionner :

```
>>> float(nb1) + float(nb2)
20.599999999999998
```

Remarque

Retenez bien l'utilisation des instructions précédentes pour extraire des valeurs numériques d'une chaîne de caractères. Elles sont régulièrement employées pour analyser des données extraites d'un fichier.

11.7 Fonction `map()`

Conseil

Si vous êtes débutant, vous pouvez sauter cette rubrique.

La fonction `map()` permet d'appliquer une fonction à plusieurs éléments d'un objet itérable. Par exemple, si on a une chaîne de caractères avec trois entiers séparés par des espaces, on peut extraire et convertir les trois nombres en entier en une seule ligne. La fonction `map()` produit un objet de type *map* qui est itérable et transformable en liste :

```
>>> ligne = "67 946   -45"
>>> ligne.split()
['67', '946', '-45']
>>> map(int, ligne.split())
<map object at 0x7fa34e573b20>
>>> for entier in map(int, ligne.split()):
...     print(entier)
...
67
946
-45
>>> list(map(int, ligne.split()))
[67, 946, -45]
```

Remarque

La fonction `map()` prend deux arguments. Le second est un objet itérable, souvent une liste comme dans notre exemple. Le premier argument est le nom d'une fonction qu'on souhaite appliquer à chaque élément de la liste, mais sans les parenthèses (ici `int` et non pas `int()`). Une fonction passée en argument d'une autre fonction est appelée fonction de rappel[5] ou *callback* en anglais. Nous reverrons cette notion dans le chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne).

La fonction `map()` est particulièrement utile lorsqu'on lit un fichier de valeurs numériques. Par exemple, si on a un fichier `data.dat` contenant trois colonnes de nombres, `map()` en conjonction avec `.split()` permet de séparer les trois nombres puis de les convertir en *float* en une seule ligne de code :

```
with open("data.dat", "r") as filin:
    for line in filin:
        x, y, z = map(float, line.split())
        print(x + y + z)
```

Sans `map()`, il aurait fallu une ligne pour séparer les données `x, y, z = line.split()` et une autre pour les transformer en *float* `x, y, z = float(x), float(y), float(z)`.

Enfin, on peut utiliser `map()` avec ses propres fonctions :

```
>>> def calc_cube(x):
...     return x**3
...
>>> list(map(calc_cube, [1, 2, 3, 4]))
[1, 8, 27, 64]
```

11.8 Test d'appartenance

L'opérateur `in` teste si une chaîne de caractères fait partie d'une autre chaîne de caractères :

```
>>> chaine = "Néfertiti"
>>> "toto" in chaine
False
>>> "titi" in chaine
True
>>> "ti" in chaine
True
```

Notez que la chaîne testée peut-être présente à n'importe quelle position dans l'autre chaîne. Par ailleurs, le test est vrai si elle est présente une ou plusieurs fois.

La variation avec l'opérateur booléen `not` permet de vérifier qu'une chaîne n'est pas présente dans une autre chaîne :

```
>>> not "toto" in chaine
True
>>> not "fer" in chaine
False
```

11.9 Conversion d'une liste de chaînes de caractères en une chaîne de caractères

On a vu dans le chapitre 2 *Variables* la conversion d'un type simple (entier, *float* et chaîne de caractères) en un autre avec les fonctions `int()`, `float()` et `str()`. La conversion d'une liste de chaînes de caractères en une chaîne de caractères est moins intuitive. Elle fait appelle à la méthode `.join()` :

5. https://fr.wikipedia.org/wiki/Fonction_de_rappel

```
>>> seq = ["A", "T", "G", "A", "T"]
>>> seq
['A', 'T', 'G', 'A', 'T']
>>> "-".join(seq)
'A-T-G-A-T'
>>> " ".join(seq)
'A T G A T'
>>> "".join(seq)
'ATGAT'
```

Les éléments de la liste initiale sont concaténés les uns à la suite des autres et intercalés par un séparateur, qui peut être n'importe quelle chaîne de caractères. Ici, on a utilisé un tiret, un espace et rien (une chaîne de caractères vide).

Attention, la méthode `.join()` ne s'applique qu'à une liste de chaînes de caractères :

```
>>> maliste = ["A", 5, "G"]
>>> " ".join(maliste)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 1: expected str instance, int found
```

On espère qu'après ce petit tour d'horizon vous serez convaincu de la richesse des méthodes associées aux chaînes de caractères. Pour avoir une liste exhaustive de l'ensemble des méthodes associées à une variable particulière, vous pouvez utiliser la fonction `dir()` :

```
>>> animaux = "girafe tigre"
>>> dir(animaux)
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
 ...,
 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Pour l'instant, vous pouvez ignorer les méthodes qui commencent et qui se terminent par deux tirets bas (*underscores*) `__`. Nous n'avons pas mis l'ensemble de la sortie de cette commande `dir()` pour ne pas surcharger le texte, mais n'hésitez pas à la tester dans l'interpréteur.

Vous pouvez également accéder à l'aide et à la documentation d'une méthode particulière avec `help()`, par exemple pour la méthode `.split()` :

```
>>> help(animaux.split)
Help on built-in function split:

split(...)
    S.split([sep [,maxsplit]]) -> list of strings

    Return a list of the words in the string S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator.
(END)
```

Attention à ne pas mettre les parenthèses à la suite du nom de la méthode. L'instruction correcte est `help(animaux.split)` et non pas `help(animaux.split())`.

11.10 *Method chaining*

Il existe de nombreuses méthodes pour traiter les chaînes de caractères. Ces méthodes renvoient la plupart du temps une chaîne de caractères modifiée.

Par exemple, si on souhaite mettre une majuscule à tous les mots d'une chaîne de caractères, puis remplacer un mot par un autre, puis transformer cette chaîne de caractères en une liste de chaînes de caractères, on peut écrire :

```
>>> message = "salut patrick salut pierre"
>>> message1 = message.title()
>>> message1
'Salut Patrick Salut Pierre'
>>> message2 = message1.replace("Salut", "Bonjour")
>>> message2
'Bonjour Patrick Bonjour Pierre'
>>> message2.split()
['Bonjour', 'Patrick', 'Bonjour', 'Pierre']
```

On a créé deux variables intermédiaires `message1` et `message2` pour stocker les chaînes de caractères modifiées par les méthodes `.title()` et `.replace()`.

Il est possible de faire la même chose en une seule ligne, en utilisant le chaînage de méthodes ou *method chaining* :

```
>>> message = "salut patrick salut pierre"
>>> message.title().replace("Salut", "Bonjour").split()
['Bonjour', 'Patrick', 'Bonjour', 'Pierre']
```

On évite ainsi de créer des variables intermédiaires.

Le *method chaining* peut créer des lignes de code très longues. On peut couper une ligne de code en plusieurs lignes en utilisant le caractère \ en fin de ligne :

```
>>> message = "salut patrick salut pierre"
>>> message.title() \
... .replace("Salut", "Bonjour") \
... .title()
'Bonjour Patrick Bonjour Pierre'
```

On peut aussi utiliser des parenthèses pour couper une ligne de code en plusieurs lignes :

```
>>> message = "salut patrick salut pierre"
>>> (message
... .title()
... .replace("Salut", "Bonjour")
... .split()
... )
['Bonjour', 'Patrick', 'Bonjour', 'Pierre']
```

L'utilisation de parenthèses permet aussi de couper une chaîne de caractères en plusieurs lignes :

```
>>> ma_chaine = (
... "voici une chaine de caractères "
... "très longue "
... "sur plusieurs lignes")
>>> ma_chaine
'voici une chaine de caractères très longue sur plusieurs lignes'
```

Nous reverrons le *method chaining* dans le chapitre 22 *Module Pandas*.

11.11 Exercices

Conseil

Pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

11.11.1 Parcours d'une liste de chaînes de caractères

Soit la liste `['girafe', 'tigre', 'singe', 'souris']`. Avec une boucle, affichez chaque élément ainsi que sa taille (nombre de caractères).

11.11.2 Lecture d'une séquence à partir d'un fichier FASTA

Le fichier `UBI4_SCerevisiae.fasta`[6] contient une séquence d'ADN au format FASTA.

Créez une fonction `lit_fasta()` qui prend comme argument le nom d'un fichier FASTA sous la forme d'une chaîne de caractères, lit la séquence dans le fichier FASTA et la renvoie sous la forme d'une chaîne de caractères.

Utilisez ensuite cette fonction pour récupérer la séquence d'ADN dans la variable `sequence` puis pour afficher les informations suivantes :

- le nom du fichier FASTA,
- la longueur de la séquence (c'est-à-dire le nombre de bases qu'elle contient),
- un message vérifiant que le nombre de bases est (ou non) un multiple de 3,
- le nombre de codons (on rappelle qu'un codon est un bloc de 3 bases),
- les 10 premières bases,
- les 10 dernières bases.

La sortie produite par le script devrait ressembler à ça :

```
UBI4_SCerevisiae.fasta
La séquence contient WWW bases
La longueur de la séquence est un multiple de 3 bases
La séquence possède XXX codons
10 premières bases : YYYYYYYYYY
10 dernières bases : ZZZZZZZZZZ
```

où `WWW` et `XXX` sont des entiers et `YYYYYYYYYY` et `ZZZZZZZZZZ` sont des bases.

Conseil

Vous trouverez des explications sur le format FASTA et des exemples de code dans l'annexe A *Quelques formats de données en biologie*.

11.11.3 Fréquence des bases dans une séquence d'ADN

Soit la séquence d'ADN `ATATACGGATCGGCTGTTGCCTGCGTAGTAGCGT`. On souhaite calculer la fréquence de chaque base A, T, C et G dans cette séquence et afficher le résultat à l'écran.

Créez pour cela une fonction `calc_composition()` à laquelle vous passez en argument votre séquence d'ADN sous forme d'une chaîne de caractères, et qui renvoie une liste de quatre *floats* indiquant respectivement la fréquence en bases `A`, `T`, `G` et `C`.

11.11.4 Distance de Hamming

La distance de Hamming[7] mesure la différence entre deux séquences de même taille en comptant le nombre de positions qui, pour chaque séquence, ne correspondent pas au même acide aminé.

Créez la fonction `dist_hamming()` qui prend en argument deux chaînes de caractères et qui renvoie la distance de Hamming (sous la forme d'un entier) entre ces deux chaînes de caractères.

Calculez la distance de Hamming entre les séquences : `AGWPSGGASAGLAIL` et `IGWPSAGASAGLWIL`

puis entre les séquences : `ATTCATACGTTACGATT` et `ATACTTACGTAACCATT`.

11.11.5 Moyenne de notes

Le fichier `notes.csv`[8] contient des noms d'étudiant ainsi que leurs notes dans différentes matières. Chaque donnée est séparée par une virgule. On trouve dans l'ordre le nom de l'étudiant, la note en géographie, la note en sport, la note en anglais.

```
Jason,17,3,1
William,9,18,15
Susan,3,8,10
[...]
```

6. https://python.sdv.u-paris.fr/data-files/UBI4_SCerevisiae.fasta
7. https://fr.wikipedia.org/wiki/Distance_de_Hamming
8. https://python.sdv.u-paris.fr/data-files/notes.csv

Créez un programme qui lit chaque ligne du fichier et construit une liste de dictionnaire du style `[{"nom": "Jason", "geo": 17, "sport": 3, "anglais": 1}, ...]`. Utilisez si possible la fonction `map()` pour convertir les nombres lus dans le fichier en entiers. Réalisez ensuite une boucle sur cette liste de dictionnaires, et affichez le nom de l'étudiant, sa note en sport et sa note en anglais. Affichez ensuite la moyenne des notes de sport et de géographie pour tous les étudiants.

11.11.6 Conversion des acides aminés du code à trois lettres au code à une lettre

Créez une fonction `convert_3_lettres_1_lettre()` qui prend en argument une chaîne de caractères avec des acides aminés en code à trois lettres et renvoie une chaîne de caractères avec les acides aminés en code à une lettre. Vous pourrez tenter d'utiliser le *method chaining* dans cette fonction.

Utilisez cette fonction pour convertir la séquence protéique `ALA GLY GLU ARG TRP TYR SER GLY ALA TRP`.

Rappel de la nomenclature des acides aminés :

Acide aminé	Code 3-lettres	Code 1-lettre	Acide aminé	Code 3-lettres	Code 1-lettre
Alanine	Ala	A	Leucine	Leu	L
Arginine	Arg	R	Lysine	Lys	K
Asparagine	Asn	N	Méthionine	Met	M
Aspartate	Asp	D	Phénylalanine	Phe	F
Cystéine	Cys	C	Proline	Pro	P
Glutamate	Glu	E	Sérine	Ser	S
Glutamine	Gln	Q	Thréonine	Thr	T
Glycine	Gly	G	Tryptophane	Trp	W
Histidine	His	H	Tyrosine	Tyr	Y
Isoleucine	Ile	I	Valine	Val	V

11.11.7 Palindrome

Un palindrome est un mot ou une phrase dont l'ordre des lettres reste le même si on le lit de gauche à droite ou de droite à gauche. Par exemple, « ressasser » et « engage le jeu que je le gagne » sont des palindromes.

Créez la fonction `est_palindrome()` qui prend en argument une chaîne de caractères et qui renvoie un booléen (`True` si l'argument est un palindrome, `False` si ce n'est pas le cas). Dans le programme principal, affichez `xxx est un palindrome` si la fonction `est_palindrome()` renvoie `True` sinon `xxx n'est pas un palindrome`. Pensez à vous débarrasser au préalable des majuscules, des signes de ponctuations et des espaces.

Testez ensuite si les expressions suivantes sont des palindromes :

- `Radar`
- `Never odd or even`
- `Karine alla en Iran`
- `Un roc si biscornu`
- `Et la marine ira vers Malte`
- `Deer Madam, Reed`
- `rotator`
- `Was it a car or a cat I saw?`

Conseil

Pour le nettoyage de la chaîne de caractères (retrait des majuscules, signes de ponctations et espaces), essayer d'utiliser le *method chaining*.

11.11.8 Mot composable

Un mot est composable à partir d'une séquence de lettres si la séquence contient toutes les lettres du mot. Chaque lettre de la séquence ne peut être utilisée qu'une seule fois. Par exemple, « coucou » est composable à partir de « uocuoceokzefhu ».

Créez la fonction `est_composable()`, qui prend en argument un mot (sous la forme d'une chaîne de caractères) et une séquence de lettres (aussi comme une chaîne de caractères), et qui renvoie `True` si le mot est composable à partir de la séquence, sinon `False`.

Dans le programme principal, créez une liste de tuples contenant les couples mot / séquence, de la forme `[('mot1', 'sequence1'), ('mot2', 'sequence2'), ...]`. Utilisez ensuite une boucle sur tous les couples mot / séquence, et appelez à chaque itération la fonction `est_composable()`. Affichez enfin `Le mot xxx est composable à partir de yyy` si le mot `xxx` est composable à partir de la séquence de lettres (`yyy`). Affichez `Le mot xxx n'est pas composable à partir de yyy` si ce n'est pas le cas.

Testez cette fonction avec les mots et les séquences suivantes :

Mot	Séquence
python	aophrtkny
python	aeiouyhpq
coucou	uocuoceokzezh
fonction	nhwfnitvkloco

11.11.9 Alphabet et pangramme

Les codes ASCII des lettres minuscules de l'alphabet vont de 97 (lettre « a ») à 122 (lettre « z »). La fonction `chr()` prend en argument un code ASCII sous la forme d'un entier et renvoie le caractère correspondant (sous la forme d'une chaîne de caractères). Ainsi `chr(97)` renvoie `'a'`, `chr(98)` renvoie `'b'` et ainsi de suite.

Créez la fonction `get_alphabet()` qui utilise une boucle et la fonction `chr()` et qui renvoie une chaîne de caractères contenant toutes les lettres de l'alphabet.

Un pangramme[9] est une phrase comportant au moins une fois chaque lettre de l'alphabet. Par exemple, « Portez ce vieux whisky au juge blond qui fume » est un pangramme.

Créez la fonction `est_pangramme()` qui utilise la fonction `get_alphabet()` précédente, qui prend en argument une chaîne de caractères `xxx`, et qui renvoie `True` si la phrase est un pangramme et `False` sinon.

Le programme affichera finalement `xxx est un pangramme` ou `xxx n'est pas un pangramme`. Pensez à vous débarrasser des majuscules le cas échéant.

Testez ensuite si les expressions suivantes sont des pangrammes :

- Portez ce vieux whisky au juge blond qui fume
- Monsieur Jack vous dactylographiez bien mieux que votre ami Wolf
- Buvez de ce whisky que le patron juge fameux
- Ceci n'est pas un pangramme

11.11.10 Lecture d'une séquence à partir d'un fichier GenBank (exercice +++)

On cherche à récupérer la séquence d'ADN du chromosome I de la levure *Saccharomyces cerevisiae* contenu dans le fichier au format GenBank `NC_001133.gbk`[10].

Le format GenBank est présenté en détail dans l'annexe A *Quelques formats de données en biologie*. Pour cet exercice, vous devez savoir que la séquence démarre après la ligne commençant par le mot `ORIGIN` et se termine avant la ligne commençant par les caractères `//` :

```
ORIGIN
        1 ccacaccaca cccacacacc cacacaccac accacacacc acaccacacc cacacacaca
       61 catcctaaca ctaccctaac acagccctaa tctaaccctg gccaacctgt ctctcaactt
[...]
   230101 tgttagtgtt agtattaggg tgtggtgtgt gggtgtggtg tgggtgtggg tgtgggtgtg
   230161 ggtgtgggtg tgggtgtggt gtggtgtgtg ggtgtggtgt gggtgtggtg tgtgtggg
//
```

Pour extraire la séquence d'ADN, nous vous proposons d'utiliser un algorithme de « drapeau », c'est-à-dire une variable qui sera à `True` lorsqu'on lira les lignes contenant la séquence et à `False` pour les autres lignes.

9. http://fr.wikipedia.org/wiki/Pangramme
10. https://python.sdv.u-paris.fr/data-files/NC_001133.gbk

Créez une fonction `lit_genbank()` qui prend comme argument le nom d'un fichier GenBank sous la forme d'une chaîne de caractères, lit la séquence dans le fichier GenBank et la renvoie sous la forme d'une chaîne de caractères.

Utilisez ensuite cette fonction pour récupérer la séquence d'ADN dans la variable `sequence` dans le programme principal. Le script affichera :

```
NC_001133.gbk
La séquence contient XXX bases
10 premières bases : YYYYYYYYYY
10 dernières bases : ZZZZZZZZZZ
```

où `XXX` est un entier et `YYYYYYYYYY` et `ZZZZZZZZZZ` sont des bases.

Vous avez toutes les informations pour effectuer cet exercice. Si toutefois vous coincez sur la mise en place du drapeau, voici l'algorithme en pseudo-code pour vous aider :

```
drapeau <- Faux
seq <- chaîne de caractères vide
Lire toutes les lignes du fichier:
    si la ligne contient //:
        drapeau <- Faux
    si drapeau est Vrai:
        on ajoute à seq la ligne (sans espace, chiffre et retour à la ligne)
    si la ligne contient ORIGIN:
        drapeau <- Vrai
```

11.11.11 Affichage des carbones alpha d'une structure de protéine

Téléchargez le fichier `1bta.pdb`[11] qui correspond à la structure tridimensionnelle de la protéine barstar[12] sur le site de la *Protein Data Bank* (PDB).

Créez la fonction `trouve_calpha()` qui prend en argument le nom d'un fichier PDB (sous la forme d'une chaîne de caractères), qui sélectionne uniquement les lignes contenant des carbones alpha, qui stocke ces lignes dans une liste et les renvoie sous la forme d'une liste de chaînes de caractères.

Utilisez la fonction `trouve_calpha()` pour afficher à l'écran les carbones alpha des deux premiers résidus (acides aminés).

Conseil

Vous trouverez des explications sur le format PDB et des exemples de code pour lire ce type de fichier en Python dans l'annexe A *Quelques formats de données en biologie*.

11.11.12 Calcul des distances entre les carbones alpha consécutifs d'une structure de protéine (exercice +++)

En utilisant la fonction `trouve_calpha()` précédente, calculez la distance interatomique entre les carbones alpha des deux premiers résidus (avec deux chiffres après la virgule).

Rappel : la distance euclidienne d entre deux points A et B de coordonnées cartésiennes respectives (x_A, y_A, z_A) et (x_B, y_B, z_B) se calcule comme suit :

$$d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2 + (z_B - z_A)^2}$$

Créez ensuite la fonction `calcule_distance()` qui prend en argument la liste renvoyée par la fonction `trouve_calpha()`, qui calcule les distances interatomiques entre carbones alpha consécutifs et affiche ces distances sous la forme :

`numero_calpha_1 numero_calpha_2 distance`

Les numéros des carbones alpha seront affichés sur deux caractères. La distance sera affichée avec deux chiffres après la virgule. Voici un exemple avec les premiers carbones alpha :

11. https://files.rcsb.org/download/1BTA.pdb
12. http://www.rcsb.org/pdb/explore.do?structureId=1BTA

```
1  2 3.80
2  3 3.80
3  4 3.83
4  5 3.82
```

Modifiez maintenant la fonction `calcule_distance()` pour qu'elle affiche à la fin la moyenne des distances.

La distance inter-carbone alpha dans les protéines est très stable et de l'ordre de 3,8 angströms. Observez avec attention les valeurs que vous avez calculées pour la protéine barstar. Repérez une valeur surprenante. Essayez de l'expliquer.

Conseil

Vous trouverez des explications sur le format PDB et des exemples de code pour lire ce type de fichier en Python dans l'annexe A *Quelques formats de données en biologie*.

11.11.13 Compteur de gènes dans un fichier GenBank

Dans cet exercice, on souhaite compter le nombre de gènes du fichier GenBank NC_001133.gbk[13] (chromosome I de la levure *Saccharomyces cerevisiae*) et afficher la longueur de chaque gène. Pour cela, il faudra récupérer les lignes décrivant la position des gènes. Voici par exemple les cinq premières lignes concernées dans le fichier NC_001133.gbk :

```
     gene            complement(<1807..>2169)
     gene            <2480..>2707
     gene            complement(<7235..>9016)
     gene            complement(<11565..>11951)
     gene            <12046..>12426
[...]
```

Lorsque la ligne contient le mot `complement` le gène est situé sur le brin complémentaire, sinon il est situé sur le brin direct. Votre code devra récupérer le premier et le second nombre indiquant respectivement la position du début et de fin du gène. Attention à bien les convertir en entier afin de pouvoir calculer la longueur du gène. Notez que les caractères `>` et `<` doivent être ignorés, et que les `..` servent à séparer la position de début et de fin.

On souhaite obtenir une sortie de la forme :

```
gène   1 complémentaire ->   362 bases
gène   2 direct         ->   227 bases
gène   3 complémentaire ->  1781 bases
[...]
gène  99 direct         ->   611 bases
gène 100 direct         ->   485 bases
gène 101 direct         ->  1403 bases
```

Conseil

Vous trouverez des explications sur le format GenBank dans l'annexe A *Quelques formats de données en biologie*.

13. https://python.sdv.u-paris.fr/data-files/NC_001133.gbk

CHAPITRE 12

Plus sur les listes

Nous avons vu les listes dès le chapitre 4 et les avons largement utilisées depuis le début de ce cours. Dans ce chapitre, nous allons plus loin avec les méthodes associées aux listes, ainsi que d'autres caractéristiques très puissantes telles que les tests d'appartenance ou les listes de compréhension.

12.1 Méthodes associées aux listes

Comme pour les chaînes de caractères, les listes possèdent de nombreuses **méthodes** qui leurs sont propres. On rappelle qu'une méthode est une fonction qui agit sur l'objet auquel elle est attachée par un point.

12.1.1 `.append()`

La méthode `.append()`, que l'on a déjà vu au chapitre 4 *Listes*, ajoute un élément à la fin d'une liste :

```
>>> liste1 = [1, 2, 3]
>>> liste1.append(5)
>>> liste1
[1, 2, 3, 5]
```

qui est équivalent à :

```
>>> liste1 = [1, 2, 3]
>>> liste1 = liste1 + [5]
>>> liste1
[1, 2, 3, 5]
```

Conseil

Préférez la version avec `.append()` qui est plus compacte et facile à lire.

12.1.2 `.insert()`

La méthode `.insert()` insère un objet dans une liste à un indice déterminé :

```
>>> liste1 = [1, 2, 3]
>>> liste1.insert(2, -15)
>>> liste1
[1, 2, -15, 3]
```

12.1.3 `del`

L'instruction `del` supprime un élément d'une liste à un indice déterminé :

```
>>> liste1 = [1, 2, 3]
>>> del liste1[1]
>>> liste1
[1, 3]
```

Remarque

Contrairement aux méthodes associées aux listes présentées dans cette rubrique, `del` est une instruction générale de Python, utilisable pour d'autres objets que des listes. Celle-ci ne prend pas de parenthèse.

12.1.4 `.remove()`

La méthode `.remove()` supprime un élément d'une liste à partir de sa valeur :

```
>>> liste1 = [1, 2, 3]
>>> liste1.remove(3)
>>> liste1
[1, 2]
```

S'il y a plusieurs fois la même valeur dans la liste, seule la première est retirée. Il faut appeler la méthode `.remove()` autant de fois que nécessaire pour retirer toutes les occurences d'un même élément :

```
>>> liste1 = [1, 2, 3, 4, 3]
>>> liste1.remove(3)
>>> liste1
[1, 2, 4, 3]
>>> liste1.remove(3)
>>> liste1
[1, 2, 4]
```

12.1.5 `.sort()`

La méthode `.sort()` trie les éléments d'une liste du plus petit au plus grand :

```
>>> liste1 = [3, 1, 2]
>>> liste1.sort()
>>> liste1
[1, 2, 3]
```

L'argument `reverse=True` spécifie le tri inverse, c'est-à-dire du plus grand au plus petit élément :

```
>>> liste1 = [3, 1, 2]
>>> liste1.sort(reverse=True)
>>> liste1
[3, 2, 1]
```

12.1.6 `sorted()`

La fonction `sorted()` trie également une liste. Contrairement à la méthode précédente `.sort()`, cette fonction renvoie la liste triée et ne modifie pas la liste initiale :

```
>>> liste1 = [3, 1, 2]
>>> sorted(liste1)
[1, 2, 3]
>>> liste1
[3, 1, 2]
```

La fonction `sorted()` supporte aussi l'argument `reverse=True` :

```
>>> liste1 = [3, 1, 2]
>>> sorted(liste1, reverse=True)
[3, 2, 1]
>>> liste1
[3, 1, 2]
```

12.1.7 `.reverse()`

La méthode `.reverse()` inverse une liste :

```
>>> liste1 = [3, 1, 2]
>>> liste1.reverse()
>>> liste1
[2, 1, 3]
```

12.1.8 `.count()`

La méthode `.count()` compte le nombre d'éléments (passés en argument) dans une liste :

```
>>> liste1 = [1, 2, 4, 3, 1, 1]
>>> liste1.count(1)
3
>>> liste1.count(4)
1
>>> liste1.count(23)
0
```

12.1.9 Particularités des méthodes associées aux listes

De nombreuses méthodes mentionnées précédemment (`.append()`, `.sort()`, etc.) modifient la liste, mais ne renvoient pas d'objet récupérable dans une variable. Il s'agit d'un exemple d'utilisation de méthode (donc de fonction particulière) qui fait une action, mais qui ne renvoie rien. Pensez-y dans vos utilisations futures des listes. Ainsi, même si l'instruction `var = liste1.reverse()` est une instruction Python valide, `var` ne contiendra que `None` c'est-à-dire un objet vide en Python, préférez-lui directement l'instruction `liste1.reverse()` :

```
>>> liste1 = [1, 2, 3]
>>> var = liste1.reverse()
>>> var
>>> print(var)
None
>>> liste1
[3, 2, 1]
>>> liste2 = [5, 6, 7]
>>> liste2.reverse()
>>> liste2
[7, 6, 5]
```

Remarque

Pour exprimer la même idée, la documentation parle de modification de la liste « sur place » (*in place* en anglais) :

```
>>> liste1 = [1, 2, 3]
>>> help(liste1.reverse)
Help on built-in function reverse:

reverse() method of builtins.list instance
    Reverse *IN PLACE*.
```

Cela signifie que la liste est modifiée « sur place », c'est-à-dire **dans la méthode** au moment où elle s'exécute. La liste étant modifiée « en dur » dans la méthode, cette dernière ne renvoie donc rien. L'explication du mécanisme sous-jacent vous sera donnée dans la rubrique 13.4 *Portée des listes* du chapitre 13 *Plus sur les fonctions*.

Par ailleurs, certaines méthodes ou instructions des listes décalent les indices d'une liste (par exemple `.insert()`,

`del`, etc.).

Enfin, pour obtenir une liste exhaustive des méthodes disponibles pour les listes, utilisez la fonction `dir(liste1)` (`liste1` étant une liste).

12.2 Construction d'une liste par itération

La méthode `.append()` est très pratique car on peut l'utiliser pour construire une liste au fur et à mesure des itérations d'une boucle.

Pour cela, il est commode de définir préalablement une liste vide avec l'instruction `liste1 = []`. Voici un exemple où une chaîne de caractères est convertie en liste :

```
>>> seq = "CAAAGGTAACGC"
>>> seq_list = []
>>> seq_list
[]
>>> for base in seq:
...   seq_list.append(base)
...
>>> seq_list
['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
```

Remarquez que dans cet exemple, vous pouvez aussi utiliser directement la fonction `list()` qui prend n'importe quel objet séquentiel (liste, chaîne de caractères, etc.) et qui renvoie une liste :

```
>>> seq = "CAAAGGTAACGC"
>>> list(seq)
['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
```

Cette méthode est certes plus simple, mais il arrive parfois qu'on doive utiliser des boucles tout de même, comme lorsqu'on lit un fichier. Nous vous rappellons que l'instruction `list(seq)` convertit un objet de type chaîne de caractères en un objet de type liste (il s'agit donc d'une opération de *casting*). De même que `list(range(10))` convertit un objet de type `range` en un objet de type `list`.

12.3 Test d'appartenance

L'opérateur `in` teste si un élément fait partie d'une liste :

```
liste1 = [1, 3, 5, 7, 9]
>>> 3 in liste1
True
>>> 4 in liste1
False
>>> 3 not in liste1
False
>>> 4 not in liste1
True
```

La variation avec `not` permet, *a contrario*, de vérifier qu'un élément n'est pas dans une liste.

12.4 Fonction `zip()`

Conseil

Si vous êtes débutant, vous pouvez sauter cette rubrique.

La fonction `zip()` de Python permet d'itérer sur plusieurs listes en parallèle :

```
>>> animaux = ["poulain", "renard", "python"]
>>> couleurs = ["alezan", "roux", "vert"]
>>> zip(animaux, couleurs)
<zip object at 0x7f6cf954a480>
>>> type(zip(animaux, couleurs))
<class 'zip'>
>>> for element in zip(animaux, couleurs):
...     print(element)
...
('poulain', 'alezan')
('renard', 'roux')
('python', 'vert')
>>> for animal, couleur in zip(animaux, couleurs):
...     print(f"le {animal} est {couleur}")
...
le poulain est alezan
le renard est roux
le python est vert
```

Lignes 3 et 6. On passe en argument deux listes à `zip()` qui génère un nouvel objet de type *zip*. Comme pour les objets de type *map* vu au chapitre 11 *Plus sur les chaînes de caractères*, les objets *zip* sont itérables.

Lignes 7 à 12. Lorsqu'on itère sur un objet *zip*, la variable d'itération est un tuple. À la première itération, on a un tuple avec le premier élément de chaque liste utilisée pour générer l'objet *zip*, à la deuxième itération, ce sera le deuxième élément, et ainsi de suite.

Lignes 13 à 18. Avec l'affectation multiple, on peut affecter à la volée les éléments à des variables différentes, comme on l'a fait avec la fonction `enumerate()` (chapitre 5 *Boucles*) et la méthode `.items()` des dictionnaires (chapitre 8 *Dictionnaires et tuples*).

Un objet *zip* est aussi utile pour générer facilement une liste de tuples.

```
>>> list(zip(animaux, couleurs))
[('poulain', 'alezan'), ('renard', 'roux'), ('python', 'vert')]
```

Si une des listes passée en argument n'a pas la même longueur, l'objet *zip* s'arrête sur la liste la plus courte :

```
>>> animaux = ["poulain", "renard", "python", "orque"]
>>> couleurs = ["alezan", "roux", "vert"]
>>> list(zip(animaux, couleurs))
[('poulain', 'alezan'), ('renard', 'roux'), ('python', 'vert')]
```

On peut empêcher ce comportement avec l'argument par mot-clé `strict`, qui renvoie une erreur si les listes n'ont pas la même longueur :

```
>>> list(zip(animaux, couleurs, strict=True))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: zip() argument 2 is shorter than argument 1
```

Enfin, il est possible de créer des objets *zip* avec autant de listes que l'on veut :

```
>>> animaux = ["poulain", "renard", "python"]
>>> couleurs = ["alezan", "roux", "vert"]
>>> numero = [1, 2, 3]
>>> list(zip(numero, animaux, couleurs))
[(1, 'poulain', 'alezan'), (2, 'renard', 'roux'), (3, 'python', 'vert')]
```

Remarque

La fonction `zip()` fonctionne sur n'importe quel objet itérable : listes, tuples, dictionnaires, objets *range*, etc.

Conseil

Pour les débutants, vous pouvez sauter cette remarque.

Un objet `zip()` comme présenté plus haut est ce qu'on appelle un itérateur. Cela implique un mode de fonctionnement particulier, notamment le fait qu'on ne peut l'utiliser qu'une fois lorsqu'on l'a créé. Vous trouverez plus d'explications sur la définition et le fonctionnement d'un itérateur dans le chapitre 26 *Remarques complémentaires*.

12.5 Copie de listes

Il est très important de savoir que l'affectation d'une liste (à partir d'une liste préexistante) crée en réalité une **référence** et non une **copie** :

```
>>> liste1 = [1, 2, 3]
>>> liste2 = liste1
>>> liste2
[1, 2, 3]
>>> liste1[1] = -15
>>> liste1
[1, -15, 3]
>>> liste2
[1, -15, 3]
```

Vous voyez que la modification de `liste1` modifie `liste2` aussi ! Pour comprendre ce qu'il se passe, nous allons de nouveau utiliser le site *Python Tutor* avec cet exemple (Figure 12.1) :

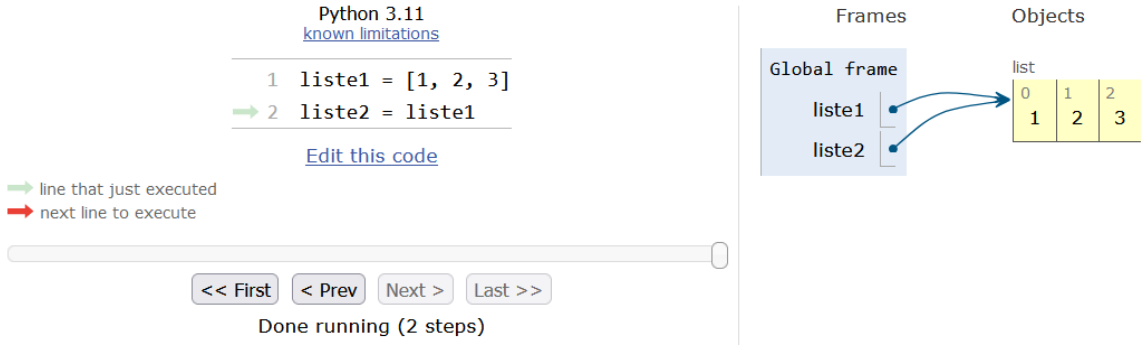


FIGURE 12.1 – Copie de liste.

Techniquement, Python utilise des pointeurs (comme dans le langage de programmation C) vers les mêmes objets. *Python Tutor* l'illustre avec des flèches qui partent des variables `liste1` et `liste2` et qui pointent vers la même liste. Donc, si on modifie la liste `liste1`, la liste `liste2` est modifiée de la même manière. Rappelez-vous de ceci dans vos futurs programmes, car cela pourrait avoir des effets désastreux !

Pour éviter ce problème, il va falloir créer une copie explicite de la liste initiale. Observez cet exemple :

```
>>> liste1 = [1, 2, 3]
>>> liste2 = liste1[:]
>>> liste1[1] = -15
>>> liste2
[1, 2, 3]
```

L'instruction `liste1[:]` a créé une copie « à la volée » de la liste `liste1`. Vous pouvez utiliser aussi la fonction `list()`, qui renvoie explicitement une liste :

```
>>> liste1 = [1, 2, 3]
>>> liste2 = list(liste1)
>>> liste1[1] = -15
>>> liste2
[1, 2, 3]
```

Si on regarde à nouveau dans *Python Tutor* (Figure 12.2), on voit clairement que l'utilisation d'une tranche `[:]` ou de la fonction `list()` crée des copies explicites. Chaque flèche pointe vers une liste différente, indépendante des autres.

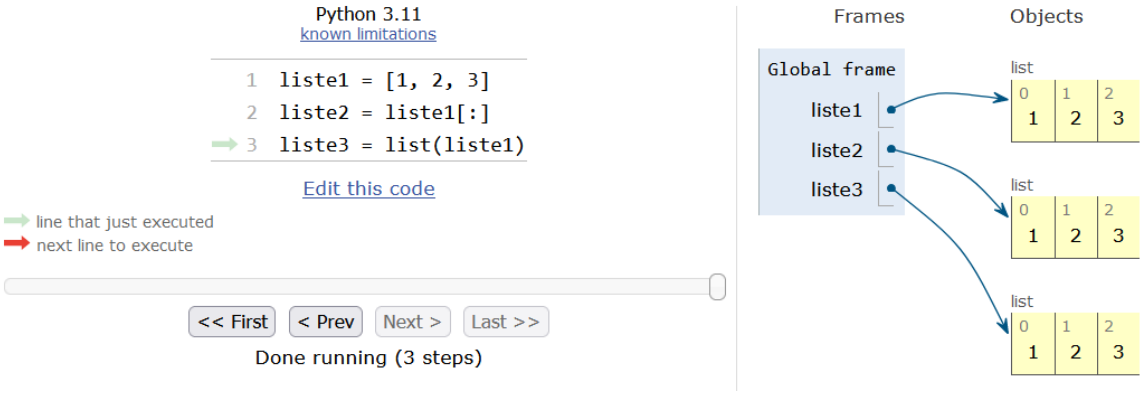


FIGURE 12.2 – Copie de liste avec une tranche `[:]` et la fonction `list()`.

Attention, les deux astuces précédentes ne fonctionnent que pour les listes à une dimension, autrement dit les listes qui ne contiennent pas elles-mêmes d'autres listes. Voyez par exemple :

```
>>> liste1 = [[1, 2], [3, 4]]
>>> liste1
[[1, 2], [3, 4]]
>>> liste2 = liste1[:]
>>> liste1[1][1] = 55
>>> liste1
[[1, 2], [3, 55]]
>>> liste2
[[1, 2], [3, 55]]
```

et

```
>>> liste2 = list(liste1)
>>> liste1[1][1] = 77
>>> liste1
[[1, 2], [3, 77]]
>>> liste2
[[1, 2], [3, 77]]
```

La méthode de copie qui **fonctionne à tous les coups** consiste à appeler la fonction `deepcopy()` du module *copy* :

```
>>> import copy
>>> liste1 = [[1, 2], [3, 4]]
>>> liste1
[[1, 2], [3, 4]]
>>> liste2 = copy.deepcopy(liste1)
>>> liste1[1][1] = 99
>>> liste1
[[1, 2], [3, 99]]
>>> liste2
[[1, 2], [3, 4]]
```

12.6 Initialisation d'une liste de listes

Un dernier écueil que vous pourrez rencontrer concerne l'initialisation d'une liste de listes avec l'opérateur `*`. Imaginons que l'on souhaite représenter un tableau de nombre et l'initialiser avec des 0. Nous pourrions être tentés d'utiliser la duplication de listes :

```
>>> liste1 = [[0, 0, 0]] * 5
>>> liste1
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Le problème est que si on modifie un élément d'une des sous-listes :

```
>>> liste1[2][0] = -12
>>> liste1
[[-12, 0, 0], [-12, 0, 0], [-12, 0, 0], [-12, 0, 0], [-12, 0, 0]]
```

Vous constatez qu'il est modifié dans chaque sous-liste ! À l'aide de *Python Tutor* on voit que Python crée une référence vers la même sous-liste (Figure 12.3) :

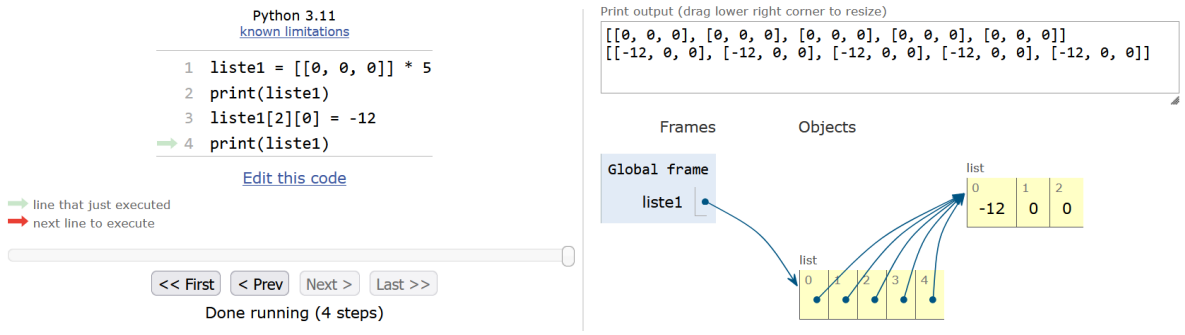


FIGURE 12.3 – Initialisation d'une liste de listes avec l'opérateur de duplication.

Comme disent les auteurs dans la documentation officielle[1] : *Note that items in the sequence are not copied ; they are referenced multiple times. This often haunts new Python programmers*. Pour éviter le problème, on peut utiliser une boucle :

```
>>> liste1 = []
>>> for i in range(5):
...     liste1.append([0, 0, 0])
...
>>> liste1
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> liste1[2][0] = -12
>>> liste1
[[0, 0, 0], [0, 0, 0], [-12, 0, 0], [0, 0, 0], [0, 0, 0]]
```

On verra dans la rubrique suivante une manière très compacte de faire cela avec les listes de compréhension.

Attention

Même si une liste de listes peut représenter un tableau de nombres, il ne faut pas la voir comme un objet mathématique de type matrice[2]. En effet, le concept de lignes et colonnes n'est pas défini clairement, on ne peut pas faire d'opérations matricielles simplement, etc. On verra dans le chapitre 20 *Module Numpy* qu'il existe des objets appelés *arrays* qui sont faits pour ça.

12.7 Liste de compréhension

Conseil

Si vous êtes débutant, vous pouvez sauter cette rubrique.

En Python, la notion de liste de compréhension (ou compréhension de listes) représente une manière originale et très puissante de générer des listes. La syntaxe de base consiste au moins en une boucle `for` au sein de crochets précédés d'une variable (qui peut être la variable d'itération ou pas) :

1. https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range
2. https://fr.wikipedia.org/wiki/Matrice

```
>>> [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [2 for i in range(10)]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

Pour plus de détails, consultez à ce sujet le site de Python [3] et celui de Wikipédia [4].

Voici quelques exemples illustrant la puissance des listes de compréhension.

12.7.1 Initialisation d'une liste de listes

Une liste de compréhension permet l'initialisation d'une liste de listes en une ligne sans avoir l'inconvénient de faire une référence vers la même sous-liste (voir rubrique précédente) :

```
>>> liste1 = [[0, 0, 0] for i in range(5)]
>>> liste1
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> liste1[2][0] = -12
>>> liste1
[[0, 0, 0], [0, 0, 0], [-12, 0, 0], [0, 0, 0], [0, 0, 0]]
```

12.7.2 Nombres pairs compris entre 0 et 30

```
>>> print([i for i in range(31) if i % 2 == 0])
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
```

12.7.3 Jeu sur la casse des mots d'une phrase

```
>>> message = "C'est sympa la BioInfo"
>>> msg_lst = message.split()
>>> print([[m.upper(), len(m)] for m in msg_lst])
[["C'EST", 5], ['SYMPA', 5], ['LA', 2], ['BIOINFO', 7]]
```

12.7.4 Formatage d'une séquence avec 60 caractères par ligne

Exemple d'une séquence constituée de 150 alanines :

```
# Exemple d'une séquence de 150 alanines.
>>> seq = "A" * 150
>>> width = 60
>>> seq_split = [seq[i:i+width] for i in range(0, len(seq), width)]
>>> print("\n".join(seq_split))
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA
```

12.7.5 Formatage FASTA d'une séquence

Exemple d'une séquence constituée de 150 alanines :

```
>>> com = "Séquence de 150 alanines"
>>> seq = "A" * 150
>>> width = 60
>>> seq_split = [seq[i:i+width] for i in range(0, len(seq), width)]
>>> print(">"+com+"\n"+"\n".join(seq_split))
>séquence de 150 alanines
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAA
```

3. http://www.python.org/dev/peps/pep-0202/
4. http://fr.wikipedia.org/wiki/Comprehension_de_liste

12.7.6 Sélection des carbones alpha dans un fichier PDB

Exemple avec la structure de la barstar[5] :

```
>>> with open("1bta.pdb", "r") as f_pdb:
...     CA_lines = [
...         line for line in f_pdb
...         if line.startswith("ATOM") and line[12:16].strip() == "CA"
...     ]
...
>>> print(len(CA_lines))
89
```

Conseil

Pour plus de lisiblité, il est possible de répartir la liste de compréhension sur plusieurs lignes.

12.7.7 Portée des variables dans une liste de compréhension

Contrairement à une boucle `for`, la variable d'itération d'une liste de compréhension n'est pas accessible en dehors de la liste de compréhension elle-même. Par exemple :

```
>>> liste_a = []
>>> for idx_a in range(10):
...     liste_a.append(idx_a)
...
>>> print(liste_a)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print(idx_a)
9
>>>
>>> liste_b = [idx_b for idx_b in range(10)]
>>> print(liste_b)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print(idx_b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'idx_b' is not defined. Did you mean: 'idx_a'?
```

La variable d'itération `idx_a` reste disponible en dehors de la boucle `for`. Par contre, la variable d'itération `idx_b` n'est pas disponible en dehors de la liste de compréhension, car elle est créée « à la volée » par Python puis éliminée une fois l'instruction exécutée.

12.8 Tris puissants de listes

Conseil

Si vous êtes débutant, vous pouvez sauter cette rubrique.

Un peu plus haut nous avons évoqué la méthode `.sort()` qui trie une liste sur place, ainsi que la fonction `sorted()` qui renvoie une nouvelle liste triée. Nous avons également vu qu'elles supportaient l'argument par mot-clé `reverse` pour trier dans le sens inverse (décroissant ou anti-ASCII). Il existe un autre argument par mot-clé nommé `key` permettant un tri avec des règles alternatives que nous pouvons customiser. On doit passer à `key` une fonction *callback* (nous avions déjà croisé cette notion avec la fonction `map()` dans le chapitre 11 *Plus sur les chaînes de caractères*, pour une définition voir le chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne)), c'est-à-dire, un nom de fonction sans les parenthèses. Par exemple, si on passe la *callback* `len` comme ça :

5. http://www.rcsb.org/pdb/explore.do?structureId=1BTA

```
>>> mots = ["babar", "bar", "ba", "bababar"]
>>> sorted(mots, key=len)
['ba', 'bar', 'babar', 'bababar']
```

Python trie la liste `mots` en considérant la longeur de chaque élément, donc ici le nombre de lettres de chaque chaîne de caractères. Si plusieurs mots ont la même longueur (`bar` et `bam` dans l'exemple suivant), `sorted()` les laisse dans l'ordre de la liste initiale.

```
>>> mots = ["bar", "babar", "bam", "ba", "bababar"]
>>> sorted(mots, key=len)
['ba', 'bar', 'bam', 'babar', 'bababar']
```

Là où `key` va se révéler puissant est quand nous allons lui passer une fonction « maison ». Voici une exemple :

```
1 >>> def compte_b(chaine):
2 ...     return chaine.count("b")
3 ...
4 >>> compte_b("babar")
5 2
6 >>> mots = ["bar", "babar", "bam", "ba", "bababar"]
7 >>> sorted(mots, key=compte_b)
8 ['bar', 'bam', 'ba', 'babar', 'bababar']
```

- **Lignes 1 à 5.** Comme son nom l'indique, la fonction `compte_b()` compte les lettres `b` dans une chaîne de caractères.
- **Lignes 7 et 8.** En donnant `compte_b` (notez l'absence de parenthèses) à l'argument `key`, Python trie en fonction du nombre de lettres `b` dans chaque mot ! Comme pour `len`, si plusieurs mots ont un nombre de lettres `b` identiques, il conserve l'ordre de la liste initiale.

Remarque

L'argument `key` fonctionne de la même manière entre `sorted()` et la méthode `.sort()` qui trie sur place. Cet argument existe aussi avec les fonctions `min()` et `max()`. Par exemple :

```
>>> mots = ["bar", "babar", "bam", "ba", "bababar"]
>>> min(mots, key=len)
'ba'
>>> max(mots, key=len)
'bababar'
```

Python renverra le premier élément avec `min()` ou le dernier élément avec `max()` après un tri sur la longueur de chaque mot.

Pour aller plus loin

En Python, trier avec une fonction maison passée à l'argument `key` se fait plutôt avec ce qu'on appelle une **fonction lambda**. Il s'agit d'une « petite » fonction que l'on écrit sur une ligne. Si vous voulez en savoir plus, vous pouvez consulter le chapitre 26 *Remarques complémentaires*.

12.9 Exercices

Conseil

Pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

12.9.1 Tri de liste

Soit la liste de nombres `[8, 3, 12.5, 45, 25.5, 52, 1]`. Triez les nombres de cette liste par ordre croissant, sans utiliser la fonction `sort()`. Les fonctions et méthodes `min()`, `.append()` et `.remove()` vous seront utiles.

12.9.2 Séquence d'ADN aléatoire

Créez une fonction `seq_alea()` qui prend comme argument un entier positif `taille` représentant le nombre de bases de la séquence et qui renvoie une séquence d'ADN aléatoire sous forme d'une chaîne de caractères. Utilisez la fonction `random.choices()` présentée dans le chapitre 9 *Modules*.

Utilisez la fonction `seq_alea()` pour générer aléatoirement une séquence d'ADN de 15 bases.

12.9.3 Séquence d'ADN complémentaire inverse

Créez une fonction `gen_comp_inv()` qui prend comme argument une séquence d'ADN sous la forme d'une chaîne de caractères, qui renvoie la séquence complémentaire inverse sous la forme d'une autre chaîne de caractères et qui utilise des méthodes associées aux listes. Dans cette fonction, utilisez un dictionnaire `{"A": "T", "T": "A", "G": "C", "C": "G"}` donnant la correspondance entre nucléotides des brins direct et complémentaire.

Utilisez cette fonction pour transformer la séquence d'ADN `TCTGTTAACCATCCACTTCG` en sa séquence complémentaire inverse.

Rappel : la séquence complémentaire inverse doit être « inversée ». Par exemple, la séquence complémentaire inverse de la séquence `ATCG` est `CGAT`.

12.9.4 Doublons

Soit la liste de nombres `liste1 = [5, 1, 1, 2, 5, 6, 3, 4, 4, 4, 2]`. À partir de `liste1`, créez une nouvelle liste sans les doublons, triez-la et affichez-la.

12.9.5 Séquence d'ADN aléatoire 2

Créez une fonction `seq_alea_2()` qui prend comme argument un entier et quatre *floats*, représentant respectivement la longueur de la séquence et les pourcentages de chacune des quatre bases A, T, G et C. La fonction générera aléatoirement une séquence d'ADN qui prend en compte les contraintes fournies en arguments et renverra la séquence sous forme d'une chaîne de caractères.

Utilisez cette fonction pour générer aléatoirement une séquence d'ADN de 50 bases contenant 10 % de A, 30 % de T, 50 % de G et 10 % de C.

Conseil

Utilisez la fonction `random.choises()` avec les paramètres `k` et `weights`. Le paramètre `k` spécifie le nombre de tirages aléatoires à réaliser et le paramètre `weights` indique les probabilités de tirage.

Par exemple, pour réaliser 10 tirages aléatoires entre les lettres `A` et `B` avec 80% de `A` et 20% de `B`, on utilise la fonction `random.choices()` de la manière suivante :

```
>>> import random
>>> random.choices("AB", k=10, weights=[80, 20])
['A', 'A', 'A', 'A', 'A', 'A', 'A', 'B', 'A', 'B']
```

N'hésitez pas à consulter la documentation[6] de la fonction `random.choices()` pour plus de détails.

12.9.6 Le nombre mystère

Trouvez le nombre mystère qui répond aux conditions suivantes :

- *Il est composé de trois chiffres.*
- *Il est strictement inférieur à 300.*
- *Il est pair.*
- *Deux de ses chiffres sont identiques.*

6. https://docs.python.org/fr/3/library/random.html#random.choices

- *La somme de ses chiffres est égale à 7.*

On vous propose d'employer une méthode dite « *brute force* », c'est-à-dire d'utiliser une boucle et à chaque itération de tester les différentes conditions.

12.9.7 Codes une et trois lettres des acides aminés

On donne les deux listes suivantes décrivant quelques acides aminés en code une et trois lettres :

```
code_1_lettre = ["A", "V", "L", "M", "P"]
code_3_lettres = ["Ala", "Val", "Leu", "Met", "Pro"]
```

Avec la fonction `zip()` et une boucle, générez la sortie suivante :

```
L'acide aminé se note A ou Ala
L'acide aminé se note V ou Val
L'acide aminé se note L ou Leu
L'acide aminé se note M ou Met
L'acide aminé se note P ou Pro
```

12.9.8 Triangle de Pascal (exercice +++)

Voici le début du triangle de Pascal :

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
[...]
```

Déduisez comment une ligne est construite à partir de la précédente. Par exemple, à partir de la ligne 2 (`1 1`), construisez la ligne suivante (ligne 3 : `1 2 1`) et ainsi de suite.

Implémentez cette construction en Python. Généralisez à l'aide d'une boucle.

Écrivez dans un fichier `pascal.out` les 10 premières lignes du triangle de Pascal.

CHAPITRE 13

Plus sur les fonctions

Avant d'aborder ce chapitre, nous vous conseillons de relire le chapitre 10 *Fonctions* et de bien en assimiler toutes les notions (et aussi d'en faire les exercices). Nous avons vu dans ce chapitre 10 le concept incontournable que représentent les **fonctions**. Nous avons également introduit la notion de variables **locales** et **globales**.

Dans ce chapitre, nous allons aller un peu plus loin sur la visibilité de ces variables dans et hors des fonctions, et aussi voir ce qui se passe lorsque ces variables sont des listes. Attention, la plupart des lignes de code ci-dessous sont données à titre d'exemple pour bien comprendre ce qui se passe, mais nombre d'entre elles sont des aberrations en terme de programmation. Nous ferons un récapitulatif des bonnes pratiques à la fin du chapitre. Enfin, nous vous conseillons de tester tous les exemples ci-dessous avec le site *Python Tutor*[1] afin de suivre l'état des variables lors de l'exécution des exemples.

13.1 Appel d'une fonction dans une fonction

Dans le chapitre 10, nous avons vu des fonctions qui étaient appelées depuis le programme principal. Il est en fait possible d'appeler une fonction depuis une autre fonction. Et plus généralement, on peut appeler une fonction de n'importe où à partir du moment où elle est visible par Python (c'est-à-dire chargée dans la mémoire). Observez cet exemple :

```
1  # Définition des fonctions.
2  def est_pair(x):
3      if x % 2 == 0:
4          return True
5      else:
6          return False
7
8  def calc_somme_nb_pairs(debut, fin):
9      somme = 0
10     for nombre in range(debut, fin+1):
11         if est_pair(nombre):
12             somme += nombre
13
14 # Programme principal.
15 somme = calc_somme_nb_pairs(1, 5)
16 print(f"La somme des nombres pairs de 1 à 5 est {somme}")
```

Nous appelons la fonction `calc_somme_nb_pairs()` depuis le programme principal, puis à l'intérieur de celle-ci nous

1. http://www.pythontutor.com/

appelons l'autre fonction `est_pair()`. Regardons ce que *Python Tutor* nous montre lorsque la fonction `calc_somme_nb_pairs()` est exécutée dans la Figure 13.1.

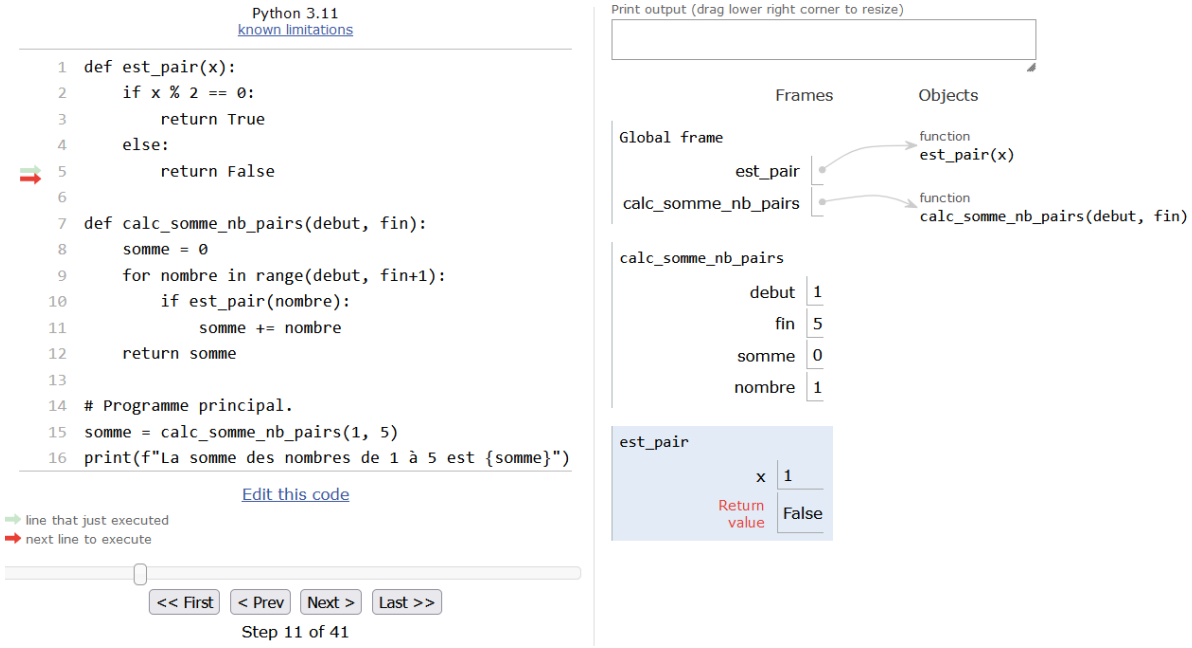


FIGURE 13.1 – Appel d'une fonction dans une fonction.

L'espace mémoire alloué à `est_pair()` est grisé, indiquant que cette fonction est en cours d'exécution. La fonction appelante `calc_somme_nb_pairs()` est toujours là (sur un fond blanc) car son exécution n'est pas terminée. Elle est en quelque sorte *figée* dans le même état qu'avant l'appel de `est_pair()`, et on pourra ainsi noter que ses variables *locales* (`debut`, `fin`) sont toujours là. De manière générale, les variables *locales* d'une fonction ne seront détruites que lorsque l'exécution de celle-ci sera terminée. Dans notre exemple, les variables *locales* de `calc_somme_nb_pairs()` ne seront détruites que lorsque la boucle sera terminée et que la variable `somme` sera retournée au programme principal. Enfin, notez bien que la fonction `calc_somme_nb_pairs()` appelle la fonction `est_pair()` à chaque itération de la boucle.

Ainsi, le programmeur est libre de faire tous les appels qu'il souhaite. Une fonction peut appeler une autre fonction, cette dernière peut appeler une autre fonction et ainsi de suite (et autant de fois qu'on le veut). Une fonction peut même s'appeler elle-même, cela s'appelle une fonction *récursive* (voir la rubrique suivante). Attention toutefois à retrouver vos petits si vous vous perdez dans les appels successifs !

Conseil

Dans la fonction `est_pair()` on teste si le nombre est pair et on renvoie `True`, sinon on renvoie `False`. Cette fonction pourrait être écrite de manière plus compacte :

```
1 def est_pair(x):
2     return x % 2
```

Comme l'expression `x % 2` renvoie un booléen directement, elle revient au même que le `if` / `else` ci-dessus. C'est bien sûr cette dernière notation plus compacte que nous vous recommandons.

13.2 Fonctions récursives

Conseil

Si vous êtes débutant, vous pouvez sauter cette rubrique.

Une fonction récursive est une fonction qui s'appelle elle-même. Les fonctions récursives permettent d'obtenir une efficacité redoutable dans la résolution de certains algorithmes, comme le tri rapide[2] (en anglais, *quicksort*).

Oublions la recherche d'efficacité pour l'instant et concentrons-nous sur l'exemple de la fonction mathématique factorielle. Nous vous rappelons que la factorielle s'écrit avec un `!` et se définit de la manière suivante :

$$3! = 3 \times 2 \times 1 = 6$$
$$4! = 4 \times 3 \times 2 \times 1 = 30$$
$$\dots$$
$$n! = n \times n-1 \times \dots \times 2 \times 1$$

Voici le code Python avec une fonction récursive :

```
1  def calc_factorielle(nb):
2      if nb == 1:
3          return 1
4      else:
5          return nb * calc_factorielle(nb - 1)
6
7  # Programme principal.
8  print(calc_factorielle(4))
```

Pas si facile à comprendre, n'est-ce pas ? À nouveau, aidons nous de *Python Tutor* pour visualiser ce qui se passe dans la figure 13.2 (nous vous conseillons bien sûr de tester vous-même cet exemple) :

FIGURE 13.2 – Fonction récursive : factorielle.

Ligne 8, on appelle la fonction `calc_factorielle()` en passant comme argument l'entier 4. Dans la fonction, la variable locale qui récupère cet argument est `nb`. Au sein de la fonction, celle-ci se rappelle elle-même (*ligne 5*), mais cette fois-ci en passant la valeur 3. Au prochain appel, ce sera avec la valeur 2, puis finalement 1. Dans ce dernier cas, le test

2. https://fr.wikipedia.org/wiki/Tri_rapide

`if nb == 1:` est vrai et l'instruction `return 1` sera exécutée. À ce moment précis de l'exécution, les appels successifs forment une sorte de *pile* (voir la figure 13.2). La valeur 1 sera ainsi renvoyée au niveau de l'appel précédent, puis le résultat $2 \times 1 = 2$ (où 2 correspond à `nb` et 1 provient de `calc_factorielle(nb - 1)`, soit 1) va être renvoyé à l'appel précédent, puis $3 \times 2 = 6$ (où 3 correspond à `nb` et 2 provient de `calc_factorielle(nb - 1)`, soit 2) va être renvoyé à l'appel précédent, pour finir par $4 \times 6 = 24$ (où 4 correspond à `nb` et 6 provient de `calc_factorielle(nb - 1)`, soit 6), soit la valeur de 4!. Les appels successifs vont donc se « dépiler » et nous reviendrons dans le programme principal.

Même si les fonctions récursives peuvent être ardues à comprendre, notre propos est ici de vous illustrer qu'une fonction qui en appelle une autre (ici il s'agit d'elle-même) reste « figée » dans le même état, jusqu'à ce que la fonction appelée lui renvoie une valeur.

13.3 Portée des variables

Il est très important lorsque l'on manipule des fonctions de connaître la portée des variables (*scope* en anglais), c'est-à-dire savoir là où elles sont visibles. On a vu que les variables créées au sein d'une fonction ne sont pas visibles à l'extérieur de celle-ci car elles étaient **locales** à la fonction. Observez le code suivant :

```
>>> def ma_fonction():
...     x = 2
...     print(f"x vaut {x} dans la fonction")
...
>>> ma_fonction()
x vaut 2 dans la fonction
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Lorsque Python exécute le code de la fonction, il connaît le contenu de la variable `x`. Par contre, de retour dans le module principal (dans ce cas, il s'agit de l'interpréteur Python), il ne la connaît plus, d'où le message d'erreur.

De même, une variable passée en argument est considérée comme **locale** lorsqu'on arrive dans la fonction :

```
>>> def ma_fonction(x):
...     print(f"x vaut {x} dans la fonction")
...
>>> ma_fonction(2)
x vaut 2 dans la fonction
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

Lorsqu'une variable est déclarée dans le programme principal, elle est visible dans celui-ci ainsi que dans toutes les fonctions. On a vu qu'on parlait de variable **globale** :

```
>>> def ma_fonction():
...     print(x)
...
>>> x = 3
>>> ma_fonction()
3
>>> print(x)
3
```

Dans ce cas, la variable `x` est visible dans le module principal et dans toutes les fonctions du module. Toutefois, Python ne permet pas la modification d'une variable globale dans une fonction :

```
>>> def ma_fonction():
...     x = x + 1
...
>>> x = 1
>>> ma_fonction()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in ma_fonction
UnboundLocalError: cannot access local variable 'x' where it is not associated with a value
```

L'erreur renvoyée montre que Python pense que `x` est une variable locale qui n'a pas été encore assignée. Si on veut vraiment modifier une variable globale dans une fonction, il faut utiliser le mot-clé `global` :

```
>>> def ma_fonction():
...     global x
...     x = x + 1
...
>>> x = 1
>>> ma_fonction()
>>> x
2
```

Dans ce dernier cas, le mot-clé `global` a forcé la variable `x` à être globale plutôt que locale au sein de la fonction.

13.4 Portée des listes

Attention

Les exemples de cette partie représentent des absurdités en termes de programmation. Ils sont donnés à titre indicatif pour comprendre ce qui se passe, mais il ne faut surtout pas s'en inspirer !

Soyez extrêmement attentifs avec les types modifiables (tels que les listes) car vous pouvez les changer au sein d'une fonction :

```
>>> def ma_fonction():
...     liste1[1] = -127
...
>>> liste1 = [1,2,3]
>>> ma_fonction()
>>> liste1
[1, -127, 3]
```

De même, si vous passez une liste en argument, elle est modifiable au sein de la fonction :

```
>>> def ma_fonction(liste_tmp):
...     liste_tmp[1] = -15
...
>>> liste1 = [1,2,3]
>>> ma_fonction(liste1)
>>> liste1
[1, -15, 3]
```

Pour bien comprendre l'origine de ce comportement, utilisons à nouveau le site *Python Tutor*[3]. La figure 13.3 vous montre le mécanisme à l'oeuvre lorsqu'on passe une liste à une fonction.

L'instruction `pass` dans la fonction est une instruction Python qui ne fait rien. Elle est là car une fonction ne peut être vide et doit contenir au moins une instruction Python valide.

On voit très clairement que la variable `liste1` passée en argument lors de l'appel de la fonction d'une part, et la variable locale `liste_tmp` au sein de la fonction d'autre part, **pointent vers le même objet dans la mémoire**. Ainsi, si on modifie `liste_tmp`, on modifie aussi `liste1`. C'est exactement le même mécanisme que pour la copie de listes (cf. rubrique 11.4 *Copie de listes* du chapitre 12 *Plus sur les listes*).

Si vous voulez éviter les problèmes de modification malencontreuse d'une liste dans une fonction, utilisez des tuples (ils ont présentés dans le chapitre 8 *Dictionnaires et tuples*), Python renverra une erreur car ces derniers sont non modifiables.

Une autre solution pour éviter la modification d'une liste, lorsqu'elle est passée comme argument à une fonction, est de la passer explicitement (comme nous l'avons fait pour la copie de liste) afin qu'elle reste intacte dans le programme principal :

3. http://www.pythontutor.com/

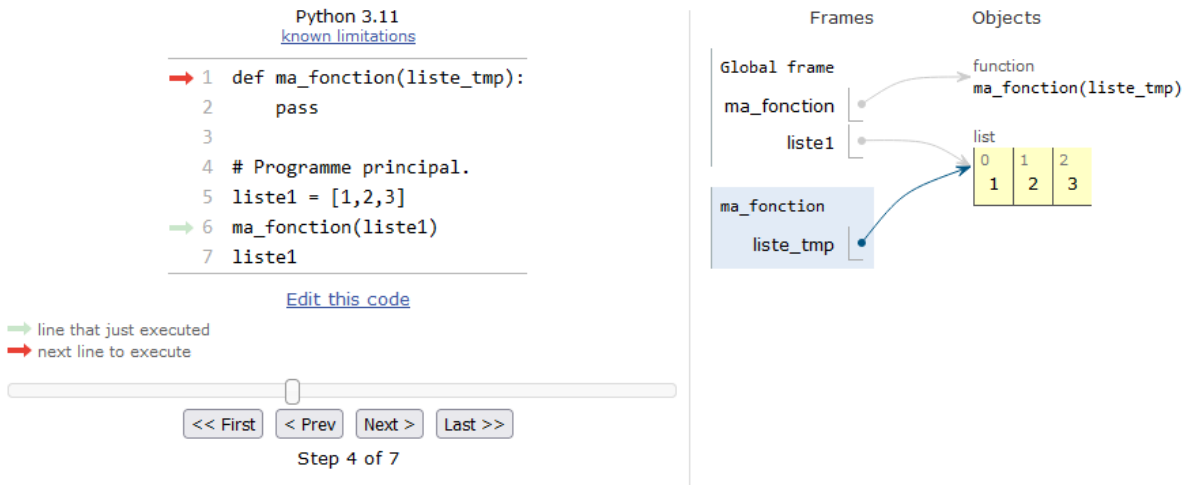


FIGURE 13.3 – Passage d'une liste à une fonction.

```
>>> def ma_fonction(liste_tmp):
...     liste_tmp[1] = -15
...
>>> liste1 = [1, 2, 3]
>>> ma_fonction(liste1[:])
>>> liste1
[1, 2, 3]
>>> ma_fonction(liste1(y))
>>> liste1
[1, 2, 3]
```

Dans ces deux derniers exemples, une copie de `y` est créée à la volée lorsqu'on appelle la fonction, ainsi la liste `y` du module principal reste intacte.

D'autres suggestions sur l'envoi de liste dans une fonction vous sont données dans la rubrique *Recommandations* ci-dessous.

13.5 Règle LGI

Lorsque Python rencontre une variable, il va traiter la résolution de son nom avec des priorités particulières. D'abord il va regarder si la variable est **locale**, puis si elle n'existe pas localement, il vérifiera si elle est **globale** et enfin si elle n'est pas globale, il testera si elle est **interne** (par exemple la fonction `len()` est considérée comme une fonction interne à Python, elle existe à chaque fois que vous lancez Python). On appelle cela la règle **LGI** pour locale, globale, interne. En voici un exemple :

```
>>> def ma_fonction():
...     x = 4
...     print(f"Dans la fonction x vaut {x}")
...
>>> x = -15
>>> ma_fonction()
Dans la fonction x vaut  4
>>> print(f"Dans le module principal x vaut {x}")
Dans le module principal x vaut -15
```

Dans la fonction, `x` a pris la valeur qui lui était définie localement en priorité sur la valeur définie dans le module principal.

Conseil

Même si Python accepte qu'une variable ait le même nom que ses propres fonctions ou variables internes, évitez

d'utiliser de tels noms, car ceci rendra votre code confus !

De manière générale, la règle LGI découle de la manière dont Python gère ce que l'on appelle « les espaces de noms ». C'est cette gestion qui définit la portée (visibilité) de chaque variable. Nous en parlerons plus longuement dans le chapitre 24 *Avoir plus la classe avec les objets* (en ligne).

13.6 Recommandations

13.6.1 Évitez les variables globales

Dans ce chapitre nous avons *joué* avec les fonctions (et les listes) afin de vous montrer comment Python réagissait. Toutefois, notez bien que **l'utilisation de variables globales est à bannir définitivement de votre pratique de la programmation**.

Parfois on veut faire vite et on crée une variable globale visible partout dans le programme (donc dans toutes les fonctions), car « *Ça va plus vite, c'est plus simple* ». C'est un très mauvais calcul, ne serait-ce que parce que vos fonctions ne seront pas réutilisables dans un autre contexte si elles utilisent des variables globales ! Ensuite, arriverez-vous à vous relire dans six mois ? Quelqu'un d'autre pourrait-il comprendre votre programme ? Il existe de nombreuses autres raisons [4] que nous ne développerons pas ici, mais libre à vous de consulter de la documentation externe.

Heureusement, Python est orienté objet et permet « d'encapsuler » des variables dans des objets et de s'affranchir définitivement des variables globales (nous verrons cela dans le chapitre 23 *Avoir la classe avec les objets*). En attendant, et si vous ne souhaitez pas aller plus loin sur les notions d'objet (on peut tout à fait « pythonner » sans cela), retenez la chose suivante sur les fonctions et les variables globales :

Conseil

Plutôt que d'utiliser des variables globales, passez vos variables explicitement aux fonctions comme des argument(s).

13.6.2 Modification d'une liste dans une fonction

Concernant les fonctions qui modifient une liste, nous vous conseillons de l'indiquer clairement dans votre code. Pour cela, faites en sorte que la fonction renvoie la liste modifiée et de récupérer cette liste renvoyée dans une variable portant le même nom. Par exemple :

```
1  def ajoute_un(liste):
2      for indice in range(len(liste)):
3          liste[indice] += 1
4      return liste
5
6  # Programme principal.
7  liste_notes = [10, 8, 16, 7, 15]
8  liste_notes = ajoute_un(liste_notes)
9  print(liste_notes)
```

La ligne 8 indique que la liste `liste_notes` passée à la fonction est écrasée par la liste renvoyée par la fonction.
Le code suivant produirait la même sortie :

```
1  def ajoute_un(liste):
2      for indice in range(len(liste)):
3          liste[indice] += 1
4
5  # Programme principal.
6  liste_notes = [10, 8, 16, 7, 15]
7  ajoute_un(liste_notes)
8  print(liste_notes)
```

Cela reste toutefois moins intuitif, car il n'est pas évident de comprendre que la liste est modifiée dans la fonction en lisant la ligne 7. Dans un tel cas, il serait essentiel d'indiquer dans la documentation de la fonction que la liste est

4. http://wiki.c2.com/?GlobalVariablesAreBad

modifiée « sur place » (*in place* en anglais) dans la fonction. Vous verrez dans le chapitre 15 *Création de modules* comment documenter vos fonctions.

Conseil

Pour les raisons évoquées ci-dessus, nous vous conseillons de privilégier la première version :

```
liste_notes = ajoute_un(liste_notes)
```

13.6.3 Conclusion

Vous connaissez maintenant les fonctions sous tous leurs angles. Comme indiqué en introduction du chapitre 10, elles sont incontournables et tout programmeur se doit de les maîtriser. Voici les derniers conseils que nous pouvons vous donner :

- Lorsque vous débutez un nouveau projet de programmation, posez-vous la question : « Comment pourrais-je décomposer en blocs chaque tâche à effectuer, chaque bloc pouvant être une fonction ? ». Et n'oubliez pas que si une fonction s'avère trop complexe, vous pouvez la décomposer en d'autres fonctions.
- Au risque de nous répéter, forcez-vous à utiliser des fonctions en permanence. Pratiquez, pratiquez… et pratiquez encore !

13.7 Exercices

Conseil

Pour le second exercice, créez un script puis exécutez-le dans un *shell*.

13.7.1 Prédire la sortie

Prédisez le comportement des codes suivants, sans les recopier dans un script ni dans l'interpréteur Python :

13.7.1.1 Code 1

```
def hello(prenom):
    print(f"Bonjour {prenom}")


# Programme principal.
hello("Patrick")
print(x)
```

13.7.1.2 Code 2

```
def hello(prenom):
    print(f"Bonjour {prenom}")


# Programme principal.
x = 10
hello("Patrick")
print(x)
```

13.7.1.3 Code 3

```
def hello(prenom):
    print(f"Bonjour {prenom}")
    print(x)


# Programme principal.
x = 10
hello("Patrick")
print(x)
```

13.7.1.4 Code 4

```
def hello(prenom):
    x = 42
    print(f"Bonjour {prenom}")
    print(x)


# Programme principal.
x = 10
hello("Patrick")
print(x)
```

13.7.2 Passage de liste à une fonction

Créez une fonction `ajoute_nb_alea()` qui prend en argument une liste et qui ajoute un nombre entier aléatoire entre -10 et 10 (inclus) à chaque élément. La fonction affichera à l'écran cette nouvelle liste modifiée.

Dans le programme principal, effectuez les actions suivantes :

1. Créez une variable `ma_liste = [7, 3, 8, 4, 5, 1, 9, 10, 2, 6]`.
2. Affichez `ma_liste` à l'écran.
3. Appelez la fonction `ajoute_nb_alea()` en lui passant `ma_liste` en argument.
4. Affichez à nouveau `ma_liste` à l'écran.

Comment expliquez-vous le résultat obtenu ?

CHAPITRE 14

Conteneurs

Dans ce chapitre, nous allons aborder la notion de conteneur, revenir sur certaines propriétés avancées des dictionnaires et tuples, et enfin aborder les types *set* et *frozenset*. Pour les débutants, ce chapitre aborde des notions relativement avancées. Avant de vous lancer, nous vous conseillons vivement de bien maitriser les chapitres 4 *Listes* et 12 *Plus sur les listes*, ainsi que le chapitre 8 *Dictionnaires et tuples*, d'avoir effectué un maximum d'exercices, et de vous sentir à l'aise avec toutes les notions abordées jusque là.

14.1 Généralités

14.1.1 Définition et propriétés

Définition

Un **conteneur** (*container* en anglais) est un nom générique pour définir un objet Python qui contient une collection d'autres objets.

Les conteneurs que nous connaissons depuis le début de ce cours sont les listes, les chaînes de caractères, les dictionnaires et les tuples. Même si on ne l'a pas vu explicitement, les objets de type *range* sont également des conteneurs.

Dans la suite de cette rubrique, nous allons examiner les différentes propriétés des conteneurs. À la fin de ce chapitre, nous ferons un tableau récapitulatif de ces propriétés.

Examinons d'abord les propriétés qui caractérisent tous les types de conteneur.

- Capacité à supporter le **test d'appartenance**. Souvenez-vous, il permet de vérifier si un élément était présent dans une liste. Cela fonctionne donc aussi sur les chaînes de caractères ou tout autre conteneur :

```
>>> liste1 = [4, 5, 6]
>>> 4 in liste1
True
>>> "to" in "toto"
True
```

- Capacité à supporter la fonction `len()` renvoyant la longueur du conteneur.

Voici d'autres propriétés générales que nous avons déjà croisées. Un conteneur peut être :

- **Ordonné** (*ordered* en anglais) : il y a un ordre précis des éléments ; cet ordre correspond à celui utilisé lors de la création ou de la modification du conteneur (si cela est permis) ; ce même ordre est utilisé lorsqu'on itère dessus.

- **Indexable** (*subscriptable* en anglais) : on peut retrouver un élément par son indice (c'est-à-dire sa position dans le conteneur) ou plusieurs éléments avec une tranche ; en général, tout conteneur indexable est ordonné.
- **Itérable** (*iterable* en anglais) : on peut faire une boucle dessus.

Certains conteneurs sont appelés objets séquentiels ou séquence.

Définition

Un **objet séquentiel** ou **séquence** est un conteneur itérable, ordonné et indexable. Les objets séquentiels sont les listes, les chaînes de caractères, les objets de type *range*, ainsi que les tuples.

Une autre propriété importante que l'on a déjà croisée, et qui nous servira dans ce chapitre, concerne la possibilité ou non de modifier un objet.

- Un objet est dit **non modifiable** lorsqu'on ne peut pas le modifier, ou lorsqu'on ne peut pas en modifier un de ses éléments si c'est un conteneur. On parle aussi d'objet immuable[1] (*immutable object* en anglais). Cela signifie qu'une fois créé, Python ne permet plus de le modifier par la suite.

Qu'en est-il des objets que nous connaissons ? Les listes sont modifiables, on peut modifier un ou plusieurs de ses éléments et ajouter ou retirer un élément. Les dictionnaires sont modifiables : pour une clé donnée, on peut changer la valeur correspondante et ajouter ou retirer un couple clé/valeur. Tous les autres types que nous avons vus précédemment sont quant à eux non modifiables : les chaînes de caractères ou *strings*, les tuples, les objets de type *range*, mais également des objets qui ne sont pas des conteneurs comme les entiers, les *floats* et les booléens.

On comprend bien l'immutabilité des *strings* comme vu au chapitre 11 *Plus sur les chaînes de caractères*, mais c'est moins évident pour les entiers, *floats* ou booléens. Nous allons démontrer cela, mais avant nous avons besoin de définir la notion d'identifiant d'un objet.

Définition

L'**identifiant** d'un objet est un nombre entier qui est garanti constant pendant toute la durée de vie de l'objet. Cet identifiant est en général unique pour chaque objet. Toutefois, pour des raisons d'optimisation, Python crée parfois le même identifiant pour deux objets non modifiables différents qui ont la même valeur. L'identifiant peut être assimilé à l'adresse mémoire de l'objet qui, elle aussi, est unique. En Python, on utilise la fonction interne `id()` qui prend en argument un objet et renvoie son identifiant.

Maintenant que l'identifiant est défini, regardons l'exemple suivant qui montre l'immutabilité des entiers :

```
1  >>> var = 4
2  >>> id(var)
3  140318876873440
4  >>> var = 5
5  >>> id(var)
6  140318876873472
```

Ligne 1 on définit l'entier `var` puis on regarde son identifiant. Ligne 4, on pourrait penser que l'on modifie `var`. Toutefois, on voit que son identifiant ligne 6 est différent de la ligne 3. En fait, l'affectation ligne 4 `var = 5` écrase l'ancienne variable `var` et en crée une nouvelle, ce n'est pas la valeur de `var` qui a été changée puisque l'identifiant n'est plus le même. Le même raisonnement peut être tenu pour les autres types numériques comme les *floats* et booléens. Si on regarde maintenant ce qu'il se passe pour une liste :

```
1  >>> liste1 = [1, 2, 3]
2  >>> id(liste1)
3  140318850324832
4  >>> liste1[1] = -15
5  >>> id(liste1)
6  140318850324832
7  >>> liste1.append(5)
8  >>> id(liste1)
9  140318850324832
```

1. https://fr.wikipedia.org/wiki/Objet_immuable

La liste `liste1` a été modifiée ligne 4 (changement de l'élément d'indice 1) et ligne 7 (ajout d'un élément). Pour autant, l'identifiant de cette liste est resté identique tout du long. Ceci démontre la mutabilité des listes : quelle que soit la manière dont on modifie une liste, celle-ci garde le même identifiant.

- Une dernière propriété importante est la capacité d'un conteneur (ou tout autre objet Python) à être **hachable**.

Définition

Un objet Python est dit **hachable** (*hashable* en anglais) s'il est possible de calculer une valeur de hachage sur celui-ci avec la fonction interne `hash()`. En programmation, la valeur de hachage peut être vue comme une empreinte numérique de l'objet. Elle est obtenue en passant l'objet dans une fonction de hachage et dépend du contenu de l'objet. En Python, cette empreinte est, comme dans la plupart des langages de programmation, un entier. Au sein d'une même session Python, deux objets hachables qui ont un contenu identique auront strictement la même valeur de hachage.

Attention

La valeur de hachage d'un objet renvoyée par la fonction `hash()` n'a pas le même sens que son identifiant renvoyé par la fonction `id()`. La valeur de hachage est obtenue en « moulinant » le contenu de l'objet dans une fonction de hachage. L'identifiant est quant à lui attribué par Python à la création de l'objet. Il est constant tout le long de la durée de vie de l'objet, un peu comme une carte d'identité. Tout objet a un identifiant, mais il doit être hachable pour avoir une valeur de hachage.

Pour aller plus loin

Pour aller plus loin, vous pouvez consulter la page Wikipedia sur les fonctions de hachage[2].

Pourquoi évoquer cette propriété de hachabilité ? D'abord, parce qu'elle est étroitement liée à l'immutabilité. En effet, un objet non modifiable est la plupart du temps hachable. Cela permet de l'identifier **en fonction de son contenu**. Par ailleurs, l'hachabilité est une implémentation qui permet un accès rapide aux éléments des conteneurs de type dictionnaire ou *set* (cf. rubriques suivantes).

Les objets hachables sont les chaînes de caractères, les entiers, les *floats*, les booléens, les objets de type *range*, les tuples (sous certaines conditions) et les *frozensets* ; par contre, les listes, les *sets* et les dictionnaires sont non hachables. Les *sets* et *frozensets* seront vus plus bas dans ce chapitre.

Voici un exemple :

```
1  >>> hash("Plouf")
2  5085648805260210718
3  >>> hash(5)
4  5
5  >>> hash(3.14)
6  322818021289917443
7  >>> hash([1, 2, 3])
8  Traceback (most recent call last):
9    File "<stdin>", line 1, in <module>
10 TypeError: unhashable type: 'list'
```

Les valeurs de hachage renvoyées par la fonction `hash()` de Python sont systématiquement des entiers. Par contre, Python renvoie une erreur pour une liste, car elle est non hachable.

14.1.2 Conteneurs de type *range*

Revenons rapidement sur les objets de type *range*. Jusqu'à maintenant, on s'en est servi pour faire des boucles ou générer des listes de nombres. Toutefois, on a vu ci-dessus qu'ils étaient aussi des conteneurs. Ils sont ordonnés, indexables, itérables, hachables et non modifiables :

2. https://fr.wikipedia.org/wiki/Fonction_de_hachage

```
>>> range1 = range(3)
>>> range1[0]
0
>>> range1[0:1]
range(0, 1)
>>> for element in range1:
...     print(element)
...
0
1
2
>>> range1[2] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'range' object does not support item assignment
>>> hash(range1)
5050907061201647097
```

La tentative de modification d'un élément ligne 12 conduit à la même erreur que lorsqu'on essaie de modifier un caractère d'une chaîne de caractères. Comme pour la plupart des objets Python non modifiables, les objets de type *range* sont hachables.

14.2 Plus sur les dictionnaires

Nous revenons sur les dictionnaires qui, on l'a vu, sont des conteneurs de correspondance où chaque valeur est associée à une clé plutôt qu'un indice. Nous allons voir certaines propriétés avancées des dictionnaires, notamment comment trier par clé ou par valeur.

14.2.1 Objets utilisables comme clé

Toutes les clés de dictionnaire vues dans le chapitre 8 *Dictionnaires et tuples* et utilisées jusqu'à présent étaient des chaînes de caractères. Toutefois, on peut utiliser d'autres types d'objets comme des entiers, des *floats*, voire des tuples, cela peut s'avérer parfois très utile. Une règle est toutefois requise : les objets utilisés comme clé doivent être **hachables** (voir la rubrique précédente pour la définition).

Pourquoi les clés doivent être des objets hachables ? C'est la raison d'être des dictionnaires qui d'ailleurs sont aussi appelés table de hachage[3] dans d'autres langages, comme Perl. Convertir chaque clé en sa valeur de hachage permet un accès très rapide à chacun des éléments du dictionnaire, ainsi que des comparaisons de clés entre dictionnaires extrêmement efficaces. Même si on a vu que deux objets pouvaient avoir la même valeur de hachage, par exemple `a = 5` et `b = 5`, on ne peut mettre qu'une seule fois la clé `5`. Ceci assure que deux clés d'un même dictionnaire ont forcément une valeur de hachage différente.

Pouvoir utiliser autre chose qu'une chaîne de caractères comme clé peut se révéler très pratique. Par exemple, pour une protéine ou un peptide, on pourrait concevoir d'utiliser comme clé le numéro de résidu, et comme valeur le nom de résidu. Imaginons par ailleurs que nous commencions à compter le premier acide aminé à 3 (souvent les fichiers PDB ne commence pas à 1 pour le premier acide aminé). Par exemple :

```
>>> sequence = {3: 'S', 4: 'E', 5: 'Q', 6: 'P', 7: 'E', 8: 'P', 9: 'T'}
>>> sequence[5]
'Q'
>>> sequence[9]
'T'
>>> for num, res in sequence.items():
...     print(num, res)
...
3 S
4 E
5 Q
6 P
7 E
8 P
9 T
```

3. https://fr.wikipedia.org/wiki/Table_de_hachage

Vous voyez l'énorme avantage, d'utiliser comme clé le numéro de résidu. Avec une liste ou une chaîne de caractère, l'indiçage commence à zéro. Ainsi, il faudrait utiliser les indices 2 et 6 pour retrouver respectivement les acides aminés 5 et 9 :

```
>>> sequence = ['S', 'E', 'Q', 'P', 'E', 'P', 'T']
>>> sequence[2]
'Q'
>>> sequence[6]
'T'
```

14.2.2 Destruction d'une paire clé/valeur

Comme pour tous les objets Python, l'instruction `del` permet de détruire un couple clé/valeur :

```
>>> dico = {'nom': 'girafe', 'taille': 5.0, 'poids': 1100}
>>> del dico["nom"]
>>> dico
{'taille': 5.0, 'poids': 1100}
```

Pour les listes, on utilise l'indice entre crochet pour détruire l'élément, par exemple `del liste[2]`. Ici, on utilise la clé.

14.2.3 Tri par clés

On peut utiliser la fonction `sorted()` vue précédemment avec les listes pour trier un dictionnaire par ses clés :

```
>>> ani2 = {'nom': 'singe', 'taille': 1.75, 'poids': 70}
>>> sorted(ani2)
['nom', 'poids', 'taille']
```

Les clés sont triées ici par ordre alphabétique.

14.2.4 Tri par valeurs

Pour trier un dictionnaire par ses valeurs, il faut utiliser la fonction `sorted()` avec l'argument `key` :

```
>>> dico = {"a": 15, "b": 5, "c":20}
>>> sorted(dico, key=dico.get)
['b', 'a', 'c']
```

L'argument `key=dico.get` indique explicitement qu'il faut réaliser le tri par les valeurs du dictionnaire. On retrouve la méthode `.get()` vue au chapitre 8 *Dictionnaires et tuples*, mais sans les parenthèses : `key=dico.get`, mais pas `key=dico.get()`. Une fonction ou méthode passée en argument sans les parenthèses est appelée *callback*, nous reverrons cela en détail dans le chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne).

Attention, ce sont les clés du dictionnaire qui sont renvoyées, pas les valeurs. Ces clés sont cependant renvoyées dans un ordre qui permet d'obtenir les clés triées par ordre croissant :

```
>>> dico = {"a": 15, "b": 5, "c":20}
>>> for key in sorted(dico, key=dico.get):
...     print(key, dico[key])
...
b 5
a 15
c 20
```

Enfin, l'argument `reverse=True` fonctionne également :

```
>>> dico = {"a": 15, "b": 5, "c":20}
>>> sorted(dico, key=dico.get, reverse=True)
['c', 'a', 'b']
```

Remarque

Lorsqu'on trie un dictionnaire par ses valeurs, il faut être sûr que cela soit possible. Ce n'est pas le cas lorsqu'on a un mélange de valeurs numériques et chaînes de caractères :

```
>>> ani2 = {'nom': 'singe', 'poids': 70, 'taille': 1.75}
>>> sorted(ani2, key=ani2.get)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

On obtient ici une erreur, car Python ne sait pas comparer une chaîne de caractères (`singe`) avec des valeurs numériques (`70` et `1.75`).

14.2.5 Clé associée au minimum ou au maximum des valeurs

Les fonctions `min()` et `max()`, que vous avez déjà manipulées dans les chapitres précédents, acceptent également l'argument `key=`. On peut ainsi obtenir la clé associée au minimum ou au maximum des valeurs d'un dictionnaire :

```
>>> dico = {"a": 15, "b": 5, "c":20}
>>> max(dico, key=dico.get)
'c'
>>> min(dico, key=dico.get)
'b'
```

14.2.6 Fonction `dict()`

La fonction `dict()` va convertir l'argument qui lui est passé en dictionnaire. Il s'agit donc d'une fonction de *casting*, comme `int()`, `str()`, etc. Toutefois, l'argument qui lui est passé doit avoir une forme particulière : un objet séquentiel contenant d'autres objets séquentiels de deux éléments. Par exemple, une liste de listes de deux éléments :

```
>>> liste_animaux = [["girafe", 2], ["singe", 3]]
>>> dict(liste_animaux)
{'girafe': 2, 'singe': 3}
```

Ou un tuple de tuples de deux éléments, ou encore une combinaison liste et tuple :

```
>>> tuple_animaux = (("girafe", 2), ("singe", 3))
>>> dict(tuple_animaux)
{'girafe': 2, 'singe': 3}
>>>
>>> dict([("girafe", 2), ("singe", 3)])
{'girafe': 2, 'singe': 3}
```

Si un des sous-éléments a plus de deux éléments (ou moins), Python renvoie une erreur :

```
>>> dict([("girafe", 2), ("singe", 3, 4)])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: dictionary update sequence element #1 has length 3; 2 is required
```

Attention

Une manière intuitive utilise simplement des arguments par mot-clés, qui deviendront des clés sous forme de chaîne de caractères :

```
>>> dict(un=1, deux=2, trois=3)
{'un': 1, 'deux': 2, 'trois': 3}
```

Nous vous déconseillons toutefois cette manière de faire, car on ne peut pas mettre d'arguments par mot-clé variables, on doit les écrire explicitement.

Une dernière manière puissante pour générer des dictionnaires combine les fonctions `dict()` et `zip()`. On se souvient que la fonction `zip()` peut générer une liste de tuples :

```
>>> animaux = ["poulain", "renard", "python"]
>>> couleurs = ["alezan", "roux", "vert"]
>>> list(zip(animaux, couleurs))
[('poulain', 'alezan'), ('renard', 'roux'), ('python', 'vert')]
```

Si on utilise l'objet *zip* avec la fonction `dict()`, on obtient un dictionnaire.

```
>>> dict(zip(animaux, couleurs))
{'poulain': 'alezan', 'renard': 'roux', 'python': 'vert'}
```

Attention à ne passer que deux listes à la fonction `zip()`, sinon Python renvoie une erreur :

```
>>> dict(zip([1, 2, 3], animaux, couleurs))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: dictionary update sequence element #0 has length 3; 2 is required
```

14.3 Plus sur les tuples

Nous revenons sur les tuples, que nous avons défini dans le chapitre 8 *Dictionnaires et tuples* et que nous avons croisé à de nombreuses reprises, notamment avec les fonctions. Les tuples sont des objets séquentiels correspondant aux listes, donc ils sont itérables, ordonnés et indexables, mais ils sont toutefois non modifiables. On verra plus bas qu'ils sont hachables sous certaines conditions. L'intérêt des tuples par rapport aux listes réside dans leur immutabilité. Cela accélère considérablement la manière dont Python accède à chaque élément et ils prennent moins de place en mémoire. Par ailleurs, on ne risque pas de modifier un de ses éléments par mégarde.

14.3.1 Immutabilité

Nous avions vu que les tuples étaient immuables :

```
>>> tuple1 = (1, 2, 3)
>>> tuple1[2] = 15
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Ce message est similaire à celui que nous avions rencontré quand on essayait de modifier une chaîne de caractères (voir chapitre 11 *Plus sur les chaînes de caractères*). De manière générale, Python renverra un message `TypeError: '[...]' does not support item assignment` lorsqu'on essaie de modifier un élément d'un objet non modifiable. Si vous voulez ajouter un élément (ou le modifier), vous devez créer un nouveau tuple :

```
1  >>> tuple1 = (1, 2, 3)
2  >>> tuple1
3  (1, 2, 3)
4  >>> id(tuple1)
5  139971081704464
6  >>> tuple1 = tuple1 + (2,)
7  >>> tuple1
8  (1, 2, 3, 2)
9  >>> id(tuple1)
10 139971081700368
```

La fonction `id()` montre que le tuple créé ligne 6 est bien différent de celui créé ligne 4, bien qu'ils aient le même nom. Comme on a vu plus haut, ceci est dû à l'opérateur d'affectation utilisé ligne 6 (`tuple1 = tuple1 + (2,)`) qui crée un nouvel objet distinct de celui de la ligne 1. Cet exemple montre que les tuples sont peu adaptés lorsqu'on a besoin d'ajouter, retirer, modifier des éléments. La création d'un nouveau tuple à chaque étape s'avère lourde et il n'y a aucune méthode pour faire cela, puisque les tuples sont non modifiables.

Conseil

Pour ce genre de tâche, les listes sont clairement mieux adaptées que les tuples.

14.3.2 Affectation multiple et fonctions

Dans le chapitre 8 *Dictionnaires et tuples*, nous avons abordé l'affectation multiple. Pour rappel, elle permet d'effectuer sur une même ligne plusieurs affectations en même temps, par exemple : `x, y, z = 1, 2, 3`. On a vu qu'il était possible de le faire également avec les listes : `[x, y, z] = [1, 2, 3]`. Toutefois, cette syntaxe étant alourdie par la présence des crochets, on préfèrera toujours la première syntaxe avec les tuples sans parenthèses.

Concernant les fonctions, nous avions croisé l'importance de l'affectation multiple dans le chapitre 10 lorsqu'une fonction renvoyait plusieurs valeurs :

```
>>> def ma_fonction():
...     return 3, 14
...
>>> x, y = ma_fonction()
>>> print(x, y)
3 14
```

La syntaxe `x, y = ma_fonction()` permet de récupérer les deux valeurs renvoyées par la fonction et de les affecter à la volée dans deux variables différentes. Cela évite l'opération laborieuse de récupérer d'abord le tuple, puis de créer les variables en utilisant l'indiçage :

```
>>> resultat = ma_fonction()
>>> resultat
(3, 14)
>>> x = resultat[0]
>>> y = resultat[1]
>>> print(x, y)
3 14
```

Conseil

Lorsqu'une fonction renvoie plusieurs valeurs sous forme de tuple, privilégiez toujours la forme `x, y = ma_fonction()`.

14.3.3 Affectation multiple et nom de variable _

Quand une fonction renvoie plusieurs valeurs, mais que l'on ne souhaite pas les utiliser toutes dans la suite du code, on peut utiliser le nom de variable _ (caractère *underscore*) pour indiquer que certaines valeurs ne nous intéressent pas :

```
>>> def ma_fonction():
...     return 1, 2, 3, 4
...
>>> x, _, y, _ = ma_fonction()
>>> x
1
>>> y
3
```

Cela envoie le message à la personne qui lit le code « je ne m'intéresse pas aux valeurs récupérées dans les variables _ ». Notez que l'on peut utiliser une ou plusieurs variables *underscore(s)*. Dans l'exemple ci-dessus, la 2e et la 4e variable renvoyées par la fonction seront ignorées dans la suite du code. Cela présente le mérite d'éviter de polluer l'attention de la personne qui lit le code.

Remarque

Dans l'interpréteur interactif, la variable _ a une signification différente. Elle prend automatiquement la dernière valeur affichée :

```
>>> 3
3
>>> _
3
>>> "mésange"
'mésange'
>>> _
'mésange'
```

Attention, ceci n'est vrai que dans l'interpréteur !

Remarque

Le caractère *underscore* (_) est couramment utilisé dans les noms de variable pour séparer les mots et être explicite, par exemple `seq_ADN` ou `liste_listes_residus`. On verra dans le chapitre 16 *Bonnes pratiques en programmation Python* que ce style de nommage est appelé *snake_case*. Toutefois, il faut éviter d'utiliser les *underscores* en début et/ou en fin de nom de variable (*leading* et *trailing underscores* en anglais), par exemple : `_var`, `var_`, `__var`, `__var__`. On verra au chapitre 23 *Avoir la classe avec les objets* que ces *underscores* ont aussi une signification particulière.

14.3.4 Tuples contenant des listes

On a vu que les tuples étaient **non modifiables**. Que se passe-t-il alors si on crée un tuple contenant des objets modifiables comme des listes ? Examinons le code suivant :

```
>>> liste1 = [1, 2, 3]
>>> tuple1 = (liste1, "Plouf")
>>> tuple1
([1, 2, 3], 'Plouf')
>>> liste1[0] = -15
>>> tuple1[0].append(-632)
>>> tuple1
([-15, 2, 3, -632], 'Plouf')
```

Si on modifie un élément de la liste `liste1` (ligne 5) ou bien qu'on ajoute un élément à `tuple1[0]` (ligne 6), Python s'exécute et ne renvoie pas de message d'erreur. Or nous avions dit qu'un tuple était non modifiable… Comment cela est-il possible ? Commençons d'abord par regarder comment les objets sont agencés avec *Python Tutor*.

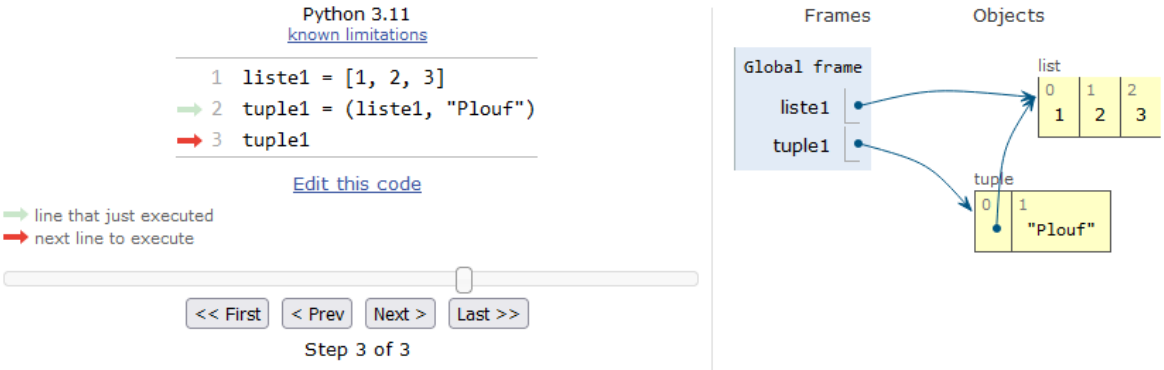


FIGURE 14.1 – Tuple contenant une liste.

La liste `liste1` pointe vers le même objet que l'élément du tuple d'indice 0. Comme pour la copie de liste (par exemple `liste_b = liste_a`), ceci est attendu car, par défaut, Python crée une copie par référence (voir le chapitre 12 *Plus sur les listes*). Ainsi, qu'on raisonne en tant que premier élément du tuple ou bien en tant que liste `liste1`, on pointe vers **la même liste**. Or, rappelez-vous, nous avons expliqué au début de ce chapitre que lorsqu'on modifiait un

élément d'une liste, celle-ci gardait le même identifiant. C'est toujours le cas ici, même si celle-ci se trouve dans un tuple. Regardons cela :

```
>>> liste1 = [1, 2, 3]
>>> tuple1 = (liste1, "Plouf")
>>> tuple1
([1, 2, 3], 'Plouf')
>>> id(liste1)
139971081980816
>>> id(tuple1[0])
139971081980816
```

Nous confirmons ici le schéma de *Python Tutor*, c'est bien la même liste que l'on considère `liste1` ou `tuple1[0]` puisqu'on a le même identifiant. Maintenant, on modifie cette liste via la variable `liste1` ou `tuple1[0]` :

```
>>> liste1[2] = -15
>>> tuple1[0].append(-632)
>>> tuple1
([1, 2, -15, -632], 'Plouf')
>>> id(liste1)
139971081980816
>>> id(tuple1[0])
139971081980816
```

Malgré la modification de cette liste, l'identifiant n'a toujours pas changé puisque la fonction `id()` nous renvoie la même valeur depuis le début. Même si la liste a été modifiée « de l'intérieur », Python considère que c'est toujours la même liste, puisqu'elle n'a pas changé d'identifiant. Si au contraire on essaie de remplacer cette sous-liste par autre chose, Python renvoie une erreur :

```
>>> tuple1[0] = "Plif"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Cette erreur s'explique par le fait que le nouvel objet `"Plif"` n'a pas le même identifiant que la sous-liste initiale. En fait, l'immutabilité selon Python signifie qu'un objet créé doit toujours garder le même identifiant. Cela est valable pour tout objet non modifiable, comme un élément d'un tuple, un caractère dans une chaîne de caractères, etc.

Conseil

Cette digression avait pour objectif de vous faire comprendre ce qu'il se passe lorsqu'on met une liste dans un tuple. Toutefois, pouvoir modifier une liste en tant qu'élément d'un tuple va à l'encontre de l'intérêt d'un objet non modifiable. Dans la mesure du possible, nous vous déconseillons de créer des listes dans des tuples afin d'éviter les déconvenues.

14.3.5 Fonction `tuple()`

Nous avions vu également la fonction `tuple()`, qui permet de convertir un objet séquentiel en tuple (opération de *casting*). Cela est possible seulement si l'objet passé en argument est itérable :

```
>>> tuple([1, 3])
(1, 3)
>>> tuple("a")
('a',)
>>> tuple(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>> tuple(True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bool' object is not iterable
```

Bien sûr, un entier ou un booléen ne sont pas itérables.

14.3.6 Hachabilité des tuples

Les tuples sont hachables s'ils ne contiennent que des éléments hachables. Si un tuple contient un ou plusieurs objet(s) non hachable(s), comme une liste, il devient non hachable :

```
>>> tuple1 = tuple(range(10))
>>> tuple1
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> hash(tuple1)
-4181190870548101704
>>> tuple2 = ("Plouf", 2, (1, 3))
>>> tuple2
('Plouf', 2, (1, 3))
>>> hash(tuple2)
286288423668065022
>>> tuple3 = (1, (3, 4), "Plaf", [3, 4, 5])
>>> tuple3
(1, (3, 4), 'Plaf', [3, 4, 5])
>>> hash(tuple3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Les tuples `tuple1` et `tuple2` sont hachables car ils ne contiennent que des éléments hachables. Par contre, `tuple3` ne l'est pas, car un de ses éléments est une liste.

Conseil

Mettre une ou plusieurs liste(s) dans un tuple le rend non hachable. Ceci le rend inutilisable comme clé de dictionnaire ou, on le verra ci-après, comme élément d'un *set* ou d'un *frozenset*. Donc, à nouveau, ne mettez pas de listes dans vos tuples !

14.4 *Sets* et *frozensets*

14.4.1 Définition et propriétés

Les objets de type *set* représentent un autre type de conteneur qui peut se révéler très pratique. Ils ont la particularité d'être modifiables, non hachables, non ordonnés, non indexables et de ne contenir qu'une seule copie maximum de chaque élément. Pour créer un nouveau *set* on peut utiliser les accolades :

```
>>> set1 = {4, 5, 5, 12}
>>> set1
{12, 4, 5}
>>> type(set1)
<class 'set'>
```

Remarquez que la répétition du chiffre 5 dans la définition du *set* ligne 1 produit finalement un seul chiffre 5, car chaque élément ne peut être présent qu'une seule fois. Comme pour les dictionnaires (jusqu'à la version 3.6), les *sets* sont non ordonnés. La manière dont Python les affiche n'a pas de sens en tant que tel et peut être différente de celle utilisée lors de leur création.

Les *sets* ne peuvent contenir que des objets **hachables**. On a déjà eu le cas avec les clés de dictionnaire. Ceci optimise l'accès à chaque élément du *set*. Pour rappel, les objets hachables que nous connaissons sont les chaînes de caractères, les tuples, les entiers, les *floats*, les booléens et les *frozensets* (voir plus bas). Les objets non hachables que l'on connait sont les listes, les *sets* et les dictionnaires. Si on essaie tout de même de mettre une liste dans un *set*, Python renvoie une erreur :

```
>>> set1 = {3, 4, "Plouf", (1, 3)}
>>> set1
{(1, 3), 3, 4, 'Plouf'}
>>> set2 = {3.14, [1, 2]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

À quoi différencie-t-on un *set* d'un dictionnaire alors que les deux utilisent des accolades ? Le *set* sera défini seulement par des valeurs `{valeur_1, valeur_2, ...}` alors que le dictionnaire aura toujours des couples clé/valeur `{clé_1: valeur_1, clé_2: valeur_2, ...}`.

La fonction interne à Python `set()` convertit un objet itérable passé en argument en un nouveau *set* (opération de *casting*) :

```
>>> set([1, 2, 4, 1])
{1, 2, 4}
>>> set((2, 2, 2, 1))
{1, 2}
>>> set(range(5))
{0, 1, 2, 3, 4}
>>> set({"clé_1": 1, "clé_2": 2})
{'clé_1', 'clé_2'}
>>> set(["ti", "to", "to"])
{'ti', 'to'}
>>> set("Maître Corbeau et Maître Renard")
{'e', 'd', 'M', 'r', 'n', 't', 'a', 'C', 'î', ' ', 'o', 'u', 'R', 'b'}
```

Nous avons dit plus haut que les *sets* ne sont ni ordonnés ni indexables, il est donc impossible de récupérer un élément par sa position. Il est également impossible de modifier un de ses éléments par l'indexation.

```
>>> set1 = set([1, 2, 4, 1])
>>> set1[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
>>> set1[1] = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support item assignment
```

Par contre, les *sets* sont itérables :

```
>>> for element in set1:
...     print(element)
...
1
2
4
```

Les *sets* ne peuvent être modifiés que par des méthodes spécifiques :

```
>>> set1 = set(range(5))
>>> set1
{0, 1, 2, 3, 4}
>>> set1.add(4)
>>> set1
{0, 1, 2, 3, 4}
>>> set1.add(472)
>>> set1
{0, 1, 2, 3, 4, 472}
>>> set1.discard(0)
>>> set1
{1, 2, 3, 4, 472}
```

La méthode `.add()` ajoute au *set* l'élément passé en argument. Toutefois, si l'élément est déjà présent dans le *set*, il n'est pas ajouté puisqu'on a au plus une copie de chaque élément. La méthode `.discard()` retire du *set* l'élément passé en argument. Si l'élément n'est pas présent dans le *set*, il ne se passe rien, le *set* reste intact. Comme les *sets* ne sont pas ordonnés ni indexables, il n'y a pas de méthode pour insérer un élément à une position précise, contrairement aux listes. Dernier point sur ces méthodes, elles modifient le *set* sur place (*in place*, en anglais) et ne renvoient rien, à l'instar des méthodes des listes (`.append()`, `.remove()`, etc.).

Enfin, les *sets* ne supportent pas les opérateurs `+` et `*`.

14.4.2 Utilité

Les conteneurs de type *set* sont très utiles pour rechercher les éléments uniques d'une suite d'éléments. Cela revient à éliminer tous les doublons. Par exemple :

```
>>> import random
>>> liste1 = [random.randint(0, 9) for i in range(10)]
>>> liste1
[7, 9, 6, 6, 7, 3, 8, 5, 6, 7]
>>> set(liste1)
{3, 5, 6, 7, 8, 9}
```

On peut bien sûr transformer dans l'autre sens un *set* en liste. Cela permet par exemple d'éliminer les doublons de la liste initiale, tout en récupérant une liste à la fin :

```
>>> list(set([7, 9, 6, 6, 7, 3, 8, 5, 6, 7]))
[3, 5, 6, 7, 8, 9]
```

On peut faire des choses très puissantes. Par exemple, un compteur de lettres en combinaison avec une liste de compréhension, le tout en une ligne !

```
>>> seq = "atctcgatcgatcgcgctagctagctcgccatacgtacgactacgt"
>>> set(seq)
{'c', 'g', 't', 'a'}
>>> [(base, seq.count(base)) for base in set(seq)]
[('c', 15), ('g', 10), ('t', 11), ('a', 10)]
```

Les *sets* permettent aussi l'évaluation d'union ou d'intersection mathématiques en conjonction avec les opérateurs, respectivement | et & :

```
>>> liste1 = [3, 3, 5, 1, 3, 4, 1, 1, 4, 4]
>>> liste2 = [3, 0, 5, 3, 3, 1, 1, 1, 2, 2]
>>> set(liste1) | set(liste2)
{0, 1, 2, 3, 4, 5}
>>> set(liste1) & set(liste2)
{1, 3, 5}
```

Notez qu'il existe les méthodes `.union()` et `.intersection` permettant de réaliser ces opérations d'union et d'intersection :

```
>>> set1 = {1, 3, 4, 5}
>>> set2 = {0, 1, 2, 3, 5}
>>> set1.union(set2)
{0, 1, 2, 3, 4, 5}
>>> set1.intersection(set2)
{1, 3, 5}
```

L'instruction `set1.difference(set2)` renvoie sous la forme d'un nouveau *set* les éléments de `set1` qui ne sont pas dans `set2`. Et inversement pour `set2.difference(set1)` :

```
>>> set1.difference(set2)
{4}
>>> set2.difference(set1)
{0, 2}
```

Enfin, deux autres méthodes sont très utiles :

```
>>> set1 = set(range(10))
>>> set2 = set(range(3, 7))
>>> set3 = set(range(15, 17))
>>> set1
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> set2
{3, 4, 5, 6}
>>> set3
{16, 15}
>>> set2.issubset(set1)
True
>>> set3.isdisjoint(set1)
True
```

La méthode `.issubset()` indique si un *set* est inclus dans un autre *set*. La méthode `isdisjoint()` indique si un *set* est disjoint d'un autre *set*, c'est-à-dire, s'ils n'ont aucun élément en commun indiquant que leur intersection est nulle.

Il existe de nombreuses autres méthodes que nous n'abordons pas ici, mais qui peuvent être consultées sur la documentation officielle de Python [4].

14.4.3 *Frozensets*

Les *frozensets* sont des *sets* non modifiables et hachables. Ainsi, un *set* peut contenir des *frozensets* mais pas l'inverse. À quoi servent-ils ? Comme la différence entre tuple et liste, l'immutabilité des *frozensets* donne l'assurance de ne pas pouvoir les modifier par erreur. Pour créer un *frozenset* on utilise la fonction interne `frozenset()`, qui prend en argument un objet itérable et le convertit (opération de *casting*) :

```
>>> frozen1 = frozenset([3, 3, 5, 1, 3, 4, 1, 1, 4, 4])
>>> frozen2 = frozenset([3, 0, 5, 3, 3, 1, 1, 1, 2, 2])
>>> frozen1
frozenset({1, 3, 4, 5})
>>> frozen2
frozenset({0, 1, 2, 3, 5})
>>> frozen1.add(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
>>> frozen1.union(frozen2)
frozenset({0, 1, 2, 3, 4, 5})
>>> frozen1.intersection(frozen2)
frozenset({1, 3, 5})
```

Les *frozensets* ne possèdent bien sûr pas les méthodes de modification des *sets* (`.add()`, `.discard()`, etc.) puisqu'ils sont non modifiables. Par contre, ils possèdent toutes les méthodes de comparaisons de *sets* (`.union()`, `.intersection()`, etc.).

Conseil

Pour aller plus loin sur les *sets* et les *frozensets*, voici deux articles sur les sites programiz [5] et towardsdatascience [6].

14.5 Récapitulation des propriétés des conteneurs

Après ce tour d'horizon des différents conteneurs, voici des tableaux qui résument leurs propriétés. La mention « `in` et `len()` » indique que l'on peut tester l'appartenance d'un élément à un conteneur avec l'opérateur `in`, et que l'on peut connaître le nombre d'éléments du conteneur avec la fonction `len()`. Les mentions « index. » et « modif. » indiquent respectivement « indexable » et « modifiable ».

14.5.1 Objets séquentiels

4. https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset
5. https://www.programiz.com/python-programming/set
6. https://towardsdatascience.com/python-sets-and-set-theory-2ace093d1607

Conteneur	`in` et `len()`	itérable	ordonné	index.	modif.	hachable
liste	oui	oui	oui	oui	oui	non
chaîne de caractères	oui	oui	oui	oui	non	oui
range	oui	oui	oui	oui	non	oui
tuple	oui	oui	oui	oui	non	oui*

* s'il ne contient que des objets hachables

14.5.2 Objets de *mapping*

Conteneur	`in` et `len()`	itérable	ordonné	index.	modif.	hachable
dictionnaire	oui	oui sur les clés	oui*	non	oui	non

* à partir de Python 3.7 uniquement

14.5.3 Objets *sets*

Conteneur	`in` et `len()`	itérable	ordonné	index.	modif.	hachable
sets	oui	oui	non	non	oui	non
frozensets	oui	oui	non	non	non	oui

14.5.4 Types de base

Il est aussi intéressant de comparer ces propriétés avec celles des types numériques de base qui ne sont pas des conteneurs.

Objet numérique	`in` et `len()`	itérable	ordonné	index.	modif.	hachable
entier	non	non	non	non	non	oui
float	non	non	non	non	non	oui
booléen	non	non	non	non	non	oui

14.5.5 Copie de conteneurs

Un dernier point qu'il peut être utile de mentionner concerne la copie de conteneurs. On avait vu dans le chapitre 12 *Plus sur les listes* que la copie de listes se fait par référence. Cela est un mécanisme général pour tous les types de conteneurs, sauf pour les chaînes de caractères. *Python Tutor* nous permet de visualiser cela (Figure 14.2).

Ainsi, il faut toujours faire attention quand on fait une copie d'un conteneur modfiable (liste, dictionnaire, set, etc.). On verra que Python se comporte de la même manière avec les objets *arrays* (chapitre 20 *module Numpy*) ou *Dataframes* (chapitre 22 *Module pandas*), car on peut les considérer également comme des conteneurs.

14.6 Dictionnaires et *sets* de compréhension

Nous avons abordé les listes de compréhension dans le chapitre 12 *Plus sur les listes*. Il est également possible de générer des dictionnaires de compréhension :

```
>>> dico = {"a": 10, "g": 10, "t": 11, "c": 15}
>>> dico.items()
dict_items([('a', 10), ('g', 10), ('t', 11), ('c', 15)])
>>> {key:val*2 for key, val in dico.items()}
{'a': 20, 'g': 20, 't': 22, 'c': 30}
>>>
>>> animaux = (("singe", 3), ("girafe", 4), ("rhinocéros", 2))
>>> {animal:nombre for animal, nombre in animaux}
{'singe': 3, 'girafe': 4, 'rhinocéros': 2}
```

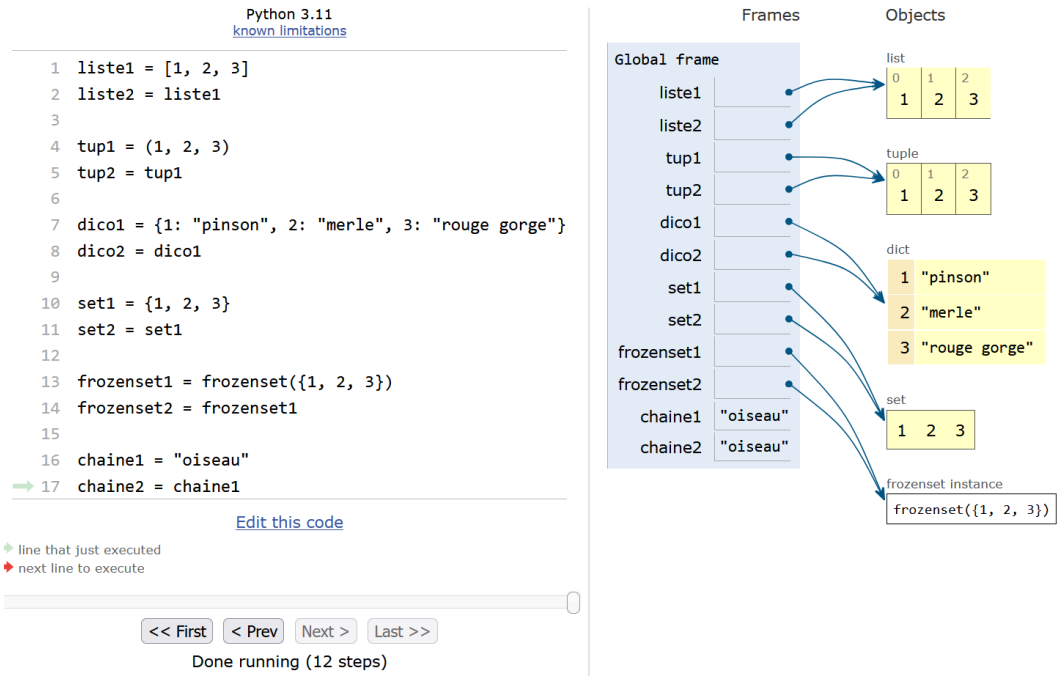


FIGURE 14.2 – Copie de conteneurs.

La méthode `.items()` vue dans le chapitre 8 *Dictionnaires et tuples* est particulièrement bien adaptée pour créer un dictionnaire de compréhension, car elle permet d'itérer en même temps sur les clés et valeurs d'un dictionnaire.

Avec un dictionnaire de compréhension, on peut rapidement compter le nombre de chaque base dans une séquence d'ADN :

```
>>> sequence = "atctcgatcgatcgcgctagctagctcgccatacgtacgactacgt"
>>> {base:seq.count(base) for base in set(sequence)}
{'a': 10, 'g': 10, 't': 11, 'c': 15}
```

De manière générale, tout objet sur lequel on peut faire une double itération du type `for var1, var2 in obj` est utilisable pour créer un dictionnaire de compréhension. Si vous souhaitez aller plus loin, vous pouvez consulter cet article sur le site *Datacamp*[7].

Il est également possible de générer des *sets* de compréhension sur le même modèle que les listes de compréhension :

```
>>> {i for i in range(10)}
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> {i**2 for i in range(10)}
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
>>>
>>> animaux = (("singe", 3), ("girafe", 4), ("rhinocéros", 2))
>>> {ani for ani, _ in animaux}
{'girafe', 'singe', 'rhinocéros'}
```

14.7 Module *collections*

Le module *collections*[8] contient d'autres types de conteneurs qui peuvent se révéler utiles, c'est une véritable mine d'or ! Nous n'aborderons pas tous ces objets ici, mais nous pouvons citer tout de même certains d'entre eux si vous souhaitez aller un peu plus loin :

- Les dictionnaires ordonnés[9], qui se comportent comme les dictionnaires classiques, mais qui sont ordonnés, c'est-à-dire que si on les affiche ou on itère dessus, l'ordre sera le même que celui utilisé pour sa création. Avant la version

7. https://www.datacamp.com/community/tutorials/python-dictionary-comprehension
8. https://docs.python.org/fr/3/library/collections.html
9. https://docs.python.org/fr/3/library/collections.html#collections.OrderedDict

3.6 de Python, ces dictionnaires ordonnés avaient un intérêt, car l'ordre des dictionnaires normaux était arbitraire. Désormais, les dictionnaires normaux se comportent presque en tout point comme les dictionnaires ordonnés.

- Les *defaultdicts*[10], qui génèrent des valeurs par défaut quand on demande une clé qui n'existe pas (cela évite que Python génère une erreur).
- Les compteurs[11], dont un exemple est présenté ci-dessous.
- Les *namedtuples*[12], que nous évoquerons au chapitre 24 *Avoir plus la classe avec les objets* (en ligne).

L'objet `collection.Counter()` est particulièrement intéressant et simple à utiliser. Il crée des compteurs à partir d'objets itérables, par exemple :

```
>>> import collections
>>> compo_seq = collections.Counter("aatctccgatcgatcgatcgatgatc")
>>> compo_seq
Counter({'a': 7, 't': 7, 'c': 7, 'g': 5})
>>> type(compo_seq)
<class 'collections.Counter'>
>>> compo_seq["a"]
7
>>> compo_seq["n"]
0
```

Dans cet exemple, Python a automatiquement compté chaque caractère `a`, `t`, `g` et `c` de la chaîne de caractères passée en argument. Cela crée un objet de type `Counter` qui se comporte ensuite comme un dictionnaire, à une exception près : si on appelle une clé qui n'existe pas dans l'itérable initiale (comme le `n` ci-dessus), la valeur renvoyée est 0.

14.8 Exercices

Conseil

Pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

14.8.1 Séquence peptidique et dictionnaire

Les numéros d'acides aminés commencent rarement à 1 dans les fichiers PDB. Créez un dictionnaire où chaque clé est un numéro de résidu de 3 à 9, et chaque valeur est un acide aminé de la séquence peptidique suivante : `SEQPEPT`. Utilisez pour cela les fonctions `dict()` et `zip()`.

14.8.2 Composition en acides aminés

En utilisant un *set* et la méthode `.count()` des chaînes de caractères, déterminez le nombre d'occurrences de chaque acide aminé dans la séquence

`AGWPSGGASAGLAILWGASAIMPGALW`.

14.8.3 Mots de deux et trois lettres dans une séquence d'ADN

Créez une fonction `compte_mots_2_lettres()`, qui prend comme argument une séquence sous la forme d'une chaîne de caractères et qui renvoie tous les mots de deux lettres qui existent dans la séquence sous la forme d'un dictionnaire. Par exemple pour la séquence `ACCTAGCCCTA`, le dictionnaire renvoyée serait :

`{'AC': 1, 'CC': 3, 'CT': 2, 'TA': 2, 'AG': 1, 'GC': 1}`

Créez une nouvelle fonction `compte_mots_3_lettres()`, qui a un comportement similaire à `compte_mots_2_lettres()`, mais avec des mots de trois lettres.

Utilisez ces fonctions pour afficher les mots de deux et trois lettres et leurs occurrences trouvés dans la séquence d'ADN :

`ACCTAGCCATGTAGAATCGCCTAGGCTTTAGCTAGCTCTAGCTAGCTG`

Voici un exemple de sortie attendue :

10. https://docs.python.org/fr/3/library/collections.html#collections.defaultdict
11. https://docs.python.org/fr/3/library/collections.html#collections.Counter
12. https://docs.python.org/fr/3/library/collections.html#collections.namedtuple

```
Mots de 2 lettres
AC : 1
CC : 3
CT : 8
[...]
Mots de 3 lettres
ACC : 1
CCT : 2
CTA : 5
[...]
```

14.8.4 Mots de deux lettres dans la séquence du chromosome I de *Saccharomyces cerevisiae*

Créez une fonction `lit_fasta()` qui prend comme argument le nom d'un fichier FASTA sous la forme d'une chaîne de caractères, lit la séquence dans le fichier FASTA et la renvoie sous la forme d'une chaîne de caractères. Inspirez-vous d'un exercice similaire du chapitre 10 *Plus sur les chaînes de caractères*.

Utilisez cette fonction et la fonction `compte_mots_2_lettres()` de l'exercice précédent pour extraire les mots de deux lettres et leurs occurrences dans la séquence du chromosome I de la levure du boulanger *Saccharomyces cerevisiae* (fichier `NC_001133.fna`[13]).

Le génome complet est fourni au format FASTA. Vous trouverez des explications sur ce format et des exemples de code dans l'annexe A *Quelques formats de données en biologie*.

14.8.5 Mots de *n* lettres dans un fichier FASTA

Créez un script `extract-words.py` qui prend comme arguments le nom d'un fichier FASTA suivi d'un entier compris entre 1 et 4. Ce script doit extraire du fichier FASTA tous les mots et leurs occurrences, en fonction du nombre de lettres passé en option.

Utilisez pour ce script la fonction `lit_fasta()` de l'exercice précédent. Créez également la fonction `compte_mots_n_lettres()` qui prend comme argument une séquence sous la forme d'une chaîne de caractères et le nombre de lettres des mots sous la forme d'un entier.

Testez ce script avec :

- la séquence du chromosome I de la levure du boulanger *Saccharomyces cerevisiae* (fichier `NC_001133.fna`[14]) ;
- le génome de la bactérie *Escherichia coli* (fichier `NC_000913.fna`[15]).

Les deux fichiers sont au format FASTA.

Cette méthode vous paraît-elle efficace sur un génome assez gros comme celui d'*Escherichia coli* ?

14.8.6 Atomes carbone alpha d'un fichier PDB

Téléchargez le fichier `1bta.pdb`[16], qui correspond à la structure tridimensionnelle de la protéine barstar[17] sur le site de la *Protein Data Bank* (PDB).

Créez la fonction `trouve_calpha()`, qui prend en argument le nom d'un fichier PDB (sous la forme d'une chaîne de caractères), qui sélectionne uniquement les lignes contenant des carbones alpha et qui les renvoie sous la forme d'une liste de dictionnaires. Chaque dictionnaire contient quatre clés :

- le numéro du résidu (`resid`) avec une valeur entière,
- la coordonnée atomique x (`x`) avec une valeur *float*,
- la coordonnée atomique y (`y`) avec une valeur *float*,
- la coordonnée atomique z (`z`) avec une valeur *float*.

Utilisez la fonction `trouve_calpha()` pour afficher à l'écran le nombre total de carbones alpha de la barstar ainsi que les coordonnées atomiques des carbones alpha des deux premiers résidus (acides aminés).

Conseil

13. https://python.sdv.u-paris.fr/data-files/NC_001133.fna
14. https://python.sdv.u-paris.fr/data-files/NC_001133.fna
15. https://python.sdv.u-paris.fr/data-files/NC_000913.fna
16. https://files.rcsb.org/download/1BTA.pdb
17. http://www.rcsb.org/pdb/explore.do?structureId=1BTA

Vous trouverez des explications sur le format PDB et des exemples de code pour lire ce type de fichier en Python dans l'annexe A *Quelques formats de données en biologie*.

14.8.7 Barycentre d'une protéine (exercice +++)

Téléchargez le fichier `1bta.pdb`[18] qui correspond à la structure tridimensionnelle de la protéine barstar[19] sur le site de la *Protein Data Bank* (PDB).

Un carbone alpha est présent dans chaque résidu (acide aminé) d'une protéine. On peut obtenir une bonne approximation du barycentre d'une protéine en calculant le barycentre de ses carbones alpha.

Le barycentre G de coordonnées (G_x, G_y, G_z) est obtenu à partir des n carbones alpha (CA) de coordonnées (CA_x, CA_y, CA_z) avec :

$$G_x = \frac{1}{n}\sum_{i=1}^{n}\mathrm{CA}_{i,x}$$

$$G_y = \frac{1}{n}\sum_{i=1}^{n}\mathrm{CA}_{i,y}$$

$$G_z = \frac{1}{n}\sum_{i=1}^{n}\mathrm{CA}_{i,z}$$

Créez une fonction `calcule_barycentre()`, qui prend comme argument une liste de dictionnaires dont les clés (`resid`, `x`, `y` et `z`) sont celles de l'exercice précédent et qui renvoie les coordonnées du barycentre sous la forme d'une liste de *floats*.

Utilisez la fonction `trouve_calpha()` de l'exercice précédent et la fonction `calcule_barycentre()` pour afficher, avec deux chiffres significatifs, les coordonnées du barycentre des carbones alpha de la barstar.

14.8.8 Kinases et protéines humaines

Nous avons extrait de la base de données de protéines UniProt la liste des protéines humaines (dans le fichier `human_proteins.txt`[20]) et la liste des kinases (dans le fichier `kinases_proteins.txt`[21]), qui sont une famille de protéines enzymatiques[22] impliquées dans la phosphorylation d'autres protéines.

Chaque fichier contient un identifiant de protéine par ligne. Voici un exemple pour le fichier `human_proteins.txt` :

```
A0A087X1C5
A0A0B4J2F0
A0A0B4J2F2
A0A0C5B5G6
A0A0K2S4Q6
A0A0U1RRE5
A0A1B0GTW7
A0AV02
A0AV96
[...]
```

L'objectif de cet exercice est de déterminer quelles sont les protéines humaines qui sont des kinases. Chaque liste de protéines contenant plusieurs milliers d'éléments, il n'est pas possible de la faire à la main. Vous aller utiliser Python et les *sets* pour cela.

1. Créez un script `compare_proteins.py`.
2. Dans ce script, créez une fonction `read_protein_file()` qui prend en argument le nom d'un fichier de protéines sous la forme d'une chaîne de caractères et qui renvoie un *set* contenant la liste des identifiants des protéines contenues dans le fichier passé en argument.

18. https://files.rcsb.org/download/1BTA.pdb
19. http://www.rcsb.org/pdb/explore.do?structureId=1BTA
20. https://python.sdv.u-paris.fr/data-files/human_proteins.txt
21. https://python.sdv.u-paris.fr/data-files/kinase_proteins.txt
22. https://fr.wikipedia.org/wiki/Kinase

3. Affichez ensuite le nombre de protéines listées dans chaque fichier.
4. En utilisant uniquement des opérations sur les *sets*, déterminez et affichez :
- le nombre de protéines humaines qui sont des kinases ;
- le nombre de protéines humaines qui ne sont pas des kinases ;
- le nombre de kinases qui ne sont pas des protéines humaines.

CHAPITRE 15

Création de modules

15.1 Pourquoi créer ses propres modules ?

Dans le chapitre 9 *Modules*, nous avons découvert quelques modules Python comme *random*, *math*, etc. Nous avons vu par ailleurs dans les chapitres 10 *Fonctions* et 13 *Plus sur les fonctions* que les fonctions sont utiles pour réutiliser une fraction de code plusieurs fois au sein d'un même programme, sans avoir à dupliquer ce code. On peut imaginer qu'une fonction utile pourrait être judicieusement réutilisée dans un autre programme Python. C'est justement l'intérêt de créer un module. On y regroupe un ensemble de fonctions que l'on peut être amené à utiliser souvent. En général, les modules sont regroupés autour d'un thème précis. Par exemple, on pourrait concevoir un module d'analyse de séquences biologiques ou encore de gestion de fichiers PDB.

15.2 Création d'un module

En Python, la création d'un module est très simple. Il suffit d'écrire un ensemble de fonctions (et éventuellement de constantes) dans un fichier, puis d'enregistrer ce dernier avec une extension `.py` (comme n'importe quel script Python). À titre d'exemple, nous allons créer un module simple que nous enregistrerons sous le nom `message.py` :

```
"""Module inutile qui affiche des messages."""

DATE = "2024-01-05"


def bonjour(nom):
    """Dit Bonjour."""
    return f"Bonjour {nom}"


def ciao(nom):
    """Dit Ciao."""
    return f"Ciao {nom}"


def hello(nom):
    """Dit Hello."""
    return f"Hello {nom}"
```

Les chaînes de caractères entre triple guillemets en tête du module et en tête de chaque fonction sont facultatives mais elles jouent néanmoins un rôle essentiel dans la documentation du code.

Remarque

Une constante est, par définition, une variable dont la valeur n'est pas modifiée. Par convention, en Python, le nom des constantes est écrit en majuscules (comme `DATE` dans notre exemple).

15.3 Utilisation de son propre module

Pour appeler une fonction ou une variable de ce module, il faut que le fichier `message.py` soit dans le répertoire courant (dans lequel on travaille) ou bien dans un répertoire listé par la variable d'environnement `PYTHONPATH` de votre système d'exploitation. Ensuite, il suffit d'importer le module et toutes ses fonctions (et constantes) vous sont alors accessibles.

Remarque

Avec Mac OS X et Linux, il faut taper la commande suivante depuis un *shell* Bash pour modifier la variable d'environnement `PYTHONPATH` :

```
export PYTHONPATH=$PYTHONPATH:/chemin/vers/mon/super/module
```

Avec Windows, mais depuis un *shell* PowerShell, il faut taper la commande suivante :

```
$env:PYTHONPATH += ";C:\chemin\vers\mon\super\module"
```

Une fois cette manipulation effectuée, vous pouvez contrôler que le chemin vers le répertoire contenant vos modules a bien été ajouté à la variable d'environnement `PYTHONPATH` :

- sous Mac OS X et Linux : `echo $PYTHONPATH`
- sous Windows : `echo $env:PYTHONPATH`

Le chargement du module se fait avec la commande `import message`. Notez que le fichier est bien enregistré avec une extension `.py`, pourtant on ne la précise pas lorsqu'on importe le module. Ensuite, on peut utiliser les fonctions comme avec un module classique :

```
>>> import message
>>> message.hello("Joe")
'Hello Joe'
>>> message.ciao("Bill")
'Ciao Bill'
>>> message.bonjour("Monsieur")
'Bonjour Monsieur'
>>> message.DATE
'2024-01-05'
```

Remarque

La première fois qu'un module est importé, Python crée un répertoire nommé `__pycache__` contenant un fichier avec une extension `.pyc` qui contient le bytecode[1], c'est-à-dire le code précompilé du module.

15.4 Les *docstrings*

Lorsqu'on écrit un module, il est important de créer de la documentation pour expliquer ce que fait le module et comment utiliser chaque fonction. Les chaînes de caractères entre triple guillemets, situées en début du module et de chaque fonction, sont là pour cela : on les appelle *docstrings* (« chaînes de documentation » en français). Les *docstrings* seront détaillées dans le chapitre 16 *Bonnes pratiques en programmation Python*.

Les *docstrings* permettent notamment de fournir de l'aide lorsqu'on invoque la commande `help()` :

1. https://docs.python.org/fr/3/glossary.html#term-bytecode

```
>>> import message
>>> help(message)

Help on module message:

NAME
    message - Module inutile qui affiche des messages.

FUNCTIONS
    bonjour(nom)
        Dit Bonjour.

    ciao(nom)
        Dit Ciao.

    hello(nom)
        Dit Hello.

DATA
    DATE = '2024-01-05'

FILE
    /home/pierre/message.py
```

Remarque

Pour quitter l'aide, pressez la touche `Q`.

Vous remarquez que Python a généré automatiquement cette page d'aide, tout comme il est capable de le faire pour les modules internes à Python (*random*, *math*, etc.) et ce grâce aux *docstrings*. Notez que l'on peut aussi appeler l'aide pour une seule fonction :

```
>>> help(message.ciao)

Help on function ciao in module message:

ciao(nom)
    Dit Ciao.
```

En résumé, les *docstrings* sont destinés aux utilisateurs du module. Leur but est différent des commentaires qui, eux, sont destinés à celui qui lit le code (pour en comprendre les subtilités). Une bonne *docstring* de fonction doit contenir tout ce dont un utilisateur a besoin pour utiliser cette fonction. Une liste minimale et non exhaustive serait :

- ce que fait la fonction,
- ce qu'elle prend en argument,
- ce qu'elle renvoie.

Pour en savoir plus sur les *docstrings* et comment les écrire, nous vous recommandons de lire le chapitre 16 *Bonnes pratiques en programmation Python*.

15.5 Visibilité des fonctions dans un module

La visibilité des fonctions au sein des modules suit des règles simples :

- Les fonctions dans un même module peuvent s'appeler les unes les autres.
- Les fonctions dans un module peuvent appeler des fonctions situées dans un autre module s'il a été préalablement importé. Par exemple, si la commande `import autremodule` est utilisée dans un module, il est possible d'appeler une fonction avec `autremodule.fonction()`.

Toutes ces règles viennent de la manière dont Python gère les **espaces de noms**. De plus amples explications sont données sur ce concept dans le chapitre 24 *Avoir plus la classe avec les objets* (en ligne).

15.6 Module ou script ?

Vous avez remarqué que notre module `message` ne contient que des fonctions et une constante. Si on l'exécutait comme un script classique, cela n'afficherait rien :

```
$ python message.py
$
```

Cela s'explique par l'absence de programme principal, c'est-à-dire de lignes de code que l'interpréteur exécute lorsqu'on lance le script.

À l'inverse, que se passe-t-il si on importe un script en tant que module alors qu'il contient un programme principal avec des lignes de code ? Prenons par exemple le script `message2.py` suivant :

```
1  """Script de test."""
2
3
4  def bonjour(nom):
5      """Dit Bonjour."""
6      return f"Bonjour {nom}"
7
8
9  # Programme principal.
10 print(bonjour("Joe"))
```

Si on l'importe dans l'interpréteur, on obtient :

```
1  >>> import message2
2  Bonjour Joe
```

Ceci n'est pas le comportement voulu pour un module, car on n'attend pas d'affichage particulier lors de son chargement. Par exemple la commande `import math` n'affiche rien dans l'interpréteur.

Afin de pouvoir utiliser un code Python en tant que module ou en tant que script, nous vous conseillons la structure suivante :

```
1  """Script de test."""
2
3
4  def bonjour(nom):
5      """Dit Bonjour."""
6      return f"Bonjour {nom}"
7
8
9  if __name__ == "__main__":
10     print(bonjour("Joe"))
```

À la ligne 9, l'instruction `if __name__ == "__main__":` indique à Python :

- Si le programme `message2.py` est exécuté en tant que script dans un *shell*, le résultat du test `if` sera alors `True` et le bloc d'instructions correspondant (ligne 10) sera exécuté :

  ```
  $ python message2.py
  Bonjour Joe
  ```

- Si le programme `message2.py` est importé en tant que module, le résultat du test `if` sera alors `False` et le bloc d'instructions correspondant ne sera pas exécuté :

  ```
  1  >>> import message2
  2  >>>
  ```

Ce comportement est possible grâce à la gestion des espaces de noms par Python (pour plus de détail, consultez le chapitre 24 *Avoir plus la classe avec les objets* (en ligne)). Au delà de la commodité de pouvoir utiliser votre script en tant que programme ou en tant que module, cela présente l'avantage de signaler clairement où se situe le programme principal quand on lit le code.

Conseil

Ainsi, au lieu d'ajouter le commentaire :

```
# Programme principal.
```

comme nous vous l'avions suggéré dans les chapitres 10 *Fonctions* et 13 *Plus sur les fonctions*, nous vous recommandons désormais d'utiliser la ligne :

```
if __name__ == "__main__":
```

15.7 Exercice

Conseil

Pour cet exercice, créez un script puis exécutez-le dans un *shell*.

15.7.1 Module ADN

Dans le script `adn.py`, construisez un module qui va contenir les fonctions et constantes suivantes.

- Fonction `lit_fasta()` : prend en argument un nom de fichier sous forme d'une chaîne de caractères et renvoie la séquence d'ADN lue dans le fichier sous forme d'une chaîne de caractères.
- Fonction `seq_alea()` : prend en argument une taille de séquence sous forme d'un entier et renvoie une séquence aléatoire d'ADN de la taille correspondante sous forme d'une chaîne de caractères.
- Fonction `comp_inv()` : prend en argument une séquence d'ADN sous forme d'une chaîne de caractères et renvoie la séquence complémentaire inverse (aussi sous forme d'une chaîne de caractères).
- Fonction `prop_gc()` : prend en argument une séquence d'ADN sous forme d'une chaîne de caractères et renvoie la proportion en GC de la séquence sous forme d'un *float*. Nous vous rappelons que la proportion de GC s'obtient comme la somme des bases Guanine (G) et Cytosine (C), divisée par le nombre total de bases (A, T, C, G).
- Constante `BASE_COMP` : dictionnaire qui contient la complémentarité des bases d'ADN (`A`→`T`, `T`→`C`, `G`→`C` et `C`→`G`). Ce dictionnaire sera utilisé par la fonction `comp_inv()`.

À la fin de votre script, proposez des exemples d'utilisation des fonctions que vous aurez créées. Ces exemples d'utilisation ne devront pas être exécutés lorsque le script est chargé comme un module.

Conseil

- Dans cet exercice, on supposera que toutes les séquences sont manipulées comme des chaînes de caractères en majuscules.
- Pour les fonctions `seq_alea()` et `comp_inv()`, n'hésitez pas à jeter un œil aux exercices correspondants dans le chapitre 12 *Plus sur les listes*.
- Voici un exemple de fichier FASTA `adn.fasta`[2] pour tester la fonction `lit_fasta()`.

2. https://python.sdv.u-paris.fr/data-files/adn.fasta

CHAPITRE 16

Bonnes pratiques en programmation Python

Comme vous l'avez constaté dans tous les chapitres précédents, la syntaxe de Python est très permissive. Afin d'uniformiser l'écriture de code en Python, la communauté des développeurs Python recommande un certain nombre de règles afin qu'un code soit lisible. Lisible par quelqu'un d'autre, mais également, et surtout, par soi-même. Essayez de relire un code que vous avez écrit « rapidement » il y a un mois, six mois ou un an. Si le code ne fait que quelques lignes, il se peut que vous vous y retrouviez, mais s'il fait plusieurs dizaines, voire centaines de lignes, vous serez perdus.

Dans ce contexte, le créateur de Python, Guido van Rossum, part d'un constat simple : « *code is read much more often than it is written* » (« le code est plus souvent lu qu'écrit »). Avec l'expérience, vous vous rendrez compte que cela est parfaitement vrai. Alors, plus de temps à perdre, voyons en quoi consistent ces bonnes pratiques.

Plusieurs choses sont nécessaires pour écrire un code lisible : la syntaxe, l'organisation du code, le découpage en fonctions (et possiblement en classes, que nous verrons dans le chapitre 23 *Avoir la classe avec les objets*), mais souvent, aussi, le bon sens. Pour cela, les « PEP » peuvent nous aider.

Définition

Afin d'améliorer le langage Python, la communauté qui développe Python publie régulièrement des *Python Enhancement Proposal*[1] (PEP), suivi d'un numéro. Il s'agit de propositions concrètes pour améliorer le code, ajouter de nouvelles fonctionnalités, mais aussi des recommandations sur la manière d'utiliser Python, bien écrire du code, etc.

On va aborder dans ce chapitre sans doute la plus célèbre des PEP, à savoir la PEP 8, qui est incontournable lorsque l'on veut écrire du code Python correctement.

Définition

On parle de code **pythonique** lorsque ce dernier respecte les règles d'écriture définies par la communauté Python, mais aussi les règles d'usage du langage.

1. https://www.python.org/dev/peps/

16.1 De la bonne syntaxe avec la PEP 8

La PEP 8 *Style Guide for Python Code*[2] est une des plus anciennes PEP (les numéros sont croissants avec le temps). Elle consiste en un nombre important de recommandations sur la syntaxe de Python. Il est vivement recommandé de lire la PEP 8 en entier au moins une fois pour avoir une bonne vue d'ensemble. On ne présentera ici qu'un rapide résumé de cette PEP 8.

16.1.1 Indentation

On a vu que l'indentation est obligatoire en Python pour séparer les blocs d'instructions. Cela vient d'un constat simple : l'indentation améliore la lisibilité d'un code. La PEP 8 recommande d'utiliser **quatre espaces** pour chaque niveau d'indentation. Nous vous recommandons de suivre impérativement cette règle.

Attention

Afin de toujours utiliser cette règle des quatre espaces pour l'indentation, il est essentiel de régler correctement votre éditeur de texte. Consultez pour cela l'annexe *Installation de Python* disponible en ligne[3]. Avant d'écrire la moindre ligne de code, faites en sorte que lorsque vous pressez la touche tabulation, cela ajoute quatre espaces (et non pas un caractère tabulation).

16.1.2 Importation des modules

Comme on l'a vu dans le chapitre 9 *Modules*, le chargement d'un module est réalisé avec l'instruction `import module` plutôt qu'avec `from module import *`.

Si on souhaite ensuite utiliser une fonction d'un module, la première syntaxe conduit à `module.fonction()`, ce qui rend explicite la provenance de la fonction. Avec la seconde syntaxe, il faudrait écrire `fonction()`, ce qui peut :

- mener à un conflit si une de vos fonctions a le même nom ;
- rendre difficile la recherche de documentation si on ne sait pas d'où vient la fonction, notamment si plusieurs modules sont chargés avec l'instruction
  `from module import *`

Par ailleurs, la première syntaxe définit un « espace de noms » spécifique au module (voir le chapitre 24 *Avoir plus la classe avec les objets* (en ligne)).

Dans un script Python, on importe un module par ligne. D'abord les modules internes (classés par ordre alphabétique), c'est-à-dire les modules de base de Python, puis les modules externes (ceux que vous avez installés en plus), et enfin, les modules que vous avez créés.

Si le nom du module est trop long, on peut utiliser un alias. L'instruction `from` est tolérée si vous n'importez que quelques fonctions clairement identifiées.

En résumé :

```
import module_interne_1
import module_interne_2
from module_interne_3 import fonction_spécifique

import module_externe_1
import module_externe_2_qui_a_un_nom_long as mod2

import module_cree_par_vous
```

16.1.3 Règles de nommage

Les noms de variables, de fonctions et de modules doivent être de la forme :

```
ma_variable
fonction_test_27()
mon_module
```

2. https://www.python.org/dev/peps/pep-0008/
3. https://python.sdv.u-paris.fr/livre-dunod

c'est-à-dire en minuscules avec un caractère « souligné » (« tiret du bas », ou *underscore* en anglais) pour séparer les différents « mots » dans le nom.

Les constantes sont écrites en majuscules :

```
MA_CONSTANTE
VITESSE_LUMIERE
```

Les noms de classes (voir le chapitre 23 *Avoir la classe avec les objets*) et les exceptions (voir le chapitre 26 *Remarques complémentaires* (en ligne)) sont de la forme :

```
MaClasse
MyException
```

Remarque

- Le style recommandé pour nommer les variables et les fonctions en Python est appelé *snake_case*. Il est différent du *CamelCase* utilisé pour les noms des classes et des exceptions.
- La variable _ est habituellement employée pour stocker des valeurs qui ne seront pas utilisées par la suite. Par exemple, dans le cas d'une affectation multiple, on peut utiliser _ pour stocker une valeur qui ne nous intéresse pas (voir chapitre 14 *Conteneurs*).

Pensez à donner à vos variables des noms qui ont du sens. Évitez autant que possible les `a1`, `a2`, `i`, `truc`, `toto`… Les noms de variables à un caractère sont néanmoins autorisés pour les indices dans les boucles :

```
>>> ma_liste = [1, 3, 5, 7, 9, 11]
>>> for i in range(len(ma_liste)):
...     print(ma_liste[i])
```

Bien sûr, une écriture plus « pythonique » de l'exemple précédent permet de se débarrasser de l'indice `i` :

```
>>> ma_liste = [1, 3, 5, 7, 9, 11]
>>> for entier in ma_liste:
...     print(entier)
```

Enfin, des noms de variable à une lettre peuvent être utilisés lorsque cela a un sens mathématique (par exemple, les noms `x`, `y` et `z` évoquent des coordonnées cartésiennes).

16.1.4 Gestion des espaces

La PEP 8 recommande d'entourer les opérateurs (`+`, `-`, `/`, `*`, `==`, `!=`, `>=`, `not`, `in`, `and`, `or`…) d'un espace avant et d'un espace après. Par exemple :

```
# Code recommandé :
ma_variable = 3 + 7
mon_texte = "souris"
mon_texte == ma_variable
# Code non recommandé :
ma_variable=3+7
mon_texte="souris"
mon_texte== ma_variable
```

Il n'y a, par contre, pas d'espace à l'intérieur de crochets, d'accolades et de parenthèses :

```
# Code recommandé :
ma_liste[1]
mon_dico{"clé"}
ma_fonction(argument)
# Code non recommandé :
ma_liste[ 1 ]
mon_dico{"clé" }
ma_fonction( argument )
```

Ni juste avant la parenthèse ouvrante d'une fonction ou le crochet ouvrant d'une liste ou d'un dictionnaire :

```
# Code recommandé :
ma_liste[1]
mon_dico{"clé"}
ma_fonction(argument)
# Code non recommandé :
ma_liste [1]
mon_dico {"clé"}
ma_fonction (argument)
```

On met un espace après les caractères `:` et `,` (mais pas avant) :

```
# Code recommandé :
ma_liste = [1, 2, 3]
mon_dico = {"clé1": "valeur1", "clé2": "valeur2"}
ma_fonction(argument1, argument2)
# Code non recommandé :
ma_liste = [1 , 2 ,3]
mon_dico = {"clé1" : "valeur1", "clé2":"valeur2"}
ma_fonction(argument1 ,argument2)
```

Par contre, pour les tranches de listes, on ne met pas d'espace autour du `:`

```
ma_liste = [1, 3, 5, 7, 9, 1]
# Code recommandé :
ma_liste[1:3]
ma_liste[1:4:2]
ma_liste[::2]
# Code non recommandé :
ma_liste[1 : 3]
ma_liste[1: 4:2 ]
ma_liste[ : :2]
```

Enfin, on n'ajoute pas plusieurs espaces autour du `=` ou des autres opérateurs pour faire joli :

```
# Code recommandé :
x1 = 1
x2 = 3
x_old = 5
# Code non recommandé :
x1    = 1
x2    = 3
x_old = 5
```

16.1.5 Longueur de ligne

Une ligne de code ne doit pas dépasser 79 caractères, pour des raisons tant historiques que de lisibilité.
On a déjà vu dans le chapitre 1 *Introduction* que le caractère `\` permet de couper des lignes trop longues. Par exemple :

```
>>> ma_variable = 3
>>> if ma_variable > 1 and ma_variable < 10 \
... and ma_variable % 2 == 1 and ma_variable % 3 == 0:
...     print(f"ma variable vaut {ma_variable}")
...
ma variable vaut 3
```

À l'intérieur de parenthèses, on peut revenir à la ligne sans utiliser le caractère `\`. C'est particulièrement utile pour préciser les arguments d'une fonction ou d'une méthode, lors de sa création ou lors de son utilisation :

```
>>> def ma_fonction(argument_1, argument_2,
...                 argument_3, argument_4):
...     return argument_1 + argument_2
...
>>> ma_fonction("texte très long", "tigre",
...             "singe", "souris")
'texte très longtigre'
```

Les parenthèses sont également très pratiques, pour répartir sur plusieurs lignes une chaîne de caractères qui sera

ensuite affichée sur une seule ligne :

```
>>> print("ATGCGTACAGTATCGATAAC"
...       "ATGACTGCTACGATCGGATA"
...       "CGGGTAACGCCATGTACATT")
ATGCGTACAGTATCGATAACATGACTGCTACGATCGGATACGGGTAACGCCATGTACATT
```

Notez qu'il n'y a pas d'opérateur + pour concaténer les trois chaînes de caractères, et que celles-ci ne sont pas séparées par des virgules. À partir du moment où elles sont entre parenthèses, Python les concatène automatiquement.

On peut aussi utiliser les parenthèses pour évaluer un expression trop longue :

```
>>> ma_variable = 3
>>> if (ma_variable > 1 and ma_variable < 10
... and ma_variable % 2 == 1 and ma_variable % 3 == 0):
...     print(f"ma variable vaut {ma_variable}")
...
ma variable vaut 3
```

Remarque

Les parenthèses sont aussi très utiles lorsqu'on a besoin d'enchaîner des méthodes les unes à la suite des autres. Cette technique du *method chaining* a été introduite dans le chapitre 11 *Plus sur les chaînes de caractères* et sera très utilisée dans le chapitre 22 *Module Pandas*.

Enfin, il est possible de créer des listes ou des dictionnaires sur plusieurs lignes, en sautant une ligne après une virgule :

```
>>> ma_liste = [1, 2, 3,
...             4, 5, 6,
...             7, 8, 9]
>>> mon_dico = {"clé1": 13,
...             "clé2": 42,
...             "clé3": -10}
```

16.1.6 Lignes vides

Dans un script, les lignes vides sont utiles pour séparer visuellement les différentes parties du code.

Il est recommandé de laisser deux lignes vides avant la définition d'une fonction ou d'une classe, et de laisser une seule ligne vide avant la définition d'une méthode (dans une classe).

On peut aussi laisser une ligne vide dans le corps d'une fonction pour séparer les sections logiques de la fonction, mais cela est à utiliser avec parcimonie.

16.1.7 Commentaires

Les commentaires débutent toujours par le symbole # suivi d'un espace. Ils fournissent des explications sur l'utilité du code et permettent de comprendre son fonctionnement.

Les commentaires sont sur le même niveau d'indentation que le code qu'ils commentent. Les commentaires sont constitués de phrases complètes, avec une majuscule au début (sauf si le premier mot est une variable qui s'écrit sans majuscule) et un point à la fin.

La PEP 8 recommande d'écrire les commentaires en anglais, sauf si vous êtes absolument certains que votre code ne sera lu que par des francophones. Dans la mesure où vous allez souvent développer des programmes scientifiques, nous vous conseillons d'écrire vos commentaires en anglais.

Soyez également cohérent entre la langue utilisée pour les commentaires et la langue utilisée pour nommer les variables. Pour un programme scientifique, les commentaires et les noms de variables sont en anglais. Ainsi `ma_liste` deviendra `my_list` et `ma_fonction` deviendra `my_function` (par exemple).

Les commentaires qui suivent le code sur la même ligne sont à éviter le plus possible et doivent être séparés du code par au moins deux espaces :

```
var_x = number / total * 100   # My useful comment.
```

Remarque

La PEP 8 ne fournit pas de recommandation[4] quant à l'usage de guillemets simples ou de guillemets doubles pour déclarer une chaîne de caractères.

```
>>> var_1 = "Ma chaîne de caractères"
>>> var_1
'Ma chaîne de caractères'
>>> var_2 = 'Ma chaîne de caractères'
>>> var_2
'Ma chaîne de caractères'
>>> var_1 == var_2
True
```

Vous constatez dans l'exemple ci-dessus que, pour Python, les guillemets simples et doubles sont équivalents. Nous vous conseillons cependant d'utiliser les **guillemets doubles** car ceux-ci sont, de notre point de vue, plus lisibles.

16.2 Les *docstrings* et la PEP 257

Les *docstrings*, que l'on pourrait traduire par « chaînes de documentation » en français, sont un élément essentiel des programmes Python, comme on l'a vu au chapitre 15 *Création de modules*. À nouveau, les développeurs de Python ont émis des recommandations dans la PEP 8, et plus exhaustivement dans la PEP 257[5], sur la manière de rédiger correctement les *docstrings*. En voici un résumé succinct.

16.2.1 Les principales règles

De manière générale, écrivez des *docstrings* pour les modules, les fonctions, les classes et les méthodes que vous développez.

Lorsque l'explication est courte et compacte, comme dans certaines fonctions ou méthodes simples, utilisez des *docstrings* d'une ligne :

```
"""Docstring simple d'une ligne se finissant par un point."""
```

Lorsque vous avez besoin de décrire plus en détail un module, une fonction, une classe ou une méthode, utilisez une *docstring* sur plusieurs lignes :

```
"""Docstring de plusieurs lignes, la première ligne est un résumé.

Après avoir sauté une ligne, on décrit les détails de cette docstring.
On termine la docstring avec les triples guillemets
sur la ligne suivante.
"""
```

Remarque

La PEP 257 recommande d'écrire des *docstrings* avec trois doubles guillemets, c'est-à-dire :

```
"""Ceci est une docstring recommandée."""
```

mais pas :

```
'''Ceci n'est pas une docstring recommandée.'''
```

Comme indiqué dans le chapitre 15 *Création de modules*, n'oubliez pas que les *docstrings* sont destinées aux utilisateurs des modules, fonctions, méthodes et classes que vous avez développés. Les éléments essentiels pour les fonctions et les méthodes sont :

1. ce que fait la fonction ou la méthode,
2. ce qu'elle prend en argument,
3. ce qu'elle renvoie.

4. https://peps.python.org/pep-0008/#string-quotes
5. https://www.python.org/dev/peps/pep-0257/

Pour les modules et les classes, on ajoute également des informations générales sur leur fonctionnement.

Pour autant, la PEP 257 ne dit pas explicitement comment organiser les *docstrings* pour les fonctions et les méthodes. Pour répondre à ce besoin, deux solutions ont émergées :

- la solution Google avec le *Google Style Python Docstrings*[6],
- la solution *NumPy* avec le *NumPy Style Python Docstrings*[7]. *NumPy* est un module complémentaire à Python, très utilisé en analyse de données et dont on parlera dans le chapitre 20.

16.2.2 Un exemple concret

On illustre ici la solution de *docstrings NumPy* pour des raisons de goût personnel. Sentez-vous libre d'explorer la proposition de Google.

Voici un exemple pour une fonction qui prend en argument deux entiers et qui renvoie leur produit :

```
1  def multiplie_nombres(nombre1, nombre2):
2      """Multiplication de deux nombres entiers.
3
4      Cette fonction ne sert pas à grand chose.
5
6      Parameters
7      ----------
8      nombre1 : int
9          Le premier nombre entier.
10     nombre2 : int
11         Le second nombre entier,
12         très important pour cette fonction.
13
14     Returns
15     -------
16     int
17         Le produit des deux nombres.
18     """
19     return nombre1 * nombre2
```

- **Lignes 6 et 7.** La section `Parameters` précise les paramètres de la fonction. Les tirets sur la ligne 7 soulignent le nom de la section pour la rendre visible.
- **Lignes 8 et 9.** On indique le nom et le type du paramètre, séparés par le caractère deux-points. Le type n'est pas obligatoire. En dessous, on indique une description du paramètre en question. La description est indentée.
- **Lignes 10 à 12.** Même chose pour le second paramètre. La description du paramètre peut s'étaler sur plusieurs lignes.
- **Lignes 14 et 15.** La section `Returns` indique ce qui est renvoyé par la fonction (le cas échéant).
- **Lignes 16 et 17.** La mention du type renvoyé est obligatoire. En dessous, on indique une description de ce qui est renvoyé par la fonction. Cette description est aussi indentée.

Attention

L'être humain a une fâcheuse tendance à la procrastination (le fameux « Bah je le ferai demain…») et écrire de la documentation peut être un sérieux motif de procrastination. Soyez vigilant sur ce point, et rédigez vos *docstrings* au moment où vous écrivez vos modules, fonctions, classes ou méthodes. Passer une journée (voire plusieurs) à écrire les *docstrings* d'un gros projet est particulièrement pénible. Croyez-nous !

16.3 Outils de contrôle qualité du code

Pour évaluer la qualité d'un code Python, c'est-à-dire sa conformité avec les recommandations de la PEP 8 et de la PEP 257, on peut utiliser les outils `pycodestyle`, `pydocstyle` et `pylint`.

Ces outils ne sont fournis dans l'installation de base de Python et doivent être installés sur votre machine. Avec la distribution Miniconda, cette étape d'installation se résume à une ligne de commande :

6. https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html
7. https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_numpy.html

```
$ conda install -c conda-forge pycodestyle pydocstyle pylint
```

Définition

Les outils `pycodestyle`, `pydocstyle` et `pylint` sont des **linters**, c'est-à-dire des programmes qui vont chercher les sources potentielles d'erreurs dans un code informatique. Ces erreurs peuvent être des erreurs de style (PEP 8 et 257) ou des erreurs logiques (manipulation d'une variable, chargement de module).

Voici le contenu du script `script_quality_not_ok.py`[8] que nous allons analyser par la suite :

```
"""Un script de multiplication.
"""

import os

def Multiplie_nombres(nombre1,nombre2 ):
    """Multiplication de deux nombres entiers
    Cette fonction ne sert pas à grand chose.

    Parameters
    ----------
    nombre1 : int
        Le premier nombre entier.
    nombre2 : int
        Le second nombre entier.
        Très utile.

    Returns
    -------
    int
        Le produit des deux nombres.

    """
    return nombre1 *nombre2


if __name__ == "__main__":
    print(f"2 x 3 = {Multiplie_nombres(2, 3)}")
    print (f"4 x 5 = {Multiplie_nombres(4, 5)}")
```

Ce script est d'ailleurs parfaitement fonctionnel :

```
$ python script_quality_not_ok.py
2 x 3 = 6
4 x 5 = 20
```

On va tout d'abord vérifier la conformité avec la PEP 8 grâce à l'outil `pycodestyle` :

```
$ pycodestyle script_quality_not_ok.py
script_quality_not_ok.py:6:1: E302 expected 2 blank lines, found 1
script_quality_not_ok.py:6:30: E231 missing whitespace after ','
script_quality_not_ok.py:6:38: E202 whitespace before ')'
script_quality_not_ok.py:26:21: E225 missing whitespace around operator
script_quality_not_ok.py:31:10: E211 whitespace before '('
```

- **Ligne 2.** Le bloc `script_quality_not_ok.py:6:1:` désigne le nom du script (`script_quality_not_ok.py`), le numéro de la ligne (6) et le numéro de la colonne (1) où se trouve la non-conformité avec la PEP 8. Ensuite, `pycodestyle` fournit un code et un message explicatif. Ici, il faut deux lignes vides avant la fonction `Multiplie_nombres()`.
- **Ligne 3.** Il manque un espace après la virgule qui sépare les arguments `nombre1` et `nombre2` dans la définition de la fonction `Multiplie_nombres()` à la ligne 6 (colonne 30) du script.

8. https://python.sdv.u-paris.fr/data-files/script_quality_not_ok.py

- **Ligne 4.** Il y un espace de trop après le second argument `nombre2` dans la définition de la fonction `Multiplie_nombres()` à la ligne 6 (colonne 38) du script.
- **Ligne 5.** Il manque un espace après l'opérateur `*` à la ligne 26 (colonne 21) du script.
- **Ligne 6.** Il y a un espace de trop entre `print` et `(` à la ligne 31 (colonne 10) du script.

Assez curieusement, `pycodestyle` n'a pas détecté que le nom de la fonction `Multiplie_nombres()` ne respecte pas la convention de nommage (pas de majuscule).

Ensuite, l'outil `pydocstyle` va vérifier la conformité avec la PEP 257 et s'intéresser particulièrement aux *docstrings* :

```
$ pydocstyle script_quality_not_ok.py
script_quality_not_ok.py:1 at module level:
        D200: One-line docstring should fit on one line with quotes (found 2)
script_quality_not_ok.py:7 in public function `Multiplie_nombres`:
        D205: 1 blank line required between summary line and description (found 0)
script_quality_not_ok.py:7 in public function `Multiplie_nombres`:
        D400: First line should end with a period (not 's')
```

- **Lignes 2 et 3.** `pydocstyle` indique que la *docstring* à la ligne 1 du script est sur deux lignes, alors qu'elle devrait être sur une seule ligne.
- **Lignes 4 et 5.** Dans la *docstring* de la fonction `Multiplie_nombres()` (ligne 7 du script), il manque une ligne vide entre la ligne résumé et la description plus complète.
- **Lignes 6 et 7.** Dans la *docstring* de la fonction `Multiplie_nombres()` (ligne 7 du script), il manque un point à la fin de la première ligne.

Les outils `pycodestyle` et `pydocstyle` vont simplement vérifier la conformité aux PEP 8 et 257. L'outil `pylint` va lui aussi vérifier une partie de ces règles mais il va également essayer de comprendre le contexte du code et proposer des éléments d'amélioration. Par exemple :

```
$ pylint script_quality_not_ok.py
************* Module script_quality_not_ok
script_quality_not_ok.py:6:0: C0103: Function name "Multiplie_nombres"
doesn't conform to snake_case naming style (invalid-name)
script_quality_not_ok.py:4:0: W0611: Unused import os (unused-import)

--------------------------------------------------------------------
Your code has been rated at 6.67/10
```

- **Lignes 3 et 4.** `pylint` indique que nom de la fonction `Multiplie_nombres()` ne respecte pas la convention PEP 8 (ligne 6 du script).
- **Ligne 5.** Le module `os` est chargé mais pas utilisé (ligne 4 du script).
- **Ligne 8.** `pylint` produit également une note sur 10. Ne soyez pas surpris si cette note est très basse (voire négative) la première fois que vous analysez votre script avec `pylint`. Cet outil fournit de nombreuses suggestions d'amélioration et la note attribuée à votre script devrait rapidement augmenter. Pour autant, la note de 10 est parfois difficile à obtenir. Ne soyez pas trop exigeant.

Une version améliorée du script précédent est disponible en ligne[9].

16.4 Outil de formatage automatique du code

Se souvenir de toutes les règles PEP 8 est fastidieux. Il existe des outils pour formater automatiquement le code Python pour qu'il soit conforme à la PEP 8. L'outil le plus connu est `black`.

Cet outil n'est pas fourni dans l'installation de base de Python et doit être installé sur votre machine. Avec la distribution Miniconda, cette étape d'installation se résume à une ligne de commande :

```
$ conda install -c conda-forge black
```

Voici un exemple d'utilisation :

```
$ black script_quality_not_ok.py
reformatted script_quality_not_ok.py

All done!
1 file reformatted.
```

9. https://python.sdv.u-paris.fr/data-files/script_quality_ok.py

Le script `script_quality_not_ok.py` a été modifié pour être conforme à la PEP 8, ce qu'on peut vérifier avec `pycodestyle` :

```
$ pycodestyle script_quality_not_ok.py
```

qui ne renvoie aucune erreur.

`black` peut modifier votre code de manière significative. Il est donc recommandé de l'utiliser avec l'option `--diff` au préalable pour afficher les modifications apportées. Par exemple, avec le programme `script_quality_not_ok.py` qui n'aurait pas été modifié :

```
$ black --diff script_quality_not_ok.py
--- script_quality_not_ok.py    2024-02-05 12:07:04.851491+00:00
+++ script_quality_not_ok.py    2024-02-05 12:07:10.418009+00:00
@@ -1,11 +1,12 @@
 """Un script de multiplication.
 """

 import os

-def Multiplie_nombres(nombre1,nombre2 ):
+
+def Multiplie_nombres(nombre1, nombre2):
[...]
```

Conseil

`black` est très pratique. N'hésitez pas à l'utiliser pour formater automatiquement votre code.

Attention

- `black` ne fait qu'une entorse à la PEP 8 : il autorise des longueurs de lignes jusqu'à 88 caractères. Si vous souhaitez respecter strictement la PEP 8, utilisez l'option `--line-length 79`.
- `black` se limite à la PEP 8. Il ne vérifie pas la conformité avec la PEP 257 ni la qualité du code (imports inutiles, etc.). Utilisez toujours `pydocstyle` et `pylint` en complément.

16.5 Organisation du code

Il est important de toujours structurer son code de la même manière. Ainsi, on sait tout de suite où trouver l'information et un autre programmeur pourra s'y retrouver. Voici un exemple de code avec les différents éléments dans le bon ordre :

```
1  """Docstring d'une ligne décrivant brièvement ce que fait le programme.
2
3  Usage:
4  ======
5      python nom_de_ce_super_script.py argument1 argument2
6
7      argument1: un entier signifiant un truc
8      argument2: une chaîne de caractères décrivant un bidule
9  """
10
11 __authors__ = ("Johny B Good", "Hubert de la Pâte Feuilletée")
12 __contact__ = ("johny@bgood.us", "hub@pate.feuilletee.fr")
13 __copyright__ = "MIT"
14 __date__ = "2030-01-01"
15 __version__ = "1.2.3"
16
17 import module_interne
18 import module_interne_2 as mod2
19
20 import module_externe
21
22 import my_module
23
24 UNE_CONSTANTE = valeur
25
26
27 def une_fonction_complexe(arg1, arg2, arg3):
28     """Résumé de la docstring décrivant la fonction.
29
30     Description détaillée.
31     """
32     [...]
33     return une_chose
34
35
36 def une_fonction_simple(arg1, arg2):
37     """Docstring d'une ligne décrivant la fonction."""
38     [...]
39     return autre_chose
40
41
42 if __name__ == "__main__":
43     # Ici débute le programme principal.
44     [...]
```

- **Lignes 1 à 9.** Cette *docstring* décrit globalement le script. Cette *docstring* (ainsi que les autres) seront visibles si on importe le script en tant que module, puis en invoquant la commande `help()` (voir chapitre 15 *Création de modules*).
- **Lignes 11 à 15.** On définit ici un certain nombre de variables avec des doubles *underscores* donnant quelques informations sur la version du script, les auteurs, etc. Il s'agit de métadonnées que la commande `help()` pourra afficher. Ces métadonnées sont utiles lorsque le code est distribué à la communauté.
- **Lignes 17 à 22.** Importation des modules. D'abord les modules internes à Python (fournis en standard), puis les modules externes (ceux qu'il faut installer en plus), puis les modules créés localement. Un module par ligne.
- **Ligne 24.** Définition des constantes. Le nom des constantes est en majuscule.
- **Lignes 27 à 39.** Définition des fonctions. Avant chaque fonction, on laisse deux lignes vides.
- **Lignes 42 à 44.** On écrit le programme principal. Le test ligne 42 n'est vrai que si le script est utilisé en tant que programme.

16.6 Conseils sur la conception d'un script

Voici quelques conseils pour vous aider à concevoir un script Python.

- **Réfléchissez** avec un papier, un crayon… et un cerveau (voire même plusieurs) ! Reformulez avec vos propres mots les consignes qui vous ont été données. Dessinez des schémas si cela vous aide.
- **Découpez en fonctions** chaque élément de votre programme. Vous pourrez ainsi tester chaque élément indépen-

damment du reste. Pensez à écrire les *docstrings* en même temps que vous écrivez vos fonctions.
- **Documentez-vous**. L'algorithme dont vous avez besoin existe-t-il déjà dans un autre module ? De quels outils mathématiques avez-vous besoin dans votre algorithme ?
- Quand l'algorithme est complexe, **commentez votre code** pour expliquer votre raisonnement. Utiliser des fonctions (ou méthodes) encore plus petites peut aussi être une solution.
- Utilisez des **noms de variables explicites**, qui signifient quelque chose. En lisant votre code, on doit comprendre ce que vous faites. Choisir des noms de variables pertinents permet aussi de réduire les commentaires.
- Quand vous construisez une structure de données complexe (par exemple une liste de dictionnaires contenant d'autres objets), **documentez** l'organisation de cette structure de données avec un exemple simple.
- Si vous créez ou manipulez une entité cohérente avec des propriétés propres, essayez de construire une **classe**. Reportez-vous, pour cela, au chapitre 23 *Avoir la classe avec les objets*.
- **Testez** votre code sur un **petit jeu de données**, pour comprendre rapidement ce qui se passe et corriger d'éventuelles erreurs. Par exemple, une séquence d'ADN de 1 000 bases est plus facile à manipuler que le génome humain $(3 \times 10^9 \text{ bases})$!
- Lorsque votre programme « plante », **lisez le message d'erreur**. Python tente de vous expliquer ce qui ne va pas. Le numéro de la ligne qui pose problème est aussi indiqué.
- **Discutez avec des gens**. Faites tester votre programme par d'autres. Les instructions d'utilisation sont-elles claires ?
- Enfin, si vous **distribuez votre code** :
  — Rédigez une **documentation** claire.
  — **Testez** votre programme (jetez un œil aux tests unitaires[10]).
  — Précisez une **licence** d'utilisation (voir le site *Choose an open source license*[11]).

16.7 Pour terminer : la PEP 20

La PEP 20 est une sorte de réflexion philosophique avec des phrases simples qui devraient guider tout programmeur. Comme les développeurs de Python ne manque pas d'humour, celle-ci est accessible sous la forme d'un « œuf de Pâques » (*easter egg*, en anglais) ou encore « fonctionnalité cachée d'un programme » en important un module nommé `this` :

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Et si l'aventure et les *easter eggs* vous plaisent, testez également la commande

```
>>> import antigravity
```

Il vous faudra un navigateur et une connexion internet.

10. https://fr.wikipedia.org/wiki/Test_unitaire
11. https://choosealicense.com/

Pour aller plus loin

- L'article *Python Code Quality : Tools & Best Practices*[12] du site *Real Python* est une ressource intéressante pour explorer plus en détail la notion de qualité pour un code Python. De nombreux *linters* y sont présentés.
- Les articles *Assimilez les bonnes pratiques de la PEP 8*[13] du site *OpenClassrooms* et *Structuring Python Programs*[14] du site *Real Python* rappellent les règles d'écriture et les bonnes pratiques vues dans ce chapitre.

12. https://realpython.com/python-code-quality/
13. https://openclassrooms.com/fr/courses/4425111-perfectionnez-vous-en-python/4464230-assimilez-les-bonnes-pratiqu
14. https://realpython.com/python-program-structure/

CHAPITRE 17

Expressions régulières et *parsing*

Le module *re* permet d'utiliser des expressions régulières avec Python. Les expressions régulières sont aussi appelées en anglais *regular expressions*, ou en plus court *regex*. Dans la suite de ce chapitre, nous utiliserons souvent le mot *regex* pour désigner une expression régulière. Les expressions régulières sont puissantes et incontournables en bioinformatique, surtout lorsque vous souhaitez récupérer des informations dans de gros fichiers.

Cette action de recherche de données dans un fichier est appelée généralement *parsing* (qui signifie littéralement « analyse syntaxique »). Le *parsing* fait partie du travail quotidien du bioinformaticien, il est sans arrêt en train de « fouiller » dans des fichiers pour en extraire des informations d'intérêt, par exemple récupérer les coordonnées 3D des atomes d'une protéine dans un fichier PDB, ou encore extraire les gènes d'un fichier GenBank.

Dans ce chapitre, nous ne ferons que quelques rappels sur les expressions régulières. Pour une documentation plus complète, référez-vous à la page d'aide des expressions régulières[1] sur le site officiel de Python.

17.1 Définition et syntaxe

Une expression régulière est une suite de caractères qui a pour but de décrire un fragment de texte. Cette suite de caractères est encore appelée **motif** (en anglais, *pattern*), qui est constitué de deux types de caractères :

- les caractères dits *normaux* ;
- les *métacaractères* ayant une signification particulière, par exemple le caractère `^` signifie début de ligne, et non pas le caractère « chapeau » littéral.

Avant de présenter les *regex* en Python, nous allons faire un petit détour par Unix. En effet, certains programmes, comme `egrep`, `sed` ou encore `awk`, savent interpréter les expressions régulières. Tous ces programmes fonctionnent généralement selon le schéma suivant :

- le programme lit un fichier ligne par ligne ;
- pour chaque ligne lue, si l'expression régulière passée en argument est retrouvée dans la ligne, alors le programme effectue une action.

Par exemple, pour le programme `egrep` :

```
$ egrep "^DEF" herp_virus.gbk
DEFINITION  Human herpesvirus 2, complete genome.
```

Ici, `egrep` affiche toutes les lignes du fichier du virus de l'herpès (`herp_virus.gbk`) dans lesquelles la *regex* `^DEF` (c'est-à-dire le mot `DEF` en début de ligne) est retrouvée.

1. https://docs.python.org/fr/3/library/re.html

Remarque

Il est intéressant de faire un point sur le vocabulaire utilisé en anglais et en français. En général, on utilise le verbe *to match* pour indiquer qu'une *regex* « a fonctionné ». Bien qu'il n'y ait pas de traduction littérale en français, on peut utiliser les verbes « retrouver » ou « correspondre ». Par exemple, on pourra traduire l'expression « *The regex matches the line* » par « La *regex* est retrouvée dans la ligne » ou encore « La *regex* correspond dans la ligne ».

Après avoir introduit le vocabulaire des *regex*, voici quelques éléments de syntaxe des métacaractères :

^ Début de chaîne de caractères ou de ligne.
Exemple : la *regex* `^ATG` est retrouvée dans la chaîne de caractères `ATGCGT` mais pas dans la chaîne `CCATGTT`.

$ Fin de chaîne de caractères ou de ligne.
Exemple : la *regex* `ATG$` est retrouvée dans la chaîne de caractères `TGCATG` mais pas dans la chaîne `CCATGTT`.

. N'importe quel caractère (mais un caractère quand même).
Exemple : la *regex* `A.G` est retrouvée dans `ATG`, `AtG`, `A4G`, mais aussi dans `A-G` ou dans `A G`.

[ABC] Le caractère A ou B ou C (un seul caractère).
Exemple : la *regex* `T[ABC]G` est retrouvée dans `TAG`, `TBG` ou `TCG`, mais pas dans `TG`.

[A-Z] N'importe quelle lettre majuscule.
Exemple : la *regex* `C[A-Z]T` est retrouvée dans `CAT`, `CBT`, `CCT`…

[a-z] N'importe quelle lettre minuscule.

[0-9] N'importe quel chiffre.

[A-Za-z0-9] N'importe quel caractère alphanumérique.

[^AB] N'importe quel caractère sauf A et B.
Exemple : la *regex* `CG[^AB]T` est retrouvée dans `CG9T`, `CGCT`… mais pas dans `CGAT` ni dans `CGBT`.

**** Caractère d'échappement (pour protéger certains caractères).
Exemple : la *regex* `\+` désigne le caractère `+` littéral. La *regex* `A\.G` est retrouvée dans `A.G` et non pas dans `A` suivi de n'importe quel caractère, suivi de `G`.

***** 0 à *n* fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : la *regex* `A(CG)*T` est retrouvée dans `AT`, `ACGT`, `ACGCGT`…

+ 1 à *n* fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : la *regex* `A(CG)+T` est retrouvée dans `ACGT`, `ACGCGT`… mais pas dans `AT`.

? 0 à 1 fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : la *regex* `A(CG)?T` est retrouvée dans `AT` ou `ACGT`.

{n} *n* fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : la *regex* `A(CG){2}T` est retrouvée dans `ACGCGT` mais pas dans `ACGT`, `ACGCGCGT` ou `ACGCG`.

{n,m} *n* à *m* fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : la *regex* `A(C){2,4}T` est retrouvée dans `ACCT`, `ACCCT` et `ACCCCT` mais pas dans `ACT`, `ACCCCCT` ou `ACCC`.

{n,} Au moins *n* fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : la *regex* `A(C){2,}T` est retrouvée dans `ACCT`, `ACCCT` et `ACCCCT` mais pas à `ACT` ou `ACCC`.

{,m} Au plus *m* fois le caractère précédent ou l'expression entre parenthèses précédente.
Exemple : la *regex* `A(C){,2}T` est retrouvée dans `AT`, `ACT` et `ACCT` mais pas dans `ACCCT` ou `ACC`.

(CG|TT) Les chaînes de caractères `CG` ou `TT`.
Exemple : la *regex* `A(CG|TT)C` est retrouvée dans `ACGC` ou `ATTC`.

Enfin, il existe des caractères spéciaux qui sont bien commodes et qui peuvent être utilisés en tant que métacaractères :

\d remplace n'importe quel chiffre (*d* signifie *digit*), équivalent à `[0-9]`.

\w remplace n'importe quel caractère alphanumérique et le caractère souligné (*underscore*) (*w* signifie *word character*), équivalent à `[0-9A-Za-z_]`.

\s remplace n'importe quel « espace blanc » (*whitespace*) (*s* signifie *space*), équivalent à `[ \t\n\r\f]`. La notion d'espace blanc a été abordée dans le chapitre 11 *Plus sur les chaînes de caractères*. Les espaces blancs les plus classiques sont l'espace , la tabulation `\t`, le retour à la ligne `\n`, mais il en existe d'autres comme `\r` et `\f` que nous ne développerons pas ici. `\s` est très pratique pour détecter une combinaison d'espace(s) et/ou de tabulation(s).

Comme vous le constatez, les métacaractères sont nombreux et leur signification est parfois difficile à maîtriser. Faites particulièrement attention aux métacaractères `.`, `+` et `*` qui, combinés ensemble, peuvent donner des résultats ambigus.

Attention

Il est important de savoir par ailleurs que les *regex* sont « avides » (*greedy* en anglais) lorsqu'on utilise les métacaractères `+` et `*`. Cela signifie que la *regex* cherchera à « s'étendre » au maximum. Par exemple, si on utilise la *regex* `A+` pour faire une recherche dans la chaîne `TTTAAAAAAAAGC`, tous les A de cette chaîne (huit en tout) seront concernés, bien que `AA`, `AAA`, etc. « fonctionnent » également avec cette *regex*.

17.2 Quelques ressources en ligne

Nous vous conseillons de tester systématiquement vos expressions régulières sur des exemples simples. Pour vous aider, nous vous recommandons plusieurs sites internet :

- RegexOne[2] : tutoriel en ligne très bien fait.
- RegExr[3] et ExtendsClass[4] : visualisent tous les endroits où une *regex* est retrouvée dans un texte.
- pythex.org[5] : interface simple et efficace, dédiée à Python.
- Regular-Expressions.info[6] : documentation exhaustive sur les *regex* (il y a même une section sur Python).

N'hésitez pas à explorer ces sites avant de vous lancer dans les exercices ou dans l'écriture de vos propres *regex* !

17.3 Le module *re*

17.3.1 La fonction `search()`

Dans le module *re*, la fonction `search()` est incontournable. Elle permet de rechercher un motif, c'est-à-dire une *regex*, au sein d'une chaîne de caractères avec une syntaxe de la forme `search(motif, chaine)`. Si `motif` est retrouvé dans `chaine`, Python renvoie un objet du type `SRE_Match`.

Sans entrer dans les détails propres au langage orienté objet, si on utilise un objet du type `SRE_Match` dans un test, il sera considéré comme vrai. Par exemple, si on recherche le motif `tigre` dans la chaîne de caractères `"girafe tigre singe"` :

```
>>> import re
>>> animaux = "girafe tigre singe"
>>> re.search("tigre", animaux)
<_sre.SRE_Match object at 0x7fefdaefe2a0>
>>> if re.search("tigre", animaux):
...     print("OK")
...
OK
```

Attention

Le motif que vous utilisez comme premier argument de la fonction `search()` sera interprété en tant que *regex*. Ainsi, `^DEF` correspondra au mot `DEF` en début de chaîne et pas au caractère littéral `^`suivi du mot `DEF`.

2. https://regexone.com/
3. https://regexr.com/
4. https://extendsclass.com/regex-tester.html#python
5. https://pythex.org/
6. https://www.regular-expressions.info

17.3.2 Les fonctions `match()` et `fullmatch()`

Il existe aussi la fonction `match()` dans le module `re` qui fonctionne sur le modèle de `search()`. La différence est qu'elle renvoie un objet du type `SRE_Match` seulement lorsque la *regex* correspond au début de la chaîne de caractères (à partir du premier caractère) :

```
>>> animaux = "girafe tigre singe"
>>> re.search("tigre", animaux)
<_sre.SRE_Match object at 0x7fefdaefe718>
>>> re.match("tigre", animaux)
>>>
>>> animaux = "tigre singe"
>>> re.match("tigre", animaux)
<_sre.SRE_Match object; span=(0, 5), match='tigre'>
>>>
```

Il existe également la fonction `fullmatch()`, qui renvoie un objet du type `SRE_Match` si et seulement si l'expression régulière correspond **exactement** à la chaîne de caractères.

```
>>> animaux = "tigre "
>>> re.fullmatch("tigre", animaux)
>>> animaux = "tigre"
>>> re.fullmatch("tigre", animaux)
<_sre.SRE_Match object; span=(0, 5), match='tigre'>
```

De manière générale, nous vous recommandons l'usage de la fonction `search()`. Si vous souhaitez avoir une correspondance avec le début de la chaîne de caractères comme dans la fonction `match()`, vous pouvez toujours utiliser l'accroche de début de ligne `^`. Si vous voulez une correspondance exacte, comme dans la fonction `fullmatch()`, vous pouvez utiliser les métacaractères `^` et `$`, par exemple `^tigre$`.

17.3.3 Compilation d'expressions régulières

Lorsqu'on a besoin de tester la même expression régulière sur plusieurs milliers de chaînes de caractères, il est pratique de compiler préalablement la *regex* à l'aide de la fonction `compile()`, qui renvoie un objet de type `SRE_Pattern` :

```
>>> regex = re.compile("^tigre")
>>> regex
<_sre.SRE_Pattern object at 0x7fefdafd0df0>
```

On peut alors utiliser directement cet objet avec la méthode `.search()` :

```
>>> animaux = "girafe tigre singe"
>>> regex.search(animaux)
>>> animaux = "tigre singe"
>>> regex.search(animaux)
<_sre.SRE_Match object at 0x7fefdaefe718>
>>> animaux = "singe tigre"
>>> regex.search(animaux)
```

17.3.4 Groupes

L'intérêt de l'objet de type `SRE_Match` renvoyé par Python lorsqu'une *regex* trouve une correspondance dans une chaîne de caractères est de pouvoir ensuite récupérer certaines zones précises :

```
>>> regex = re.compile("([0-9]+)\.([0-9]+)")
```

Dans cet exemple, on recherche un nombre décimal, c'est-à-dire une chaîne de caractères :

- qui débute par un ou plusieurs chiffres `[0-9]+`,
- suivi d'un point `\.` (le point a d'habitude une signification de métacaractère, donc il faut l'échapper avec `\` pour qu'il retrouve sa signification de point),
- et qui se termine encore par un ou plusieurs chiffres `[0-9]+`.

Les parenthèses dans la *regex* créent des groupes (`[0-9]+` deux fois) qui seront récupérés ultérieurement par la méthode `.group()`.

```
>>> resultat = regex.search("pi vaut 3.14")
>>> resultat.group(0)
'3.14'
>>> resultat.group(1)
'3'
>>> resultat.group(2)
'14'
>>> resultat.start()
8
>>> resultat.end()
12
```

La totalité de la correspondance est donnée par `.group(0)`, le premier élément entre parenthèses est donné par `.group(1)` et le second par `.group(2)`.

Les méthodes `.start()` et `.end()` donnent respectivement la position de début et de fin de la zone qui correspond à la *regex*. Notez que la méthode `.search()` ne renvoie que la première zone qui correspond à l'expression régulière, même s'il en existe plusieurs :

```
>>> resultat = regex.search("pi vaut 3.14 et e vaut 2.72")
>>> resultat.group(0)
'3.14'
```

17.3.5 La méthode `.findall()`

Pour récupérer chaque zone dans la *regex*, s'il y en a plusieurs, vous pouvez utiliser la méthode `.findall()` qui renvoie une liste des éléments en correspondance :

```
>>> regex = re.compile("[0-9]+\.[0-9]+")
>>> resultat = regex.findall("pi vaut 3.14 et e vaut 2.72")
>>> resultat
['3.14', '2.72']
```

L'utilisation des groupes entre parenthèses est également possible, ceux-ci sont alors renvoyés sous la forme de tuples :

```
>>> regex = re.compile("([0-9]+)\.([0-9]+)")
>>> resultat = regex.findall("pi vaut 3.14 et e vaut 2.72")
>>> resultat
[('3', '14'), ('2', '72')]
```

17.3.6 La méthode `.sub()`

Enfin, la méthode `.sub()` permet d'effectuer des remplacements assez puissants. Par défaut, la méthode `.sub( chaine1, chaine2)` remplace toutes les occurrences trouvées par l'expression régulière dans `chaine2` par `chaine1`. Si vous souhaitez ne remplacer que les *n* premières occurrences, utilisez l'argument `count=n` :

```
>>> regex = re.compile("[0-9]+\.[0-9]+")
>>> regex.sub("quelque chose", "pi vaut 3.14 et e vaut 2.72")
'pi vaut quelque chose et e vaut quelque chose'
>>> regex.sub("quelque chose", "pi vaut 3.14 et e vaut 2.72", count=1)
'pi vaut quelque chose et e vaut 2.72'
```

Encore plus puissant, il est possible d'utiliser dans le remplacement des groupes qui ont été « capturés » avec des parenthèses :

```
>>> regex = re.compile("([0-9]+)\.([0-9]+)")
>>> phrase = "pi vaut 3.14 et e vaut 2.72"
>>> regex.sub("approximativement \\1", phrase)
'pi vaut approximativement 3 et e vaut vaut approximativement 2'
>>> regex.sub("approximativement \\1 (puis .\\2)",phrase)
'pi vaut approximativement 3 (puis .14) et e vaut approximativement 2 (puis .72)'
```

Si vous avez capturé des groupes, il suffit d'utiliser `\\1`, `\\2` (etc.) pour utiliser les groupes correspondants dans la chaîne de caractères substituée. On notera que la syntaxe générale pour récupérer des groupes dans les outils qui gèrent les *regex* est `\1`, `\2`, etc. Toutefois, Python nous oblige à mettre un deuxième *backslash* car il y a ici deux niveaux : un

premier niveau Python où on veut mettre un *backslash* littéral (donc `\\`), puis un second niveau *regex* dans lequel on veut retrouver `\1`. Si cela est confus, retenez seulement qu'il faut mettre un `\\` devant le numéro de groupe.

Enfin, sachez que la réutilisation d'un groupe précédemment capturé est aussi utilisable lors d'une utilisation classique de *regex*. Par exemple :

```
1 >>> re.search("(pan)\\1", "bambi et panpan")
2 <_sre.SRE_Match object; span=(9, 15), match='panpan'>
3 >>> re.search("(pan)\\1", "le pistolet a fait pan !")
4 >>>
```

Dans la *regex* `(pan)\\1`, on capture d'abord le groupe `(pan)` grâce aux parenthèses (il s'agit du groupe 1, puisque c'est le premier jeu de parenthèses), immédiatement suivi du même groupe grâce au `\\1`. Dans cet exemple, on capture donc le mot `panpan` (lignes 1 et 2). Si, par contre, on a une seule occurrence du mot `pan`, cette *regex* ne fonctionne pas, ce qui est le cas ligne 3.

Bien sûr, si on avait eu un deuxième groupe, on aurait pu le réutiliser avec `\\2`, un troisième groupe avec `\\3`, etc.

Nous espérons vous avoir convaincu de la puissance du module *re* et des expressions régulières. Alors, plus de temps à perdre, à vos *regex* !

17.4 Exercices

Conseil

Pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

17.4.1 *Regex* de base

Dans cet exercice, nous allons manipuler le fichier GenBank `NC_001133.gbk`[7] correspondant au chromosome I de la levure *Saccharomyces cerevisiae*.

Créez un script `regex_genbank.py` :

- qui recherche le mot `DEFINITION` en début de ligne dans le fichier GenBank, puis affiche la ligne correspondante ;
- qui recherche tous les journaux (mot-clé `JOURNAL`) dans lesquels ont été publiés les travaux sur cette séquence, puis affiche les lignes correspondantes.

Conseil

- Utilisez des *regex* pour trouver les lignes demandées.
- Des explications sur le format GenBank et des exemples de code sont fournies dans l'annexe A *Quelques formats de données en biologie*.

17.4.2 Enzyme de restriction

Une enzyme de restriction est une protéine capable de couper une molécule d'ADN. Cette coupure se fait sur le site de restriction de l'ADN qui correspond à une séquence particulière de nucléotides (bases).

Pour chacune des enzymes ci-dessous, déterminez les expressions régulières qui décrivent leurs sites de restriction. Le symbole N correspond aux bases A, T, C ou G. W correspond à A ou T. Y correspond à C ou T. R correspond à A ou G.

Enzyme	Site de restriction
HinFI	GANTC
EcoRII	CCWGG
BbvBI	GGYRCC
BcoI	CYCGRG
Psp5II	RGGWCCY

7. https://python.sdv.u-paris.fr/data-files/NC_001133.gbk

Enzyme	Site de restriction
BbvAI	GAANNNNTTC

17.4.3 Nettoyeur d'espaces

Le fichier `cigale_fourmi.txt`[8] contient la célèbre fable de Jean de la Fontaine. Malheureusement, la personne qui l'a recopié a parfois mis plusieurs espaces au lieu d'un seul entre les mots.

Créez un script `cigale_fourmi.py` qui, grâce à une *regex* et à la fonction `sub()`, remplace plusieurs espaces par un seul dans le texte ci-dessus. Le nouveau texte, ainsi nettoyé, sera enregistré dans un fichier `cigale_fourmi_propre.txt`.

17.4.4 Liste des protéines humaines

Téléchargez le fichier `human-proteome.fasta`[9] qui contient le protéome humain, c'est-à-dire les séquences de l'ensemble des protéines chez l'Homme. Ce fichier est au format FASTA.

On souhaite lister toutes ces protéines et les indexer avec un numéro croissant.

Créez un script `liste_proteome.py` qui :

- lit le fichier `human-proteome.fasta` ;
- extrait, avec une *regex*, le numéro d'accession de la protéine de toutes les lignes de commentaires des séquences ;
- affiche le mot `protein`, suivi d'un numéro qui s'incrémente, suivi du numéro d'accession.

Voici un exemple de sortie attendue :

```
protein 00001 O95139
protein 00002 O75438
protein 00003 Q8N4C6
[...]
protein 20371 Q8IZJ1
protein 20372 Q9UKP6
protein 20373 Q96HZ7
```

Conseil

- Des explications sur le format FASTA et des exemples de code sont fournis dans l'annexe A *Quelques formats de données en biologie*.
- La ligne de commentaire d'une séquence au format FASTA est de la forme
  `>sp|O95139|NDUB6_HUMAN NADH dehydrogenase [...]`
  Elle débute toujours pas le caractère `>`. Le numéro d'accession `O95139` se situe entre le premier et le second symbole `|` (symbole *pipe*). Attention, il faudra « échapper » ce symbole car il a une signification particulière dans une *regex*.
- Le numéro qui s'incrémente débutera à 1 et sera affiché sur 5 caractères, avec des 0 à sa gauche si nécessaires (formatage `{:05d}`).

17.4.5 Le défi du dé-HTMLiseur (exercice +++)

Le format HTML permet d'afficher des pages web dans un navigateur. Il s'agit d'un langage à balise qui fonctionne avec des balises ouvrantes `<balise>` et des balises fermantes `</balise>`.

Créez un script `dehtmliseur.py` qui lit le fichier `fichier_a_dehtmliser.html`[10] au format HTML et qui renvoie à l'écran tout le texte de ce fichier sans les balises HTML.

Nous vous conseillons tout d'abord d'ouvrir le fichier HTML dans un éditeur de texte et de bien l'observer. N'hésitez pas à vous aider des sites mentionnés dans les ressources en ligne.

17.4.6 Nettoyeur de doublons (exercice +++)

Téléchargez le fichier `breves_doublons.txt`[11] qui contient des mots répétés deux fois. Par exemple :

8. https://python.sdv.u-paris.fr/data-files/cigale_fourmi.txt
9. https://python.sdv.u-paris.fr/data-files/human-proteome.fasta
10. https://python.sdv.u-paris.fr/data-files/fichier_a_dehtmliser.html
11. https://python.sdv.u-paris.fr/data-files/breves_doublons.txt

```
Le cinéma est devenu parlant, la radio radio finira en images.
La sardine, c'est un petit petit poisson sans tête qui vit dans l'huile.
[...]
```

Écrivez un script `ote_doublons.py` qui lit le fichier `breves_doublons.txt` et qui supprime tous les doublons à l'aide d'une *regex*. Le script affichera le nouveau texte à l'écran.

Conseil

Utilisez la méthode `.sub()`.

CHAPITRE 18

Jupyter et ses *notebooks*

Les *notebooks* Jupyter sont des cahiers électroniques qui, dans le même document, peuvent rassembler du texte, des images, des formules mathématiques, des tableaux, des graphiques et du code informatique exécutable. Ils sont manipulables interactivement dans un navigateur web.

Initialement développés pour les langages de programmation Julia, Python et R (d'où le nom *Jupyter*), les *notebooks* Jupyter supportent près de 40 langages différents.

La cellule est l'élément de base d'un *notebook* Jupyter. Elle peut contenir du texte formaté au format Markdown ou du code informatique qui pourra être exécuté.

Voici un exemple de *notebook* Jupyter (figure 18.1) :

Ce *notebook* est constitué de cinq cellules : deux avec du texte en Markdown (la première et la dernière) et trois avec du code Python (légèrement grisées).

18.1 Installation

Avec la distribution Miniconda, les *notebooks* Jupyter s'installent avec la commande :

```
$ conda install -c conda-forge -y jupyterlab
```

Pour être exact, la commande précédente installe un peu plus que les *notebooks* Jupyter, mais nous verrons cela par la suite.

18.2 JupyterLab

En 2018, le consortium Jupyter a lancé *JupyterLab*, qui est un environnement complet de programmation et d'analyse de données.

Pour obtenir cette interface, lancez la commande suivante depuis un *shell* :

```
$ jupyter lab
```

Une nouvelle page devrait s'ouvrir dans votre navigateur web et vous devriez obtenir une interface similaire à la figure 18.2, avec à gauche un navigateur de fichiers et à droite le « *Launcher* », qui permet de créer un nouveau *notebook* Jupyter, de lancer un terminal ou d'éditer un fichier texte, un fichier Mardown, un script Python…

L'interface proposée par JupyterLab est très riche. On peut y organiser un *notebook* Jupyter, un éditeur de fichier texte, un terminal… Les possibilités sont nombreuses et nous vous invitons à explorer cette interface par vous-même.

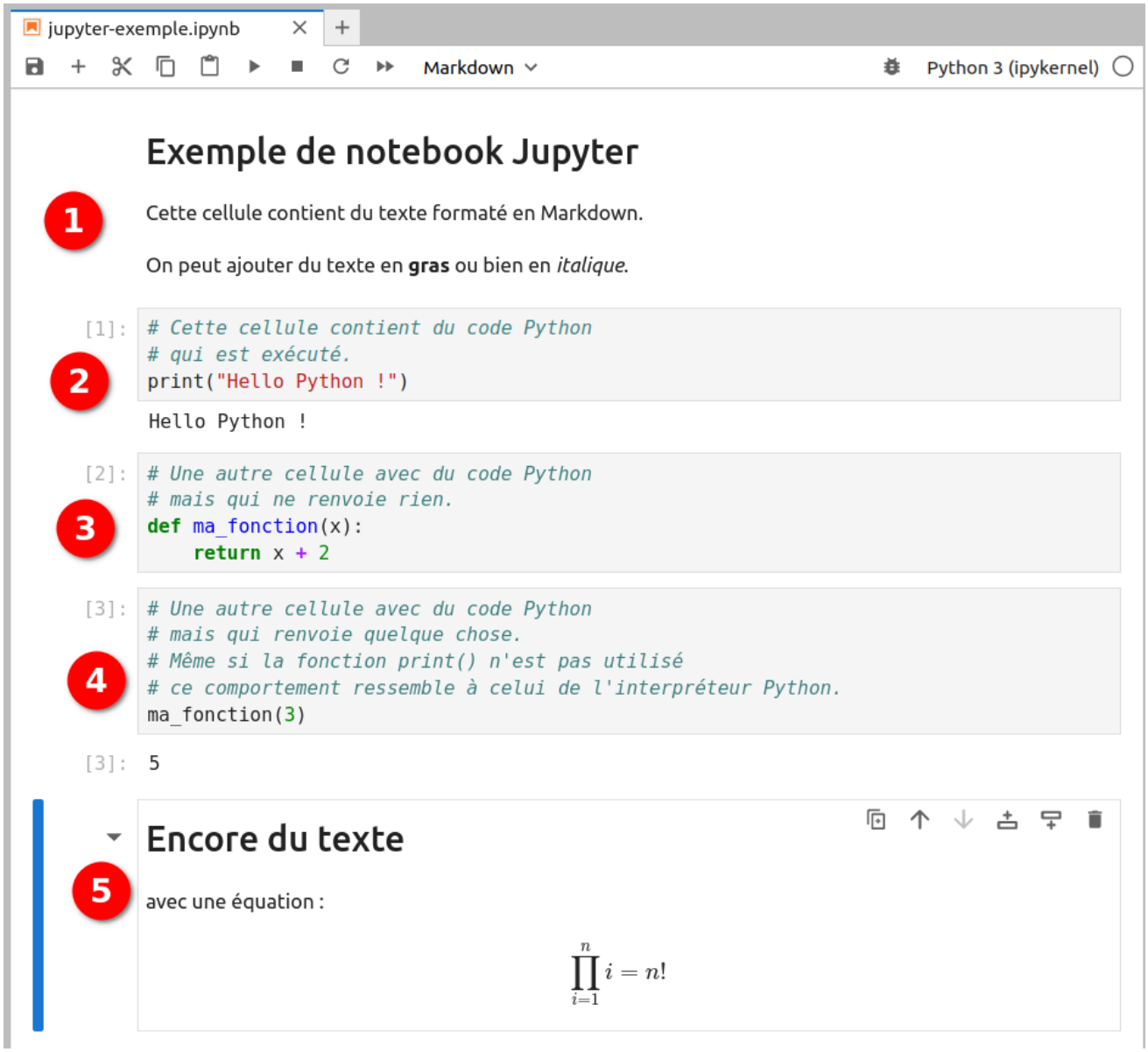


FIGURE 18.1 – Exemple de *notebook* Jupyter. Les chiffres entourés désignent les différentes cellules.

18.3 Création d'un *notebook*

Pour créer un *notebook*, cliquez sur le bouton *Python 3* situé dans la rubrique *Notebook* dans le *Launcher* (figure 18.3).

Le *notebook* fraîchement créé ne contient qu'une cellule vide.

La première chose à faire est de donner un nom à votre *notebook*. Pour cela, cliquer avec le bouton droit de la souris sur *Untitled.ipynb*, en haut du *notebook*. Si le nom de votre *notebook* est *test.ipynb*, alors le fichier *test.ipynb* sera créé dans le répertoire depuis lequel vous avez lancé JupyterLab.

Remarque

L'extension *.ipynb* est l'extension de fichier des *notebooks* Jupyter.

Vous pouvez entrer des instructions Python dans la première cellule. Par exemple :

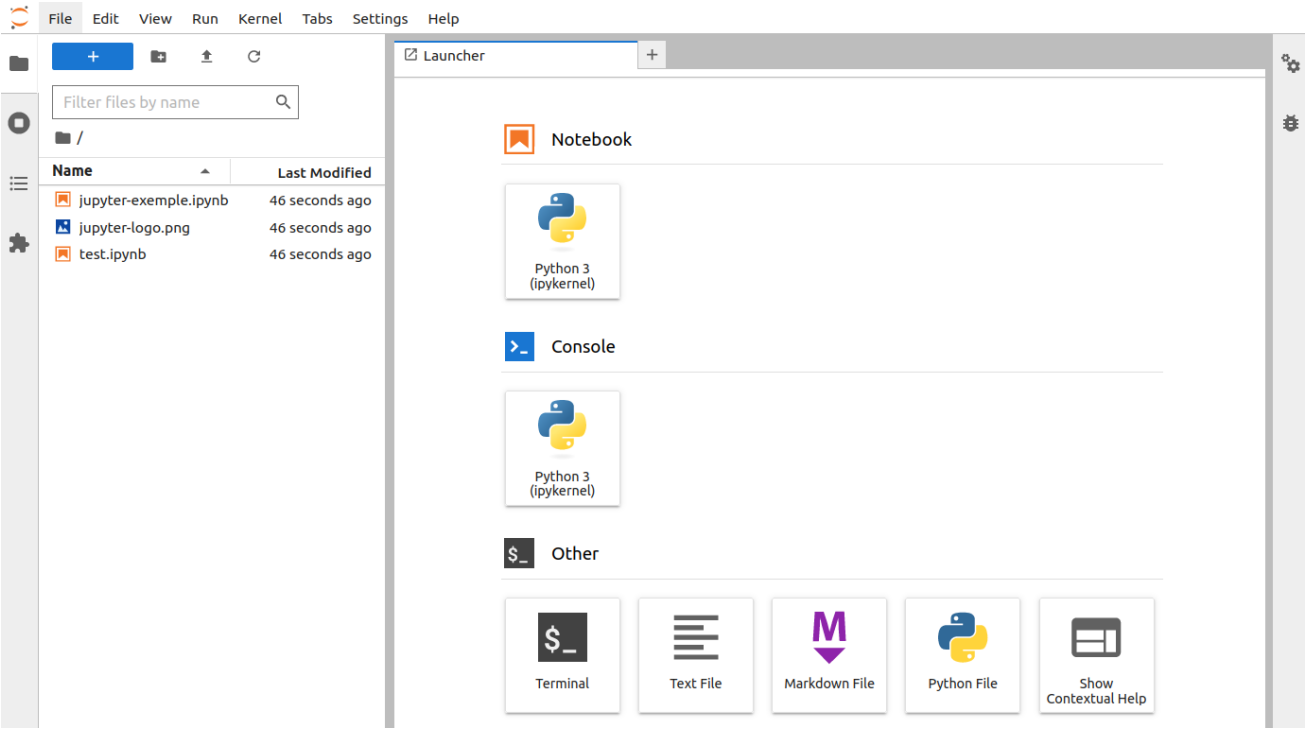


FIGURE 18.2 – Interface de JupyterLab.

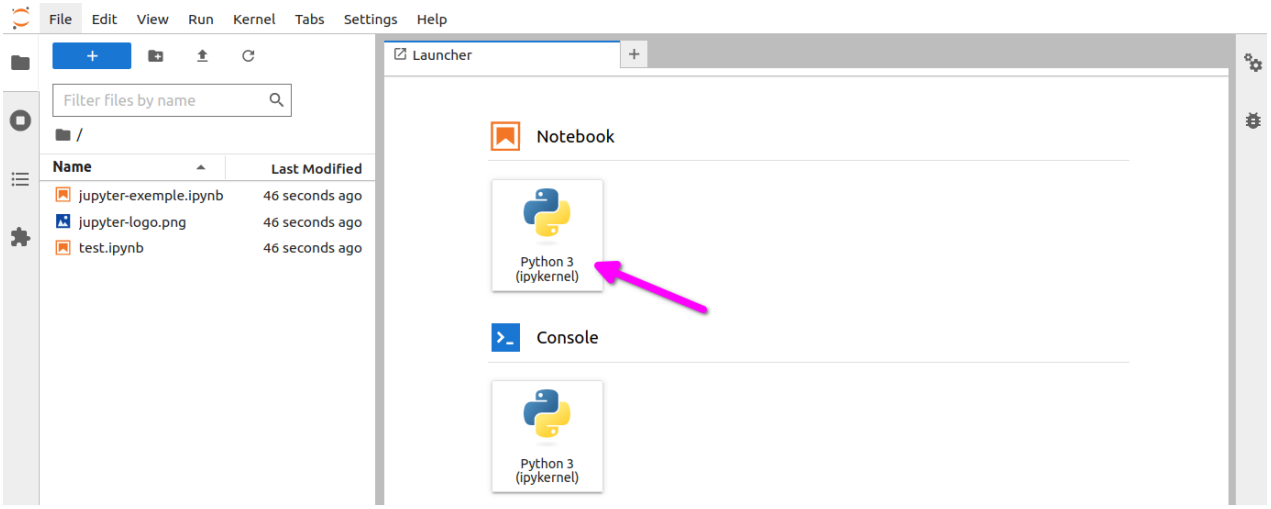


FIGURE 18.3 – Création d'un nouveau *notebook*.

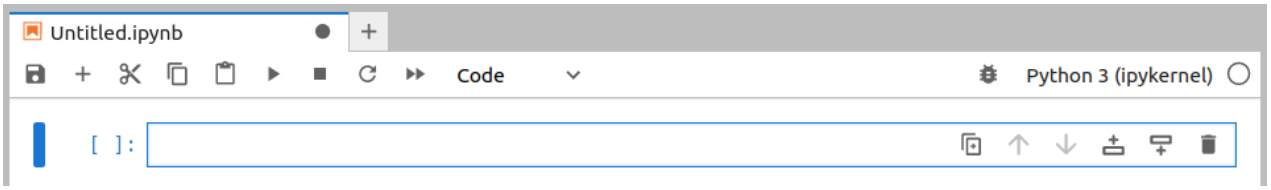


FIGURE 18.4 – Nouveau *notebook* avec une cellule vide.

```
1  a = 2
2  b = 3
3  print(a+b)
```

Pour exécuter le contenu de cette cellule, vous avez plusieurs possibilités :

- Cliquer sur le menu *Run*, puis *Run Selected Cells*.
- Cliquer sur le bouton dans la barre de menu au dessus du *notebook*.
- Presser simultanément les touches *Ctrl* + *Entrée*.

Dans tous les cas, vous devriez obtenir un résultat similaire à la figure 18.5. La notation `[1]` à gauche de la cellule indique qu'il s'agit de la première cellule de code qui a été exécutée.

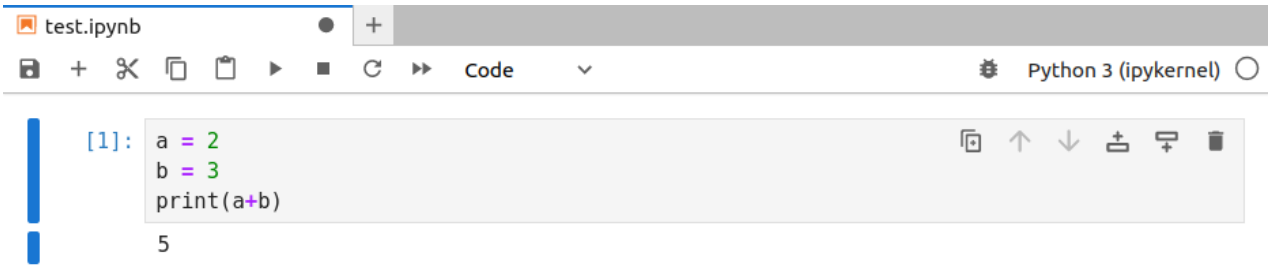


FIGURE 18.5 – Exécution d'une première cellule.

Pour créer une nouvelle cellule, vous avez, ici encore, plusieurs possibilités :

- Cliquer sur l'icône + dans la barre de menu au dessus du *notebook*.
- Cliquer sur la 2e icône à partir de la droite (juste à côté de la poubelle), dans les icônes situées à l'intérieur de la cellule, à droite.

Une nouvelle cellule vide devrait apparaître.

Vous pouvez également créer une nouvelle cellule, en positionnant votre curseur dans la première cellule, puis en pressant simultanément les touches *Alt* + *Entrée*. Si vous utilisez cette combinaison de touches, vous remarquerez que le numéro à gauche de la première cellule est passée de `[1]` à `[2]`, car vous avez exécuté une nouvelle fois la première cellule *puis* créé une nouvelle cellule.

Vous pouvez ainsi créer plusieurs cellules les unes à la suite des autres. Un objet créé dans une cellule antérieure sera disponible dans les cellules suivantes. Par exemple, dans la figure 18.6, nous avons quatre cellules.

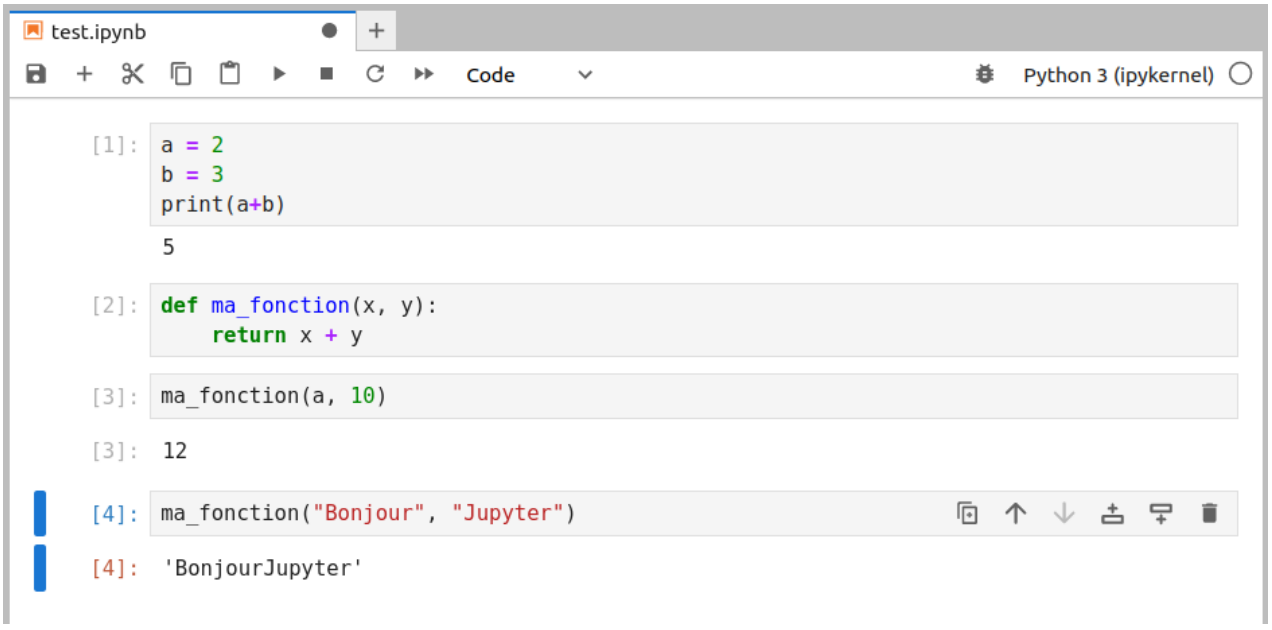


FIGURE 18.6 – *Notebook* avec plusieurs cellules de code Python.

Dans un *notebook* Jupyter, il est parfaitement possible de réexécuter une cellule précédente. Par exemple la première cellule, qui porte désormais à sa gauche la numérotation `[5]` (voir figure 18.7).

FIGURE 18.7 – *Notebook* avec une cellule ré-exécutée.

Attention

La possibilité d'exécuter les cellules d'un *notebook* Jupyter dans un ordre arbitraire peut prêter à confusion, notamment si vous modifiez la même variable dans plusieurs cellules.

Nous vous recommandons de régulièrement relancer complètement l'exécution de toutes les cellules de votre *notebook*, de la première à la dernière, en cliquant sur le menu *Kernel* puis *Restart Kernel and Run All Cells* et enfin de valider le message *Restart Kernel ?* en cliquant sur le bouton *Restart*.

18.4 Le format Markdown

Dans le tout premier exemple (figure 18.1), nous avons vu qu'il était possible de mettre du texte au format Markdown dans une cellule.

Il faut cependant indiquer à Jupyter que cette cellule est au format Markdown en cliquant sur *Code*, sous la barre de menu au dessus du *notebook*, puis en choisissant *Markdown*.

Le format Markdown permet de rédiger du texte formaté (gras, italique, liens, titres, images, formules mathématiques…) avec quelques balises très simples. Voici un exemple dans un *notebook* Jupyter (figure 18.8 (A)) et le rendu lorsque la cellule est exécutée (figure 18.8 (B)). Notez qu'une cellule Markdown est sur fond blanc (comme sur la figure 18.8 (B)).

Le format Markdown permet de rédiger du texte structuré rapidement et simplement. Ce cours est par exemple complètement rédigé en Markdown. Nous vous conseillons d'explorer les possibilités du Markdown en consultant la page Wikipédia [1] ou directement la page de référence [2].

18.5 Des graphiques dans les *notebooks*

Un autre intérêt des *notebooks* Jupyter est de pouvoir y incorporer des graphiques réalisés avec la bibliothèque *matplotlib* (que nous verrons prochainement).

Voici un exemple, d'un graphique qui sera présenté dans le chapitre 21 *Module Matplotlib* (figure 18.9).

L'instruction `%matplotlib inline` n'est pas nécessaire dans les versions récentes de JupyterLab. Mais avec d'anciennes versions, vous pourriez en avoir besoin pour que les graphiques s'affichent dans le *notebook*.

1. https://fr.wikipedia.org/wiki/Markdown
2. https://daringfireball.net/projects/markdown/syntax

Remarque

Pour quitter l'interface JupyterLab, il y a plusieurs possibilités :

- Dans le menu en haut à gauche de l'interface, cliquer sur *File*, puis *Shut Down*, puis confirmer en cliquant sur le bouton *Shut Down*.
- Une méthode plus radicale est de revenir sur le *shell* depuis lequel JupyterLab a été lancé, puis de presser deux fois de suite la combinaison de touches *Ctrl* + *C*.

18.6 Les *magic commands*

La commande précédente (`%matplotlib inline`) est une *magic command*. Les *magic commands*[3] apportent des fonctionnalités supplémentaires dans un *notebook*. Il en existe beaucoup, nous allons en aborder ici quelques unes.

Remarque

Dans cette rubrique, nous vous montrerons quelques exemples d'utilisation de *magic commands* exécutées dans un *notebook* Jupyter.

```
Les cellules de code apparaitront de cette manière
dans un notebook Jupyter, avec des numéros de lignes à gauche.
```

```
Les résultats seront affichés de cette manière,
éventuellement sur plusieurs lignes.
```

18.6.1 %whos

La commande `%whos` liste tous les objets (variables, fonctions, modules…) utilisés dans un *notebook*.

Si une cellule précédente contenait le code :

```
a = 2
b = 3

def ma_fonction(x, y):
    return x + y

resultat_1 = ma_fonction(a, 10)
resultat_2 = ma_fonction("Bonjour", "Jupyter")
```

alors l'exécution de :

```
%whos
```

renvoie :

```
Variable      Type        Data/Info
-----------------------------------
a             int         2
b             int         3
ma_fonction   function    <function ma_fonction at 0x7f219c2d04a0>
resultat_1    int         12
resultat_2    str         BonjourJupyter
```

18.6.2 %history

La commande `%history` liste toutes les commandes Python lancées dans un *notebook* :

```
%history
```

3. https://ipython.readthedocs.io/en/stable/interactive/magics.html

```
a = 2
b = 3
print(a + b)
def ma_fonction(x, y):
    return x + y
ma_fonction(a, 10)
ma_fonction("Bonjour", "Jupyter")
%whos
%history
```

18.6.3 %%time

La commande `%%time` (avec deux symboles `%`) va mesurer le temps d'exécution d'une cellule. C'est très utile pour faire des tests de performance. On peut, par exemple, comparer les vitesses de parcours d'une liste avec une boucle `for`, par les éléments ou par les indices des éléments.

Ainsi, cette cellule :

```
%%time
concentrations = [5.5, 7.2, 11.8, 13.6, 19.1, 21.7, 29.4]
somme_carres = 0.0
for conc in concentrations:
    somme_carres += conc**2
```

renvoie :

```
CPU times: user 8 µs, sys: 2 µs, total: 10 µs
Wall time: 11.9 µs
```

et celle-ci :

```
%%time
concentrations = [5.5, 7.2, 11.8, 13.6, 19.1, 21.7, 29.4]
somme_carres = 0.0
for idx in range(len(concentrations)):
    somme_carres += concentrations[idx]**2
```

renvoie :

```
CPU times: user 26 µs, sys: 5 µs, total: 31 µs
Wall time: 37.4 µs
```

Comme attendu, la première méthode (itération par les éléments) est plus rapide que la seconde (itération par les indices des éléments). Les temps obtenus dépendent de la machine sur laquelle vous exécutez ces commandes. Mais, sur une même machine, les résultats peuvent fluctuer d'une exécution à l'autre en fonction de l'activité de la machine. Ces fluctuations seront d'autant plus importantes que le temps d'exécution est court.

18.6.4 %%timeit

Pour palier à ce problème, la *magic command* `%%timeit` va exécuter plusieurs fois la cellule et donner une estimation du temps d'exécution moyen. Python détermine automatiquement le nombre d'itérations et le nombre de répétitions à effectuer pour obtenir un temps global d'exécution raisonnable.

En reprenant l'exemple précédent, on obtient :

```
%%timeit
concentrations = [5.5, 7.2, 11.8, 13.6, 19.1, 21.7, 29.4]
somme_carres = 0.0
for conc in concentrations:
    somme_carres += conc**2
```

```
492 ns ± 11.8 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

et

```
%%timeit
concentrations = [5.5, 7.2, 11.8, 13.6, 19.1, 21.7, 29.4]
somme_carres = 0.0
for idx in range(len(concentrations)):
    somme_carres += concentrations[idx]**2
```

```
606 ns ± 21.6 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Ici, chaque cellule sera exécutée un million de fois sur sept répétitions, soit sept millions de fois au total. Comme nous l'avions expliqué dans le chapitre 5 *Boucles et comparaisons*, itérer une liste sur ses éléments est la méthode la plus efficace (et la plus élégante).

18.7 Lancement d'une commande Unix

Enfin, dans les environnements Linux ou Mac OS X, il est possible de lancer une commande Unix depuis un *notebook* Jupyter. Il faut pour cela faire précéder la commande du symbole « `!` ». Par exemple, la commande `ls` affiche le contenu du répertoire courant :

```
!ls
```

```
jupyter-exemple.ipynb  markdown.ipynb    test.ipynb
jupyter-logo.png       matplotlib.ipynb
```

Pour aller plus loin

Le lancement d'une commande Unix depuis un *notebook* Jupyter (en précédant cette commande de `!`) est très utile pour réaliser de grosses analyses de données. Pour vous en rendre compte, explorez ce *notebook*[4] qui reproduit une analyse complète de données de séquençage haut débit. Ces résultats ont donné lieu à la publication de l'article scientifique « An open RNA-Seq data analysis pipeline tutorial with an example of reprocessing data from a recent Zika virus study[5] » (F1000 Research, 2016).

Conseil

Les *notebooks* Jupyter sont particulièrement adaptés à l'analyse de données en combinaison avec les modules *matplotlib* et *pandas*, qui seront abordés dans les prochains chapitres.

4. https://github.com/MaayanLab/Zika-RNAseq-Pipeline/blob/master/Zika.ipynb
5. https://f1000research.com/articles/5-1574/

FIGURE 18.8 – *Notebook* avec : (A) une cellule au format Markdown et (B) le rendu après exécution.

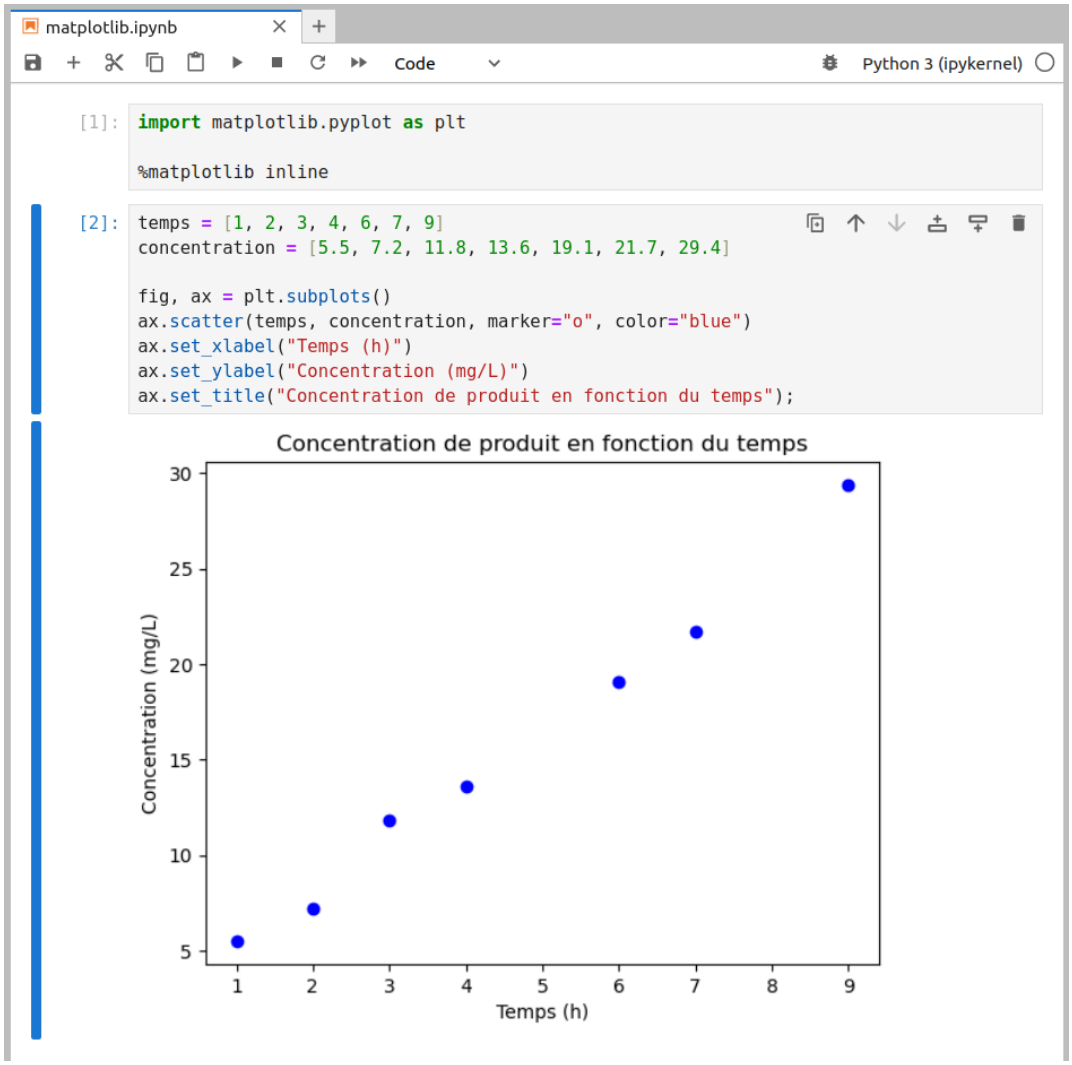


FIGURE 18.9 – Incorporation d'un graphique dans un *notebook* Jupyter.

CHAPITRE 19

Module Biopython

Nous allons aborder dans ce chapitre un module incontournable en bioinformatique. En effet, le module *Biopython*[1] permet de manipuler des données biologiques, comme des séquences (nucléiques et protéiques) ou des structures (fichiers PDB), et d'interroger des bases de données comme PubMed. Le tutoriel[2] est particulièrement bien fait, n'hésitez pas à le consulter.

19.1 Installation et convention

Contrairement aux autres modules vus précédemment, *Biopython* n'est pas fourni avec la distribution Python de base. Avec la distribution Miniconda que nous vous conseillons d'utiliser (consultez pour cela la documentation en ligne[3]), vous pouvez rapidement l'installer avec la commande :

```
$ conda install -c conda-forge biopython
```

Dans ce chapitre, nous vous montrerons quelques exemples d'utilisation du module *Biopython* pour vous convaincre de sa pertinence. Ces exemples seront exécutés dans un *notebook* Jupyter.

```
1  Les cellules de code apparaitront de cette manière
2  dans un notebook Jupyter, avec des numéros de lignes à gauche.
```

```
Les résultats seront affichés de cette manière,
éventuellement sur plusieurs lignes.
```

19.2 Chargement du module

On charge le module *Biopython* avec la commande :

```
1  import Bio
```

Attention

Le nom du module *Biopython* n'est pas `biopython`, mais `Bio` (avec un B majuscule).

1. http://biopython.org/
2. http://biopython.org/DIST/docs/tutorial/Tutorial.html
3. https://python.sdv.u-paris.fr/livre-dunod

19.3 Manipulation de séquences

Voici quelques exemples de manipulation de séquences avec *Biopython*.

19.3.1 Définition d'une séquence

```
1  import Bio
2  from Bio.Seq import Seq
3  ADN = Seq("ATATCGGCTATAGCATGC")
4  ADN
```

```
Seq('ATATCGGCTATAGCATGC')
```

- **Ligne 1.** Le module *Biopython* s'appelle `Bio`.
- **Ligne 2.** On charge la classe `Seq` du sous-module `Bio.Seq`.
- **Ligne 3.** La variable `ADN` est de type `Seq`, comme affiché dans le résultat.

19.3.2 Obtention de la séquence complémentaire et de la séquence complémentaire inverse

```
1  ADN.complement()
```

```
Seq('TATAGCCGATATCGTACG')
```

```
1  ADN.reverse_complement()
```

```
Seq('GCATGCTATAGCCGATAT')
```

19.3.3 Traduction en séquence protéique

```
1  ADN.translate()
```

```
Seq('ISAIAC')
```

Conseil

Dans l'annexe A *Quelques formats de données en biologie*, vous trouverez de nombreux exemples d'utilisation de *Biopython* pour manipuler des données aux formats FASTA, GenBank et PDB.

19.4 Interrogation de la base de données PubMed

Le sous-module *Entrez* de *Biopython* permet d'utiliser les ressources du NCBI et notamment d'interroger la base de données PubMed[4]. Nous allons par exemple utiliser PubMed pour chercher des articles scientifiques relatifs à la transferrine (*transferrin* en anglais) :

```
1  from Bio import Entrez
2  Entrez.email = "votremail@provider.fr"
3  req_esearch = Entrez.esearch(db="pubmed", term="transferrin")
4  res_esearch = Entrez.read(req_esearch)
```

- **Ligne 1.** On charge directement le sous-module *Entrez*.
- **Ligne 2.** Lors d'une requête sur le site du NCBI, il est important de définir correctement la variable `Entrez.email`, qui sera transmise au NCBI lors de la requête et qui pourra être utilisée pour vous contacter en cas de difficulté avec le serveur.

4. https://www.ncbi.nlm.nih.gov/pubmed/

- **Ligne 3.** On lance la requête (`transferrin`) sur le moteur de recherche `pubmed`. La requête est stockée dans la variable `req_esearch`.
- **Ligne 4.** Le résultat est lu et stocké dans la variable `res_esearch`.

Sans être un vrai dictionnaire, la variable `res_esearch` en a cependant plusieurs propriétés. Voici ses clés :

```
1  res_esearch.keys()
```

```
dict_keys(['Count', 'RetMax', 'RetStart', 'IdList', 'TranslationSet',
'TranslationStack', 'QueryTranslation'])
```

La valeur associée à la clé `IdList` est une liste qui contient les identifiants (PMID) des articles scientifiques associés à la requête (ici `transferrin`) :

```
1  res_esearch["IdList"]
```

```
['30411489', '30409795', '30405884', '30405827', '30402883', '30401570',
'30399508', '30397276', '30395963', '30394734', '30394728', '30394123',
'30393423', '30392910', '30392664', '30391706', '30391651', '30391537',
'30391296', '30390672']
```

```
1  len(res_esearch["IdList"])
```

```
20
```

Cette liste ne contient les identifiants que de 20 publications, alors que, si nous faisons cette même requête directement sur le site de PubMed depuis un navigateur web, nous obtenons plus de 45 700 résultats.

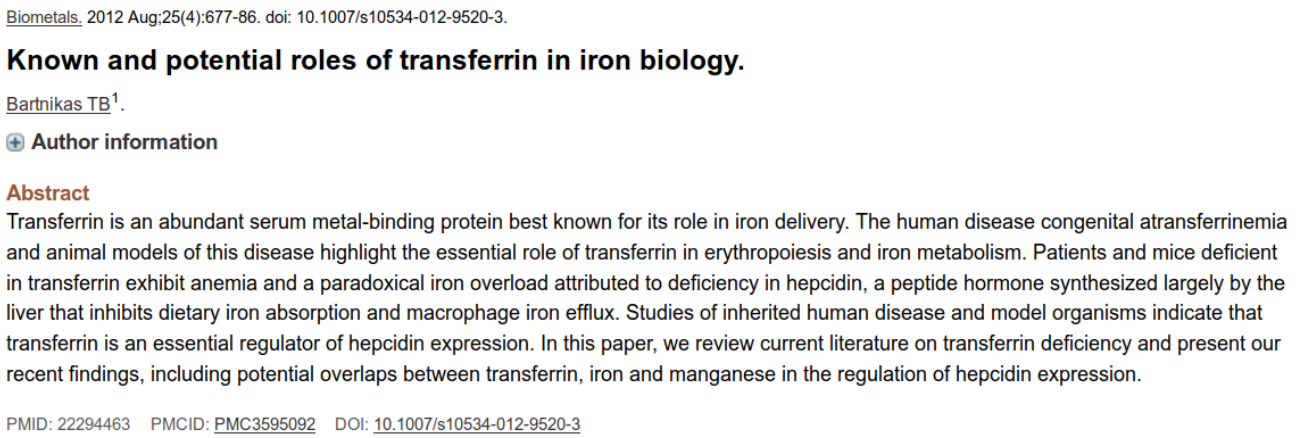
En réalité, le nombre exact de publications (en janvier 2024) est connu :

```
1  res_esearch["Count"]
```

```
'45717'
```

Pour ne pas saturer les serveurs du NCBI, seulement 20 PMID sont renvoyés par défaut. Mais vous pouvez augmenter cette limite en utilisant le paramètre `retmax` dans la fonction `Entrez.esearch()`.

Nous pouvons maintenant récupérer des informations sur une publication précise en connaissant son PMID, par exemple, l'article avec le PMID 22294463[5], dont un aperçu est sur la figure 19.1.


Biometals. 2012 Aug;25(4):677-86. doi: 10.1007/s10534-012-9520-3.

Known and potential roles of transferrin in iron biology.

Bartnikas TB[1].

Author information

Abstract

Transferrin is an abundant serum metal-binding protein best known for its role in iron delivery. The human disease congenital atransferrinemia and animal models of this disease highlight the essential role of transferrin in erythropoiesis and iron metabolism. Patients and mice deficient in transferrin exhibit anemia and a paradoxical iron overload attributed to deficiency in hepcidin, a peptide hormone synthesized largely by the liver that inhibits dietary iron absorption and macrophage iron efflux. Studies of inherited human disease and model organisms indicate that transferrin is an essential regulator of hepcidin expression. In this paper, we review current literature on transferrin deficiency and present our recent findings, including potential overlaps between transferrin, iron and manganese in the regulation of hepcidin expression.

PMID: 22294463 PMCID: PMC3595092 DOI: 10.1007/s10534-012-9520-3

FIGURE 19.1 – Aperçu de la publication *Known and potential roles of transferrin in iron biology* depuis le site PubMed.

Nous allons pour cela utiliser la fonction `Entrez.esummary()`

5. https://www.ncbi.nlm.nih.gov/pubmed/22294463

```
req_esummary = Entrez.esummary(db="pubmed", id="22294463")
res_esummary = Entrez.read(req_esummary)
```

La variable `res_esummary` n'est pas réellement une liste (son type exacte est `Bio.Entrez.Parser.ListElement`), mais elle est indexable (voir chapitre 14 *Conteneurs*). Cette pseudo-liste n'a qu'un seul élément, qui est lui-même un dictionnaire dont voici les clés :

```
res_esummary[0].keys()
```

```
dict_keys(['Item', 'Id', 'PubDate', 'EPubDate', 'Source', 'AuthorList',
'LastAuthor', 'Title', 'Volume', 'Issue', 'Pages', 'LangList',
'NlmUniqueID', 'ISSN', 'ESSN', 'PubTypeList', 'RecordStatus', 'PubStatus',
'ArticleIds', 'DOI', 'History', 'References', 'HasAbstract', 'PmcRefCount',
'FullJournalName', 'ELocationID', 'SO'])
```

Nous pouvons alors facilement obtenir le titre, le DOI et la date de publication (`PubDate`) de cet article, ainsi que le journal (`Source`) dans lequel il a été publié :

```
res_esummary[0]["Title"]
```

```
'Known and potential roles of transferrin in iron biology.'
```

```
res_esummary[0]["DOI"]
```

```
'10.1007/s10534-012-9520-3'
```

```
res_esummary[0]["PubDate"]
```

```
'2012 Aug'
```

```
res_esummary[0]["Source"]
```

```
'Biometals'
```

Enfin, pour récupérer le résumé de la publication précédente, nous allons utiliser la fonction `Entrez.efetch()` :

```
req_efetch = Entrez.efetch(
    db="pubmed", id="22294463",
    rettype="abstract", retmode="text")
req_efetch.read()
```

```
'1. Biometals. 2012 Aug;25(4):677-86. doi: 10.1007/s10534-012-9520-3.
\n\nKnown and potential roles of transferrin in iron biology.\n\nBart
nikas TB(1).\n\nAuthor information:\n(1)Department of Pathology, Chil’
drens Hospital, Enders 1110, 300 Longwood \nAvenue, Boston, MA 02115
, USA. mas.Bartnikas@childrens.harvard.edu\n\nTransferrin is an abund
ant serum metal-binding protein best known for its role \nin iron del
[...]
```

Le résultat n'est pas très lisible, car il apparait comme un seul bloc. Le caractère `\n` désigne un retour à la ligne. L'instruction `print()` affichera le résultat de manière plus lisible :

```
req_efetch = Entrez.efetch(
    db="pubmed", id="22294463",
    rettype="abstract", retmode="text")
print(req_efetch.read())
```

```
1. Biometals. 2012 Aug;25(4):677-86. doi: 10.1007/s10534-012-9520-3.

Known and potential roles of transferrin in iron biology.

Bartnikas TB(1).

Author information:
(1)Department of Pathology, 'Childrens Hospital, Enders 1110, 300 Longwood
Avenue, Boston, MA 02115, USA. mas.Bartnikas@childrens.harvard.edu

Transferrin is an abundant serum metal-binding protein best known for its role
in iron delivery. The human disease congenital atransferrinemia and animal
models of this disease highlight the essential role of transferrin in
erythropoiesis and iron metabolism. Patients and mice deficient in transferrin
exhibit anemia and a paradoxical iron overload attributed to deficiency in
hepcidin, a peptide hormone synthesized largely by the liver that inhibits
dietary iron absorption and macrophage iron efflux. Studies of inherited human
disease and model organisms indicate that transferrin is an essential regulator
of hepcidin expression. In this paper, we review current literature on
transferrin deficiency and present our recent findings, including potential
overlaps between transferrin, iron and manganese in the regulation of hepcidin
expression.

DOI: 10.1007/s10534-012-9520-3
PMCID: PMC3595092
PMID: 22294463 [Indexed for MEDLINE]
```

Le résultat contient bien le résumé de la figure 19.1, mais aussi d'autres informations comme le titre, le DOI, la date de publication…

19.5 Exercices

Conseil

Pour ces exercices, utilisez des *notebooks* Jupyter.

19.5.1 Pourcentage de GC de gènes de *Plasmodium falciparum*

Plasmodium falciparum (*P. falciparum*) est un des parasites responsables du paludisme chez les êtres humains. Le fichier `p_falciparum_500.fasta`[6] contient 500 gènes du génome de *P. falciparum*.

Écrivez un code Python qui calcule le pourcentage de GC de chaque gène. Les valeurs seront stockées dans un dictionnaire, avec comme clés les identifiants des gènes et comme valeurs le pourcentage de GC.

On rappelle que le pourcentage de GC d'une séquence est calculé avec la formule suivante :

$$\text{pourcentage GC} = \frac{\text{nombre de bases G} + \text{nombre de bases C}}{\text{longueur de la séquence}} \times 100$$

Affichez ensuite :

- Le nombre total de gènes.
- L'identifiant de la séquence qui a le pourcentage de GC le plus élevé, avec la valeur du pourcentage affichée avec deux chiffres après la virgule.
- L'identifiant de la séquence qui a le pourcentage de GC le plus faible, avec la valeur du pourcentage affichée avec deux chiffres après la virgule.

Conseil

Pour cet exercice, n'hésitez pas à consulter :

- Le chapitre 14 *Conteneurs* pour trier un dictionnaire.
- L'annexe A *Quelques formats de données en biologie* pour lire un fichier FASTA avec *Biopython*.

6. https://python.sdv.u-paris.fr/data-files/p_falciparum_500.fasta

19.5.2 Années de publication des articles relatifs à la barstar

L'objectif de cet exercice est d'interroger automatiquement la base de données bibliographique PubMed pour déterminer le nombre d'articles relatifs à la protéine barstar publiés chaque année.

Vous utiliserez le module *Biopython* et le module *matplotlib*, qui sera vu un peu plus loin (les principales instructions vous seront fournies).

19.5.2.1 Requête avec un mot-clé

Sur le site de PubMed[7], cherchez combien d'articles scientifiques sont relatifs à la barstar.

Effectuez la même chose avec Python et la méthode `Entrez.esearch()` de *Biopython*.

Choisissez un des PMID renvoyé et vérifiez dans PubMed que l'article associé est bien à propos de la barstar. Pour cela, indiquez le PMID choisi dans la barre de recherche de PubMed et cliquez sur *Search*. Attention, l'association n'est pas toujours évidente. Cherchez éventuellement dans le résumé de l'article si besoin.

Est-ce que le nombre total d'articles trouvés est cohérent avec celui obtenu sur le site de PubMed ?

19.5.2.2 Récupération des informations d'une publication

Récupérez les informations de la publication dont le PMID est 29701945[8]. Vous utiliserez la méthode `Entrez.esummary()`.

Affichez le titre, le DOI, le nom du journal (`Source`) et la date de publication (`PubDate`) de cet article. Vérifiez que cela correspond bien à ce que vous avez lu sur PubMed.

19.5.2.3 Récupération du résumé d'une publication

Récupérez le résumé de la publication dont le PMID est 29701945. Vous utiliserez la méthode `Entrez.efetch()`.

Affichez ce résumé.

19.5.2.4 Distribution des années de publication des articles relatifs à la barstar

En utilisant la méthode `Entrez.esearch()`, récupérez tous les PMID relatifs à la barstar. Pour cela, pensez à augmenter le paramètre `retmax`. Vos PMID seront stockés dans la liste `pmids` sous forme de chaînes de caractères. Vérifiez sur PubMed que vous avez récupéré le bon nombre d'articles.

En utilisant maintenant la méthode `Entrez.esummary()` dans une boucle, récupérez la date de publication de chaque article. Stockez l'année sous forme d'un nombre entier dans la liste `years`. Cette étape peut prendre une dizaine de minutes, soyez patient. Vous pouvez afficher dans votre boucle un message qui indique où vous en êtes dans la récupération des articles.

Vérifiez que votre liste `years` contient bien autant d'éléments que la liste `pmids`.

Calculez maintenant le nombre de publications par année. Vous créerez pour cela un dictionnaire `freq` qui aura pour clé les années (oui, une clé de dictionnaire peut aussi être un entier) et pour valeur le nombre de publications associées à une année donnée.

Créez une liste `x` qui contient les clés du dictionnaire `freq`. Ordonnez les valeurs dans `x` avec la méthode `.sort()`. Créez maintenant une seconde liste `y` qui contient, dans l'ordre, le nombre de publications associées à chaque année. Bien évidemment, les listes `x` et `y` doivent avoir la même taille. Au fait, en quelle année la barstar apparaît pour la première fois dans une publication scientifique ?

Ensuite, avec le module *matplotlib* (que nous aborderons prochainement), vous allez pouvoir afficher la distribution des publications en fonction des années :

7. https://www.ncbi.nlm.nih.gov/pubmed/
8. https://www.ncbi.nlm.nih.gov/pubmed/29701945

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.bar(x, y)
```

Vous pouvez également ajouter un peu de cosmétique et enregistrer le graphique sur votre disque dur :

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.bar(x, y)

# Étiquetage des axes.
ax.set_xlabel("Années")
ax.set_ylabel("Nombre de publications")

# Ajout du titre du graphique.
ax.set_title("Distribution des publications qui mentionnent la barstar")

# Enregistrement sur le disque.
fig.savefig("distribution_barstar_annee.png")
```

CHAPITRE 20

Module NumPy

Le module *NumPy*[1] est incontournable en bioinformatique. Il permet d'effectuer des calculs sur des vecteurs ou des matrices, élément par élément, via un nouveau type d'objet appelé *array*.

20.1 Installation et convention

Contrairement aux modules vus précédemment, *NumPy* n'est pas fourni avec la distribution Python de base. Avec la distribution Miniconda que nous vous conseillons d'utiliser (consultez pour cela la documentation en ligne[2]), vous pouvez rapidement l'installer avec la commande :

```
$ conda install -c conda-forge numpy
```

Dans ce chapitre, nous vous montrerons quelques exemples d'utilisation du module *NumPy* pour vous convaincre de sa pertinence. Ces exemples seront exécutés dans un *notebook* Jupyter.

```
1  Les cellules de code apparaitront de cette manière
2  dans un notebook Jupyter, avec des numéros de lignes à gauche.
```

```
Les résultats seront affichés de cette manière,
éventuellement sur plusieurs lignes.
```

20.2 Chargement du module

On charge le module *NumPy* avec la commande :

```
1  import numpy
```

Par convention, on utilise `np` comme nom raccourci pour *NumPy* :

```
1  import numpy as np
```

20.3 Objets de type *array*

Les objets de type *array* correspondent à des tableaux à une ou plusieurs dimensions et permettent d'effectuer du calcul vectoriel. La fonction `array()` convertit un conteneur (comme une liste ou un tuple) en un objet de type *array*.

1. http://numpy.scipy.org/
2. https://python.sdv.u-paris.fr/livre-dunod

Voici un exemple de conversion d'une liste à une dimension en objet *array* :

```
import numpy as np
a = [1, 2, 3]
np.array(a)
```

```
array([1, 2, 3])
```

```
b = np.array(a)
b
```

```
array([1, 2, 3])
```

```
type(b)
```

```
numpy.ndarray
```

Nous avons converti la liste `[1, 2, 3]` en *array*. La fonction `np.array()` accepte aussi comme argument un tuple, ou un objet de type *range*.

Par ailleurs, lorsqu'on demande à Python d'afficher le contenu d'un objet *array*, le mot `array` et les symboles `([` et `])` sont utilisés pour le distinguer d'une liste (délimitée par les caractères `[` et `]`) ou d'un tuple (délimité par les caractères `(` et `)`).

Remarque

Un objet *array* ne contient que des données homogènes, c'est-à-dire d'un type identique. Il est possible de créer un objet *array* à partir d'une liste contenant des entiers et des chaînes de caractères, mais, dans ce cas, toutes les valeurs seront comprises par *NumPy* comme des chaînes de caractères :

```
a = np.array([1, 2, "tigre"])
a
```

```
array(['1', '2', 'tigre'], dtype='<U21')
```

Dans cet exemple, toutes les valeurs du *array* sont entre guillemets, indiquant qu'il s'agit de chaînes de caractères.

De même, il est possible de créer un objet *array* à partir d'une liste constituée d'entiers et de *floats*, mais toutes les valeurs seront alors comprises par *NumPy* comme des *floats* :

```
b = np.array([1, 2, 3.5])
b
```

```
array([1. , 2. , 3.5])
```

Ici, la notation `1.` indique qu'il s'agit du *float* `1.0000...` et pas de l'entier `1`.

Sur un modèle similaire à la fonction `range()`, la fonction `arange()` permet de construire un *array* à une dimension :

```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Comme avec `range()`, on peut spécifier en argument une borne de début, une borne de fin et un pas :

```
np.arange(10, 0, -1)
```

```
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
```

Un autre avantage de la fonction `arange()` est qu'elle génère des objets *array* qui contiennent des entiers ou des *floats* (ce qui n'est pas possible avec `range()`) selon l'argument qu'on lui passe. D'abord un entier :

```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Puis un *float* :

```
np.arange(10.0)
```

```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
```

La différence fondamentale entre un objet *array* à une dimension et une liste (ou un tuple) est que celui-ci est considéré comme un **vecteur**. Par conséquent, on peut effectuer des opérations vectorielles **élément par élément** sur ce type d'objet, ce qui est bien commode lorsqu'on analyse de grandes quantités de données. Regardez ces exemples :

```
v = np.arange(4)
v
```

```
array([0, 1, 2, 3])
```

On ajoute `1` à **chacun** des éléments de l'*array* `v` :

```
v + 1
```

```
array([1, 2, 3, 4])
```

On multiplie par `2` **chacun** des éléments de l'*array* `v` :

```
v * 2
```

```
array([0, 2, 4, 6])
```

Avec les listes, ces opérations n'auraient été possibles qu'en utilisant des boucles. Nous vous encourageons donc à utiliser dorénavant les objets *array* lorsque vous aurez besoin de faire des opérations élément par élément.

Il est aussi possible de multiplier deux *arrays* entre eux. Le résultat correspond alors à la multiplication **élément par élément** des deux *arrays* initiaux :

```
v * v
```

```
array([0, 1, 4, 9])
```

20.3.1 *Array* et dimensions

Il est aussi possible de construire des objets *arrays* à deux dimensions, il suffit de passer en argument une liste de listes à la fonction `array()` :

```
w = np.array([[1, 2], [3, 4], [5, 6]])
w
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

On peut aussi créer des tableaux à trois dimensions en passant comme argument à la fonction `array()` une liste de listes de listes :

```
x = np.array([[[1, 2], [2, 3]], [[4, 5], [5, 6]]])
x
```

```
array([[[1, 2],
        [2, 3]],

       [[4, 5],
        [5, 6]]])
```

La fonction `array()` peut créer des tableaux à n'importe quel nombre de dimensions. Toutefois, cela devient vite compliqué lorsqu'on dépasse trois dimensions. Retenez qu'un objet *array* à une dimension peut être assimilé à un **vecteur**, un *array* à deux dimensions à une **matrice**. On peut généraliser ces objets mathématiques avec un nombre arbitraires de dimensions, on parle alors de **tenseur**, qui sont représentés avec NumPy en *array* à n dimensions. Nous nous focaliserons dans la suite sur des *arrays* à une dimension (1D) ou deux dimensions (2D).

Avant de continuer, il est important de définir comment sont organisés ces *arrays* 2D qui représentent des matrices. Il s'agit de tableaux de nombres qui sont organisés en lignes et en colonnes comme le montre la figure 20.1. Les indices indiqués dans cette figure seront définis un peu plus loin dans la rubrique *Indices*.

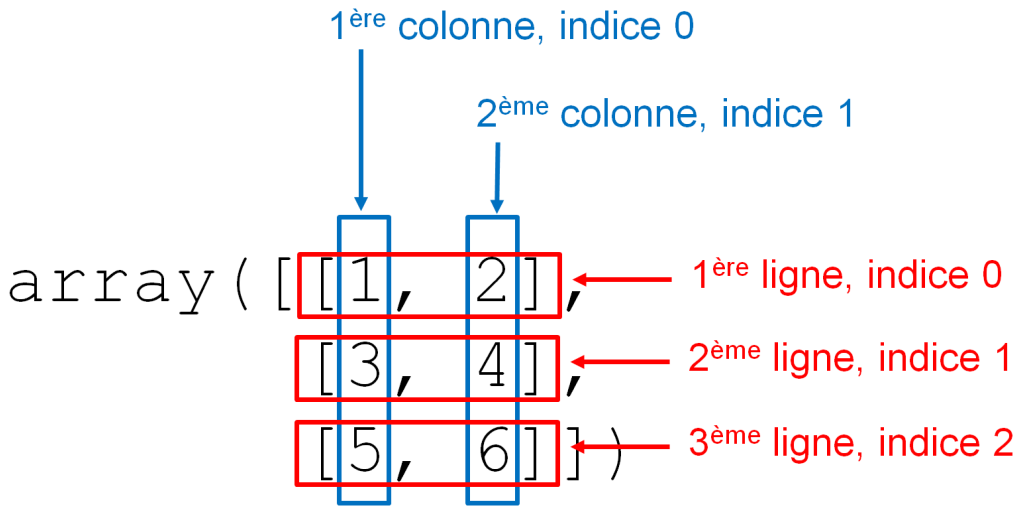


FIGURE 20.1 – Définition des lignes et colonnes dans un *array* 2D.

Voici quelques attributs intéressants pour décrire un objet *array* :

```
1  v = np.arange(4)
2  v
```

```
array([0, 1, 2, 3])
```

```
1  w = np.array([[1, 2], [3, 4], [5, 6]])
2  w
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

L'attribut `.ndim` renvoie le nombre de dimensions de l'*array*. Par exemple, 1 pour un vecteur et 2 pour une matrice :

```
1  v.ndim
```

```
1
```

```
1  w.ndim
```

```
2
```

L'attribut `.shape` renvoie les dimensions sous forme d'un tuple. Dans le cas d'une matrice (*array* à deux dimensions), la première valeur du tuple correspond au nombre de lignes et la seconde au nombre de colonnes.

```
1  v.shape
```

```
(4,)
```

```
1  w.shape
```

```
(3, 2)
```

Enfin, l'attribut `.size` renvoie le nombre total d'éléments contenus dans l'`array` :

```
1  v.size
```

```
4
```

```
1  w.size
```

```
6
```

20.3.2 Redimensionnement d'*array*

La méthode `.reshape()` renvoie un nouvel *array* avec les dimensions spécifiées en argument :

```
1  a = np.arange(0, 6)
2  a
```

```
array([0, 1, 2, 3, 4, 5])
```

```
1  a.shape
```

```
(6,)
```

```
1  b = a.reshape((2, 3))
2  b
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

```
1  b.shape
```

```
(2, 3)
```

```
1  a
```

```
array([0, 1, 2, 3, 4, 5])
```

Notez bien que l'*array* initial `a` n'a pas été modifié et que `a.reshape((2, 3))` n'est pas la même chose que `a.reshape((3, 2))` :

```
c = a.reshape((3, 2))
c
```

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
c.shape
```

```
(3, 2)
```

La méthode `.reshape()` attend que les nouvelles dimensions soient **compatibles** avec la dimension initiale de l'objet *array*, c'est-à-dire que le nombre d'éléments contenus dans les différents *arrays* soit le même. Dans nos exemples précédents, $6 = 2 \times 3 = 3 \times 2$.

Si les nouvelles dimensions ne sont pas compatibles avec les dimensions initiales, la méthode `.reshape()` génère une erreur.

```
a = np.arange(0, 6)
a
```

```
array([0, 1, 2, 3, 4, 5])
```

```
a.shape
```

```
(6,)
```

```
d = a.reshape((3, 4))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[36], line 1
----> 1 d = a.reshape((3, 4))

ValueError: cannot reshape array of size 6 into shape (3,4)
```

La méthode `.resize()`, par contre, ne déclenche pas d'erreur dans une telle situation et ajoute des 0 jusqu'à ce que le nouvel *array* soit rempli, ou bien coupe la liste initiale :

```
a = np.arange(0, 6)
a.shape
```

```
(6,)
```

```
a.resize((3, 3), refcheck=False)
a.shape
```

```
(3, 3)
```

```
a
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [0, 0, 0]])
```

```
b = np.arange(0, 10)
b.shape
```

```
(10,)
```

```
b.resize((2, 3), refcheck=False)
b.shape
```

```
(2, 3)
```

```
b
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

Attention

- Cette modification de la forme de l'*array* par la méthode `.resize()` est faite « sur place » (*in place*), c'est-à-dire que la méthode ne renvoie rien, mais l'*array* initial est bel et bien modifié (comme des méthodes sur les listes telles que la méthode `.reverse()`, voir le chapitre 13 *Plus sur les listes*).
- Si l'option `refcheck=False` n'est pas présente, Python peut parfois renvoyer une erreur s'il existe des références vers l'*array* qu'on souhaite modifier.

Enfin, il existe la fonction `np.resize()` qui, dans le cas d'un nouvel *array* plus grand que l'*array* initial, va répéter l'*array* initial afin de remplir les cases manquantes :

```
a = np.arange(0, 6)
a.shape
```

```
(6,)
```

```
c = np.resize(a, (3, 5))
c.shape
```

```
(3, 5)
```

```
c
```

```
array([[0, 1, 2, 3, 4],
       [5, 0, 1, 2, 3],
       [4, 5, 0, 1, 2]])
```

```
a
```

```
array([0, 1, 2, 3, 4, 5])
```

Notez que la fonction `np.resize()` renvoie un nouvel *array* mais ne modifie pas l'*array* initial, contrairement à la méthode `.resize()`, décrite ci-dessus.

Remarque

Depuis le début de ce chapitre, nous avons toujours montré l'affichage d'un *array* tel quel dans un *notebook* Jupyter :

```
a = np.array(range(10))
a
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
a2 = np.ones((3, 3))
a2
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Nous avons déjà indiqué que Python affiche systématiquement le mot *array* ainsi que les parenthèses, crochets et virgules pour séparer les éléments. Toutefois, si vous utilisez la fonction `print()`, l'affichage sera différent. Le mot *array*, les parenthèses et les virgules disparaissent :

```
print(a)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

```
print(a2)
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

Ceci peut amener des confusions, en particulier entre un *array* 1D :

```
[0 1 2 3 4 5 6 7 8 9]
```

et une liste contenant les mêmes éléments :

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Dans ce cas, seule la présence ou l'absence de virgules permet de savoir s'il s'agit d'un *array* ou d'une liste.

20.3.3 Méthodes de calcul sur les *arrays* et l'argument `axis`

Chaque *array NumPy* possède une multitude de méthodes. Nombre d'entre elles permettent de faire des calculs de base comme `.mean()` pour la moyenne, `.sum()` pour la somme, `.std()` pour l'écart-type, `.max()` pour extraire le maximum, `.min()` pour extraire le minimum, etc. La liste exhaustive est disponible en ligne[3]. Par défaut, chacune de ces méthodes effectuera l'opération sur l'*array* entier, quelle que soit sa dimensionnalité. Par exemple :

```
import random
ma_liste = list(range(8))
random.shuffle(ma_liste)
ma_liste
```

```
[2, 7, 6, 4, 0, 3, 1, 5]
```

```
a = np.resize(ma_liste, (4, 2))
a
```

3. https://numpy.org/doc/stable/reference/arrays.ndarray.html#calculation

```
array([[2, 7],
       [6, 4],
       [0, 3],
       [1, 5]])
```

```
a.max()
```

```
7
```

La méthode `.max()` a bien renvoyé la valeur maximale 7. Un argument *très* utile existant dans toutes ces méthodes est `axis`. Pour un *array* 2D, `axis=0` signifie qu'on fera l'opération le long de l'axe 0, à savoir les lignes. C'est-à-dire que l'opération se fait en variant les lignes. On récupère ainsi une valeur par colonne :

```
a.max(axis=0)
```

```
array([6, 7])
```

Dans l'*array* 1D récupéré, le premier élément vaut 6 (maximum de la 1ère colonne) et le second vaut 7 (maximum de la seconde colonne).

Avec `axis=1`, on fait une opération similaire, mais en faisant varier les colonnes. On récupère ainsi une valeur par ligne :

```
a.max(axis=1)
```

```
array([7, 6, 3, 5])
```

L'*array* 1D récupéré a quatre éléments correspondant au maximum de chaque ligne.

On comprend la puissance de l'argument `axis`. À nouveau, il est possible, en une ligne, de faire des calculs qui pourraient être fastidieux avec les listes traditionnelles.

20.3.4 Indices

Pour récupérer un ou plusieurs élément(s) d'un objet *array*, vous pouvez utiliser les indices, de la même manière qu'avec les listes :

```
a = np.arange(10)
a
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
a[1]
```

```
1
```

L'utilisation des tranches est aussi possible :

```
a[5:]
```

```
array([5, 6, 7, 8, 9])
```

Ainsi que les pas :

```
a[::2]
```

```
array([0, 2, 4, 6, 8])
```

Dans le cas d'un objet *array* à deux dimensions, vous pouvez récupérer une ligne complète (d'indice *i*), une colonne complète (d'indice *j*) ou bien un seul élément. La figure 20.1 montre comment sont organisés les indices des lignes et des colonnes :

```
a = np.array([[1, 2], [3, 4]])
a
```

```
array([[1, 2],
       [3, 4]])
```

```
a[:,0]
```

```
array([1, 3])
```

```
a[0,:]
```

```
array([1, 2])
```

La syntaxe `a[i,:]` renvoie la ligne d'indice `i`, et `a[:,j]` renvoie la colonne d'indice `j`. Les tranches sont aussi utilisables sur un *array* à deux dimensions.

```
a[1, 1]
```

```
4
```

La syntaxe `a[i, j]` renvoie l'élément à la ligne d'indice `i` et à la colonne d'indice `j`. Notez que *NumPy* suit la convention mathématiques des matrices[4], à savoir, qu'**on définit toujours un élément par sa ligne puis par sa colonne**. En mathématiques, l'élément a_{ij} d'une matrice A se trouve à la i^{me} ligne et à la j^{me} colonne :

Remarque

Pour un *array* 2D, si un seul indice est donné, par exemple `a[i]`, on récupère la ligne d'indice `i` sous forme d'*array* 1D :

```
a = np.array([[1, 2], [3, 4]])
a
```

```
array([[1, 2],
       [3, 4]])
```

```
a[0]
```

```
array([1, 2])
```

```
a[1]
```

```
array([3, 4])
```

Pour cette raison, la syntaxe `a[i][j]` est également valide pour récupérer un élément :

```
a[1, 1]
```

```
4
```

4. https://fr.wikipedia.org/wiki/Matrice_(math%C3%A9matiques)#D%C3%A9finitions

```
a[1][1]
```

```
4
```

Nous vous recommandons la syntaxe `a[i, j]`, qui est plus proche de la définition mathématique d'un élément de matrice [5].

20.3.5 Copie d'*arrays*

Comme pour les listes, nous attirons votre attention sur la copie d'*arrays* :

```
a = np.arange(5)
a
```

```
array([0, 1, 2, 3, 4])
```

```
b = a
b[2] = -300
b
```

```
array([   0,    1, -300,    3,    4])
```

```
a
```

```
array([   0,    1, -300,    3,    4])
```

Attention

Par défaut la copie d'*arrays* se fait par référence, comme pour tous les conteneurs en Python (listes, *tuples*, dictionnaires, etc.).

Afin d'éviter le problème, vous pouvez soit utiliser la fonction `np.array()`, qui crée une nouvelle copie distincte de l'*array* initial, soit la fonction `copy.deepcopy()`, comme pour les listes (voir chapitre 12 *Plus sur les listes*) :

```
a = np.full((2, 2), 0)
a
```

```
array([[0, 0],
       [0, 0]])
```

```
b = np.array(a)
b[1, 1] = -300
import copy
c = copy.deepcopy(a)
c[1, 1] = -500
a
```

```
array([[0, 0],
       [0, 0]])
```

```
b
```

5. https://fr.wikipedia.org/wiki/Matrice_(math%C3%A9matiques)#D%C3%A9finitions

```
array([[   0,    0],
       [   0, -300]])
```

```
c
```

```
array([[   0,    0],
       [   0, -500]])
```

La fonction `np.full()` est expliquée dans la rubrique suivante.

Remarque

L'instruction `b = np.array(a)` réalise bien une copie distincte de l'*array* `a`, quelle que soit sa dimensionnalité. Ceci n'était pas le cas avec la fonction `list()` pour les copies de listes à partir de la dimension deux (liste de listes) :

```
liste_1 = [[0, 0], [1, 1]]
liste_2 = list(liste_1)
import copy
liste_3 = copy.deepcopy(liste_1)
liste_1[1][1] = -365
liste_2
```

```
[[0, 0], [1, -365]]
```

```
liste_3
```

```
[[0, 0], [1, 1]]
```

20.4 Construction automatique de matrices

Il est parfois pénible de construire une matrice (*array* à deux dimensions) à l'aide d'une liste de listes. Le module *NumPy* possède quelques fonctions pratiques pour initialiser des matrices. Par exemple, Les fonctions `zeros()` et `ones()` construisent des objets *array* contenant des 0 ou des 1. Il suffit de leur passer en argument un tuple indiquant les dimensions voulues :

```
np.zeros((2, 3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

```
np.ones((3, 3))
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Par défaut, les fonctions `zeros()` et `ones()` génèrent des *floats*, mais vous pouvez demander des entiers en passant le type (par exemple `int`, `float`, etc.) en second argument :

```
np.zeros((2,3), int)
```

```
array([[0, 0, 0],
       [0, 0, 0]])
```

Enfin, si vous voulez construire une matrice avec autre chose que des 0 ou des 1, vous avez à votre disposition la fonction `full()` :

```
1  np.full((2, 3), 7, int)
```

```
array([[7, 7, 7],
       [7, 7, 7]])
```

```
1  np.full((2, 3), 7, float)
```

```
array([[ 7.,  7.,  7.],
       [ 7.,  7.,  7.]])
```

Nous construisons ainsi une matrice constituée de 2 lignes et 3 colonnes. Celle-ci ne contient que le chiffre 7 sous formes d'entiers (`int`) dans le premier cas et de *floats* dans le second.

20.5 Chargement d'un *array* depuis un fichier

Le module *NumPy* contient aussi des fonctions pour lire des données à partir de fichiers et créer des *arrays* automatiquement. C'est très pratique, car la plupart du temps les données que l'on analyse proviennent de fichiers. La fonction la plus simple à prendre en main est `np.loadtxt()`. Celle-ci lit un fichier organisé en lignes et colonnes. Par exemple, imaginons que nous ayons un fichier `donnees.dat` contenant :

```
  1   7 310
 15  -4  35
 78  95  79
```

La fonction prend en argument le nom du fichier et renvoie un *array* 2D directement :

```
1  np.loadtxt("donnees.dat")
```

```
array([[  1.,   7., 310.],
       [ 15.,  -4.,  35.],
       [ 78.,  95.,  79.]])
```

Pratique, non ? Attention toutefois aux points suivants :

- Chaque ligne doit avoir le même nombre de colonnes, la fonction ne gère pas les données manquantes.
- Chaque donnée est convertie en *float*, donc si une chaîne de caractères est rencontrée la fonction renvoie une erreur.
- Par défaut, les données doivent être séparées par n'importe quelle combinaison d'espace(s) et/ou de tabulations.

Nous vous conseillons de consulter la documentation complète[6] de cette fonction. En effet, `np.loadtxt()` contient de nombreux arguments permettant de récupérer telles ou telles lignes ou colonnes, d'ignorer des lignes de commentaire, de changer le séparateur par défaut (par exemple la virgule `,` pour les fichiers .csv)... qui peuvent se révéler utiles.

L'opération inverse qui consiste à sauver un *array* dans un fichier se fait avec la fonction `np.savetxt()` :

```
1  a = np.reshape(range(1, 10), (3, 3))
2  a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
1  np.savetxt("out.dat", a)
```

Ceci générera le fichier `out.dat` contenant les lignes suivantes :

```
1.00000000000000000e+00 2.00000000000000000e+00 3.00000000000000000e+00
4.00000000000000000e+00 5.00000000000000000e+00 6.00000000000000000e+00
7.00000000000000000e+00 8.00000000000000000e+00 9.00000000000000000e+00
```

6. https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html

La fonction `np.savetxt()` écrit par défaut les données comme des *floats* en notation scientifique. Il existe de nombreuses options possibles[7] permettant de changer le format, les séparateurs, etc.

Pour aller plus loin

Il existe d'autres fonctions plus avancées telles que np.genfromttxt()[8], gérant les données manquantes, ou encore np.load()[9] et np.fromfile()[10], permettant de lire des données au format binaire. De même, il existe des fonctions ou méthodes permettant d'écrire au format binaire : np.save()[11] ou .tofile()[12]. Le format binaire possède en général l'extension `.npy` ou `.npz` lorsque les données sont compressées. L'avantage d'écrire au format binaire est que cela prend moins de place pour de gros tableaux de données.

20.6 Concaténation d'*arrays*

Il peut être très utile de concaténer un ou plusieurs *arrays*. Il existe pour cela plusieurs fonctions dans *NumPy*, nous développerons celle qui nous parait la plus intuitive et directe : `np.concatenate()`.

Pour les *arrays* 1D, `np.concatenate()` prend en argument un tuple contenant les *arrays* à concaténer :

```
a1 = np.array((0, 1))
a2 = np.array((3, 4))
a1
```

```
array([0, 1])
```

```
a2
```

```
array([3, 4])
```

```
np.concatenate((a1, a2))
```

```
array([0, 1, 3, 4])
```

L'ordre de la concaténation est important :

```
np.concatenate((a2, a1))
```

```
array([3, 4, 0, 1])
```

```
np.concatenate((a1, a2, a1, a2))
```

```
array([0, 1, 3, 4, 0, 1, 3, 4])
```

Pour les *arrays* 2D, ça se complique un peu, car on peut concaténer des lignes ou des colonnes ! Ainsi, `np.concatenate()` prend un argument optionnel, à savoir `axis`. Comme nous l'avions expliqué plus haut, celui-ci va indiquer à *NumPy* si on veut concaténer le long de l'axe 0 (les lignes) ou le long de l'axe 1 (les colonnes). Voyons un exemple :

```
a1 = np.reshape(np.array(range(6)), (3, 2))
a2 = a1 * 5
a1
```

7. https://numpy.org/doc/stable/reference/generated/numpy.savetxt.html
8. https://numpy.org/doc/stable/reference/generated/numpy.genfromtxt.html
9. https://numpy.org/doc/stable/reference/generated/numpy.load.html
10. https://numpy.org/doc/stable/reference/generated/numpy.fromfile.html
11. https://numpy.org/doc/stable/reference/generated/numpy.save.html
12. https://numpy.org/doc/stable/reference/generated/numpy.ndarray.tofile.html#numpy.ndarray.tofile

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
1 a2
```

```
array([[ 0,  5],
       [10, 15],
       [20, 25]])
```

On concatène d'abord par ligne (`axis=0`), c'est-à-dire qu'on ajoute les lignes du second *array* `a2` à celles de l'*array* `a1` :

```
1 np.concatenate((a1, a2), axis=0)
```

```
array([[ 0,  1],
       [ 2,  3],
       [ 4,  5],
       [ 0,  5],
       [10, 15],
       [20, 25]])
```

Ensuite, on concatène par colonne (`axis=1`). Attention, il vaut bien veiller à ce que la concaténation soit possible en terme de dimensionalité. Par exemple, lors de la concaténation par colonne, il faut que les deux *arrays* `a1` et `a2` aient le même nombre de lignes :

```
1 np.concatenate((a1, a2), axis=1)
```

```
array([[ 0,  1,  0,  5],
       [ 2,  3, 10, 15],
       [ 4,  5, 20, 25]])
```

Ces opérations de concaténation sont très importantes. On les utilise par exemple si on a des données dans plusieurs fichiers différents et qu'on veut les agréger dans un *array* unique. On verra qu'on peut faire le même genre de chose avec les fameux *Dataframes* du module *pandas*. Lisez bien également les recommandations dans la dernière rubrique *17.1.10 Quelques conseils* sur quand utiliser la concaténation d'*arrays* avec *NumPy*.

20.7 Un peu d'algèbre linéaire

Après avoir manipulé les objets *array* comme des vecteurs et des matrices, voici quelques fonctions pour faire de l'algèbre linéaire.

La fonction `transpose()` renvoie la transposée[13] d'un *array*. Par exemple, pour une matrice :

```
1 a = np.resize(np.arange(1, 10), (3, 3))
2 a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
1 np.transpose(a)
```

```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

13. https://fr.wikipedia.org/wiki/Matrice_transpos%C3%A9e

Tout objet *array* possède un attribut `.T` qui contient la transposée, il est ainsi possible d'utiliser cette notation objet plus compacte :

```
a.T
```

```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

La fonction `dot()` permet de multiplier deux matrices[14] :

```
a = np.resize(np.arange(4), (2, 2))
a
```

```
array([[0, 1],
       [2, 3]])
```

```
np.dot(a, a)
```

```
array([[ 2,  3],
       [ 6, 11]])
```

```
a * a
```

```
array([[0, 1],
       [4, 9]])
```

Notez bien que `dot(a, a)` renvoie le **produit matriciel** entre deux matrices, alors que l'opération `a * a` renvoie le produit **élément par élément**.

Remarque

Dans le module *NumPy*, il existe également des objets de type *matrix* pour lesquels les multiplications de matrices sont différents, mais nous ne les aborderons pas ici.

Pour toutes les opérations suivantes, nous utiliserons des fonctions du sous-module *linalg* de *NumPy*.

La fonction `diag()` permet de générer une matrice diagonale :

```
a = np.diag((1, 2, 3))
a
```

```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

La fonction `inv()` renvoie l'inverse d'une matrice carrée[15] :

```
np.linalg.inv(a)
```

```
array([[1.        , 0.        , 0.        ],
       [0.        , 0.5       , 0.        ],
       [0.        , 0.        , 0.33333333]])
```

La fonction `det()` renvoie le déterminant[16] d'une matrice carrée :

14. https://fr.wikipedia.org/wiki/Produit_matriciel#Produit_matriciel_ordinaire
15. https://fr.wikipedia.org/wiki/Matrice_inversible
16. https://fr.wikipedia.org/wiki/Calcul_du_d%C3%A9terminant_d%27une_matrice

```
np.linalg.det(a)
```

```
6.0
```

Enfin, la fonction `eig()` renvoie les vecteurs et valeurs propres :

```
np.linalg.eig(a)
```

```
EigResult(eigenvalues=array([1., 2., 3.]), eigenvectors=array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]]))
```

La fonction `eig()` renvoie un objet `EigResult`, qui contient les valeurs propres (`eigenvalues`) et les vecteurs propres (`eigenvectors`), qu'on peut ensuite récupérer par affectation multiple :

```
eigvals, eigvecs = np.linalg.eig(a)
eigvals
```

```
array([1., 2., 3.])
```

`eigvals` est un *array* 1D contenant les trois valeurs propres.

```
eigvecs
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

`eigvecs` est un *array* 2D contenant les trois vecteurs propres (un par ligne).

20.8 Parcours de matrice et affectation de lignes et colonnes

Lorsqu'on a une matrice, on est souvent amené à la parcourir par ligne ou par colonne. *NumPy* permet d'itérer directement sur les lignes d'une *array* :

```
a = np.reshape(np.arange(1, 10), (3, 3))
a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
for row in a:
    print(row, type(row))
```

```
[1 2 3] <class 'numpy.ndarray'>
[4 5 6] <class 'numpy.ndarray'>
[7 8 9] <class 'numpy.ndarray'>
```

À chaque itération, la variable `row` est un *array* 1D correspondant à chaque ligne de la matrice `a`. Cela est du au fait que l'utilisation d'un indiçage unique `a[i]` pour un *array* 2D correspond à sa ligne d'indice `i` (voir la rubrique *Indices* ci-dessus).

Pour itérer sur les colonnes, on peut utiliser l'astuce d'itérer sur la transposée de l'*array* `a`, c'est-à-dire `a.T` :

```
for col in a.T:
    print(col, type(col))
```

```
[1 4 7] <class 'numpy.ndarray'>
[2 5 8] <class 'numpy.ndarray'>
[3 6 9] <class 'numpy.ndarray'>
```

À chaque itération, la variable `col` est un *array* 1D correspondant à chaque colonne de `a`.

On se souvient de l'affectation multiple `x, y = 1, 2` qui permettait d'affecter des valeurs à plusieurs variables à la fois. Il est possible d'utiliser cette fonctionnalité aussi avec les *arrays NumPy* :

```
a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
a1, a2, a3 = a
a1
```

```
array([1, 2, 3])
```

```
a2
```

```
array([4, 5, 6])
```

```
a3
```

```
array([7, 8, 9])
```

Par défaut, l'affectation multiple se fait sur les lignes de l'*array* 2D. Cette fonctionnalité s'explique à nouveau par le fait que pour *NumPy*, `a[i]` correspond à la ligne d'indice `i` d'un *array* 2D.

Pour utiliser l'affectation multiple sur les colonnes, il suffit d'utiliser la transposée `a.T` :

```
c1, c2, c3 = a.T
c1
```

```
array([1, 4, 7])
```

```
c2
```

```
array([2, 5, 8])
```

```
c3
```

```
array([3, 6, 9])
```

20.9 Masques booléens

Une fonctionnalité puissante des *arrays NumPy* est l'utilisation des **masques booléens**. Avant de les définir, il est important d'introduire le concept d'*arrays* de booléens. Jusqu'à maintenant nous avions définis uniquement des *arrays* avec des types numériques *int* ou *float*. Il est tout à fait possible de définir des *arrays* de booléens. La fonction `np.full()` vue précédemment nous permet d'en construire facilement :

```
np.full((2, 2), True)
```

```
array([[ True,  True],
       [ True,  True]])
```

```
np.full((2, 2), False)
```

```
array([[False, False],
       [False, False]])
```

Au premier abord, nous n'en voyons pas forcément l'utilité… Mais qu'en est-il lorsqu'on utilise les opérateurs de comparaison avec un *array* ? Et bien cela renvoie un *array* de booléens !

```
a = np.reshape(np.arange(1, 10), (3, 3))
a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
a > 5
```

```
array([[False, False, False],
       [False, False,  True],
       [ True,  True,  True]])
```

```
a == 2
```

```
array([[False,  True, False],
       [False, False, False],
       [False, False, False]])
```

Tous les éléments de l'*array* satisfaisant la condition seront à `True`, les autres à `False`. Il est même possible de combiner plusieurs conditions avec les opérateurs logiques `&` et `|` (respectivement **ET** et **OU**) :

```
a = np.reshape(np.arange(1, 10), (3, 3))
a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
(a > 3) & (a % 2 == 0)
```

```
array([[False, False, False],
       [ True, False,  True],
       [False,  True, False]])
```

```
(a > 3) | (a % 2 == 0)
```

```
array([[False,  True, False],
       [ True,  True,  True],
       [ True,  True,  True]])
```

- Les opérateurs logiques `&` et `|` s'appliquent sur les *arrays* et sont différents des opérateurs logiques `and` et `or`, qui eux s'appliquent sur les booléens (`True` ou `False`).
- Il est conseillé de mettre entre parenthèses chaque condition afin d'éviter les ambiguïtés.

Maintenant que les *arrays* de booléens ont été introduits, nous pouvons définir les masques booléens :

Définition

Les masques booléens sont des *arrays* de booléens qui sont utilisés en tant qu'« indice » d'un *array* initial. Cela permet de récupérer ou de modifier une partie de l'*array* initial.

Concrètement, il suffira d'utiliser un *array* et un opérateur de comparaison entre les crochets qui étaient dédiés à l'indiçage :

```
a = np.reshape(np.arange(1, 10), (3, 3))
a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Pour isoler tous les éléments de l'*array* `a` qui sont supérieurs à 5 :

```
a[a > 5]
```

```
array([6, 7, 8, 9])
```

Pour isoler tous les éléments de l'*array* `a` qui sont égaux à 2 :

```
a[a == 2]
```

```
array([2])
```

Pour isoler tous les éléments de l'*array* `a` qui sont non nuls :

```
a[a != 0]
```

```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

À chaque fois, on ne récupère que les éléments de l'*array* `a` qui satisfont la sélection. Toutefois, il est important de remarquer que l'*array* renvoyé perd la dimensionnalité de l'array `a` initial, il s'agit systématiquement d'un *array* 1D.

La grande puissance de ce mécanisme est que l'on peut utiliser les masques booléens pour modifier les éléments que l'on sélectionne :

```
a = np.reshape(np.arange(1, 10), (3, 3))
a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

On sélectionne les éléments de l'*array* `a` supérieurs à 5 :

```
a[a > 5]
```

```
array([6, 7, 8, 9])
```

On affecte la valeur `-1` aux éléments de l'*array* `a` supérieurs à 5 :

```
a[a > 5] = -1
a
```

```
array([[ 1,  2,  3],
       [ 4,  5, -1],
       [-1, -1, -1]])
```

On peut bien sûr combiner plusieurs conditions avec les opérateurs logiques :

```
1  a = np.reshape(np.arange(1, 10), (3, 3))
2  a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
1  a[(a > 3) | (a % 2 == 0)] = 0
2  a
```

```
array([[1, 0, 3],
       [0, 0, 0],
       [0, 0, 0]])
```

Ce mécanisme de sélection avec des masques booléens se révèle très puissant pour manipuler de grandes quantités de données. On verra qu'il peut être également utilisé avec les *Dataframes* du module *pandas*.

Remarque

Les masques booléens ne doivent pas être confondus avec les *masked arrays*[17], qui sont des *arrays* dans lesquels on peut trouver des valeurs manquantes ou invalides.

Enfin, une application possible des masques est de « binariser » une matrice de nombre :

```
1  import random
2  import numpy as np
3  a = np.resize([random.random() for i in range(16)], (4, 4))
4  a
```

```
array([[0.58704728, 0.50212977, 0.70652863, 0.24158108],
       [0.93102132, 0.41864373, 0.45807961, 0.98288744],
       [0.48198211, 0.16877376, 0.14431518, 0.74784176],
       [0.92913469, 0.08383269, 0.10670144, 0.14554345]])
```

```
1  seuil = 0.3
2  a[a < seuil] = 0
3  a[a > seuil] = 1
4  a
```

```
array([[1., 1., 1., 0.],
       [1., 1., 1., 1.],
       [1., 0., 0., 1.],
       [1., 0., 0., 0.]])
```

On obtient ce résultat avec deux lignes de code en utilisant des *arrays*, alors qu'il aurait fallu faire des boucles avec des listes classiques.

20.10 Quelques conseils

Nous vous avons présenté une petite partie du module *NumPy*, mais vous avez pu en constater son extraordinaire puissance. On pourrait au premier abord être tenté d'abandonner les listes, toutefois elles gardent toute leur importance.

17. https://numpy.org/doc/1.18/reference/maskedarray.html

Alors, quand utiliser les listes ou quand utiliser les *arrays NumPy* ? Voici une liste non exhaustive d'éléments qui peuvent guider votre choix :

Utilisez *NumPy* pour :

- les opérations vectorielles (éléments par éléments) ;
- lorsque vous souhaitez manipuler des objets mathématiques (vecteurs, matrices, etc.) et les outils associés (algèbre linéaire) ;
- tout ce qui est numérique de manière générale.

Utilisez les listes :

- Lorsque vous avez besoin d'un conteneur pour accumuler des valeurs (fussent-elles des sous-listes), surtout lorsqu'elles ne sont pas homogènes (c'est-à-dire du même type).
- Lorsque vous souhaitez accumuler des valeurs au fur et à mesure des itérations d'une boucle. Pour cela, la méthode `.append()` des listes est bien plus efficace que de faire grandir un *array* ligne par ligne (c'est-à-dire en ajoutant une ligne avec `np.concatenate()` à chaque itération).
- Lorsqu'on ne peut pas utiliser les fonctions de lecture de fichier de *NumPy* pour quelque raison que ce soit, il est tout à fait classique de faire grandir une liste au fur et à mesure de la lecture du fichier puis de la convertir à la fin en *array*. De manière générale, utilisez `np.concatenate()` seulement pour concaténer des gros *arrays*, pas pour ajouter une seule ligne.

Enfin, comme nous vous le conseillons depuis le début, soignez votre documentation (*docstrings*) et vos commentaires lorsque vous utilisez des *arrays*. *NumPy* permet de réaliser des opérations vectorielles de manière très compacte. Il est donc essentiel de se mettre à la place du lecteur de votre script (y compris vous dans quelques semaines ou mois) et de documenter ce que contient chaque *array* ainsi que sa dimensionnalité (1D, 2D, etc.).

Le module *NumPy* est la brique de base du calcul numérique en Python. Associé aux modules *SciPy*[18] et *matplotlib*, ainsi qu'aux *notebooks Jupyter* (voir le chapitre précédent), il permet de faire du calcul scientifique de manière très efficace. On verra dans le chapitre 22 *Module Pandas* que la puissance de *NumPy* est également utilisée par le module *pandas* pour faire de l'analyse de données.

Pour aller plus loin

- Le livre de Nicolas Rougier *From Python to Numpy*[19] est une excellente ressource pour explorer plus en détails les possibilités de *NumPy*.
- Les tutoriels[20] proposés par les développeurs de *NumPy* sont également un bon moyen de poursuivre votre exploration de cette bibliothèque incontournable en sciences.

20.11 Exercices

Conseil

Pour ces exercices, utilisez des *notebooks* Jupyter.

20.11.1 Nombres pairs et impairs

Soit `impairs` un *array NumPy* qui contient les nombres :

```
1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21
```

En une seule instruction, construisez l'*array* `pairs` dans lequel tous les éléments de `impairs` sont incrémentés de 1. Comparez ce que vous venez de faire avec l'exercice « Nombres pairs et impairs » du chapitre 5 *Boucles et comparaisons*.

18. https://scipy.org/
19. https://www.labri.fr/perso/nrougier/from-python-to-numpy/
20. https://numpy.org/numpy-tutorials/index.html

20.11.2 Distance entre deux atomes carbones alpha consécutifs de la barstar

La barstar est un inhibiteur de ribonucléase. C'est une protéine relativement simple qui contient 89 acides aminés. Sa structure tridimensionnelle, obtenue par résonance magnétique nucléaire (RMN), se trouve dans la *Protein Data Bank* (PDB) sous le code 1BTA.

L'objectif de cet exercice est de calculer la distance entre carbones alpha consécutifs le long de la chaîne peptidique avec module *NumPy* et de découvrir une anomalie.

Le morceau de code suivant vous sera utile pour extraire les coordonnées atomiques des carbones alpha de la barstar depuis un fichier PDB :

```
1  with open("1bta.pdb", "r") as f_pdb, open("1bta_CA.txt", "w") as f_CA:
2      for ligne in f_pdb:
3          if ligne.startswith("ATOM") and ligne[12:16].strip() == "CA":
4              x = ligne[30:38]
5              y = ligne[38:46]
6              z = ligne[46:54]
7              f_CA.write(f"{x} {y} {z} ")
```

- **Ligne 1**. On ouvre deux fichiers simultanément. Ici, le fichier `1bta.pdb` est ouvert en lecture (`r`) et le fichier `1bta_CA.txt` est ouvert en écriture (`w`).
- Pour chaque ligne du fichier PDB (**ligne 2**), si la ligne débute par `ATOM` et le nom de l'atome est `CA` (**ligne 3**), alors on extrait les coordonnées atomiques (**lignes 4 à 6**) et on les écrit dans le fichier `1bta_CA.txt` (**ligne 7**). Les coordonnées sont toutes enregistrées sur une seule ligne, les unes après les autres.

Voici les étapes à suivre :

1. **Extraction des coordonnées atomiques**
   - Téléchargez le fichier `1bta.pdb` qui correspond à la structure de la barstar[21] sur le site de la PDB (lien direct vers le fichier[22]).
   - Utilisez le code précédent pour extraire les coordonnées atomiques des carbones alpha de la barstar.
2. **Lecture des coordonnées**
   - Ouvrez le fichier `1bta_CA.txt` avec Python et créez une liste contenant toutes les coordonnées sous forme de *floats* avec les fonctions `split()` et `float()`.
   - Affichez à l'écran le nombre total de coordonnées.
3. **Construction de la matrice de coordonnées**
   - En ouvrant dans un éditeur de texte le fichier `1bta.pdb`, trouvez le nombre d'acides aminés qui constituent la barstar.
   - Avec la fonction `array()` du module *NumPy*, convertissez la liste de coordonnées en `array`. Avec la fonction `reshape()` de *NumPy*, construisez ensuite une matrice à deux dimensions contenant les coordonnées des carbones alpha de la barstar. Affichez les dimensions de cette matrice.
4. **Calcul de la distance**
   - Créez maintenant une matrice qui contient les coordonnées des $n-1$ premiers carbones alpha et une autre qui contient les coordonnées des $n-1$ derniers carbones alpha. Affichez les dimensions des matrices pour vérification.
   - En utilisant les opérateurs mathématiques habituels (`-`, `+`, `**2`) et les fonctions `sqrt()` et `sum()` du module *NumPy*, calculez la distance entre les atomes n et $n+1$.
   - Pour chaque atome, affichez le numéro de l'atome et la distance entre carbones alpha consécutifs avec un chiffre après la virgule. Repérez la valeur surprenante.

20.11.3 Jour le plus chaud

Le fichier `temperature.dat`[23] contient un relevé de quatre températures pour chaque jour de la semaine :

```
Lun 12 11 14 12
Mar 12 10 14 11
Mer 11 11 14 13
[...]
```

21. http://www.rcsb.org/pdb/explore.do?structureId=1BTA
22. https://files.rcsb.org/download/1BTA.pdb
23. https://python.sdv.u-paris.fr/data-files/temperatures.dat

À l'aide du module *NumPy*, on souhaite déterminer quel est le jour de la semaine le plus chaud. Pour cela nous vous proposons les étapes suivantes :

1. Récupérez le nom des jours de la semaine depuis le fichier et stockez-les dans une liste `days`.
2. Récupérez les valeurs de températures depuis le fichier et stockez-les dans un *array* 2D. La fonction `np.loadtxt()` [24] et son argument `usecols` vous seront utiles.
3. Parcourez chaque ligne de la matrice, calculez la température moyenne de chaque jour puis stockez-la dans une liste `mean_temps`.
4. À l'aide des deux listes `days` et `mean_temps`, déterminez et affichez le jour le plus chaud.

20.11.4 Calcul du centre de masse d'une membrane

L'image de gauche de la figure 20.2 montre le cliché d'une membrane de POPC (cyan) entourée d'eau (bleu) (coordonnées trouvées ici [25]). Les atomes de phosphore des groupes phosphates sont représentés en boule de van der Waals brune. Dans cet exercice, on cherche à calculer le centre de masse de la membrane, ainsi que le centre de masse (COM) de chaque monocouche de phosphores. Ces COM sont représentés sous forme de croix dans le graphique de droite de la figure 20.2.

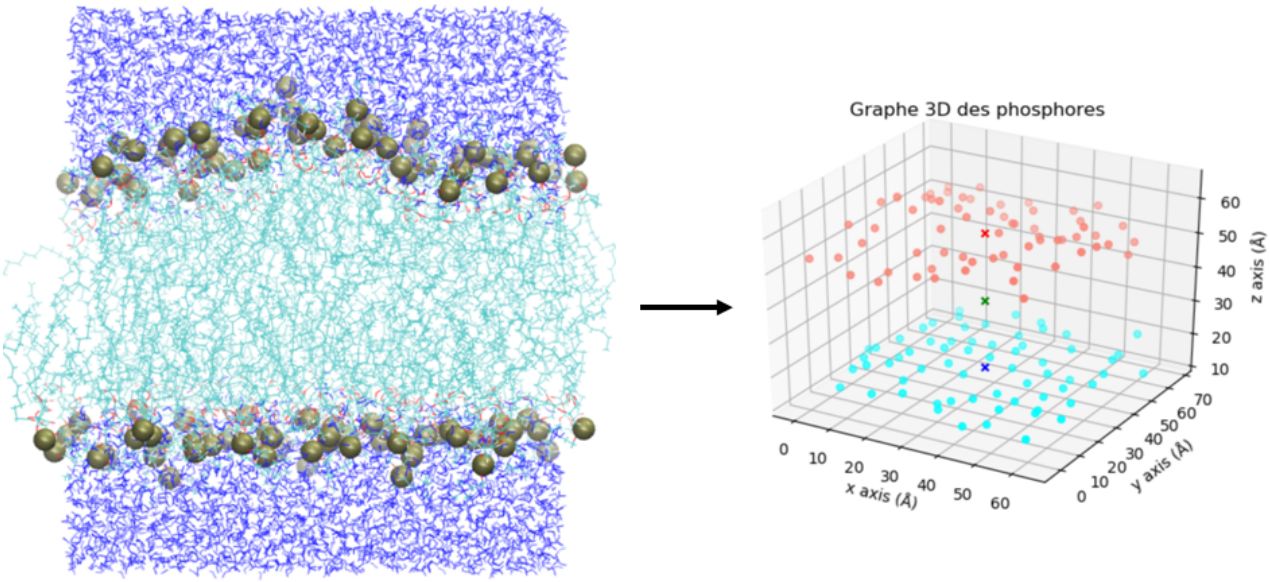


FIGURE 20.2 – Cliché d'une membrane de POPC.

Les coordonnées cartésiennes (x, y, z) de chaque atome de phosphore (en Å) sont stockées dans le fichier coors_P.dat [26], à raison d'un atome par ligne.

Nous vous proposons les étapes suivantes pour résoudre cet exercice à l'aide du module *NumPy* :

1. Récupérez les coordonnées des atomes de phosphore depuis le fichier `coors_P.dat` et stockez-les dans un *array* 2D (matrice) `coors_P`. La dimensionnalité de cette matrice est $n \times 3$, avec n le nombre de phosphores.
2. Calculez le z moyen de tous les phosphores (nombre réel) et stockez-le dans la variable `mean_z`. La méthode `.mean()` vous sera utile.
3. Avec des masques de booléens, récupérez les coordonnées des phosphores de la monocouche du haut dans un *array* 2D `upper`. Faites de même avec la monocouche du bas dans un *array* 2D `lower`.
4. Calculez le centre de masse `COM` de la membrane, ainsi que de la monocouche du haut `COM_upper` et du bas `COM_lower`. Pensez aux méthodes de calcul sur les *arrays* et l'argument `axis`.
5. Une fois tout cela effectué, créez un graphique 3D pour représenter les différents centres de masse. Utilisez la fonction `scatter()` du module *matplotlib* pour l'affichage en 3D [27]. Voici un squelette de programme pour vous

24. https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html
25. https://zenodo.org/record/153944
26. https://python.sdv.u-paris.fr/data-files/coors_P.dat
27. https://matplotlib.org/3.2.1/gallery/mplot3d/scatter3d.html

aider :

```
# Initialisation du graphique.
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
[...]
# X, Y et Z sont des arrays 1D de n éléments.
# Par exemple X représente tous les x des P de la monocouche upper.
[...]
# Affichage de la couche upper.
ax.scatter(X, Y, Z, c="salmon", marker="o")
# Affichage du COM de la couche upper.
ax.scatter(x, y, z, c="red", marker="x")
[...]
# Affichage des étiquettes des axes et du titre.
ax.set_xlabel("x (Å)")
ax.set_ylabel("y (Å)")
ax.set_zlabel("z (Å)")
ax.set_title("Graphe 3D des phosphores")
plt.show()
```

CHAPITRE 21

Module Matplotlib

Le module *matplotlib*[1] permet de générer des graphiques depuis Python. Il est l'outil complémentaire des modules *NumPy*, *scipy* ou *pandas* (que l'on verra juste après) lorsqu'on veut faire de l'analyse de données.

21.1 Installation et convention

Le module *matplotlib* n'est pas fourni avec la distribution Python de base. Avec la distribution Miniconda que nous vous conseillons d'utiliser (consultez pour cela la documentation en ligne[2]), vous pouvez l'installer avec la commande :

```
$ conda install -c conda-forge matplotlib
```

Dans ce chapitre, nous vous montrerons quelques exemples d'utilisation du module *matplotlib* pour vous convaincre de sa pertinence. Ces exemples seront exécutés dans un *notebook* Jupyter.

```
1  Les cellules de code apparaitront de cette manière
2  dans un notebook Jupyter, avec des numéros de lignes à gauche.
```

21.2 Chargement du module

On importe le module *matplotlib* avec la commande :

```
1  import matplotlib.pyplot as plt
```

Remarque

On n'importe pas le module *matplotlib* directement, mais plutôt son sous-module `pyplot`. Par convention, et pour l'utiliser plus rapidement, ce sous-module prendre l'alias `plt`.

21.3 Représentation en nuage de points

Dans cet exemple, nous considérons l'évolution de la concentration d'un produit dans le sang (exprimé en mg/L) en fonction du temps (exprimé en heures). Cet exemple est purement fictif.

1. https://matplotlib.org/
2. https://python.sdv.u-paris.fr/livre-dunod

Voici les valeurs mesurées :

Temps (h)	Concentration (mg/L)
1	3.5
2	5.8
3	9.1
4	11.8
6	17.5
7	21.3
9	26.8

Nous allons maintenant représenter l'évolution de la concentration en fonction du temps :

```
import matplotlib.pyplot as plt

temps = [1, 2, 3, 4, 6, 7, 9]
concentration = [5.5, 7.2, 11.8, 13.6, 19.1, 21.7, 29.4]

fig, ax = plt.subplots()
ax.scatter(temps, concentration, marker="o", color="blue")
ax.set_xlabel("Temps (h)")
ax.set_ylabel("Concentration (mg/L)")
ax.set_title("Concentration de produit en fonction du temps")
plt.show()
```

Dans un *notebook* Jupyter, vous devriez obtenir un graphique ressemblant à celui de la figure 21.1.

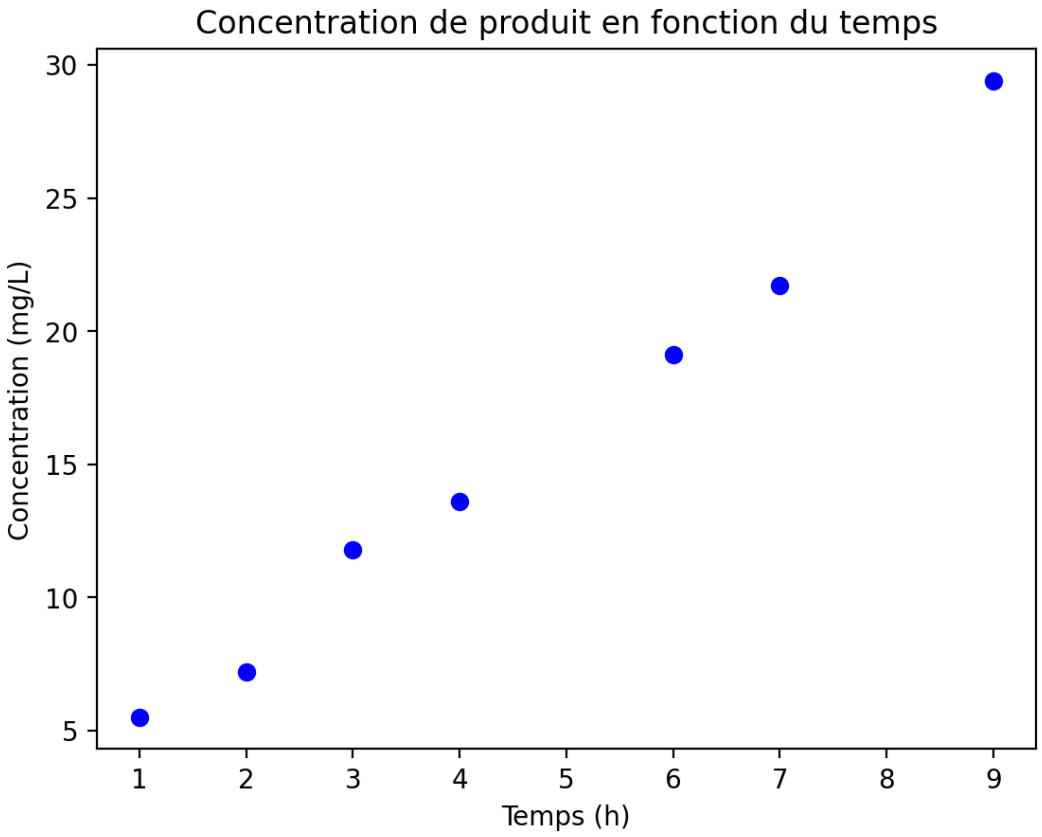


FIGURE 21.1 – Graphique produit par *matplotlib*.

Revenons maintenant sur le code :

- **Ligne 1.** Tout d'abord, on importe le sous-module `pyplot` du module *matplotlib* et on lui donne l'alias `plt` pour l'utiliser plus rapidement ensuite. Cet alias est standard, utilisez-le systématiquement.
- **Lignes 3 et 4.** On définit les variables `temps` et `concentration` comme des listes. Les deux listes doivent avoir la même longueur (sept éléments dans le cas présent).
- **Ligne 6.** On crée une figure avec la fonction `subplots()` qui renvoie deux objets : une figure (`fig`) et un axe (`ax`). L'axe est l'objet qui contient le graphique à proprement dit. On peut avoir plusieurs axes dans une même figure.
- **Ligne 7.** La méthode `.scatter()` permet de représenter des points sous forme de nuage de points. Les deux premiers arguments correspondent aux valeurs en abscisse et en ordonnée des points, fournis sous forme de listes. Des arguments facultatifs sont ensuite précisés comme le symbole (`marker`) et la couleur (`color`).
- **Lignes 8 et 9.** Les méthodes `.set_xlabel()` et `.set_ylabel()` donnent une légende aux axes des abscisses et des ordonnées.
- **Ligne 10.** La méthode `.set_title()` définit le titre du graphique.
- **Ligne 11.** L'instruction `plt.show()` affiche le graphique. Elle n'est pas nécessaire dans un *notebook* Jupyter, car le graphique est affiché automatiquement, mais elle est indispensable dans un script Python.

21.4 Représentation sous forme de courbe

On sait par ailleurs que l'évolution de la concentration du produit en fonction du temps peut-être modélisée par la fonction $f(x) = 2 + 3 \times x$.

Remarque

Le modèle présenté ici est purement fictif. Vous découvrirez dans le chapitre 22 *Module Pandas* comment réaliser une régression linéaire pour modéliser des données expérimentales.

Représentons ce modèle avec les points expérimentaux et sauvegardons le graphique obtenu sous forme d'une image :

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  temps = [1, 2, 3, 4, 6, 7, 9]
5  concentration = [5.5, 7.2, 11.8, 13.6, 19.1, 21.7, 29.4]
6
7  fig, ax = plt.subplots()
8  ax.scatter(temps, concentration, marker="o", color = "blue")
9  ax.set_xlabel("Temps (h)")
10 ax.set_ylabel("Concentration (mg/L)")
11 ax.set_title("Concentration de produit en fonction du temps")
12
13 x = np.linspace(min(temps), max(temps), 50)
14 y = 2 + 3 * x
15
16 ax.plot(x, y, color="green", ls="--")
17 ax.grid()
18 fig.savefig("concentration_vs_temps_1.png", bbox_inches="tight", dpi=200)
```

Le résultat est représenté sur la figure 21.2.

Les étapes supplémentaires par rapport au graphique précédent (figure 21.1) sont :

- **Ligne 1.** On charge le module *numpy* sous le nom `np`.
- **Ligne 13.** On crée la variable `x` avec la fonction `linspace()` du module *NumPy*, qui renvoie une liste de valeurs régulièrement espacées entre deux bornes, ici entre le minimum (`min(temps)`) et le maximum (`max(temps)`) de la variable `temps`. Dans notre exemple, nous générons une liste de 50 valeurs. La variable `x` ainsi créée est du type *array*.
- **Ligne 14.** On construit ensuite la variable `y` à partir de la formule modélisant l'évolution de la concentration du produit en fonction du temps. Cette manipulation n'est possible que parce que `x` est du type *array*. Cela ne fonctionnerait pas avec une liste classique.
- **Ligne 16.** La méthode `.plot()` construit une courbe à partir des coordonnées en abscisse et en ordonnées des

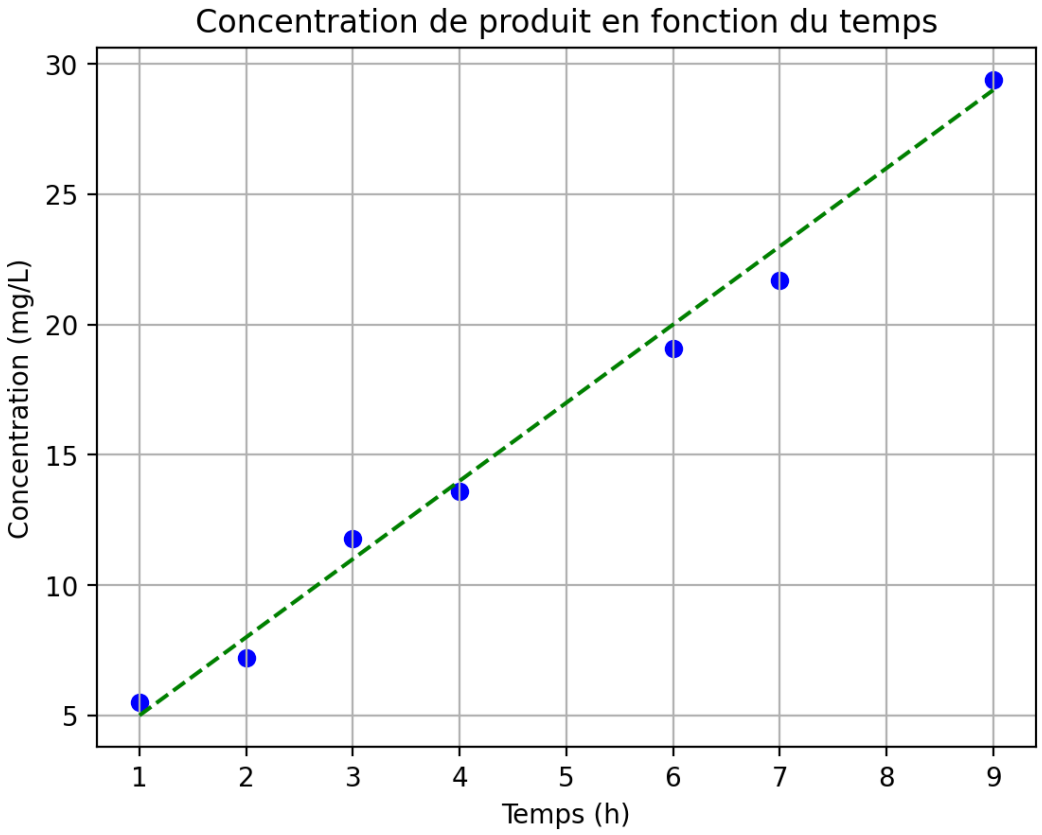


FIGURE 21.2 – Concentration du produit en fonction du temps.

points à représenter. On indique ensuite des arguments facultatifs comme le style de la ligne (`ls` pour *line style*) et sa couleur (`color`).

- **Ligne 17.** La méthode `.grid()` affiche une grille.
- **Ligne 18.** Enfin, l'instruction `fig.savefig()` enregistre le graphique produit sous la forme d'une image au format png. Des arguments par mot-clé définissent la manière de générer les marges autour du graphique (`bbox_inches`) et la résolution de l'image (`dpi`).

Pour terminer, on peut améliorer un peu plus le graphique en ajoutant une légende et en modifiant l'étendue des axes des abscisses et des ordonnées :

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  temps = [1, 2, 3, 4, 6, 7, 9]
5  concentration = [5.5, 7.2, 11.8, 13.6, 19.1, 21.7, 29.4]
6
7  fig, ax = plt.subplots()
8  ax.scatter(temps, concentration, marker="o", color="blue", label="mesures")
9  ax.set_xlabel("Temps (h)")
10 ax.set_ylabel("Concentration (mg/L)")
11 ax.set_title("Concentration de produit en fonction du temps")
12
13 x = np.linspace(min(temps), max(temps), 50)
14 y = 2 + 3 * x
15
16 ax.plot(x, y, color="green", ls="--", label="modèle")
17
18 ax.grid()
19 ax.set_xlim(0, 10)
20 ax.set_ylim(0, 35)
21
22 ax.legend(loc="upper left")
23 fig.savefig("concentration_vs_temps_2.png", bbox_inches="tight", dpi=200)
```

On obtient alors le graphique représenté dans la figure 21.3.

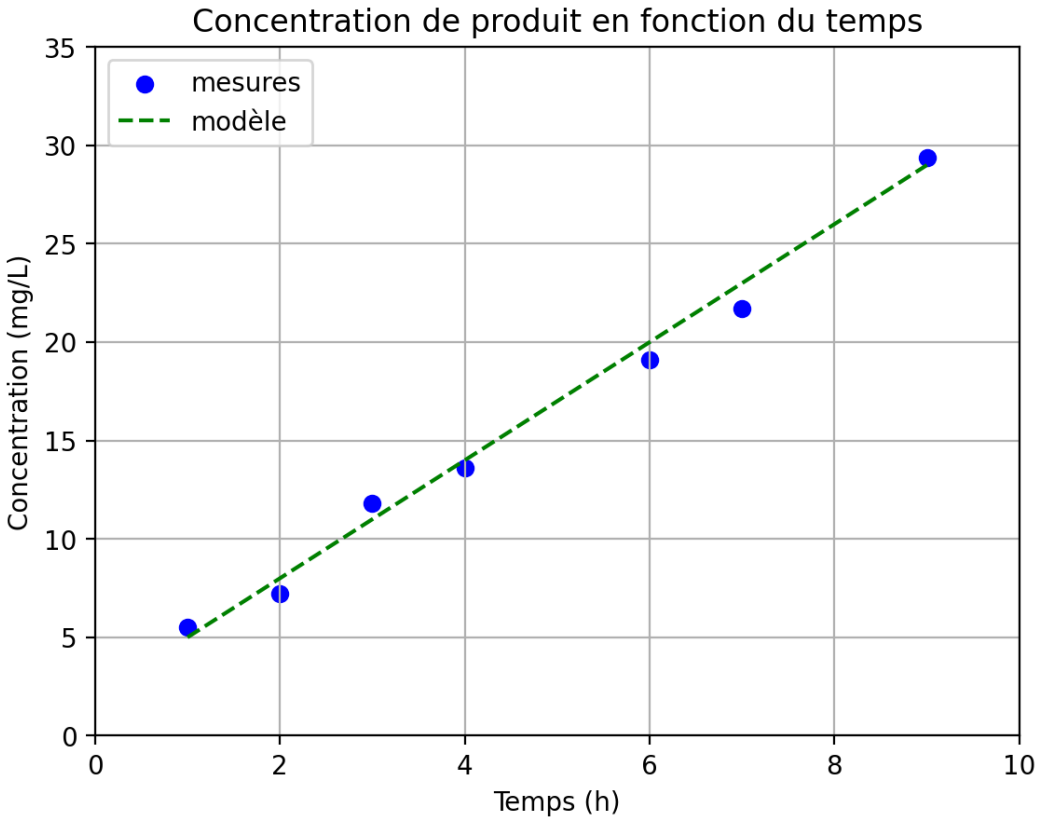


FIGURE 21.3 – Concentration du produit en fonction du temps, version améliorée.

Les différences notables par rapport au code précédent sont :

- **Lignes 8 et 16.** On ajoute le paramètre `label` pour donner un nom au nuage de points (`.scatter()`) ou à la courbe (`.plot()`).
- **Lignes 19 et 20.** On définit l'étendue de l'axe des abscisses avec la méthode `.set_xlim()` et de l'axe des

ordonnées avec la méthode `.set_ylim()`.

- **Lignes 22.** On affiche la légende avec la méthode `.legend()`. L'argument `loc` permet de préciser la position de la légende dans le graphique. Dans notre exemple, la légende est placée en haut à gauche (`"upper left"`).

21.5 Représentation en diagramme en bâtons

On souhaite maintenant représenter graphiquement la distribution des différentes bases dans une séquence d'ADN.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  sequence = "ACGATCATAGCGAGCTACGTAGAA"
5  bases = ["A", "C", "G", "T"]
6  distribution = []
7  for base in bases:
8      distribution.append(sequence.count(base))
9
10 x = np.arange(len(bases))
11
12 fig, ax = plt.subplots()
13 ax.bar(x, distribution)
14 ax.set_xticks(x, bases)
15 ax.set_xlabel("Bases")
16 ax.set_ylabel("Nombre")
17 ax.set_title(f"Distribution des bases\n dans la séquence {sequence}")
18 fig.savefig("distribution_bases.png", bbox_inches="tight", dpi=200)
```

On obtient alors le graphique de la figure 21.4.

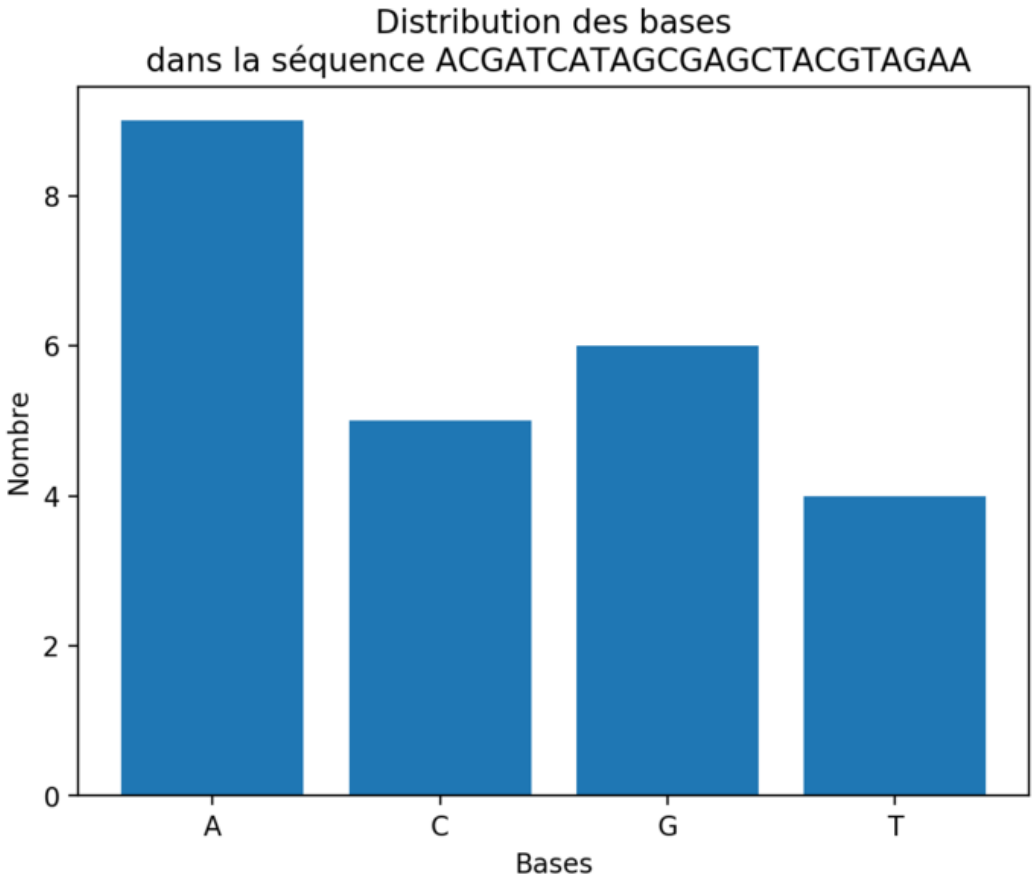


FIGURE 21.4 – Distribution des bases.

Prenons le temps d'examiner les différentes étapes du script précédent :

- **Lignes 4 à 6.** On définit les variables `sequence`, `bases` et `distribution`.
- **Lignes 7 et 8.** On calcule la distribution des différentes bases dans la séquence. On utilise pour cela la méthode `count()`, qui renvoie le nombre de fois qu'une chaîne de caractères (les différentes bases) se trouve dans une autre (la séquence).
- **Ligne 10.** On définit la position en abscisse des barres. Dans cet exemple, la variable `x` vaut `array([0, 1, 2, 3])`.
- **Ligne 12.** On crée le graphique.
- **Ligne 13.** La méthode `.bar()` construit le diagramme en bâtons. Elle prend en argument la position des barres (`x`) et leurs hauteurs (`distribution`).
- **Ligne 14.** La méthode `.set_xtics()` redéfinit les étiquettes (c'est-à-dire le nom des bases) sur l'axe des abscisses.
- **Lignes 15 à 17.** On définit les légendes des axes et le titre du graphique. On insère un retour à la ligne `\n` dans le titre pour qu'il soit réparti sur deux lignes.
- **Ligne 18.** Enfin, on enregistre le graphique généré au format png.

On espère que ces courts exemples vous auront convaincu de l'utilité du module *matplotlib*. Sachez qu'il peut faire bien plus, par exemple générer des histogrammes ou toutes sortes de graphiques utiles en analyse de données. Il est existe par ailleurs d'autres bibliothèques pour produire des graphiques avec Python, comme *Seaborn*[3], *Bokeh*[4] ou *Plotly*[5]. Ces deux dernières permettent de générer des graphiques interactifs, c'est-à-dire des graphiques dans lesquels on peut zoomer, se déplacer, etc. Nous vous invitons à les découvrir par vous-même.

Pour aller plus loin

- Le site de *matplotlib* fournit de nombreux exemples détaillés[6], n'hésitez pas à le consulter.
- Le site Python Graph Gallery[7] propose aussi des exemples de code pour différents types de graphiques, réalisés avec *matplotlib* ou d'autres bibliothèques.
- Enfin, des *cheat sheets*[8] de *matplotlib* sont extrêmement utiles et très bien faites.

3. https://seaborn.pydata.org/
4. http://bokeh.org/
5. https://plotly.com/
6. https://matplotlib.org/gallery/index.html
7. https://www.python-graph-gallery.com/matplotlib/
8. https://matplotlib.org/cheatsheets/

CHAPITRE 22

Module Pandas

Le module *pandas*[1] a été conçu pour l'analyse de données. Il est particulièrement puissant pour manipuler des données structurées sous forme de tableau.

22.1 Installation et convention

Le module *pandas* n'est pas fourni avec la distribution Python de base. Avec la distribution Miniconda que nous vous conseillons d'utiliser (consultez pour cela la documentation en ligne[2]), vous pouvez rapidement l'installer avec la commande :

```
$ conda install -c conda-forge pandas
```

Vous aurez également besoin des modules *matplotlib* pour créer des graphiques et *scipy* pour réaliser une régression linaire, que vous pouvez installer ainsi :

```
$ conda install -c conda-forge matplotlib scipy
```

Dans ce chapitre, nous vous montrerons quelques exemples d'utilisation du module *pandas* pour vous convaincre de sa pertinence. Ces exemples seront exécutés dans un *notebook* Jupyter.

```
1  Les cellules de code apparaitront de cette manière
2  dans un notebook Jupyter, avec des numéros de lignes à gauche.
```

```
Les résultats seront affichés de cette manière,
éventuellement sur plusieurs lignes.
```

22.2 Chargement du module

Pour charger *pandas* dans la mémoire de Python, on utilise la commande `import` habituelle :

```
1  import pandas
```

Par convention, on utilise `pd` comme nom raccourci pour *pandas* :

```
1  import pandas as pd
```

1. https://pandas.pydata.org/
2. https://python.sdv.u-paris.fr/livre-dunod

22.3 *Series*

Le premier type de données apporté par *pandas* est la *Series*, qui correspond à un vecteur à une dimension.

```
1  s = pd.Series([10, 20, 30, 40], index = ['a', 'b', 'c', 'd'])
2  s
```

```
a    10
b    20
c    30
d    40
dtype: int64
```

22.3.1 Sélections par étiquette ou indice

Avec *pandas*, chaque élément de la série de données possède une étiquette qui permet d'appeler les éléments qui la composent. Ainsi, pour appeler le premier élément de la série, on peut utiliser son étiquette (ici, `"a"`) :

```
1  s["a"]
```

```
10
```

Pour accéder au premier élément par son indice (ici 0), comme on le ferait avec une liste, on utilise la méthode `.iloc` :

```
1  s.iloc[0]
```

```
10
```

Bien sûr, on peut extraire plusieurs éléments, par leurs indices ou leurs étiquettes :

```
1  s[["b", "d"]]
```

```
b    20
d    40
dtype: int64
```

et

```
1  s.iloc[[1, 3]]
```

```
b    20
d    40
dtype: int64
```

22.3.2 Modifications de *Series*

Les étiquettes permettent de modifier et d'ajouter des éléments :

```
1  s["c"] = 300
2  s["z"] = 50
3  s
```

```
a     10
b     20
c    300
d     40
z     50
dtype: int64
```

22.3.3 Filtres

Enfin, on peut filtrer une partie de la *Series* :

```
s[s>30]
```

```
c    300
d     40
z     50
dtype: int64
```

Remarque

Cette écriture rappelle celle des masques booléens dans le chapitre 20 *Module NumPy*.

Enfin, on peut aussi combiner plusieurs critères de sélection avec les opérateurs logiques `&` (pour **ET**) et `|` (pour **OU**) :

```
s[(s>20) & (s<100)]
```

```
d    40
z    50
dtype: int64
```

```
s[(s<15) | (s>150)]
```

```
a     10
c    300
dtype: int64
```

22.4 *Dataframes*

Un autre type d'objet particulièrement intéressant introduit par *pandas* sont les *Dataframes*. Ceux-ci correspondent à des tableaux à deux dimensions avec des étiquettes pour nommer les lignes et les colonnes.

Remarque

Si vous êtes familier avec le langage de programmation et d'analyse statistique R, les *Dataframes* de *pandas* se rapprochent de ceux trouvés dans R.

22.4.1 Création

Voici comment créer un *Dataframe* avec *pandas* à partir de données fournies comme liste de lignes :

```
import numpy as np
df = pd.DataFrame(columns=["a", "b", "c", "d"],
                  index=["chat", "singe", "souris"],
                  data=[np.arange(10, 14),
                        np.arange(20, 24),
                        np.arange(30, 34)])
df
```

```
         a   b   c   d
chat    10  11  12  13
singe   20  21  22  23
souris  30  31  32  33
```

Voici quelques commentaires sur le code précédent :

- **Ligne 1.** On charge le module *NumPy* utilisé ensuite.
- **Ligne 2.** Le *Dataframe* est créé avec la fonction `DataFrame()` à laquelle on fournit plusieurs arguments. L'argument `columns` indique le nom des colonnes, sous forme d'une liste.
- **Ligne 3.** L'argument `index` définit le nom des lignes, sous forme de liste également.
- **Lignes 4 à 6.** L'argument `data` fournit le contenu du *Dataframe*, sous la forme d'une liste de valeurs correspondantes à des lignes. Ainsi, `np.arange(10, 14)` qui est équivalent à `[10, 11, 12, 13]` correspond à la première ligne du *Dataframe*.

Le même *Dataframe* peut aussi être créé à partir des valeurs fournies en colonnes sous la forme d'un dictionnaire :

```
1  data = {"a": np.arange(10, 40, 10),
2          "b": np.arange(11, 40, 10),
3          "c": np.arange(12, 40, 10),
4          "d": np.arange(13, 40, 10)}
5  df = pd.DataFrame(data)
6  df.index = ["chat", "singe", "souris"]
7  df
```

```
        a   b   c   d
chat    10  11  12  13
singe   20  21  22  23
souris  30  31  32  33
```

- **Lignes 1 à 4.** Le dictionnaire `data` contient les données en colonnes. La clé associée à chaque colonne est le nom de la colonne.
- **Ligne 5.** Le *dataframe* est créé avec la fonction `pd.DataFrame()` à laquelle on passe `data` en argument.
- **Ligne 6.** On peut définir les étiquettes des lignes de n'importe quel *dataframe* avec l'attribut `df.index`.

22.4.2 Quelques propriétés

Les dimensions d'un *dataframe* sont données par l'attribut `.shape` :

```
1  df.shape
```

```
(3, 4)
```

Ici, le *dataframe* `df` possède trois lignes et quatre colonnes.

L'attribut `.columns` renvoie le nom des colonnes et permet aussi de renommer les colonnes d'un *dataframe* :

```
1  df.columns
```

```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

```
1  df.columns = ["Paris", "Lyon", "Nantes", "Pau"]
2  df
```

```
        Paris  Lyon  Nantes  Pau
chat       10    11      12   13
singe      20    21      22   23
souris     30    31      32   33
```

La méthode `.head(n)` renvoie les *n* premières lignes du *Dataframe* (par défaut, *n* vaut 5) :

```
1  df.head(2)
```

```
       Paris  Lyon  Nantes  Pau
chat      10    11      12   13
singe     20    21      22   23
```

Remarque

Les *Dataframes* utilisés ici comme exemples sont volontairement petits. Si vous êtes confrontés à des *Dataframes* de grande taille, ceux-ci seront affichés partiellement dans un *notebook* Jupyter. Des ascenseurs en bas et à droite du *Dataframe* permettront de naviguer dans les données.

22.4.3 Sélections

Les mécanismes de sélection fournis avec *pandas* sont très puissants. En voici un rapide aperçu :

22.4.3.1 Sélection de colonnes

On peut sélectionner une colonne par son étiquette :

```
1  df["Lyon"]
```

```
chat      11
singe     21
souris    31
```

La notation `df["Lyon"]` sélectionne une colonne et renvoie un objet *Series* :

```
1  type(df["Lyon"])
```

```
pandas.core.series.Series
```

Attention

On trouve parfois l'écriture `df.Lyon` pour sélectionner une colonne. C'est une très mauvaise pratique, car cette écriture peut être confondue avec un attribut de l'objet `df` (par exemple `.shape`). Par ailleurs, elle ne fonctionne pas pour des noms de colonnes qui contiennent des espaces ou des caractères spéciaux (ce qui n'est pas non plus une bonne pratique).

Nous vous conseillons de toujours utiliser la notation **`df["nom_de_colonne"]`**.

Pour sélectionner plusieurs colonnes, il faut fournir une liste de noms de colonnes :

```
1  df[["Lyon", "Pau"]]
```

```
        Lyon  Pau
chat      11   13
singe     21   23
souris    31   33
```

On obtient cette fois un *Dataframe* avec les colonnes sélectionnées :

```
1  type(df[["Lyon", "Pau"]])
```

```
pandas.core.frame.DataFrame
```

Remarque

La sélection de plusieurs colonnes nécessite une liste entre les crochets, par exemple `df[["Lyon", "Pau"]]`. Si on utilise un tuple du type `df[("Lyon", "Pau")]`, Python renvoie une erreur `KeyError: ('Lyon', 'Pau')`.

22.4.3.2 Sélection de lignes

Pour sélectionner une ligne, il faut utiliser l'instruction `.loc` et l'étiquette de la ligne :

```
df.loc["singe"]
```

```
Paris     20
Lyon      21
Nantes    22
Pau       23
Name: singe, dtype: int64
```

Ici aussi, on peut sélectionner plusieurs lignes :

```
df.loc[["singe", "chat"]]
```

```
       Paris  Lyon  Nantes  Pau
singe     20    21      22   23
chat      10    11      12   13
```

Enfin, on peut aussi sélectionner des lignes avec l'instruction `.iloc` et l'indice de la ligne (la première ligne ayant l'indice 0) :

```
df.iloc[1]
```

```
Paris     20
Lyon      21
Nantes    22
Pau       23
Name: singe, dtype: int64
```

```
df.iloc[[1, 0]]
```

```
       Paris  Lyon  Nantes  Pau
singe     20    21      22   23
chat      10    11      12   13
```

On peut également utiliser les tranches (comme pour les listes) :

```
df.iloc[0:2]
```

```
       Paris  Lyon  Nantes  Pau
chat      10    11      12   13
singe     20    21      22   23
```

22.4.3.3 Sélection sur les lignes et les colonnes

On peut bien sûr combiner les deux types de sélection (en ligne et en colonne) :

```
df.loc["souris", "Pau"]
```

```
33
```

```
df.loc[["singe", "souris"], ["Nantes", "Lyon"]]
```

```
        Nantes  Lyon
singe       22    21
souris      32    31
```

Notez qu'à partir du moment où on souhaite effectuer une sélection sur des lignes, il faut utiliser `.loc` (ou `.iloc` si on utilise les indices).

22.4.3.4 Sélection par condition

Remémorons-nous d'abord le contenu du *dataframe* `df` :

```
df
```

```
       Paris  Lyon  Nantes  Pau
chat      10    11      12   13
singe     20    21      22   23
souris    30    31      32   33
```

Sélectionnons maintenant toutes les lignes pour lesquelles les effectifs à Pau sont supérieurs à 15 :

```
df[ df["Pau"]>15 ]
```

```
       Paris  Lyon  Nantes  Pau
singe     20    21      22   23
souris    30    31      32   33
```

De cette sélection, on ne souhaite garder que les valeurs pour Lyon :

```
df[ df["Pau"]>15 ]["Lyon"]
```

```
singe     21
souris    31
Name: Lyon, dtype: int64
```

On peut aussi combiner plusieurs conditions avec `&` pour l'opérateur **et** :

```
df[ (df["Pau"]>15) & (df["Lyon"]>25) ]
```

```
       Paris  Lyon  Nantes  Pau
souris    30    31      32   33
```

et `|` pour l'opérateur **ou** :

```
df[ (df["Pau"]>15) | (df["Lyon"]>25) ]
```

```
       Paris  Lyon  Nantes  Pau
singe     20    21      22   23
souris    30    31      32   33
```

22.4.4 Combinaison de *dataframes*

En biologie, on a souvent besoin de combiner deux tableaux à partir d'une colonne commune. Par exemple, si on considère les deux *dataframes* suivants :

```
data1 = {"Lyon": [10, 23, 17], "Paris": [3, 15, 20]}
df1 = pd.DataFrame.from_dict(data1)
df1.index = ["chat", "singe", "souris"]
df1
```

```
       Lyon  Paris
chat     10      3
singe    23     15
souris   17     20
```

et

```
data2 = {"Nantes": [3, 9, 14], "Strasbourg": [5, 10, 8]}
df2 = pd.DataFrame.from_dict(data2)
df2.index = ["chat", "souris", "lapin"]
df2
```

```
        Nantes  Strasbourg
chat         3           5
souris       9          10
lapin       14           8
```

On souhaite combiner ces deux *dataframes*, c'est-à-dire connaître pour les quatre villes (Lyon, Paris, Nantes et Strasbourg) le nombre d'animaux. On remarque d'ores et déjà qu'il y a des singes à Lyon et Paris, mais pas de lapin et qu'il y a des lapins à Nantes et Strasbourg, mais pas de singe. Nous allons voir comment gérer cette situation.

Pandas propose pour cela la fonction `concat()` [3], qui prend comme argument une liste de *dataframes* :

```
pd.concat([df1, df2])
```

```
        Lyon  Nantes  Paris  Strasbourg
chat    10.0     NaN    3.0         NaN
singe   23.0     NaN   15.0         NaN
souris  17.0     NaN   20.0         NaN
chat     NaN     3.0    NaN         5.0
souris   NaN     9.0    NaN        10.0
lapin    NaN    14.0    NaN         8.0
```

Ici, `NaN` indique des valeurs manquantes, cela signifie littéralement *Not a Number*. Mais le résultat obtenu n'est pas celui que nous attendions, puisque les lignes de deux *dataframes* ont été recopiées.

L'argument supplémentaire `axis=1` produit le résultat attendu :

```
pd.concat([df1, df2], axis=1)
```

```
        Lyon  Paris  Nantes  Strasbourg
chat    10.0    3.0     3.0         5.0
lapin    NaN    NaN    14.0         8.0
singe   23.0   15.0     NaN         NaN
souris  17.0   20.0     9.0        10.0
```

Par défaut, *pandas* va conserver le plus de lignes possible. Si on ne souhaite conserver que les lignes communes aux deux *dataframes*, il faut ajouter l'argument `join="inner"` :

```
pd.concat([df1, df2], axis=1, join="inner")
```

```
        Lyon  Paris  Nantes  Strasbourg
chat      10      3       3           5
souris    17     20       9          10
```

Un autre comportement par défaut de `concat()` est que cette fonction va combiner les *dataframes* en se basant sur leurs index. Il est néanmoins possible de préciser, pour chaque *dataframe*, le nom de la colonne qui sera utilisée comme référence avec l'argument `join_axes`.

22.4.5 Opérations vectorielles

Pour cette rubrique, créons un *Dataframe* composé de nombres aléatoires compris entre 100 et 200, répartis en trois colonnes (`a`, `b` et `c`) et 1 000 lignes :

3. https://pandas.pydata.org/pandas-docs/stable/merging.html

```
import numpy as np
import pandas as pd

nb_rows = 1000
df = pd.DataFrame(
    {
        "a": np.random.randint(100, 200, nb_rows),
        "b": np.random.randint(100, 200, nb_rows),
        "c": np.random.randint(100, 200, nb_rows),
    }
)
```

Vérifions que ce *Dataframe* a bien les propriétés attendues :

```
df.shape
```

```
(1000, 3)
```

```
df.head()
```

```
     a    b    c
0  105  156  122
1  116  135  138
2  125  190  113
3  196  175  179
4  129  184  153
```

On souhaite maintenant créer une nouvelle colonne (`d`) qui sera le résultat de la multiplication des colonnes `a` et `b`, à laquelle on ajoute ensuite la colonne `c`.

Une première manière de faire est de procéder ligne par ligne. La méthode `.iterrows()` permet de parcourir les lignes d'un *Dataframe* et renvoie un tuple contenant l'indice de la ligne (sous la forme d'un entier) et la ligne elle-même (sous la forme d'une *Series*) :

```
for idx, row in df.iterrows():
    df.at[idx, "d"] = (row["a"] * row["b"]) + row["c"]
```

Ici, l'instruction `.at` ajoute une cellule à la ligne d'indice `idx` et de colonne `d`. Cette instruction est plus efficace que `.loc` pour ajouter une cellule à un *Dataframe*.

L'approche précédente produit le résultat attendu, mais elle n'est pas optimale, car très lente. Pour évaluer le temps moyen pour réaliser ces opérations, on utilise la commande magique `%%timeit` abordée dans le chapitre 18 *Jupyter et ses notebooks* :

```
%%timeit
for idx, row in df.iterrows():
    df.at[idx, "d"] = (row["a"] * row["b"]) + row["c"]
```

qui renvoie :

```
52.4 ms ± 3.6 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Cette cellule de code s'exécute en moyenne en 52,4 ms.

Une autre approche, plus efficace, consiste à réaliser les opérations directement sur les colonnes (et non plus ligne par ligne) :

```
%%timeit
df["d"] = (df["a"] * df["b"]) + df["c"]
```

qui renvoie :

```
250 µs ± 36.1 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Ici, la cellule de code s'exécute en moyenne en 250 µs, soit environ 200 fois $(52400/250)$ plus rapidement qu'avec

`.iterrows()`. Tout comme avec les *arrays* du chapitre 20 *Numpy*, les opérations vectorielles avec les *Dataframes* sont rapides et efficaces. Privilégiez toujours ce type d'approche avec les *arrays* de *NumPy* ou les *Series* et *Dataframes* de *pandas*.

Remarque

Dans l'exemple précédent, l'utilisation de la commande magique `%%timeit` calcule le temps d'exécution moyen d'une cellule. Python détermine automatiquement le nombre d'itérations à réaliser pour que le calcul se fasse dans un temps raisonnable. Ainsi, pour la méthode `.iterrows()`, le calcul est réalisé 10 fois sur sept répétitions alors que pour les opérations vectorielles, le calcul est effectué 1000 fois sur sept répétitions.

22.5 Un exemple plus concret avec les kinases

Pour illustrer les possibilités de *pandas*, voici un exemple plus concret sur un jeu de données de kinases[4]. Les kinases sont des protéines responsables de la phosphorylation d'autres protéines.

Le fichier `kinases.csv` que vous pouvez télécharger en ligne[5] contient des informations tirées de la base de données de séquences UniProt pour quelques kinases.

Si vous n'êtes pas familier avec le format de fichier `.csv`, nous vous conseillons de consulter l'annexe A *Quelques formats de données en biologie*.

Remarque

Avant de nous lancer dans l'analyse de ce fichier, nous vous proposons cette petite devinette :

Qu'est-ce qu'une protéine dans une piscine ?

La réponse sera donnée à la fin de ce chapitre.

22.5.1 Prise de contact avec le jeu de données

Une fonctionnalité très intéressante de *pandas* est d'ouvrir très facilement un fichier au format `.csv` :

```
1  df = pd.read_csv("kinases.csv")
```

Le contenu est chargé sous la forme d'un *Dataframe* dans la variable `df`.

Le fichier contient 1 442 lignes de données plus une ligne d'en-tête. Cette dernière est automatiquement utilisée par *pandas* pour nommer les différentes colonnes. Voici un aperçu des premières lignes :

```
1  df.head()
```

```
        Entry   Organism  Length Creation date    Mass  PDB
0  A0A0B4J2F2      Human     783    2018-06-20   84930  NaN
1      A4L9P5        Rat    1211    2007-07-24  130801  NaN
2  A0A1D6E0S8      Maize     856    2023-05-03   93153  NaN
3  A0A8I5ZNK2        Rat     528    2023-09-13   58360  NaN
4      A1Z7T0  Fruit fly    1190    2012-01-25  131791  NaN
```

Nous avons six colonnes de données :

- l'identifiant de la protéine (`Entry`) ;
- l'organisme d'où provient cette protéine (`Organism`) ;
- le nombre d'acides aminés qui constituent la protéine (`Length`) ;
- la date à laquelle cette protéine a été déréférencée dans *UniProt* (`Creation date`) ;
- la masse de la protéine (`Mass`), exprimée en Dalton ;
- les éventuelles structures 3D de la protéine (`PDB`).

4. https://fr.wikipedia.org/wiki/Kinase
5. https://python.sdv.u-paris.fr/data-files/kinases.csv

La colonne d'entiers tout à gauche est un index automatiquement créé par *pandas*.

Nous pouvons demander à *pandas* d'utiliser une colonne particulière comme index. On utilise pour cela le paramètre `index_col` de la fonction `read_csv()`. Ici, la colonne `Entry` s'y prête très bien, car cette colonne ne contient que des identifiants uniques :

```
df = pd.read_csv("kinases.csv", index_col="Entry")
df.head()
```

Entry	Organism	Length	Creation date	Mass	PDB
A0A0B4J2F2	Human	783	2018-06-20	84930	NaN
A4L9P5	Rat	1211	2007-07-24	130801	NaN
A0A1D6E0S8	Maize	856	2023-05-03	93153	NaN
A0A8I5ZNK2	Rat	528	2023-09-13	58360	NaN
A1Z7T0	Fruit fly	1190	2012-01-25	131791	NaN

Remarque

La fonction `.read_csv()` permet également d'ouvrir un fichier au format TSV (voir l'annexe A *Quelques formats de données en biologie*). Il faut pour cela préciser que le séparateur des colonnes de données est une tabulation (`\t`), avec l'argument `sep="\t"`.

Avant d'analyser un jeu de données, il est intéressant de l'explorer un peu. Par exemple, connaître ses dimensions :

```
df.shape
```

```
(1442, 5)
```

Notre jeu de données contient donc 1 442 lignes et 5 colonnes. En effet, la colonne `Entry` est maintenant utilisée comme index et n'est donc plus prise en compte.

Il est aussi intéressant de savoir de quel type de données est constituée chaque colonne :

```
df.dtypes
```

```
Organism         object
Length            int64
Creation date    object
Mass              int64
PDB              object
dtype: object
```

Les colonnes `Length` et `Mass` contiennent des valeurs numériques, en l'occurrence des entiers (`int64`). Le type `object` est un type par défaut.

La méthode `.info()` permet d'aller un peu plus loin dans l'exploration du jeu de données en combinant les informations produites par les propriétés `.shape` et `.dtypes` :

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1442 entries, A0A0B4J2F2 to Q5F361
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Organism       1442 non-null   object
 1   Length         1442 non-null   int64
 2   Creation date  1442 non-null   object
 3   Mass           1442 non-null   int64
 4   PDB            488 non-null    object
dtypes: int64(2), object(3)
memory usage: 67.6+ KB
```

Avec l'argument `memory_usage="deep"`, la méthode `.info()` permet de connaitre avec précision la quantité de mémoire vive occupée par le *Dataframe* :

```
df.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1442 entries, A0A0B4J2F2 to Q5F361
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Organism       1442 non-null   object
 1   Length         1442 non-null   int64
 2   Creation date  1442 non-null   object
 3   Mass           1442 non-null   int64
 4   PDB            488 non-null    object
dtypes: int64(2), object(3)
memory usage: 351.0 KB
```

Ici, le *Dataframe* occupe 351 kilo-octets (ko) en mémoire.

22.5.2 Recherche de valeurs manquantes

Il est aussi utile de savoir si des valeurs manquantes sont présentes dans le jeu de données. Ces valeurs manquantes correspondent à des champs pour lesquels aucune valeur n'ont été fournies. Elles sont souvent représentées par `NaN` (pour *Not a Number*).

La méthode `.isna()` renvoie un *Dataframe* de la même dimension que le *Dataframe* initial, mais avec des valeurs booléennes (`True` si la valeur est manquante (`NaN`) ou `False` sinon). En le combinant avec la méthode `.sum()`, on peut compter le nombre de valeurs manquantes pour chaque colonne :

```
df.isna().sum()
```

```
Organism           0
Length             0
Creation date      0
Mass               0
PDB              954
dtype: int64
```

Ici, la seule colonne qui contient des valeurs manquantes est la colonne `PDB`, qui contient 954 valeurs manquantes. Cela signifie que pour 954 protéines, aucune structure 3D n'est disponible. Nous reviendrons plus tard sur cette colonne `PDB`.

22.5.3 Conversion en date

Le type `object` correspond la plupart du temps à des chaînes de caractères. C'est tout à fait légitime pour la colonne `Organism`. Mais on sait par contre que la colonne `Creation date` est une date sous la forme *année-mois-jour*.

Si le format de date utilisé est homogène sur tout le jeu de données et non ambigu, on peut demander à *pandas* de considérer la colonne `Creation Date` comme une date. *pandas* détectera alors automatiquement le format de date utilisé :

```
df["Creation date"] = pd.to_datetime(df["Creation date"])
```

L'affichage des données n'est pas modifié :

```
df.head()
```

Entry	Organism	Length	Creation date	Mass	PDB
A0A0B4J2F2	Human	783	2018-06-20	84930	NaN
A4L9P5	Rat	1211	2007-07-24	130801	NaN
A0A1D6E0S8	Maize	856	2023-05-03	93153	NaN
A0A8I5ZNK2	Rat	528	2023-09-13	58360	NaN
A1Z7T0	Fruit fly	1190	2012-01-25	131791	NaN

Mais le type de données de la colonne `Creation date` est maintenant une date (`datetime64[ns]`) :

```
df.dtypes
```

```
Organism                object
Length                   int64
Creation date    datetime64[ns]
Mass                     int64
PDB                     object
dtype: object
```

22.5.4 Statistiques descriptives et table de comptage

Pour les colonnes qui contiennent des données numériques, on peut obtenir rapidement quelques statistiques descriptives avec la méthode `.describe()` :

```
df.describe()
```

```
            Length                  Creation date           Mass
count  1442.000000                           1442    1442.000000
mean    756.139390  2001-01-25 16:10:39.112344064   84710.753814
min      81.000000            1986-07-21 00:00:00    9405.000000
25%     476.250000            1996-10-01 00:00:00   54059.000000
50%     632.000000            2002-03-10 00:00:00   71613.000000
75%     949.250000            2005-11-22 00:00:00  105485.250000
max    2986.000000            2023-09-13 00:00:00  340261.000000
std     404.195273                            NaN   44764.273097
```

On apprend ainsi que la taille de la protéine (colonne `Length`) a une valeur moyenne de 756,14 acides aminés et que la plus petite protéine est composée de 81 acides aminés et la plus grande de 2 986. Pratique !

Des statistiques sont également proposées pour la colonne `Creation date`. La protéine la plus récente a ainsi été référencée le 13 septembre 2023.

La colonne `Organism` contient des chaînes de caractères, on peut rapidement déterminer le nombre de protéines pour chaque organisme :

```
df["Organism"].value_counts()
```

```
Organism
Human        489
Mouse        489
Rat          253
Fruit fly    103
Chicken       75
Rabbit        25
Maize          8
Name: count, dtype: int64
```

On apprend ainsi que 489 protéines sont d'origine humaine (`Human`) et 8 proviennent du maïs (`Maize`).

22.5.5 Statistiques par groupe

On peut aussi déterminer, pour chaque organisme, la taille et la masse moyenne des kinases :

```
df.groupby(["Organism"])[["Length", "Mass"]].mean()
```

```
              Length          Mass
Organism
Chicken    720.160000  81120.880000
Fruit fly  784.844660  88154.669903
Human      771.004090  86281.190184
Maize      666.875000  73635.000000
Mouse      768.092025  85942.274029
Rabbit     591.480000  66754.200000
Rat        722.379447  81081.822134
```

La méthode `.groupby()` rassemble d'abord les données suivant la colonne `Organism`. Puis on sélectionne les colonnes `Length` et `Mass`. Enfin, la méthode `.mean()` calcule la moyenne pour chaque groupe.

Si on souhaite obtenir deux statistiques (par exemple les valeurs minimale et maximale) en une seule fois, il convient alors d'utiliser la méthode `.pivot_table()`, méthode plus complexe, mais aussi beaucoup plus puissante :

```
df.pivot_table(
    index="Organism",
    values=["Length", "Mass"],
    aggfunc=["min", "max"]
)
```

```
            min           max
         Length   Mass Length    Mass
Organism
Chicken      303  34688   2311  260961
Fruit fly    294  33180   2554  287025
Human        253  28160   2986  340261
Maize        294  33834    996  105988
Mouse        244  27394   2964  337000
Rabbit        81   9405   1382  158347
Rat          274  31162   2959  336587
```

- L'argument `index` précise la colonne dont on veut agréger les données.
- L'argument `values` indique sur quelles colonnes les statistiques sont calculées.
- Enfin, `aggfunc` liste les statistiques calculées, ici les valeurs minimale et maximale.

Notez que les valeurs renvoyées sont d'abord les valeurs minimales pour `Length` et `Mass` puis les valeurs maximales pour `Length` et `Mass`.

22.5.6 Analyse de données numériques

On peut, sans trop de risque, émettre l'hypothèse que plus il y a d'acides aminés dans la protéine, plus sa masse va être élevée.

Pour vérifier cela graphiquement, on représente la masse de la protéine en fonction de sa taille (c'est-à-dire du nombre d'acides aminés) :

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.scatter(df["Length"], df["Mass"])
ax.set_xlabel("Taille (nombre d'acides aminés)")
ax.set_ylabel("Masse (Dalton)")
fig.savefig("kinases1.png")
```

On obtient un graphique similaire à celui de la figure 22.1.

Avec *pandas*, on peut aussi appeler une méthode `.plot()` sur un *Dataframe* pour obtenir une représentation graphique identique à la figure 22.1 :

```
import matplotlib.pyplot as plt

df.plot(
    kind="scatter",
    x="Length",
    y="Mass",
    xlabel="Taille (nombre d'acides aminés)",
    ylabel="Masse (Dalton)"
)
plt.savefig("kinases1.png")
```

- **Ligne 4.** On spécifie le type de graphique. Ici, un nuage de points.
- **Lignes 5 et 6.** On précise les colonnes à utiliser pour les abscisses et les ordonnées.

Le graphique de la figure 22.1 met en évidence une relation linéaire entre le nombre de résidus d'une protéine et sa masse.

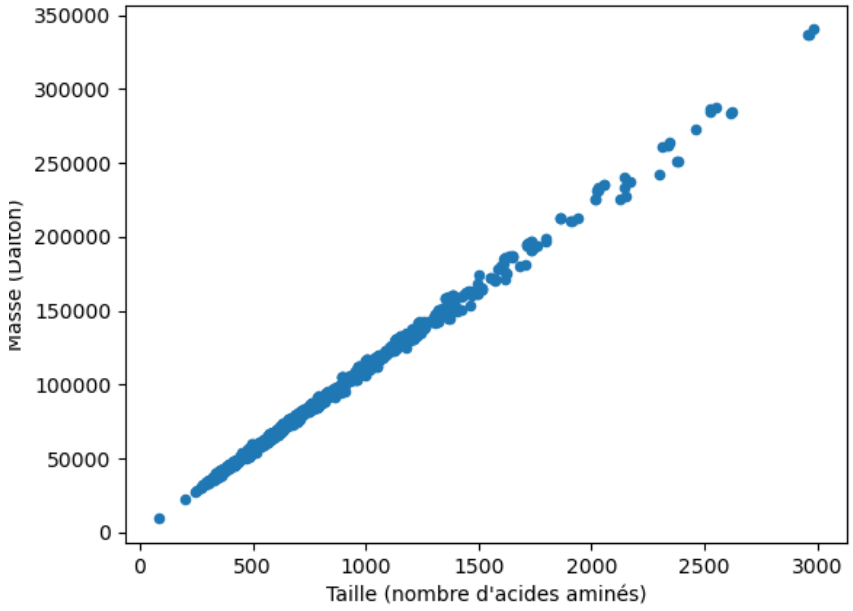


FIGURE 22.1 – Masse en fonction de la taille.

En réalisant une régression linéaire, on peut déterminer les paramètres de la droite qui passent le plus proche possible des points du graphique. On utilise pour cela la fonction `linregress()` [6] du module `scipy.stats` :

```
from scipy.stats import linregress
model = linregress(df["Length"], df["Mass"])
model
```

```
LinregressResult(slope=110.63478918698122, intercept=1055.431834679228,
rvalue=0.9989676084416755, pvalue=0.0, stderr=0.13258187632073232,
intercept_stderr=113.66584551734655)
```

Ce modèle linéaire nous indique qu'un résidu a une masse d'environ 111 Dalton, ce qui est cohérent. On peut également comparer ce modèle aux différentes protéines :

```
fig, ax = plt.subplots()
ax.scatter(df["Length"], df["Mass"], label="données")
ax.plot(
        df["Length"],
        df["Length"]*model.slope + model.intercept,
        ls=":",
        label="modèle"
)
ax.set_xlabel("Taille (nombre d'acides aminés)")
ax.set_ylabel("Masse (Dalton)")
ax.legend()
fig.savefig("kinases2.png")
```

On obtient ainsi le graphique de la figure 22.2.

22.5.7 Analyse de données temporelles

Il peut être intéressant de savoir, pour chaque organisme, quand les premières et les dernières séquences de kinases ont été référencées dans UniProt.

6. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.linregress.html

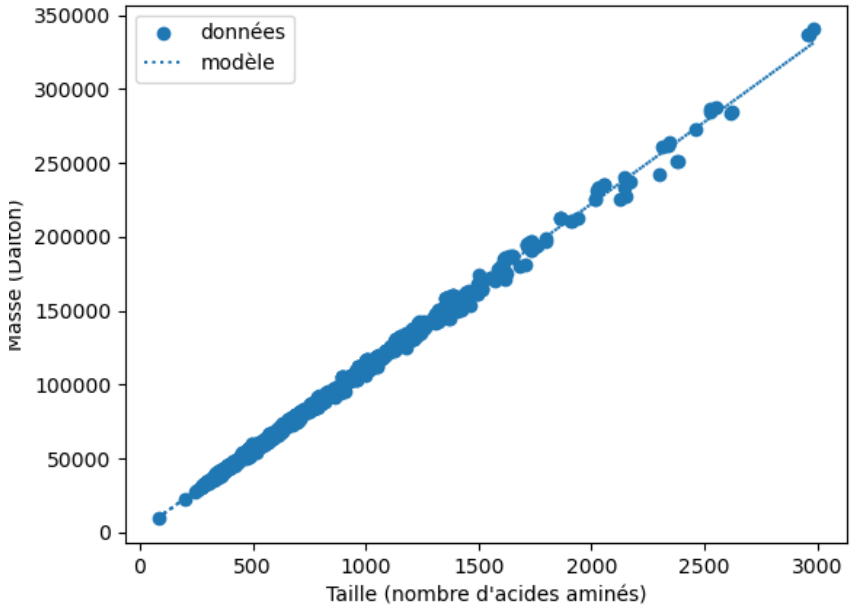


FIGURE 22.2 – Masse en fonction de la taille des protéines.

La méthode `.pivot_table()` apporte des éléments de réponse :

```
df.pivot_table(
    index="Organism",
    values=["Creation date"],
    aggfunc=["min", "max"]
)
```

```
                  min           max
        Creation date Creation date
Organism
Chicken      1986-07-21    2021-02-10
Fruit fly    1986-07-21    2023-09-13
Human        1986-07-21    2018-06-20
Maize        1990-08-01    2023-05-03
Mouse        1986-07-21    2017-03-15
Rabbit       1986-07-21    2010-03-02
Rat          1986-07-21    2023-09-13
```

Chez le poulet (*Chicken*), la première séquence a été référencée le 21 juillet 1986 et la dernière le 10 février 2021.

Une autre question est de savoir combien de kinases ont été référencées en fonction du temps.

La méthode `.value_counts()` peut être utilisée, mais elle ne renvoie que le nombre de protéines référencées dans UniProt pour un jour donné. Par exemple, 40 structures ont été référencées le 28 novembre 2006 :

```
df["Creation date"].value_counts().head()
```

```
Creation date
1997-11-01    72
1996-10-01    58
2000-12-01    43
2000-05-30    41
2006-11-28    40
Name: count, dtype: int64
```

Si on souhaite une réponse plus globale, par exemple à l'échelle de l'année, la méthode `.resample()` calcule le nombre de protéines référencées par an (en fournissant l'argument `YE`). En utilisant le *method chaining* présenté dans le chapitre 11 *Plus sur les chaînes de caractères*, nous pouvons écrire toutes ces transformations en une seule instruction, répartie sur plusieurs lignes pour plus de lisibilité (en utilisant des parenthèses) :

```
(df["Creation date"]
    .value_counts()
    .resample("YE")
    .sum()
    .head()
)
```

```
Creation date
1986-12-31    11
1987-12-31    12
1988-12-31    32
1989-12-31    29
1990-12-31    40
Freq: YE-DEC, Name: count, dtype: int64
```

Les dates apparaissent maintenant comme le dernier jour de l'année (31 décembre), mais désignent bien l'année complète. Dans cet exemple, 11 kinases ont été référencées dans UniProt entre le 1er janvier et le 31 décembre 1986.

Pour connaître en quelle année le plus de kinases ont été référencées dans UniProt, il faut trier les valeurs obtenues du plus grand au plus petit avec la méthode `.sort_values()`. Comme on ne veut connaître que les premières dates (celles où il y a eu le plus de protéines référencées), on utilisera également la méthode `.head()` :

```
(df["Creation date"]
    .value_counts()
    .resample("YE")
    .sum()
    .sort_values(ascending=False)
    .head()
)
```

```
Creation date
2006-12-31    167
2005-12-31    136
2004-12-31    118
2003-12-31    104
2007-12-31     88
Name: count, dtype: int64
```

En 2006, 167 kinases ont été référencées dans UniProt. La deuxième « meilleure » année est 2005 avec 136 protéines.

Toutes ces méthodes, enchaînées les unes à la suite des autres, peuvent vous sembler complexes, mais chacune d'elles correspond à une étape du traitement des données. Bien sûr, on aurait pu créer des variables intermédiaires pour chaque étape, mais cela aurait été plus lourd :

```
date1 = df["Creation date"].value_counts()
date2 = date1.resample("YE")
date3 = date2.sum()
date4 = date3.sort_values(ascending=False)
date4.head()
```

On aurait obtenu exactement le même résultat.

Remarque

Le *method chaining*[7] est une manière efficace et élégante de traiter des données avec *pandas*.

Enfin, pour obtenir un graphique de l'évolution du nombre de kinases référencées dans UniProt en fonction du temps,

7. https://www.youtube.com/watch?v=39MEeDLxGGg

on peut encore utiliser le *method chaining* :

```
import matplotlib.pyplot as plt
(df["Creation date"]
    .value_counts()
    .resample("YE")
    .sum()
    .plot()
)
plt.savefig("kinases3.png")
```

On obtient ainsi le graphique de la figure 22.3.

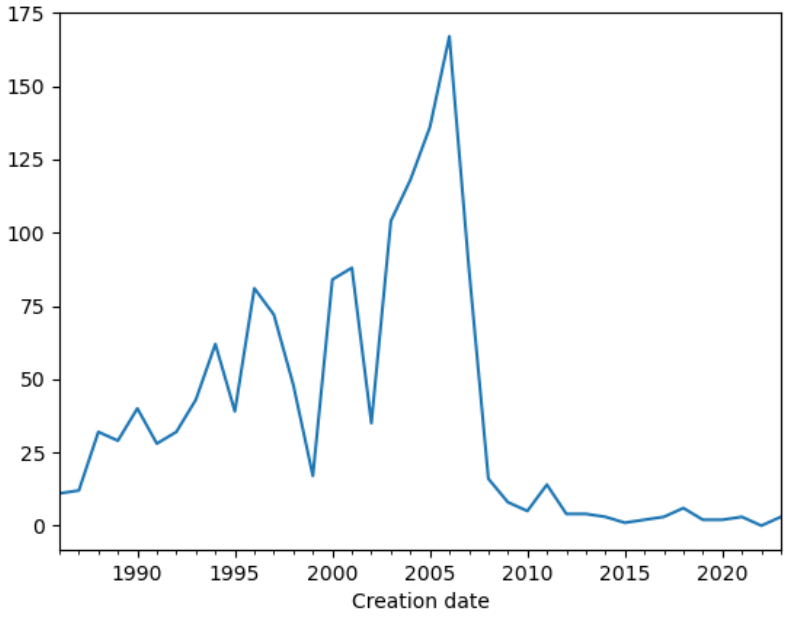


FIGURE 22.3 – Évolution temporelle du nombre de kinases référencées dans UniProt.

On observe un pic du nombre de kinases référencées dans UniProt sur la période 2003-2007.

22.5.8 Transformation d'une colonne

Nous avons vu précédemment que la colonne `PDB` contenait de nombreuses valeurs manquantes (`NaN`). Toutefois, il est intéressant de savoir ce que peut contenir cette colonne quand elle n'est pas vide :

```
(df
 .loc[ ~ df["PDB"].isna() ]
 .head(3)
)
```

Entry	Organism	Length	Creation date	Mass	PDB
A2CG49	Mouse	2964	2007-10-23	337000	1WFW;7UR2;
D3ZMK9	Rat	1368	2018-07-18	147716	6EWX;
O00141	Human	431	1998-12-15	48942	2R5T;3HDM;3HDN;7PUE;

- **Ligne 2.** La méthode `isna()` sélectionne les lignes qui contiennent des valeurs manquantes dans la colonne `PDB`, puis l'opérateur `~` inverse cette sélection.
- **Ligne 3.** On limite l'affichage aux trois premières lignes.

On découvre que la colonne `PDB` contient des identifiants de structures 3D de protéines. Ces identifiants sont séparés par des points-virgules, y compris pour la dernière valeur.

Nous souhaitons compter le nombre de structures 3D pour chaque protéine. Pour cela, nous allons d'abord créer une fonction qui compte le nombre de points-virgules dans une chaîne de caractères :

```
1  def count_structures(row):
2      if pd.isna(row["PDB"]):
3          return 0
4      else:
5          return row["PDB"].count(";")
```

Dans la ligne 2, la méthode `.isna()` teste si la valeur est manquante et si ce n'est pas le cas, la fonction renvoie le nombre de points-virgules dans la chaîne de caractères de la colonne `PDB` (ligne 5).

On applique ensuite la fonction `count_structures()` au *Dataframe* avec la méthode `.apply()`. On crée la nouvelle colonne `nb_structures` en même temps :

```
1  df["nb_structures"] = df.apply(count_structures, axis=1)
2  df.head()
```

Entry	Organism	Length	Creation date	Mass	PDB	nb_structures
A0A0B4J2F2	Human	783	2018-06-20	84930	NaN	0
A4L9P5	Rat	1211	2007-07-24	130801	NaN	0
A0A1D6E0S8	Maize	856	2023-05-03	93153	NaN	0
A0A8I5ZNK2	Rat	528	2023-09-13	58360	NaN	0
A1Z7T0	Fruit fly	1190	2012-01-25	131791	NaN	0

Les premières lignes ne sont pas très intéressantes, car elles ne contiennent pas de structures 3D. Mais on peut chercher les kinases qui ont le plus de structures 3D :

```
1  (df
2   .sort_values(by="nb_structures", ascending=False)
3   .filter(["Organism", "nb_structures"])
4   .head()
5  )
```

- **Ligne 2.** On trie les données par ordre décroissant de la colonne `nb_structures`.
- **Ligne 3.** On ne conserve que les colonnes `Organism` et `nb_structures` à afficher.
- **Ligne 4.** On limite l'affichage aux cinq premières lignes.

Entry	Organism	nb_structures
P24941	Human	453
P00533	Human	284
Q16539	Human	245
P68400	Human	238
P11309	Human	176

La kinase `P24941` possède 453 structures 3D référencées dans UniProt. Les cinq kinases qui ont le plus de structures 3D sont toutes d'origine humaine.

Pour aller plus loin

Les ouvrages *Python for Data Analysis* (2022) de Wes McKinney et *Effective Pandas* (2021) de Matt Harrison sont d'excellentes références pour *pandas*.

Remarque

La réponse à la devinette précédente est :

Une protéine kinase

(Une protéine qui nage… dans une piscine… Vous l'avez ?)

22.6 Exercices

Conseil

Pour ces exercices, utilisez des *notebooks* Jupyter.

22.6.1 Analyse d'un jeu de données

Le jeu de données `people.tsv` contient les caractéristiques de quelques individus : prénom, sexe, taille (en cm) et âge (en années). Par exemple :

name	sex	size	age
simon	male	175	33
clara	female	167	45
serge	male	181	44
claire	female	174	31
…	…	…	…

L'objectif de cet exercice est de manipuler ce jeu de données avec *pandas*, de sélectionner des données et d'en calculer quelques statistiques.

Conseil

Si vous n'êtes pas familier avec le format de fichier `.tsv`, nous vous conseillons de consulter l'annexe A *Quelques formats de données en biologie*.

1. **Chargement du jeu de données**
   - Téléchargez le fichier people.tsv[8].
   - Ouvrez ce fichier avec *pandas* et la fonction `.read_csv()`. N'oubliez pas de préciser le séparateur par défaut avec l'argument `sep="\t"`. Utilisez également l'argument `index_col` pour utiliser la colonne `name` comme index.
   - Affichez les six premières lignes du jeu de données.
   - Combien de lignes contient le jeu de données ?
2. **Sélections**
   - Déterminez la taille de Claire.
   - Déterminez l'âge de Baptiste.
   - Affichez, en une seule commande, l'âge de Paul et Bob.
3. **Statistiques descriptives et table de comptage**
   - Déterminez la moyenne et la valeur minimale de la taille et l'âge des individus.
   - Comptez ensuite le nombre de personnes de chaque sexe.
4. **Statistiques par groupe**
   - Déterminez la taille et l'âge moyen chez les hommes et les femmes. Utilisez pour cela la méthode `.groupby()`.
5. **Sélections par filtre**
   - Déterminez combien de d'individus mesurent plus de 1,80 m.
   - Quelle femme a moins de 35 ans ?
6. **Sélections et statistiques**
   - Déterminez l'âge moyen des individus qui mesurent plus de 1,80 m.

8. https://python.sdv.u-paris.fr/data-files/people.tsv

- Déterminez la taille maximale des femmes qui ont plus de 35 ans.

CHAPITRE 23

Avoir la classe avec les objets

La programmation orientée objet (POO) est un concept de programmation très puissant qui permet de structurer ses programmes d'une manière nouvelle. En POO, on définit un « objet » qui peut contenir des « attributs » ainsi que des « méthodes » qui agissent sur lui-même. Par exemple, on définit un objet « citron » qui contient les attributs « saveur » et « couleur », ainsi qu'une méthode « presser » permettant d'en extraire le jus. En Python, on utilise une « classe » pour construire un objet. Dans notre exemple, la classe correspondrait au « moule » utilisé pour construire autant d'objets citrons que nécessaire.

Définition

Une **classe** définit des **objets**, qui sont des **instances** (des représentants) de cette classe. Dans ce chapitre, on utilisera les mots *objet* ou *instance* pour désigner la même chose. Les objets peuvent posséder des **attributs** (variables associées aux objets) et des **méthodes** (qui sont des fonctions associées aux objets et qui peuvent agir sur ces derniers, ou encore les utiliser).

Dans les chapitres précédents, nous avons déjà mentionné qu'en Python tout est objet. Une variable de type *int* est en fait un objet de type *int*, donc construit à partir de la classe *int*. Même chose pour les *float* et *string*, mais aussi pour les *list*, *tuple*, *dict*, etc. Voilà pourquoi nous avons rencontré de nombreuses notations et mots de vocabulaire associés à la POO depuis le début de ce cours.

La POO permet de produire du code plus compact et plus facilement réutilisable. L'utilisation de classes évite l'utilisation de variables globales en créant ce qu'on appelle un *espace de noms*, propre à chaque objet et permettant d'y *encapsuler* des attributs et des méthodes. De plus, la POO amène de nouveaux concepts tels que le *polymorphisme* (capacité à redéfinir le comportement des opérateurs), ou bien encore l'*héritage* (capacité à définir une classe à partir d'une classe pré-existante et d'y ajouter de nouvelles fonctionnalités). Tous ces concepts seront définis dans ce chapitre.

Malgré tous ces avantages, la POO peut paraître difficile à aborder pour le débutant, spécialement dans la conception des programmes. Elle nécessite donc la lecture de nombreux exemples, mais surtout beaucoup de pratique. Bien structurer ses programmes en POO est un véritable art. Il existe même des langages qui formalisent la construction de programmes orientés objets, par exemple le langage UML[1].

Dans ce chapitre, nous vous donnerons tous les éléments pour démarrer la construction de vos premières classes. Le chapitre 24 *Avoir plus la classe avec les objets* (en ligne) abordera des aspects plus poussés de la POO, comme le polymorphisme, la composition, l'héritage, certains pièges à éviter, ainsi que des bonnes pratiques.

1. https://fr.wikipedia.org/wiki/UML_(informatique)

Après la lecture de ces deux chapitres sur la POO avec Python, vous verrez d'un autre œil de nombreux exemples évoqués dans les chapitres précédents, et vous comprendrez sans doute de nombreuses subtilités qui avaient pu vous paraître absconses.

Enfin, il est vivement recommandé de lire ces deux chapitres sur la POO avant d'aborder le chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne).

23.1 Construction d'une classe

Nous allons voir dans cette rubrique comment définir une classe en reprenant notre exemple sur le citron, que nous allons faire évoluer et complexifier. Attention, certains exemples sont destinés à vous montrer comment les classes fonctionnent, mais leur utilisation n'aurait pas de sens dans un vrai programme. Ainsi, nous vous donnerons plus loin dans ce chapitre les pratiques recommandées.

23.1.1 La classe minimale

En Python, le mot-clé `class` permet de créer sa propre classe, suivi du nom de cette classe. On se souvient, un nom de classe commence toujours par une majuscule (voir le chapitre 16 *Bonnes pratiques en programmation Python*). Comme d'habitude, cette ligne attend un bloc d'instructions indenté définissant le corps de la classe. Voyons un exemple simple dans l'interpréteur :

```
1  >>> class Citron:
2  ...     pass
3  ...
4  >>> Citron
5  <class '__main__.Citron'>
6  >>> type(Citron)
7  <class 'type'>
8  >>> citron1 = Citron()
9  >>> citron1
10 <__main__.Citron object at 0x7ff2193a20f0>
11 >>>
```

Ligne 1. La classe `Citron` est définie. Pas besoin de parenthèses comme avec les fonctions dans un cas simple comme celui-là (nous verrons d'autres exemples plus loin où elles seront nécessaires).

Ligne 2. La classe ne contient rien, mais il faut mettre au moins une ligne, on met donc ici le mot-clé Python `pass` qui ne fait rien (comme dans une fonction qui ne fait rien).

Lignes 4 et 5. Quand on tape le nom de notre classe `Citron`, Python nous indique que cette classe est connue.

Lignes 6 et 7. Lorsqu'on regarde le type de notre classe `Citron`, Python nous indique qu'il s'agit d'un type au même titre que `type(int)`. Nous avons donc créé un nouveau type !

Ligne 8. On crée une instance de la classe `Citron`, c'est-à-dire qu'on fabrique un représentant ou objet de la classe `Citron`, que nous nommons `citron1`.

Lignes 9 et 10. Lorsqu'on tape le nom de l'instance `citron1`, l'interpréteur nous rappelle qu'il s'agit d'un objet de type `Citron`, ainsi que son adresse en mémoire.

Il est également possible de vérifier qu'une instance est bien issue d'une classe donnée avec la fonction `isinstance()` :

```
1  >>> isinstance(citron1, Citron)
2  True
```

23.1.2 Ajout d'un attribut d'instance

Reprenons notre classe `Citron` et l'instance `citron1` créée précédemment. Regardons les attributs et méthodes que cet objet possède, puis tentons de lui ajouter un attribut :

```
1  >>> dir(citron1)
2  ['__class__', '__delattr__', '__dict__', [...], '__weakref__']
3  >>> citron1.couleur = "jaune"
4  >>> dir(citron1)
5  ['__class__', '__delattr__', '__dict__', [...], '__weakref__', 'couleur']
6  >>> citron1.couleur
7  'jaune'
```

Lignes 1 et 2. L'objet possède de nombreuses méthodes ou attributs, qui commencent et qui se terminent par deux caractères *underscores*. On se souvient que les *underscores* indiquent qu'il s'agit de méthodes ou attributs destinés au fonctionnement interne de l'objet. Nous reviendrons sur certains d'entre-eux dans la suite.

Ligne 3. Ici on ajoute un attribut `.couleur` à l'instance `citron1`. Notez bien la syntaxe `instance.attribut` et le point qui lie les deux.

Lignes 4 à 5. La fonction `dir()` nous montre que l'attribut `.couleur` a bien été ajouté à l'objet.

Lignes 6. La notation `instance.attribut` donne accès à l'attribut de l'objet.

L'attribut nommé `.__dict__` est particulièrement intéressant. Il s'agit d'un dictionnaire qui listera les attributs créés dynamiquement dans l'instance en cours :

```
1  >>> citron1 = Citron()
2  >>> citron1.__dict__
3  {}
4  >>> citron1.couleur = "jaune"
5  >>> citron1.__dict__
6  {'couleur': 'jaune'}
```

L'ajout d'un attribut depuis l'extérieur de la classe (on parle aussi du côté « client ») avec une syntaxe `instance.nouvel_attribut = valeur`, créera ce nouvel attribut uniquement pour cette instance :

```
1  citron1 = Citron()
2  citron1.couleur = "jaune"
3  >>> citron1.__dict__
4  {'couleur': 'jaune'}
5  >>> citron2 = Citron()
6  >>> citron2.__dict__
7  {}
```

Si on crée une nouvelle instance de `Citron`, ici `citron2`, elle n'aura pas l'attribut `couleur` à sa création.

Définition

Une **variable** ou **attribut d'instance** est une variable accrochée à une instance et qui lui est spécifique. Cet attribut n'existe donc pas forcément pour toutes les instances d'une classe donnée et, d'une instance à l'autre, il ne prendra pas forcément la même valeur. On peut retrouver tous les attributs d'instance d'une instance donnée avec une syntaxe `instance.__dict__`.

L'instruction `del` fonctionne bien sûr pour détruire un objet (par exemple : `del citron1`), mais permet également de détruire un attribut d'instance. Si on reprend notre exemple `citron1` ci-dessus :

```
1  >>> citron1.__dict__
2  {'couleur': 'jaune'}
3  >>> del citron1.couleur
4  >>> citron1.__dict__
5  {}
```

Dans la suite, on montrera du code à tester dans un script : n'hésitez pas, comme d'habitude, à le tester par vous-même.

23.1.3 Les attributs de classe

Si on ajoute une variable dans une classe comme on créait une variable locale dans une fonction, on crée ce qu'on appelle un attribut de classe :

```
1  class Citron:
2      couleur = "jaune"
```

Définition

Une **variable de classe** ou **attribut de classe** est un attribut qui sera identique pour chaque instance. On verra plus bas que de tels attributs suivent des règles différentes par rapport aux attributs d'instance.

À l'extérieur ou à l'intérieur d'une classe, un attribut de classe peut se retrouver avec une syntaxe `NomClasse.attribut` :

```
1  print(Citron.couleur)
```

Ce code affiche `jaune`. L'attribut de classe est aussi visible depuis n'importe quelle instance :

```
1  class Citron:
2      couleur = "jaune"
3
4
5  if __name__ == "__main__":
6      citron1 = Citron()
7      print(citron1.couleur)
8      citron2 = Citron()
9      print(citron2.couleur)
```

L'exécution de ce code affichera :

```
jaune
jaune
```

Attention

Même si on peut retrouver un attribut de classe avec la syntaxe `instance.attribut`, un tel attribut ne peut pas être modifié avec une instruction de cette forme :

```
1  instance.attribut = nouvelle_valeur
```

(voir la rubrique *Différence entre les attributs de classe et d'instance*).

23.1.4 Les méthodes

Dans notre classe, on pourra aussi ajouter des fonctions.

Définition

Une fonction définie au sein d'une classe est appelée **méthode**. Pour exécuter une méthode à l'extérieur de la classe, la syntaxe générale est `instance.méthode()`. En général, on distingue attributs et méthodes (comme nous le ferons systématiquement dans ce chapitre). Toutefois, il faut garder à l'esprit qu'une méthode est finalement un objet de type fonction. Ainsi, elle peut être vue comme un attribut également, concept que vous croiserez peut-être en consultant de la documentation externe.

Voici un exemple d'ajout d'une fonction, ou plus exactement d'une méthode, au sein d'une classe (attention à l'indentation !) :

```
1  class Citron:
2      def coucou(self):
3          print("Coucou, je suis la mth .coucou() dans la classe Citron !")
4
5
6  if __name__ == "__main__":
7      citron1 = Citron()
8      citron1.coucou()
```

Lignes 2 et 3. On définit une méthode nommée `.coucou()`, qui va afficher un petit message. Attention, cette méthode prend obligatoirement un argument que nous avons nommé ici `self`. Nous verrons dans les deux prochaines

rubriques la signification de ce `self`. Si on a plusieurs méthodes dans une classe, on saute toujours une ligne entre elles afin de faciliter la lecture (comme pour les fonctions).

Ligne 7 et 8. On crée l'instance `citron1` de la classe `Citron`, puis on exécute la méthode `.coucou()` avec une syntaxe `instance.méthode()`.

Une méthode étant une fonction, elle peut bien sûr retourner une valeur :

```
1  class Citron:
2      def recup_saveur(self):
3          return "acide"
4
5
6  if __name__ == "__main__":
7      citron1 = Citron()
8      saveur_citron1 = citron1.recup_saveur()
9      print(saveur_citron1)
```

Vous l'aurez deviné, ce code affichera `acide` à l'écran. Comme pour les fonctions, une valeur retournée par une méthode est récupérable dans une variable, ici `saveur_citron1`.

23.1.5 Le constructeur

Lors de l'instanciation d'un objet à partir d'une classe, il peut être intéressant de lancer des instructions, comme, d'initialiser certaines variables. Pour cela, on ajoute une méthode spéciale nommée `.__init__()` : cette méthode s'appelle le « constructeur » de la classe. Il s'agit d'une méthode spéciale dont le nom est entouré de doubles *underscores* : en effet, elle sert au fonctionnement interne de notre classe et, sauf cas extrêmement rare, elle n'est pas supposée être lancée comme une fonction classique par l'utilisateur de la classe. Ce constructeur est exécuté à chaque instanciation de notre classe, et ne renvoie pas de valeur, il ne possède donc pas de `return`.

Remarque

Pour les débutants, vous pouvez sauter cette remarque. Certains auteurs préfèrent nommer `.__init__()` « instantiateur » ou « initialisateur », pour signifier qu'il existe une autre méthode appelée `.__new__()`, qui participe à la création d'une instance. Vous n'avez bien sûr pas à retenir ces détails pour continuer la lecture de ce chapitre, retenez simplement que nous avons décidé de nommer la méthode `.__init__()` « constructeur » dans cet ouvrage.

Pour bien comprendre comment cela fonctionne, nous allons suivre un exemple simple avec le site *Python Tutor*[2] (déjà utilisé dans les chapitres 10 et 13 sur les fonctions). N'hésitez pas à copier/coller ce code dans *Python Tutor* pour le tester vous-même :

```
1  class Citron:
2      def __init__(self):
3          self.couleur = "jaune"
4
5
6  if __name__ == "__main__":
7      citron1 = Citron()
8      print(citron1.couleur)
```

Étape 1

Figure 23.1. Au départ, *Python Tutor* nous montre que la classe `Citron` a été mise en mémoire, elle contient pour l'instant la méthode `.__init__()`.

Étape 2

Figure 23.2. Nous créons ensuite l'instance `citron1` à partir de la classe `Citron`. Notre classe `Citron` contenant une méthode `.__init__()` (le constructeur), celle-ci est immédiatement exécutée au moment de l'instanciation. Cette méthode prend un argument nommé `self` : cet argument est **obligatoire**. Il s'agit en fait d'une référence vers l'instance

2. http://www.pythontutor.com

FIGURE 23.1 – Fonctionnement d'un constructeur (étape 1).

en cours (instance que nous appellerons `citron1` dans le programme principal, mais cela serait vrai pour n'importe quel autre nom d'instance). *Python Tutor* nous indique cela par une flèche pointant vers un espace nommé `Citron instance`. La signification du `self` est expliquée en détail dans la rubrique suivante.

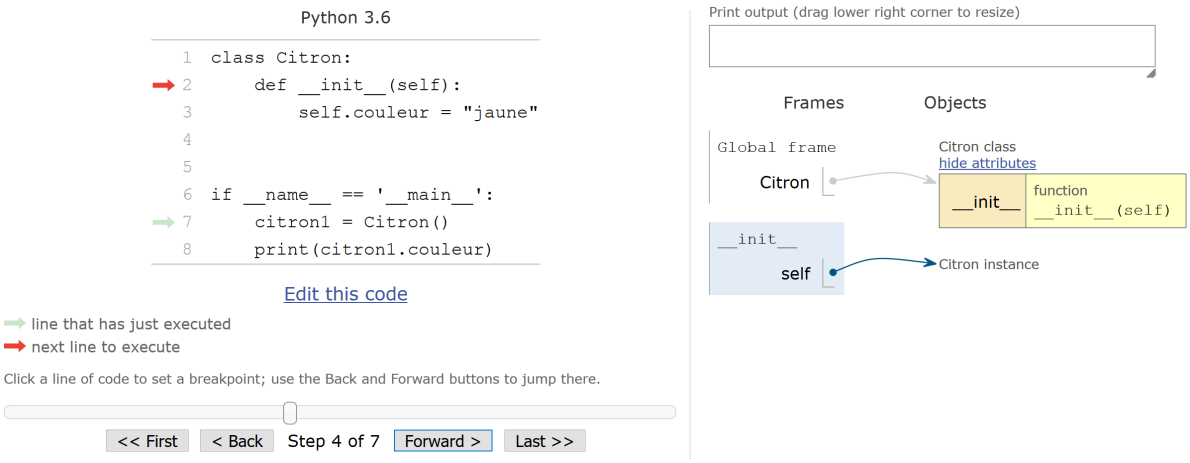


FIGURE 23.2 – Fonctionnement d'un constructeur (étape 2).

Étape 3

Figure 23.3. Un nouvel attribut est créé s'appelant `self.couleur`. La chaîne de caractères `couleur` est ainsi « accrochée » (grâce au caractère point) à l'instance en cours référencée par le `self`. *Python Tutor* nous montre cela par une flèche qui pointe depuis le `self` vers la variable `couleur` (qui se trouve elle-même dans l'espace nommé `Citron instance`). Si d'autres attributs étaient créés, ils seraient tous répertoriés dans cet espace `Citron instance`. Vous l'aurez compris, l'attribut `couleur` est donc une variable d'instance (voir rubrique *Ajout d'un attribut d'instance* ci-dessus). La méthode `.__init__()` étant intrinsèquement une fonction, *Python Tutor* nous rappelle qu'elle ne renvoie rien (d'où le `None` dans la case *Return value*), une fois son exécution terminée. Et comme avec les fonctions classiques, l'espace mémoire contenant les variables locales à cette méthode va être détruit une fois son exécution terminée.

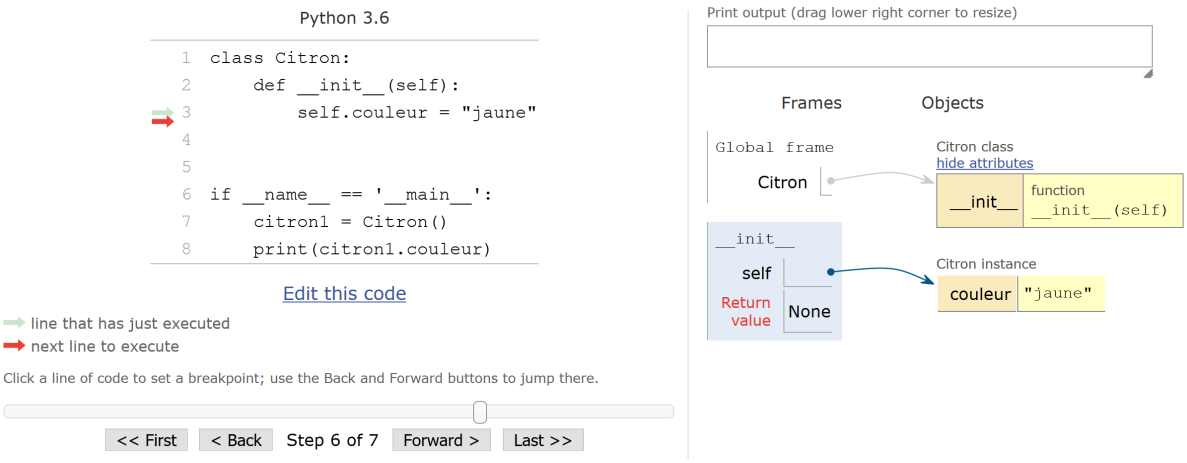


FIGURE 23.3 – Fonctionnement d'un constructeur (étape 3).

Étape 4

Figure 23.4. De retour dans le programme principal, *Python Tutor* nous indique que `citron1` est une instance de la classe `Citron` par une flèche pointant vers l'espace `Citron instance`. Cette instance contient un attribut nommé `couleur` auquel on accéde avec la syntaxe `citron1.couleur` dans le `print()`. Notez que si l'instance s'était appelée `enorme_citron`, on aurait utilisé `enorme_citron.couleur` pour accéder à l'attribut `couleur`.

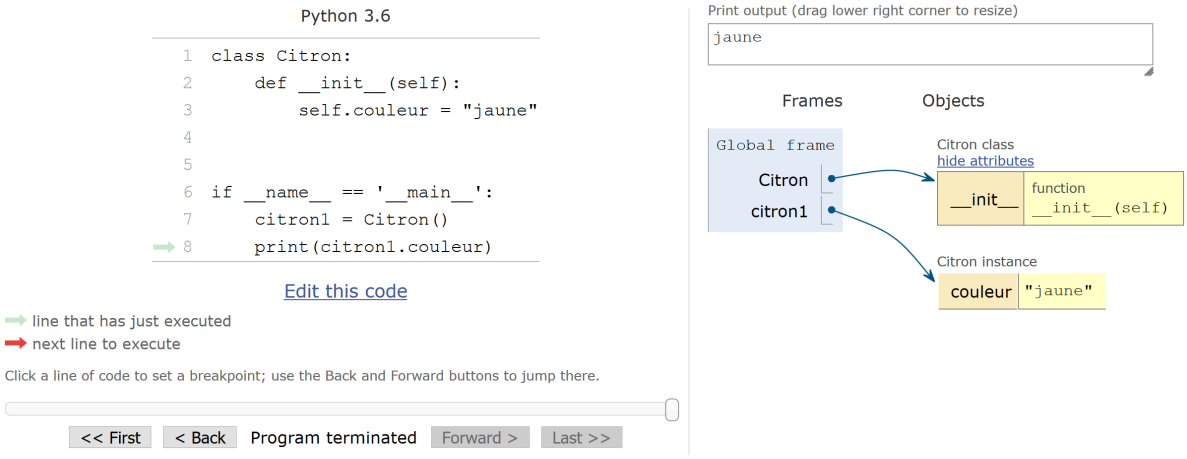


FIGURE 23.4 – Fonctionnement d'un constructeur (étape 4).

Conseil

Dans la mesure du possible, nous vous conseillons de créer tous les attributs d'instance dont vous aurez besoin dans le constructeur `.__init__()` plutôt que dans toute autre méthode. Ainsi, ils seront visibles dans toute la classe dès l'instanciation.

23.1.6 Passage d'argument(s) à l'instanciation

Lors de l'instanciation, il est possible de passer des arguments au constructeur. Comme pour les fonctions, on peut passer des arguments positionnels ou par mot-clé, et en créer autant que l'on veut (voir chapitre 10 *Fonctions*). Voici un exemple :

```
1  class Citron:
2      def __init__(self, masse, couleur="jaune"):
3          self.masse = masse
4          self.couleur = couleur
5
6
7  if __name__ == "__main__":
8      citron1 = Citron(100)
9      print("citron1:", citron1.__dict__)
10     citron2 = Citron(150, couleur="blanc")
11     print("citron2:", citron2.__dict__)
```

On a ici un argument positionnel (`masse`) et un autre par mot-clé (`couleur`). Le code donnera la sortie suivante :

```
1  citron1: {'masse': 100, 'couleur': 'jaune'}
2  citron2: {'masse': 150, 'couleur': 'blanc'}
```

23.1.7 Mieux comprendre le rôle du `self`

Cette rubrique va nous aider à mieux comprendre le rôle du `self` à travers quelques exemples simples. Regardons le code suivant dans lequel nous créons une nouvelle méthode `.affiche_attributs()` :

```
1  class Citron:
2      def __init__(self, couleur="jaune"):
3          self.couleur = couleur
4          var = 2
5
6      def affiche_attributs(self):
7          print(self)
8          print(self.couleur)
9          print(var)
10
11
12 if __name__ == "__main__":
13     citron1 = Citron()
14     citron1.affiche_attributs()
```

Ligne 3. On crée l'attribut `couleur` que l'on accroche à l'instance avec `self`.

Ligne 4. Nous créons cette fois-ci une variable `var` sans l'accrocher à `self`.

Ligne 6. Nous créons une nouvelle méthode dans la classe `Citron` qui se nomme `.affiche_attributs()`. Comme pour le constructeur, cette méthode prend comme premier argument une variable obligatoire, que nous avons à nouveau nommée `self`. Il s'agit encore une fois d'une référence vers l'objet ou instance créé(e).

Attention

On peut appeler cette référence comme on veut, toutefois nous vous conseillons vivement de l'appeler `self`, car c'est une convention en Python. Ainsi, quelqu'un qui lira votre code comprendra immédiatement de quoi il s'agit.

Ligne 7. Cette ligne va afficher le contenu de la variable `self`.

Lignes 8 et 9. On souhaite que notre méthode `.affiche_attributs()` affiche ensuite l'attribut de classe `.couleur` ainsi que la variable `var` créée dans le constructeur `.__init__()`.

L'exécution de ce code donnera :

```
$ python classe_exemple1.py
<__main__.Citron object at 0x7f4e5fb71438>
jaune
Traceback (most recent call last):
  File "classe_exemple1.py", line 14, in <module>
    citron1.affiche_attributs()
  File "classe_exemple1.py", line 9, in affiche_attributs
    print(var)
          ^^^
NameError: name 'var' is not defined. Did you mean: 'vars'?
```

Ligne 2. La méthode `.affiche_attributs()` montre que le `self` est bien une référence vers l'instance (ou objet) `citron1` (ou vers n'importe quelle autre instance : par exemple, si on crée `citron2 = Citron()`, le `self` sera une référence vers `citron2`).

Ligne 3. La méthode `.affiche_attributs()` affiche l'attribut `.couleur`, qui avait été créé précédemment dans le constructeur. Vous voyez ici l'intérêt principal de l'argument `self` passé en premier à chaque méthode d'une classe : il « accroche » n'importe quel attribut qui sera visible partout dans la classe, y compris dans une méthode où il n'a pas été défini.

Lignes 4 à 9. La création de la variable `var` dans la méthode `.__init__()` sans l'accrocher à l'objet `self` fait qu'elle n'est plus accessible en dehors de `.__init__()`. C'est exactement comme pour les fonctions classiques, `var` est finalement une variable locale au sein de la méthode `.__init__()` et n'est plus visible lorsque l'exécution de cette dernière est terminée (voir les chapitres 10 et 13 sur les fonctions). Ainsi, Python renvoie une erreur, car `var` n'existe pas lorsque `.affiche_attributs()` est en exécution.

En résumé, le `self` est nécessaire lorsqu'on a besoin d'accéder à différents attributs dans les différentes méthodes d'une classe. Le `self` est également nécessaire pour appeler une méthode de la classe depuis une autre méthode :

```
1  class Citron:
2      def __init__(self, couleur="jaune"):
3          self.couleur = couleur
4          self.affiche_message()
5
6      def affiche_message(self):
7          print("Le citron c'est trop bon !")
8
9
10 if __name__ == "__main__":
11     citron1 = Citron("jaune pâle")
```

Ligne 4. Nous appelons ici la méthode `.affiche_message()` depuis le constructeur. Pour appeler cette méthode interne à la classe `Citron`, on doit utiliser une syntaxe `self.méthode()`. Le `self` sert donc pour accéder aux attributs, mais aussi aux méthodes, ou plus généralement à tout ce qui est accroché à la classe.

Lignes 6 et 7. La méthode `.affiche_message()` est exécutée. On peut se poser la question « Pourquoi passer l'argument self à cette méthode alors qu'on ne s'en sert pas dans celle-ci ? »

Attention

Même si on ne se sert d'aucun attribut dans une méthode, l'argument `self` (ou quel que soit son nom) est **strictement obligatoire**. En fait, la notation `citron1.affiche_message()` est équivalente à `Citron.affiche_message(citron1)`. Testez les deux pour voir ! Dans cette dernière instruction, on appelle la méthode accrochée à la classe `Citron` et on lui passe explicitement l'instance `citron1` en tant qu'argument. La notation `citron1.affiche_message()` contient donc en filigrane un argument, à savoir la référence vers l'instance `citron1` que l'on appelle `self` au sein de la méthode.

Conseil

C'est la première notation `citron1.affiche_attributs()` (ou plus généralement `instance.méthode()`), plus compacte, qui sera toujours utilisée.

Ligne 11. On crée l'instance `citron1` en lui passant l'argument `"jaune pâle"`. La variable d'instance `couleur` prendra ainsi cette valeur au lieu de celle par défaut (`"jaune"`). À noter, l'instanciation affichera le message `Le citron c'est trop bon !` puisque la méthode `.affiche_attributs()` est appelée dans le constructeur `.__init__()`.

Afin de bien comprendre les différentes étapes des codes de cette rubrique, nous vous conseillons de les retester de votre côté dans *Python Tutor*.

23.1.8 Remarque finale

Dans ce chapitre, nous avons vu les bases pour construire une classe. Toutefois, nous avons encore de nombreuses notions à vous montrer afin de pouvoir utiliser la POO à plein régime. Dans le chapitre 24 *Avoir plus la classe avec les objets* (en ligne), nous verrons les concepts de polymorphisme, composition et héritage qui donnent toute la puissance à la POO. D'autres notions comme les décorateurs `property` seront abordées permettant le contrôle des attributs par un utilisateur de la classe. Nous donnerons également des conseils généraux quand vous utilisez la POO. Le chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne) illustrera l'utilisation de la POO pour concevoir des fenêtres graphiques avec le module *Tkinter*.

23.2 Exercices

Conseil

Pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

23.2.1 Classe `Rectangle`

Téléchargez le script `rectangle.py`[3] qui implémente la classe `Rectangle`.
Complétez le programme principal pour que le script :

- crée une instance `rectangle` de la classe `Rectangle` ;
- affiche les attributs d'instance `largeur`, `longueur` et `couleur` ;
- calcule et affiche la surface de `rectangle` ;
- affiche une ligne vide ;
- change le rectangle en carré de 30 m de côté ;
- calcule et affiche la surface de ce carré ;
- crée une autre instance `rectangle2`, aux dimensions et à la couleur que vous souhaitez (soyez créatif !) et qui affiche les attributs et la surface de ce nouveau rectangle.

23.2.2 Classe `Rectangle` améliorée

Entraînez-vous avec la classe `Rectangle`. Créez la méthode `calcule_perimetre()` qui calcule le périmètre d'un objet rectangle. Testez sur un exemple simple (largeur = 10 m, longueur = 20 m).

23.2.3 Classe `Atome`

Créez une nouvelle classe `Atome` avec les attributs `x`, `y`, `z`, qui contiennent les coordonnées atomiques, et la méthode `calcul_distance()`, qui calcule la distance entre deux atomes. Testez cette classe sur plusieurs exemples.

23.2.4 Classe `Atome` améliorée

Améliorez la classe `Atome` en lui ajoutant un nouvel attribut `masse`, qui correspond à la masse atomique, ainsi qu'une nouvelle méthode `.calcule_centre_masse()`. Que se passe-t-il quand vous utilisez l'instruction `print()` avec une instance d'un objet `Atome` ? Dans votre classe, ajoutez la méthode suivante :

```
1  def __str__(self):
2      """Redéfinition du comportement avec print()."""
3      return f"coords({self.x}, {self.y}, {self.z}) ; mass = {self.masse}"
```

Utilisez à nouveau l'instruction `print()` avec un objet de la classe `Atome`. Que constatez-vous par rapport au précédent `print()` ?

3. https://python.sdv.u-paris.fr/data-files/rectangle.py

CHAPITRE 24

Avoir plus la classe avec les objets

Dans le chapitre précédent, nous avons vu les bases sur comment créer une classe, les notions d'attributs d'instance et de classe, le fonctionnent d'un constructeur et comment passer des arguments lors de l'instanciation. Nous avons vu qu'une classe pouvait être vue comme un constructeur de conteneur (chaque conteneur construit est une instance), qu'on pouvait y mettre tout un tas de variables ou objets (les attributs d'instance), mais également nous pouvions définir des méthodes réalisant des actions pour modifier ce que contient l'objet.

Dans le présent chapitre, nous abordons de nouvelles notions qui augmentent la puissance des classes, à savoir le polymorphisme, l'héritage et la composition. Nous verrons également les décorateurs *property* permettant le contrôle de l'accès aux attributs. À la fin du chapitre, nous vous donnerons des bonnes pratiques pour construire vos classes. Mais avant d'aborder ces sujets, nous revenons sur un concept important en Python, à savoir les espaces de noms.

24.1 Espace de noms

La notion d'**espace de noms** est importante lorsqu'on étudie les classes. Nous avons déjà croisé ce concept à plusieurs reprises. D'abord dans le chapitre 13 *Plus sur les fonctions*, puis dans le chapitre 15 *Création de modules*, et maintenant dans ce chapitre. De quoi s'agit-il ?

Définition

Dans la documentation officielle[1], un espace de noms est défini comme suit : « *a namespace is a mapping from names to objects* ». Un espace de noms, c'est finalement une correspondance entre des noms et des objets. Un espace de noms peut être vu aussi comme une capsule dans laquelle on trouve des noms d'objets. Par exemple, le programme principal ou une fonction représentent chacun un espace de noms, un module aussi, et bien sûr une classe ou l'instance d'une classe également.

Différents espaces de noms peuvent contenir des objets de même nom sans que cela ne pose de problème. Parce qu'ils sont chacun dans un espace différent, ils peuvent cohabiter sans risque d'écrasement de l'un par l'autre. Par exemple, à chaque fois que l'on appelle une fonction, un espace de noms est créé pour cette fonction. *Python Tutor* nous montre cet espace sous la forme d'une zone dédiée (voir les chapitres 10 et 13 sur les fonctions). Si cette fonction appelle une autre fonction, un nouvel espace est créé, bien distinct de la fonction appelante (ce nouvel espace peut donc contenir

1. https://docs.python.org/fr/3/tutorial/classes.html#python-scopes-and-namespaces

un objet de même nom). En définitive, ce qui va compter, c'est de savoir quelles règles Python va utiliser pour chercher dans les différents espaces de noms pour finalement accéder à un objet.

Nous allons dans cette rubrique refaire le point sur ce que l'on a appris dans cet ouvrage sur les espaces de noms en Python, puis se pencher sur les spécificités de ce concept dans les classes.

24.1.1 Rappel sur la règle LGI

Comme vu dans le chapitre 10 *Fonctions*, la règle LGI peut être résumée ainsi : *Local* > *Global* > *Interne*. Lorsque Python rencontre un objet, il utilise cette règle de priorité pour accéder à la valeur de celui-ci. Si on est dans une fonction (ou une méthode), Python va d'abord chercher l'espace de noms *local* à cette fonction. S'il ne trouve pas de nom il va ensuite chercher l'espace de noms du programme principal (ou celui du module), donc des variables *globales* s'y trouvant. S'il ne trouve pas de nom, il va chercher dans les commandes *internes* à Python (on parle des *Built-in Functions*[2] et des *Built-in Constants*[3]). Si aucun objet n'est trouvé, Python renvoie une erreur.

24.1.2 Gestion des noms dans les modules

Les modules représentent aussi un espace de noms en soi. Afin d'illustrer cela, jetons un coup d'œil à ce programme `test_var_module.py` :

```
1  import mod
2
3  i = 1000000
4  j = 2
5
6  print("Dans prog principal i:", i)
7  print("Dans prog principal j:", j)
8
9  mod.fct()
10 mod.fct2()
11
12 print("Dans prog principal i:", i)
13 print("Dans prog principal j:", j)
```

Le module `mod.py` contient les instructions suivantes :

```
1  def fct():
2      i = -27478524
3      print("Dans module, i local:", i)
4
5
6  def fct2():
7      print("Dans module, j global:", j)
8
9
10 i = 3.14
11 j = -76
```

L'exécution de `test_var_module.py` donnera :

```
$ python ./test_var_module.py
Dans prog principal i: 1000000
Dans prog principal j: 2
Dans module, i local: -27478524
Dans module, j global: -76
Dans prog principal i: 1000000
Dans prog principal j: 2
```

Lignes 3 et 4. On a bien les valeurs de `i` et `j` définies dans le programme principal de `test.py`.

Lignes 9 et 10. Lorsqu'on exécute `mod.fct()`, la valeur de `i` sera celle définie localement dans cette fonction. Lorsqu'on exécute `mod.fct2()`, la valeur de `j` sera celle définie de manière globale dans le module.

Lignes 12 et 13. De retour dans notre programme principal, les variables `i` et `j` existent toujours et n'ont pas été modifiées par l'exécution de fonctions du module `mod.py`.

2. https://docs.python.org/fr/3/library/functions.html%20comme%20par%20exemple%20%60print()%60
3. https://docs.python.org/fr/3/library/constants.html

En résumé, lorsqu'on lance une méthode d'un module, c'est l'espace de noms de celui-ci qui est utilisé. Bien sûr, toutes les variables du programme principal / fonction / méthode appelant ce module sont conservées telles quelles, et on les retrouve intactes lorsque l'exécution de la fonction du module est terminée. Un module a donc son propre espace de noms qui est bien distinct de tout programme principal / fonction / méthode appelant un composant de ce module. Enfin, les variables globales créées dans notre programme principal ne sont pas accessibles dans le module lorsque celui-ci est en exécution.

24.1.3 Gestion des noms avec les classes

On vient de voir qu'un module avait son propre espace de noms, mais qu'en est-il des classes ? En utilisant les exemples vus depuis le début de ce chapitre, vous avez certainement la réponse. Une classe possède par définition son propre espace de noms qui ne peut être en aucun cas confondu avec celui d'une fonction ou d'un programme principal. Reprenons un exemple simple :

```
class Citron:
    def __init__(self, saveur="acide", couleur="jaune"):
        self.saveur = saveur
        self.couleur = couleur
        print("Dans __init__(), vous venez de créer un citron:",
              self.affiche_attributs())

    def affiche_attributs(self):
        return f"{self.saveur}, {self.couleur}"


if __name__ == "__main__":
    saveur = "sucrée"
    couleur = "orange"
    print(f"Dans le programme principal: {saveur}, {couleur}")
    citron1 = Citron("très acide", "jaune foncé")
    print("Dans citron1.affiche_attributs():", citron1.affiche_attributs())
    print(f"Dans le programme principal: {saveur}, {couleur}")
```

Lorsqu'on exécutera ce code, on obtiendra :

```
Dans le programme principal: sucrée, orange
Dans __init__(), vous venez de créer un citron: très acide, jaune foncé
Dans citron1.affiche_attributs(): très acide, jaune foncé
Dans le programme principal: sucrée, orange
```

Les deux variables globales `saveur` et `couleur` du programme principal ne peuvent pas être confondues avec les variables d'instance portant le même nom. Au sein de la classe, on utilisera pour récupérer ces dernières `self.saveur` et `self.couleur`. À l'extérieur, on utilisera `instance.saveur` et `instance.couleur`. Il n'y a donc aucun risque de confusion possible avec les variables globales `saveur` et `couleur`, on accède à chaque variable de la classe avec un nom distinct (qu'on soit à l'intérieur ou à l'extérieur de la classe).

Ceci est également vrai pour les méthodes. Si par exemple, on a une méthode avec un certain nom, et une fonction du module principal avec le même nom, regardons ce qui se passe :

```
class Citron:
    def __init__(self):
        self.couleur = "jaune"
        self.affiche_coucou()
        affiche_coucou()

    def affiche_coucou(self):
        print("Coucou interne !")


def affiche_coucou():
    print("Coucou externe")


if __name__ == "__main__":
    citron1 = Citron()
    citron1.affiche_coucou()
    affiche_coucou()
```

Lorsqu'on va exécuter le code, on obtiendra :

```
Coucou interne !
Coucou externe
Coucou interne !
Coucou externe
```

À nouveau, il n'y a pas de conflit possible pour l'utilisation d'une méthode ou d'une fonction avec le même nom. À l'intérieur de la classe on utilise `self.affiche_coucou()` pour la méthode et `affiche_coucou()` pour la fonction. À l'extérieur de la classe, on utilise `instance.affiche_coucou()` pour la méthode et `affiche_coucou()` pour la fonction.

Dans cette rubrique, nous venons de voir une propriété des classes extrêmement puissante : **une classe crée automatiquement son propre espace de noms**. Cela permet d'encapsuler à l'intérieur tous les attributs et méthodes dont on a besoin, sans avoir aucun risque de conflit de nom avec l'extérieur (variables locales, globales ou provenant de modules). L'utilisation de classes évitera ainsi l'utilisation de variables globales qui, on l'a vu aux chapitres 10 et 13 sur les fonctions, sont à proscrire absolument. Tout cela concourt à rendre le code plus lisible.

Dans le chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne), vous verrez une démonstration de l'utilité de tout encapsuler dans une classe afin d'éviter les variables globales.

24.1.4 Gestion des noms entre les attributs de classe et d'instance

Si vous lisez cette rubrique sur l'espace de noms sans avoir lu ce chapitre depuis le début, nous vous conseillons vivement de lire attentivement la rubrique *Différence entre les attributs de classe et d'instance*. La chose importante à retenir sur cette question est la suivante : si un attribut de classe et un attribut d'instance ont le même nom, c'est l'attribut d'instance qui est **prioritaire**.

Pour aller plus loin

Il existe d'autres règles concernant les espace de noms. L'une d'elle, que vous pourriez rencontrer, concerne la gestion des noms avec des fonctions imbriquées. Et oui, Python autorise cela ! Par exemple :

```
def fonction1():
    [...]

    def fct_dans_fonction1():
        [...]
```

Là encore, il existe certaines règles de priorités d'accès aux objets spécifiques à ce genre de cas, avec l'apparition d'un nouveau mot-clé nommé `nonlocal`. Toutefois ces aspects vont au-delà du présent ouvrage. Pour plus d'informations sur les fonctions imbriquées et la directive `nonlocal`, vous pouvez consulter la documentation officielle[4].

D'autres subtilités concerneront la gestion des noms en cas de définition d'une nouvelle classe héritant d'une classe

4. https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces

mère. Ces aspects sont présentés dans la rubrique *Héritage* de ce chapitre.

24.2 Polymorphisme

Nous allons voir maintenant des propriétés très importantes des classes en Python, le polymorphisme dans cette rubrique et l'héritage dans la suivante. Ces deux concepts donnent un surplus de puissance à la POO par rapport à la programmation classique.

24.2.1 Principe

Commençons par le polymorphisme. Dans la vie, celui-ci évoque la capacité à prendre plusieurs apparences, qu'en est-il en programmation ?

Définition

En programmation, le polymorphisme est la capacité d'une fonction (ou méthode) à se comporter différemment en fonction de l'objet qui lui est passé. Une fonction donnée peut donc avoir plusieurs définitions.

Prenons un exemple concret de polymorphisme : la fonction Python `sorted()` va trier par ordre ASCII si l'argument est une chaîne de caractères, et elle va trier par ordre croissant lorsque l'argument est une liste d'entiers :

```
>>> sorted("citron")
['c', 'i', 'n', 'o', 'r', 't']
>>> sorted([1, -67, 42, 0, 81])
[-67, 0, 1, 42, 81]
```

Le polymorphisme est intimement lié au concept de *redéfinition des opérateurs* que nous avons déjà croisé à plusieurs reprises dans ce livre.

Définition

La redéfinition des opérateurs est la capacité à redéfinir le comportement d'un opérateur en fonction des opérandes utilisées (on rappelle dans l'expression `1 + 1`, `+` est l'opérateur d'addition et les deux `1` sont les opérandes).

Un exemple classique de redéfinition des opérateurs concerne l'opérateur `+`. Si les opérandes sont de type numérique, il fait une addition, si elles sont des chaînes de caractère il fait une concaténation :

```
>>> 2 + 2
4
>>> "ti" + "ti"
'titi'
```

Nous verrons dans la rubrique suivante sur *l'héritage* qu'il est également possible de redéfinir des méthodes d'une classe, c'est-à-dire leur donner une nouvelle définition.

24.2.2 Méthodes *dunder* ou magiques

Comment Python permet-il ces prouesses que sont le polymorphisme et la redéfinition des opérateurs ? Et bien, il utilise des méthodes dites *dunder* ou *magiques*.

Définition

Une méthode *dunder* (*dunder method*) est une méthode spéciale dont le nom est entouré de double *underscores*. Par exemple, la méthode `.__init__()` est une méthode *dunder*. Ces méthodes sont, la plupart du temps, destinées au fonctionnement interne de la classe. Nombre d'entre elles sont destinées à changer le comportement de fonctions ou opérateurs internes à Python avec les instances d'une classe que l'on a créée. Le mot *dunder* signifie littéralement *double underscore*. On parle aussi parfois de méthodes *magiques*.

Nous allons prendre un exemple concret. Imaginons que suite à la création d'une classe, nous souhaitions que Python affiche un message personnalisé lors de l'utilisation de la fonction `print()` avec une instance de cette classe. La méthode *dunder* qui permettra cela est nommée `.__str__()` : elle redéfinit le comportement d'une instance avec la fonction `print()`.

```
1  class CitronBasique:
2      def __init__(self, couleur="jaune", taille="standard"):
3          self.couleur = "jaune"
4          self.taille = "standard"
5
6
7  class CitronCool:
8      def __init__(self, couleur="jaune", taille="standard"):
9          self.couleur = couleur
10         self.taille = taille
11
12     def __str__(self):
13         return (f"Votre citron est de couleur {self.couleur} "
14                 f"et de taille {self.taille}")
15
16
17 if __name__ == "__main__":
18     citron1 = CitronBasique()
19     print(citron1)
20     citron2 = CitronCool("jaune foncée", "minuscule")
21     print(citron2)
```

Lignes 1 à 4. Création d'une classe `CitronBasique` dans laquelle il n'y a qu'un constructeur.

Lignes 7 à 14. Création d'une classe `CitronCool` où nous avons ajouté la nouvelle méthode `.__str__()`. Cette dernière renvoie une chaîne de caractères contenant la description de l'instance.

Lignes 18 à 21. On crée une instance de chaque classe, et on utilise la fonction `print()` pour voir leur contenu.

L'exécution de ce code affichera la sortie suivante :

```
1  <__main__.CitronBasique object at 0x7ffe23e717b8>
2  Votre citron est de couleur jaune foncée et de taille minuscule 8-)
```

L'utilisation de la fonction `print()` sur l'instance `citron1` construite à partir de la classe `CitronBasique` affiche le message abscons que nous avons déjà croisé. Par contre, pour l'instance `citron2` de la classe `CitronCool`, le texte correspond à celui retourné par la méthode *dunder* `.__str__()`. Nous avons donc redéfini comment la fonction `print()` se comportait avec une instance de la classe `CitronCool`. Notez que `str(citron2)` donnerait le même message que `print(citron2)`.

Ce mécanisme pourra être reproduit avec de très nombreux opérateurs et fonctions de bases de Python. En effet, il existe une multitude de méthodes *dunder*, en voici quelques unes :

- `.__repr__()` : redéfinit le message obtenu lorsqu'on tape le nom de l'instance dans l'interpréteur ;
- `.__add__()` : redéfinit le comportement de l'opérateur `+` ;
- `.__mul__()` : redéfinit le comportement de l'opérateur `*` ;
- `.__del__()` : redéfinit le comportement de la fonction `del`.

Si on conçoit une classe produisant des objets séquentiels (comme des listes ou des *tuples*), il existe des méthodes *dunder* telles que :

- `.__len__()` : redéfinit le comportement de la fonction `len()` ;
- `.__getitem__()` : redéfinit le comportement pour récupérer un élément ;
- `.__getslice__()` : redéfinit le comportement avec les tranches.

Certaines méthodes *dunder* font des choses assez impressionnantes. Par exemple, la méthode `.__call__()` crée des instances que l'on peut appeler comme des fonctions ! Dans cet exemple, nous allons vous montrer que l'on peut ainsi créer un moyen inattendu pour mettre à jour des attributs d'instance :

```
1  class Citronnier:
2      def __init__(self, nb_citrons, age):
3          self.nb_citrons, self.age = nb_citrons, age
4
5      def __call__(self, nb_citrons, age):
6          self.nb_citrons, self.age = nb_citrons, age
7
8      def __str__(self):
9          return (f"Ce citronnier a {self.age} ans "
10                 f"et {self.nb_citrons} citrons")
11
12
13 if __name__ == "__main__":
14     citronnier1 = Citronnier(10, 3)
15     print(citronnier1)
16     citronnier1(30, 4)
17     print(citronnier1)
```

À la ligne 16, on utilise une notation `instance(arg1, arg2)`, ce qui va automatiquement appeler la méthode *dunder* `.__call__()` qui mettra à jour les deux attributs d'instance `nbcitrons` et `age` (lignes 5 et 6). Ce code affichera la sortie suivante :

```
Ce citronnier a 3 ans et 10 citrons
Ce citronnier a 4 ans et 30 citrons
```

Pour aller plus loin

- Nous vous avons montré l'idée qu'il y avait derrière le polymorphisme, et avec cela vous avez assez pour vous jeter à l'eau et commencer à construire vos propres classes. L'apprentissage de toutes les méthodes *dunder* va bien sûr au-delà du présent ouvrage. Toutefois, si vous souhaitez aller plus loin, nous vous conseillons la page de Trey Hunner[5] qui est particulièrement complète et très bien faite. Une autre page qui a un peu vieilli mais reste intéressante est celle de Rafe Kettler[6]. Enfin, nous développons les méthodes *dunder* `.__iter__()` et `.__next__()` dans la rubrique sur les itérateurs du chapitre 26 *Remarques complémentaires*.

24.3 Héritage

24.3.1 Prise en main

L'héritage peut évoquer la capacité qu'ont nos parents à nous transmettre certains traits physiques ou de caractère (ne dit-on pas, j'ai hérité ceci ou cela de ma mère ou de mon père ?). Qu'en est-il en programmation ?

Définition

En programmation, l'héritage est la capacité d'une classe d'hériter des propriétés d'une classe pré-existante. On parle de classe mère et de classe fille. En Python, l'héritage peut être multiple lorsqu'une classe fille hérite de plusieurs classes mères.

En Python, lorsque l'on veut créer une classe héritant d'une autre classe, on ajoutera après le nom de la classe fille le nom de la ou des classe(s) mère(s) entre parenthèses :

5. https://www.pythonmorsels.com/every-dunder-method/
6. https://rszalski.github.io/magicmethods

```
class Mere1:
    # contenu de la classe mère 1


class Mere2:
    # contenu de la classe mère 2


class Fille1(Mere1):
    # contenu de la classe fille 1


class Fille2(Mere1, Mere2):
    # contenu de la classe fille 2
```

Dans cet exemple, la classe `Fille1` hérite de la classe `Mere1` et la classe `Fille2` hérite des deux classes `Mere1` et `Mere2`. Dans le cas de la classe `Fille2`, on parle d'héritage multiple. Voyons maintenant un exemple concret :

```
class Mere:
    def bonjour(self):
        return "Vous avez le bonjour de la classe mère !"


class Fille(Mere):
    def salut(self):
        return "Un salut de la classe fille !"


if __name__ == "__main__":
    fille = Fille()
    print(fille.salut())
    print(fille.bonjour())
```

Lignes 1 à 3. On définit une classe `Mere` avec une méthode `.bonjour()`.

Lignes 6 à 8. On définit une classe `Fille` qui hérite de la classe `Mere`. Cette classe fille contient une nouvelle méthode `.salut()`.

Lignes 12 à 14. Après instanciation de la classe `Fille`, on utilise la méthode `.salut()`, puis la méthode `.bonjour()` héritée de la classe mère.

Ce code affiche la sortie suivante :

```
Un salut de la classe fille !
Vous avez le bonjour de la classe mère !
```

Nous commençons à entrevoir la puissance de l'héritage. Si on possède une classe avec de nombreuses méthodes et que l'on souhaite en ajouter de nouvelles, il suffit de créer une classe fille héritant d'une classe mère.

En revenant à notre exemple, une instance de la classe `Fille` sera automatiquement une instance de la classe `Mere`. Regardons dans l'interpréteur :

```
>>> fille = Fille()
>>> isinstance(fille, Fille)
True
>>> isinstance(fille, Mere)
True
```

Si une méthode de la classe fille possède le même nom que celle de la classe mère, c'est la première qui prend la priorité. Dans ce cas, on dit que la méthode est *redéfinie* (en anglais on parle de *method overriding*), tout comme on parlait de *redéfinition des opérateurs* un peu plus haut. C'est le même mécanisme, car la redéfinition des opérateurs revient finalement à redéfinir une méthode *dunder* (comme par exemple la méthode `.__add__()` pour l'opérateur `+`).

Voyons un exemple :

```
class Mere:
    def bonjour(self):
        return "Vous avez le bonjour de la classe mère !"


class Fille2(Mere):
    def bonjour(self):
        return "Vous avez le bonjour de la classe fille !"


if __name__ == "__main__":
    fille = Fille2()
    print(fille.bonjour())
```

Ce code va afficher `Vous avez le bonjour de la classe fille !`. La méthode `.bonjour()` de la classe fille a donc pris la priorité sur celle de la classe mère. Ce comportement provient de la gestion des espaces de noms par Python, il est traité en détail dans la rubrique suivante.

Remarque

À ce point, nous pouvons faire une note de sémantique importante. Python utilise le mécanisme de *redéfinition de méthode* (*method overriding*), c'est-à-dire qu'on redéfinit une méthode héritée d'une classe mère. Il ne faut pas confondre cela avec la *surcharge de fonction* (*function overloading*) qui désigne le fait d'avoir plusieurs définitions d'une fonction selon le nombres d'arguments et/ou leur type (la surcharge n'est pas supportée par Python contrairement à d'autres langages orientés objet).

24.3.2 Ordre de résolution des noms

Vous l'avez compris, il y aura un ordre pour la résolution des noms d'attributs ou de méthodes en fonction du ou des héritage(s) de notre classe (à nouveau, cela provient de la manière dont Python gère les espaces de noms). Prenons l'exemple d'une classe déclarée comme suit `class Fille(Mere1, Mere2):`. Si on invoque un attribut ou une méthode sur une instance de cette classe, Python cherchera d'abord dans la classe `Fille`. S'il ne trouve pas, il cherchera ensuite dans la première classe mère (`Mere1` dans notre exemple). S'il ne trouve pas, il cherchera dans les ancêtres de cette première mère (si elle en a), et ce en remontant la filiation (d'abord la grand-mère, puis l'arrière grand-mère, etc). S'il n'a toujours pas trouvé, il cherchera dans la deuxième classe mère (`Mere2` dans notre exemple) puis dans tous ses ancêtres. Et ainsi de suite, s'il y a plus de deux classes mères. Bien sûr, si aucun attribut ou méthode n'est trouvé, Python renverra une erreur.

Il est en général possible d'avoir des informations sur l'ordre de résolution des méthodes d'une classe en évoquant la commande `help()` sur celle-ci ou une de ses instances. Par exemple, nous verrons dans le chapitre suivant le module *Tkinter*, imaginons que nous créions une instance de la classe principale du module *Tkinter* nommée `Tk` :

```
>>> import tkinter as tk
>>> racine = tk.Tk()
```

En invoquant la commande `help(racine)`, l'interpréteur nous montre :

```
Help on class Tk in module tkinter:

class Tk(Misc, Wm)
 |  Toplevel widget of Tk which represents mostly the main window
 |  of an application. It has an associated Tcl interpreter.
 |
 |  Method resolution order:
 |      Tk
 |      Misc
 |      Wm
 |      builtins.object
[...]
```

On voit tout de suite que la classe `Tk` hérite de deux autres classes `Misc` et `Wm`. Ensuite, le *help* indique l'ordre de résolution des méthodes : d'abord la classe `Tk` elle-même, ensuite ses deux mères `Misc` puis `Wm`, et enfin une dernière

classe nommée `builtins.object` dont nous allons voir la signification maintenant.

Remarque

En Python, il existe une classe interne nommée `object` qui est en quelque sorte la classe ancêtre de tous les objets. Toutes les classes héritent de `object`.

Pour vous en convaincre, nous pouvons recréer une classe vide :

```
>>> class Citron:
...     pass
```

Puis ensuite regarder l'aide sur l'une de ses instances :

```
Help on class Citron in module __main__:

class Citron(builtins.object)
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
[...]
```

L'aide nous montre que `Citron` a hérité de `builtins.object` bien que nous ne l'ayons pas déclaré explicitement. Cela se fait donc de manière implicite.

Remarque

Le module *builtins* possède toutes les fonctions internes à Python. Il est donc pratique pour avoir une liste de toutes ces fonctions internes en un coup d'œil. Regardons cela avec les deux instructions `import builtins` puis `dir(builtins)` :

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', [...]
'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', [...]
'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', [...]
'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

Au début, on y trouve les exceptions commençant par une lettre majuscule (voir le chapitre 26 *Remarques complémentaires* (en ligne) pour la définition d'une exception), puis les fonctions Python de base tout en minuscule. On retrouve par exemple `list` ou `str`, mais il y a aussi `object`. Toutefois ces fonctions étant chargées de base dans l'interpréteur, l'importation de `builtins` n'est pas obligatoire : par exemple `list` revient au même que `builtins.list`, ou `object` revient au même que `builtins.object`.

En résumé, la syntaxe `class Citron:` sera équivalente à
`class Citron(builtins.object):`
ou à `class Citron(object):`.

Ainsi, même si on crée une classe `Citron` vide (contenant seulement une commande `pass`), elle possède déjà tout une panoplie de méthodes héritées de la classe `object`. Regardez l'exemple suivant :

```
>>> class Citron:
...     pass
...
>>> c = Citron()
>>> dir(c)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__']
>>> o = object()
>>> dir(o)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
'__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

La quasi-totalité des attributs / méthodes de base de la classe `Citron` sont donc hérités de la classe *object*. Par exemple, lorsqu'on instancie un objet Citron `c = Citron()`, Python utilisera la méthode `.__init__()` héritée de la classe *object* (puisque nous ne l'avons pas définie dans la classe `Citron`).

24.3.3 Un exemple concret d'héritage

Nous allons maintenant prendre un exemple un peu plus conséquent pour illustrer la puissance de l'héritage en programmation. D'abord quelques mots à propos de la conception. Imaginons que nous souhaitions créer plusieurs classes correspondant à nos fruits favoris, par exemple le citron (comme par hasard !), l'orange, le kaki, etc. Chaque fruit a ses propres particularités, mais il y a aussi de nombreux points communs. Nous pourrions donc concevoir une classe `Fruit` permettant, par exemple, d'instancier un fruit et ajouter des méthodes d'affichage commune à n'importe quel fruit, et ajouter (ou toute autre méthode) pouvant être utilisée pour n'importe quel fruit. Nous pourrions alors créer des classes comme `Citron`, `Orange`, etc., héritant de la classe `Fruit` et ainsi nous économiser des lignes de code identiques à ajouter pour chaque fruit. Regardons l'exemple suivant que nous avons garni de `print()` pour bien comprendre ce qui se passe :

```
1  class Fruit:
2      def __init__(self, taille=None, masse=None, saveur=None, forme=None):
3          print("(2) Je suis dans le constructeur de la classe Fruit")
4          self.taille = taille
5          self.masse = masse
6          self.saveur = saveur
7          self.forme = forme
8          print("Je viens de créer self.taille, self.masse, self.saveur "
9                "et self.forme")
10
11     def affiche_conseil(self, type_fruit, conseil):
12         print("(2) Je suis dans la méthode .affiche_conseil() de la "
13               "classe Fruit\n")
14         return (f"Instance {type_fruit}\n"
15                 f"taille: {self.taille}, masse: {self.masse}\n"
16                 f"saveur: {self.saveur}, forme: {self.forme}\n"
17                 f"conseil: {conseil}\n")
18
19
20 class Citron(Fruit):
21     def __init__(self, taille=None, masse=None, saveur=None, forme=None):
22         print("(1) Je rentre dans le constructeur de Citron, et je vais "
23               "appeler\n"
24               "le constructeur de la classe mère Fruit !")
25         Fruit.__init__(self, taille, masse, saveur, forme)
26         print("(3) J'ai fini dans le constructeur de Citron, "
27               "les attributs sont :\n"
28               f"self.taille: {self.taille}, self.masse: {self.masse}\n"
29               f"self.saveur: {self.saveur}, self.forme: {self.forme}\n")
30
31     def __str__(self):
32         print("(1) Je rentre dans la méthode .__str__() de la classe "
33               "Citron")
34         print("Je vais lancer la méthode .affiche_conseil() héritée "
35               "de la classe Fruit")
36         return self.affiche_conseil("Citron", "Bon en tarte :-p !")
37
38
39 if __name__ == "__main__":
40     # On crée un citron.
41     citron1 = Citron(taille="petite", saveur="acide",
42                      forme="ellipsoïde", masse=50)
43     print(citron1)
```

Lignes 1 à 9. On crée la classe `Fruit` avec son constructeur qui initialisera tous les attributs d'instance décrivant le fruit.

Lignes 11 à 17. Création d'une méthode `.affiche_conseil()` qui retourne une chaîne contenant le type de fruit, les attributs d'instance du fruit, et un conseil de consommation.

Lignes 20 à 29. Création de la classe `Citron` qui hérite de la classe `Fruit`. Le constructeur de `Citron` prend les mêmes arguments que ceux du constructeur de `Fruit`. La ligne 24 est une étape importante que nous n'avons encore jamais vue : l'instruction `Fruit.__init__()` est un appel au constructeur de la classe mère (cf. explications plus bas). Notez bien que le premier argument passé au constructeur de la classe mère sera systématiquement l'instance en cours `self`. Le `print()` en lignes 26-29 illustre qu'après l'appel du constructeur de la classe mère tous les attributs d'instance (`self.taille`, `self.poids`, etc.) ont bel et bien été créés.

Lignes 31 à 36. On définit la méthode `.__str__()` qui va modifier le comportement de notre classe avec `print()`. Celle-ci fait également appel à une méthode hértitée de la classe mère nommée `.affiche_conseil()`. Comme on a l'a héritée, elle est directement accessible avec un `self.méthode()` (et de l'extérieur ce serait `instance.méthode()`).

Lignes 39 à 43. Dans le programme principal, on instancie un objet `Citron`, puis on utilise `print()` sur l'instance.

L'exécution de ce code affichera la sortie suivante :

```
(1) Je rentre dans le constructeur de Citron, et je vais appeler
le constructeur de la classe mère Fruit !
(2) Je suis dans le constructeur de la classe Fruit
Je viens de créer self.taille, self.masse, self.saveur et self.forme
(3) J'ai fini dans le constructeur de Citron, les attributs sont:
self.taille: petite, self.masse: 50
self.saveur: acide, self.forme: ellipsoïde

(1) Je rentre dans la méthode .__str__() de la classe Citron
Je vais lancer la méthode .affiche_conseil() héritée de la classe Fruit
(2) Je suis dans la méthode .affiche_conseil() de la classe Fruit

Instance Citron
taille: petite, masse: 50
saveur: acide, forme: ellipsoïde
conseil: Bon en tarte :-p !
```

Prenez bien le temps de suivre ce code pas à pas pour bien en comprendre toutes les étapes.

Vous pourrez vous poser la question « *Pourquoi utilise-t-on en ligne 24 la syntaxe* `Fruit.__init__()` *?* ». Cette syntaxe est souvent utilisée lorsqu'une classe hérite d'une autre classe pour faire appel au constructeur de la classe mère. La raison est que nous souhaitons appeler une méthode de la classe mère qui a le même nom qu'une méthode de la classe fille. Dans ce cas, si on utilisait `self.__init__()`, cela correspondrait à la fonction de notre classe fille Citron. En mettant systématiquement une syntaxe `ClasseMere.__init__()` on indique sans ambiguïté qu'on appelle le constructeur de la classe mère, en mettant explicitement son nom. Ce mécanisme est assez souvent utilisé dans le module *Tkinter* (voir le chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne)) pour la construction d'interfaces graphiques, nous en verrons de nombreux exemples.

Remarque

Si vous utilisez des ressources externes, il se peut que vous rencontriez une syntaxe `super().__init__()`. La fonction Python interne `super()` appelle automatiquement la classe mère sans que vous ayez à donner son nom. Même si cela peut paraître pratique, nous vous conseillons d'utiliser dans un premier temps la syntaxe `ClasseMere.__init__()` qui est selon nous plus lisible (on voit explicitement le nom de la classe employée, même s'il y a plusieurs classes mères).

Ce mécanisme n'est pas obligatoirement utilisé, mais il est très utile lorsqu'une classe fille a besoin d'initialiser des attributs définis dans la classe mère. On le croise donc souvent car :

- Cela donne la garantie que toutes les variables de la classe mère sont bien initialisées. On réduit ainsi les risques de dysfonctionnement des méthodes héritées de la classe mère.
- Finalement, autant ré-utiliser les « moulinettes » de la classe mère, c'est justement à ça que sert l'héritage ! Au final, on écrit moins de lignes de code.

Conseil

Pour les deux raisons citées ci-dessus, nous vous conseillons de systématiquement utiliser le constructeur de la classe mère lors de l'instanciation.

Vous avez à présent bien compris le fonctionnement du mécanisme de l'héritage. Dans notre exemple, nous pourrions créer de nouveaux fruits avec un minimum d'effort. Ceux-ci pourraient hériter de la classe mère `Fruit` à nouveau, et nous n'aurions pas à réécrire les mêmes méthodes pour chaque fruit, simplement à les appeler. Par exemple :

```
class Kaki(Fruit):
    def __init__(self, taille=None, masse=None, saveur=None, forme=None):
        Fruit.__init__(self, taille, masse, saveur, forme)

    def __str__(self):
        return Fruit.affiche_conseil(self, "Kaki",
                                     "Bon à manger cru, miam !")


class Orange(Fruit):
    def __init__(self, taille=None, masse=None, saveur=None, forme=None):
        Fruit.__init__(self, taille, masse, saveur, forme)

    def __str__(self):
        return Fruit.affiche_conseil(self, "Orange", "Trop bon en jus !")
```

Cet exemple illuste la puissance de l'héritage et du polymorphisme et la facilité avec laquelle on les utilise en Python. Pour chaque fruit, on utilise la méthode `.affiche_conseil()` définie dans la classe mère sans avoir à la réécrire. Bien sûr cet exemple reste simpliste et n'est qu'une « mise en bouche ». Vous verrez des exemples concrets de la puissance de l'héritage dans le chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne) ainsi que dans les exercices du présent chapitre. Avec le module *Tkinter*, chaque objet graphique (bouton, zone de texte, etc.) est en fait une classe. On peut ainsi créer de nouvelles classes héritant des classes *Tkinter* afin de personnaliser chaque objet graphique.

24.4 Composition

Un autre concept puissant rencontré en POO est la composition.

Définition

La composition désigne le fait qu'une classe peut contenir des instances provenant d'autres classes. On parle parfois de classe *Composite* contenant des instances d'une classe *Component* (qu'on pourrait traduire par *élément*).

Pour vous illustrer cela, nous allons prendre un exemple sur notre fruit préféré, le citron. Un citron (classe *Composite*) contient de la pulpe (classe *Component*). Voilà comment nous pourrions l'implémenter :

```
1  class Pulpe:
2      def __init__(self, quantite_jus):
3          self.quantite_jus = quantite_jus # En cL.
4
5      def __str__(self):
6          return f"Cette pulpe contient {self.quantite_jus} cL de jus"
7
8
9  class Citron:
10     def __init__(self, pulpe=None):
11         self.pulpe = pulpe
12
13     def presse_citron(self):
14         if self.pulpe:
15             print(f"Le pressage de la pulpe délivre "
16                   f"{self.pulpe.quantite_jus} cL de jus")
17             self.pulpe = None
18         else:
19             print("Il n'y a plus rien à presser dans votre citron !")
20
21     def __str__(self):
22         if self.pulpe:
23             return f"Votre citron contient {self.pulpe.quantite_jus} cL de jus"
24         else:
25             return "Ce citron ne contient pas de pulpe"
26
27
28 if __name__ == "__main__":
29     pulpe = Pulpe(10)
30     print(pulpe)
31     citron1 = Citron()
32     print(citron1)
33     print()
34     citron2 = Citron(pulpe)
35     print(citron2.pulpe)
36     print(citron2)
37     print()
38     citron2.presse_citron()
39     citron2.presse_citron()
40     print(citron2)
```

Lignes 1 à 6. On crée une classe `Pulpe` qui prend en argument à l'instanciation une quantité de jus (en cL) qu'elle peut délivrer si on la presse.

Lignes 9 à 25. On crée une classe `Citron` qui prend un objet Pulpe à l'instanciation. Si aucun objet est passé, on affecte `None`. Cette classe contient une méthode `.presse_citron()` qui pressera la pulpe pour délivrer le jus de citron. Une fois le pressage effectué, il n'y aura plus de jus à délivrer.

La sortie sera la suivante :

```
Cette pulpe contient 10 cL de jus
Ce citron ne contient pas de pulpe

Cette pulpe contient 10 cL de jus
Votre citron contient 10 cL de jus

Le pressage de la pulpe délivre 10 cL de jus
Il n'y a plus rien à presser dans votre citron !
Ce citron ne contient pas de pulpe
```

Dans cet exemple, la classe `Citron` a utilisé une instance de la classe `Pulpe` pour fonctionner. Un avantage de la composition est qu'on pourrait réutiliser cette classe `Pulpe` dans une classe `Orange` ou `Pamplemousse`. Par ailleurs, si on change des détails dans la classe `Pulpe`, cela affectera peu la classe `Citron` à partir du moment où on garde l'attribut `.quantite_jus`.

De manière générale, la composition est considérée comme plus flexible que l'héritage car les classes *Composite* et *Component* sont peu couplées. Le changement de l'une d'entre elle aura peu d'effet sur l'autre. Au contraire, pour l'héritage le changement d'une classe mère peut avoir des répercussions importantes pour les classes filles. Toutefois,

dans certains cas l'héritage peut s'avérer plus naturel. Nous vous parlions en introduction du chapitre 23 *Avoir la classe avec les objets* de l'art pour concevoir des classes interagissant harmonieusement entre elles. Et bien nous y sommes !

Si on a deux classes `A` et `B`, la relation entre elles dans l'héritage sera de type `B` **is a** `A` (avec `B` qui hérite de `A`). Dans la composition, ce sera plutôt `A` **has a** `B`. Cela peut vous servir de piste dans la conception des relations entre vos classes. A-t-il plus de sens d'y avoir une relation **is a** ou bien **has a** ? Dans le premier cas vous irez plutôt vers l'héritage, alors que dans le deuxième plutôt vers la composition. C'est ici que le langage UML [7] peut être pratique pour avoir une vision d'ensemble sur comment les classes interagissent entre elles.

Bien sûr, il faudra vous entraîner sur des cas concrets pour acquérir l'expérience qui vous mènera aux bons choix. A la fin de ce chapitre, nous vous présentons un exercice pour vous entraîner dans un premier temps à la composition. Dans le chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne), vous aurez des illustrations et des exercices sur l'héritage qui est très utilisé en *Tkinter*.

Pour aller plus loin

Nous vous conseillons ce très bon article sur le site *RealPython* qui explique de manière approfondie la problématique entre la composition et l'héritage [8].

Pour aller plus loin

Le polymorphisme, l'héritage et la composition donnent toute la puissance à la POO. Toutefois, concevoir ses classes sur un projet, surtout au début de celui-ci, n'est pas chose aisée. Nous vous conseillons de lire d'autres ressources et de vous entraîner sur un maximum d'exemples. Si vous souhaitez allez plus loin sur la POO, nous vous conseillons de lire des ressources supplémentaires. En langue française, vous trouverez les livres de Gérard Swinnen [9], Bob Cordeau et Laurent Pointal [10], Vincent Legoff [11] et Xavier Olive [12].

24.5 Différence entre les attributs de classe et d'instance

Dans cette rubrique, nous souhaitons éclairer le rôle des attributs de classe et des attributs d'instance, et comment ils sont gérés par Python.

On a vu dans le chapitre précédent comment créer un attribut de classe, il suffit de créer une variable au sein de la classe (en dehors de toute méthode). En général, les attributs de classe contiennent des propriétés générales à la classe puisqu'ils vont prendre la même valeur quelle que soit l'instance.

Au contraire, les attributs d'instance sont spécifiques à chaque instance. Pour en créer, on a vu qu'il suffisait de les initialiser dans la méthode `.__init__()` en utilisant une syntaxe `self.nouvel_attribut = valeur`. On a vu aussi dans la rubrique *Ajout d'un attribut d'instance* que l'on pouvait ajouter un attribut d'instance de l'extérieur avec une syntaxe `instance.nouvel_attribut = valeur` .

Bien que les deux types d'attributs soient fondamentalement différents au niveau de leur finalité, il existe des similitudes lorsqu'on veut accéder à leur valeur. Le code suivant illustre cela :

7. https://fr.wikipedia.org/wiki/UML_(informatique)
8. https://realpython.com/inheritance-composition-python/
9. https://inforef.be/swi/python.htm
10. https://perso.limsi.fr/pointal/python:courspython3
11. https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python
12. https://www.xoolive.org/python/

```
1  class Citron:
2      forme = "ellipsoïde" # attribut de classe
3      saveur = "acide" # attribut de classe
4
5      def __init__(self, couleur="jaune", taille="standard", masse=0):
6          self.couleur = couleur # attribut d'instance
7          self.taille = taille # attribut d'instance
8          self.masse = masse # attribut d'instance (masse en gramme)
9
10     def augmente_masse(self, valeur):
11         self.masse += valeur
12
13
14 if __name__ == "__main__":
15     citron1 = Citron()
16     print("Attributs de classe :", citron1.forme, citron1.saveur)
17     print("Attributs d'instance :", citron1.taille, citron1.couleur,
18           citron1.masse)
19     citron1.augmente_masse(100)
20     print("Attributs d'instance :", citron1.taille, citron1.couleur,
21           citron1.masse)
```

Lignes 2 et 3. Nous créons deux variables de classe qui seront communes à toutes les instances (disons qu'un citron sera toujours ellipsoïde et acide !).

Lignes 6 à 8. Nous créons trois variables d'instance qui seront spécifiques à chaque instance (disons que la taille, la couleur et la masse d'un citron peuvent varier !), avec des valeurs par défaut.

Lignes 10 et 11. On crée une nouvelle méthode `.augmente_masse()` qui augmente l'attribut d'instance `.masse`.

Ligne 14 à 21. Dans le programme principal, on instancie la classe `Citron` sans passer d'argument (les valeurs par défaut `"jaune"`, `"standard"` et `0` seront donc prises), puis on imprime les attributs.

La figure 24.1 montre l'état des variables après avoir exécuté ce code grâce au site *Python Tutor*[13].

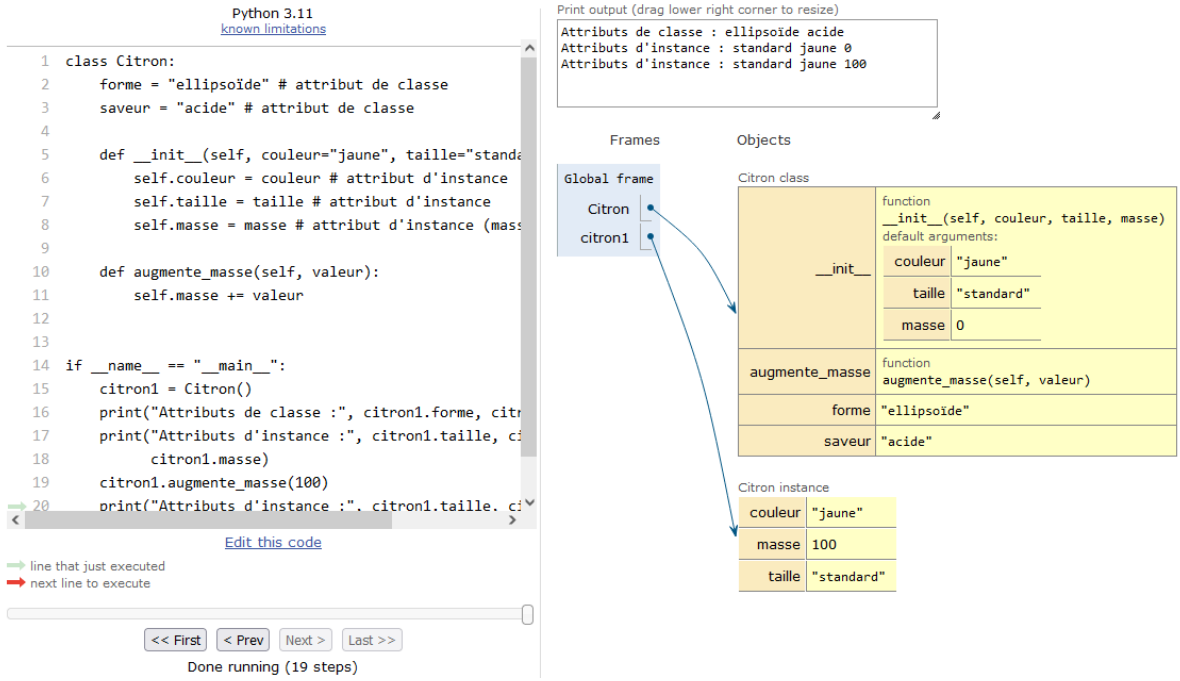


FIGURE 24.1 – Illustration de la signification des attributs de classe et d'instance avec *Python Tutor*.

Python Tutor montre bien la différence entre les variables de classe `forme` et `saveur` qui apparaissent directement dans les attributs de la classe `Citron` lors de sa définition et les trois variables d'instance `couleur`, `taille` et `masse`

13. http://www.pythontutor.com

qui sont liées à l'instance `citron1`. Pour autant, on voit dans la dernière instruction `print()` qu'on peut accéder de la même manière aux variables de classe ou d'instance, lorsqu'on est à l'extérieur, avec une syntaxe `instance.attribut`.

Au sein des méthodes, on accède également de la même manière aux attributs de classe ou d'instance, avec une syntaxe `self.attribut` :

```
1  class Citron:
2      saveur = "acide" # attribut de classe
3
4      def __init__(self, couleur="jaune"):
5          self.couleur = couleur # attribut d'instance
6
7      def affiche_attributs(self):
8          print(f"attribut de classe: {self.saveur}")
9          print(f"attribut d'instance: {self.couleur}")
10
11
12 if __name__ == "__main__":
13     citron1 = Citron()
14     citron1.affiche_attributs()
```

Ce code va afficher la phrase :

```
attribut de classe: acide
attribut d'instance: jaune
```

En résumé, qu'on ait des attributs de classe ou d'instance, on peut accéder à eux de l'extérieur par `instance.attribut` et de l'intérieur par `self.attribut`.

Qu'en est-il de la manière de modifier ces deux types d'attributs ? Les attributs d'instance peuvent se modifier sans problème de l'extérieur avec une syntaxe `instance.attribut_d_instance = nouvelle_valeur` et de l'intérieur avec une syntaxe `self.attribut_d_instance = nouvelle_valeur`. Ce n'est pas du tout le cas avec les attributs de classe.

Attention

Les attributs de classe ne peuvent pas être modifiés ni à l'extérieur d'une classe via une syntaxe `instance.attribut_de_classe = nouvelle_valeur`, ni à l'intérieur d'une classe via une syntaxe `self.attribut_de_classe = nouvelle_valeur`. Puisqu'ils sont destinés à être identiques pour toutes les instances, cela est logique de ne pas pouvoir les modifier via une instance. Les attributs de classe Python ressemblent en quelque sorte aux attributs statiques du C++.

Regardons l'exemple suivant illustrant cela :

```
1  class Citron:
2      saveur = "acide"
3
4
5  if __name__ == "__main__":
6      citron1 = Citron()
7      print(citron1.saveur)
8      citron1.saveur = "sucrée"
9      print(citron1.saveur) # on regarde ici avec Python Tutor
10     del citron1.saveur
11     print(citron1.saveur) # on regarde ici avec Python Tutor
12     del citron1.saveur
```

À la ligne 7, on pourrait penser qu'on modifie l'attribut de classe `saveur` avec une syntaxe `instance.attribut_de_classe = nouvelle_valeur`. Que se passe-t-il exactement ? La figure 24.3 nous montre l'état des variables grâce au site *Python Tutor*. Celui-ci indique que la ligne 7 a en fait créé un nouvel attribut d'instance `citron1.saveur` (contenant la valeur `sucrée`) qui est bien distinct de l'attribut de classe auquel on accédait avant par le même nom ! Tout ceci est dû à la manière dont Python gère les **espaces de noms** (voir rubrique *Espaces de noms*). Dans ce cas, l'attribut d'instance est **prioritaire** sur l'attribut de classe.

À la ligne 9, on détruit finalement l'attribut d'instance `citron1.saveur` qui contenait la valeur `sucrée`. *Python*

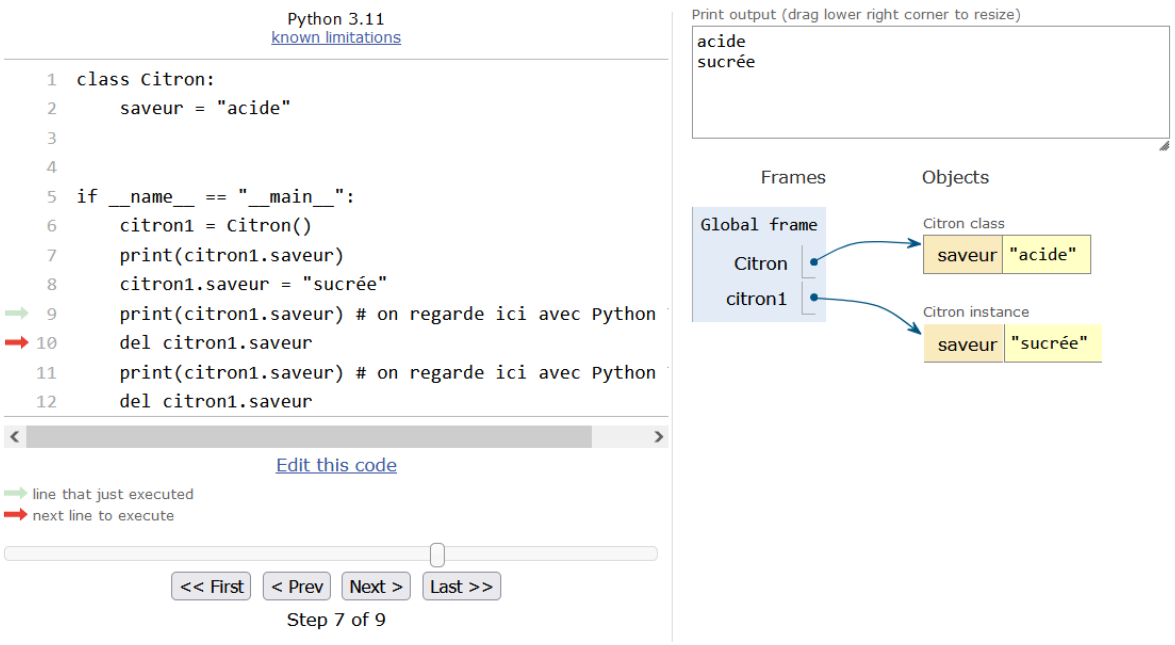


FIGURE 24.2 – Illustration avec *Python Tutor* de la non destruction d'un attribut de classe (étape 1).

Tutor nous montre que `citron1.saveur` n'existe pas dans l'espace `Citron instance` qui est vide ; ainsi, Python utilisera l'attribut de classe `.saveur` qui contient toujours la valeur `acide` (cf. figure 24.3).

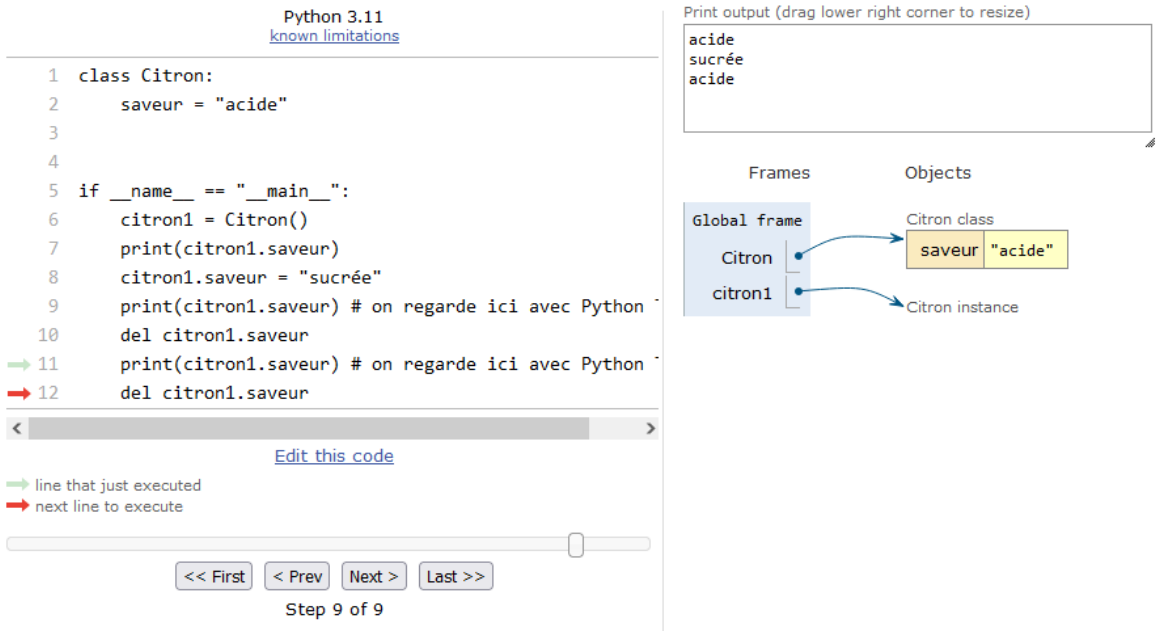


FIGURE 24.3 – Illustration avec *Python Tutor* de la non destruction d'un attribut de classe (étape 2).

La ligne 11 va tenter de détruire l'attribut de classe `.saveur`. Toutefois, Python interdit cela, ainsi l'erreur suivante sera générée :

```
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    del citron1.saveur
        ^^^^^^^^^^^^^^
AttributeError: 'Citron' object has no attribute 'saveur'
```

En fait, la seule manière de modifier un attribut de classe est d'utiliser une syntaxe `NomClasse.attribut_de_classe = nouvelle_valeur`,
dans l'exemple ci-dessus cela aurait été `Citron.saveur = "sucrée"`. De même, pour sa destruction, il faudra utiliser la même syntaxe : `del Citron.saveur`.

Conseil

Même si on peut modifier un attribut de classe, nous vous déconseillons de le faire. Une utilité des attributs de classe est par exemple de définir des constantes (mathématique ou autre), donc cela n'a pas de sens de vouloir les modifier ! Il est également déconseillé de créer des attributs de classe avec des objets modifiables comme des listes et des dictionnaires, cela peut avoir des effets désastreux non désirés. Nous verrons plus bas un exemple concret d'attribut de classe qui est très utile, à savoir le concept d'objet de type *property*.

Si vous souhaitez avoir des attributs modifiables dans votre classe, créez plutôt des attributs d'instance dans le `.__init__()`.

24.6 Accès et modifications des attributs depuis l'extérieur

24.6.1 Le problème

On a vu jusqu'à maintenant que Python était très permissif concernant le changement de valeur de n'importe quel attribut depuis l'extérieur. On a vu aussi qu'il était même possible de créer de nouveaux attributs depuis l'extérieur ! Dans d'autres langages orientés objet ceci n'est pas considéré comme une bonne pratique. Il est plutôt recommandé de définir une *interface*, c'est-à-dire tout un jeu de méthodes accédant ou modifiant les attributs. Ainsi, le concepteur de la classe a la garantie que celle-ci est utilisée correctement du « côté client ».

Remarque

Cette stratégie d'utiliser uniquement l'interface de la classe pour accéder aux attributs provient des langages orientés objet comme Java et C++. Les méthodes accédant ou modifiant les attributs s'appellent aussi des *getters* et *setters* (en français on dit accesseurs et mutateurs). Un des avantages est qu'il est ainsi possible de vérifier l'intégrité des données grâce à ces méthodes : si par exemple on souhaitait avoir un entier seulement, ou bien une valeur bornée, on peut facilement ajouter des tests dans le *setter* et renvoyer une erreur à l'utilisateur de la classe s'il n'a pas envoyé le bon type (ou la bonne valeur dans l'intervalle imposé).

Regardons à quoi pourrait ressembler une telle stratégie en Python :

```
1  class Citron:
2      def __init__(self, couleur="jaune", masse=0):
3          self.couleur = couleur
4          self.masse = masse # masse en g
5
6      def get_couleur(self):
7          return self.couleur
8
9      def set_couleur(self, value):
10         self.couleur = value
11
12     def get_masse(self):
13         return self.masse
14
15     def set_masse(self, value):
16         if value < 0:
17             raise ValueError("Z'avez déjà vu une masse négative ?")
18         self.masse = value
19
20
21 if __name__ == "__main__":
22     # définition de citron1
23     citron1 = Citron()
24     print(citron1.get_couleur(), citron1.get_masse())
25     # on change les attributs de citron1 avec les setters
26     citron1.set_couleur("jaune foncé")
27     citron1.set_masse(100)
28     print(citron1.get_couleur(), citron1.get_masse())
```

Lignes 6 à 10. On définit deux méthodes *getters* pour accéder à chaque attribut.

Lignes 12 à 18. On définit deux méthodes *setters* pour modifier chaque attribut. Notez qu'en ligne 16 nous testons si la masse est négative, si tel est le cas nous générons une erreur avec le mot-clé `raise` (voir le chapitre 26 *Remarques complémentaires* (en ligne)). Ceci représente un des avantages des *setters* : contrôler la validité des attributs (on pourrait aussi vérifier qu'il s'agit d'un entier, etc.).

Lignes 22 à 28. Après instanciation, on affiche la valeur des attributs avec les deux fonctions *getters*, puis on les modifie avec les *setters* et on les affiche à nouveau.

L'exécution de ce code donnera la sortie suivante :

```
jaune 0
jaune foncé 100
```

Si on avait mis `citron1.set_masse(-100)` en ligne 26, la sortie aurait été la suivante :

```
1  jaune 0
2  Traceback (most recent call last):
3    File "getter_setter.py", line 27, in <module>
4      citron1.set_masse(-100)
5    File "getter_setter.py", line 17, in set_masse
6      raise ValueError("Z'avez déjà vu une masse négative ???")
7  ValueError: Z'avez déjà vu une masse négative ???
```

La fonction interne `raise` nous a permis de générer une erreur, car l'utilisateur de la classe (c'est-à-dire nous dans le programme principal) n'a pas rentré une valeur correcte.

On comprend bien l'utilité d'une stratégie avec des *getters* et *setters* dans cet exemple. Toutefois, en Python, on peut très bien accéder et modifier les attributs même si on a des *getters* et des *setters* dans la classe. Imaginons la même classe `Citron` que ci-dessus, mais on utilise le programme principal suivant (notez que nous avons simplement ajouté les lignes 9 à 12 ci-dessous) :

```
if __name__ == "__main__":
    # définition de citron1
    citron1 = Citron()
    print(citron1.get_couleur(), citron1.get_masse())
    # on change les attributs de citron1 avec les setters
    citron1.set_couleur("jaune foncé")
    citron1.set_masse(100)
    print(citron1.get_couleur(), citron1.get_masse())
    # on les rechange sans les setters
    citron1.couleur = "pourpre profond"
    citron1.masse = -15
    print(citron1.get_couleur(), citron1.get_masse())
```

Cela donnera la sortie suivante :

```
jaune 0
jaune foncé 100
pourpre profond -15
```

Malgré la présence des *getters* et des *setters*, nous avons réussi à accéder et à modifier la valeur des attributs. De plus, nous avons pu mettre une valeur aberrante (masse négative) sans que cela ne génère une erreur !

Vous vous posez sans doute la question : mais dans ce cas, quel est l'intérêt de mettre des *getters* et des *setters* en Python ? La réponse est très simple : cette stratégie n'est pas une manière « pythonique » d'opérer (voir le chapitre 16 *Bonnes pratiques en programmation Python* pour la définition de « pythonique »). En Python, la lisibilité est la priorité. Souvenez-vous du Zen de Python « *Readability counts* » (voir le chapitre 16).

De manière générale, une syntaxe avec des *getters* et *setters* du côté client surcharge la lecture. Imaginons que l'on ait une instance nommée `obj` et que l'on souhaite faire la somme de ses trois attributs `x`, `y` et `z` :

```
# pythonique
obj.x + obj.y + obj.z

# non pythonique
obj.get_x() + obj.get_y() + obj.get_z()
```

La méthode pythonique est plus « douce » à lire, on parle aussi de *syntactic sugar* ou littéralement en français « *sucre syntaxique* ». De plus, à l'intérieur de la classe, il faut définir un *getter* et un *setter* pour chaque attribut, ce qui multiple les lignes de code.

Très bien. Donc en Python, on n'utilise pas comme dans les autres langages orientés objet les *getters* et les *setters* ? Mais, tout de même, cela avait l'air une bonne idée de pouvoir contrôler comment un utilisateur de la classe interagit avec certains attributs (par exemple, rentre-t-il une bonne valeur ?). N'existe-t-il pas un moyen de faire ça en Python ? La réponse est : bien sûr il existe un moyen pythonique, la classe `property`. Nous allons voir cette nouvelle classe dans la prochaine rubrique et nous vous dirons comment opérer systématiquement pour accéder, modifier, voire détruire, chaque attribut d'instance de votre classe.

24.6.2 La solution : la classe `property`

Dans la rubrique précédente, on vient de voir que les *getters* et *setters* traditionnels rencontrés dans d'autres langages orientés objet ne représentent pas une pratique pythonique. En Python, pour des raisons de lisibilité, il faudra dans la mesure du possible conserver une syntaxe `instance.attribut` pour l'accès aux attributs d'instance, et une syntaxe `instance.attribut = nouvelle_valeur` pour les modifier.

Toutefois, si on souhaite contrôler l'accès, la modification (voire la destruction) de certains attributs stratégiques, Python met en place une classe nommée `property`. Celle-ci permet de combiner le maintien de la syntaxe lisible `instance.attribut`, tout en utilisant en filigrane des fonctions pour accéder, modifier, voire détruire l'attribut (à l'image des *getters* et *setters* évoqués ci-dessus, ainsi que des *deleters* ou encore destructeurs en français). Pour faire cela, on utilise la fonction Python interne `property()` qui crée un objet (ou instance) `property` :

```
attribut = property(fget=accesseur, fset=mutateur, fdel=destructeur)
```

Les arguments passés à `property()` sont systématiquement des méthodes dites *callback*, c'est-à-dire des noms de méthodes que l'on a définies précédemment dans notre classe, mais on ne précise ni argument, ni parenthèse, ni `self`

(voir le chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne)). Avec cette ligne de code, `attribut` est un objet de type *property* qui fonctionne de la manière suivante à l'extérieur de la classe :

- L'instruction `instance.attribut` appellera la méthode `.accesseur()`.
- L'instruction `instance.attribut = valeur` appellera la méthode `.mutateur()`.
- L'instruction `del instance.attribut` appellera la méthode `.destructeur()`.

L'objet `attribut` est de type *property*, et la vraie valeur de l'attribut est stockée par Python dans une variable d'instance qui s'appellera par exemple `_attribut` (même nom, mais commençant par un *underscore* unique, envoyant un message à l'utilisateur qu'il s'agit d'une variable associée au comportement interne de la classe).

Comment cela fonctionne-t-il concrètement dans un code ? Regardons cet exemple (nous avons mis des `print()` un peu partout pour bien comprendre ce qui se passe) :

```
1  class Citron:
2      def __init__(self, masse=0):
3          print("(2) J'arrive dans le .__init__()")
4          self.masse = masse
5
6      def get_masse(self):
7          print("Coucou je suis dans le get")
8          return self._masse
9
10     def set_masse(self, valeur):
11         print("Coucou je suis dans le set")
12         if valeur < 0:
13             raise ValueError("Un citron ne peut pas avoir"
14                              " de masse négative !")
15         self._masse = valeur
16
17     masse = property(fget=get_masse, fset=set_masse)
18
19
20 if __name__ == "__main__":
21      print("(1) Je suis dans le programme principal, "
22            "je vais instancier un Citron")
23      citron = Citron(masse=100)
24      print("(3) Je reviens dans le programme principal")
25      print(f"La masse de notre citron est {citron.masse} g")
26      # On mange le citron.
27      citron.masse = 25
28      print(f"La masse de notre citron est {citron.masse} g")
29      print(citron.__dict__)
```

Pour une fois, nous allons commenter les lignes dans le désordre :

Ligne 17. Il s'agit de la commande clé pour mettre en place le système : `masse` devient ici un objet de type *property* (si on regarde son contenu avec une syntaxe `NomClasse.attribut_property`, donc ici `Citron.masse`, Python nous renverra quelque chose de ce style : `<property object at 0x7fd3615aeef8>`). Qu'est-ce que cela signifie ? Et bien la prochaine fois qu'on voudra accéder au contenu de cet attribut `.masse`, Python appellera la méthode `.get_masse()`, et quand on voudra le modifier, Python appellera la méthode `.set_masse()` (ceci sera valable de l'intérieur ou de l'extérieur de la classe). Comme il n'y a pas de méthode destructeur (passée avec l'argument `fdel`), on ne pourra pas détruire cet attribut : un `del c.masse` conduirait à une erreur de ce type : `AttributeError: can't delete attribute`.

Ligne 4. Si vous avez bien suivi, cette commande `self.masse = masse` dans le constructeur va appeler automatiquement la méthode `.set_masse()`. Attention, dans cette commande, la variable `masse` à droite du signe `=` est une variable *locale* passée en argument. Par contre, `self.masse` sera l'objet de type *property*. Si vous avez bien lu la rubrique *Différence entre les attributs de classe et d'instance*, l'objet `masse` créé en ligne 16 est un attribut de classe, on peut donc y accéder avec une syntaxe `self.masse` au sein d'une méthode.

Conseil

Notez bien l'utilisation de `self.masse` dans le constructeur (en ligne 4) plutôt que `self._masse`. Comme `self.masse` appelle la méthode `.set_masse()`, cela permet de contrôler si la valeur est correcte dès l'instanciation. C'est

donc une pratique que nous vous recommandons. Si on avait utilisé `self._masse`, il n'y aurait pas eu d'appel à la fonction mutateur et on aurait pu mettre n'importe quoi, y compris une valeur aberrante, lors de l'instanciation.

Lignes 6 à 15. Dans les méthodes accesseur et mutateur, on utilise la variable `self._masse` qui contiendra la vraie valeur de la masse du citron (cela serait vrai pour tout autre objet de type *property*).

Attention

Dans les méthodes accesseur et mutateur, il ne faut surtout pas utiliser `self.masse` à la place de `self._masse`. Pourquoi ? Par exemple, dans l'accesseur, si on met `self.masse` cela signifie que l'on souhaite accéder à la valeur de l'attribut (comme dans le constructeur !). Ainsi, Python rappellera l'accesseur et retombera sur `self.masse`, ce qui rappellera l'accesseur et ainsi de suite : vous l'aurez compris, cela partira dans une récursion infinie et mènera à une erreur du type `RecursionError: maximum recursion depth exceeded`. Cela serait vrai aussi si vous aviez une fonction destructeur, il faudrait utiliser `self._masse`.

L'exécution de ce code donnera :

```
(1) Dans le programme principal, je vais instancier un Citron
(2) J'arrive dans le .__init__()
Coucou je suis dans le set
(3) Je reviens dans le programme principal
Coucou je suis dans le get
La masse de notre citron est 100 g
Coucou je suis dans le set
Coucou je suis dans le get
La masse de notre citron est 25 g
{'_masse': 25}
```

Cette exécution montre qu'à chaque appel de `self.masse` ou `citron.masse` on va utiliser les méthodes accesseur ou mutateur. La dernière commande qui affiche le contenu de `citron.__dict__` montre que la vraie valeur de l'attribut est stockée dans la variable d'instance `._masse` (`instance._masse` de l'extérieur et `self._masse` de l'intérieur).

24.6.3 Une meilleure solution : les décorateurs `@property`, `@attribut.setter` et `@attribut.deleter`

Nous venons de voir les objets *property* pour contrôler l'accès, la mutation et la supression d'attributs. Toutefois la syntaxe est relativement lourde. Afin de la simplifier, une manière plus pythonique (sucre syntaxique) est d'utiliser un décorateur. La syntaxe pour décorer une fonction est la suivante :

```
1 @decorateur
2 def fonction():
3     [...]
```

La ligne 1 précise que `fonction()` va être modifiée par une autre fonction nommée `decorateur()`. Le symbole `@` en ligne 1 attend un nom de fonction qui sera la fonction décoratrice. Pour plus de détails sur comment les décorateurs fonctionnent, vous pouvez consulter le chapitre 26 *Remarques complémentaires* où une rubrique leur est consacrée. Ici, nous avons juste à savoir qu'un décorateur est une fonction qui modifie le comportement d'une autre fonction.

En reprenant l'exemple vu dans la rubrique précédente, voici comment on peut l'écrire avec des décorateurs :

```
1  class Citron:
2      def __init__(self, masse=0):
3          print(f"(2) J'arrive dans le .__init__(), je vais mettre la masse = {masse}")
4          self.masse = masse
5
6      @property
7      def masse(self):
8          print("Coucou je suis dans le getter")
9          return self._masse
10
11     @masse.setter
12     def masse(self, valeur):
13         print("Coucou je suis dans le setter")
14         if valeur < 0:
15             raise ValueError("Un citron ne peut pas avoir"
16                              " de masse négative !")
17         self._masse = valeur
18
19     @masse.deleter
20     def masse(self):
21         print("Coucou, je suis dans le deleter")
22         del self._masse
```

On voit que la syntaxe est plus lisible que celle de la rubrique précédente. Examinons les différences. La première chose est que les méthodes *getter* (ligne 7), *setter* (ligne 11) et *deleter* (ligne 19) s'appellent toutes `.masse()`, `masse` étant le nom de notre objet *property*. Comme dans la syntaxe de la rubrique précédente, la masse réelle se trouve dans un attribut nommée `._masse` pour ne pas confondre avec notre objet *property*. Afin de comprendre ce qu'il se passe, nous vous avons concocté le programme principal suivant avec des `print()` un peu partout :

```
1  if __name__ == "__main__":
2      print("(1) Je suis dans le programme principal et "
3            "je vais instancier un Citron")
4      print()
5      citron = Citron(masse=100)
6      print()
7      print("(3) Je reviens dans le programme principal, je vais afficher "
8            "la masse du citron")
9      print(f"La masse de notre citron est {citron.masse} g")
10     print()
11     # On mange une partie du citron.
12     print("(4)Je suis dans le prog principal "
13           "et je vais changer la masse du citron")
14     citron.masse = 25
15     print()
16     print(f"(5) Je suis dans le prog principal, je vais afficher "
17           "la masse du citron")
18     print(f"La nouvelle masse de notre citron est {citron.masse} g")
19     print(f"L'attribut citron.__dict__ m'indique bien le nom réel "
20           f"de l'attribut contenant la masse :")
21     print(citron.__dict__)
22     print()
23     # On mange la fin du citron.
24     print(f"(6)  Je suis dans le prog principal, "
25           f"je détruis l'attribut .masse")
26     del citron.masse
27     print(f"Ainsi, citron.__dict__ est maintenant vide :")
28     print(citron.__dict__)
```

L'exécution donnera la sortie suivante :

```
(1) Je suis dans le programme principal et je vais instancier un Citron

(2) J'arrive dans le .__init__(), je vais mettre la masse = 100
Coucou je suis dans le setter

(3) Je reviens dans le programme principal, je vais afficher la masse du citron
Coucou je suis dans le getter
La masse de notre citron est 100 g

(4)Je suis dans le prog principal et je vais changer la masse du citron
Coucou je suis dans le setter

(5) Je suis dans le prog principal, je vais afficher la masse du citron
Coucou je suis dans le getter
La nouvelle masse de notre citron est 25 g
L'attribut citron.__dict__ m'indique bien le nom réel de l'attribut contenant la masse :
{'_masse': 25}

(6)  Je suis dans le prog principal, je détruis l'attribut .masse
Coucou, je suis dans le deleter
Ainsi, citron.__dict__ est maintenant vide :
{}
```

Examinez bien les phrases `Coucou je suis dans [...]` et essayez de comprendre pourquoi elles apparaissent. Bien que nos trois méthodes soient définies comme `def masse()`, vous pouvez constater qu'elles sont appelées lorsque on invoque `citron.masse`, `citron.masse = 25` ou `del citron.masse` (à l'intérieur de la classe, ce serait `self.masse`, `self.masse = 25` ou `del self.masse`). Autrement dit, on n'utilise jamais la syntaxe `.masse()`. Ceci est justement dû au fait que `.masse` est un objet de type *property*.

Conseil

Lorsque vous souhaitez créer des objets *property* , nous vous conseillons la syntaxe pythonique `@property`,`@nom_attribut .setter` et `@nom_attribut.deleter` plutôt que celle de la rubrique précédente avec la ligne `masse = property( fget=get_masse, fset=set_masse, fdel=del_masse)`. Cette syntaxe améliore grandement la lisibilité.

24.6.4 Le décorateur `@property` seul

Une méthode décorée avec `@property` peut être utile seule sans avoir le *setter* et/ou le *deleter* correspondant(s). On rencontre cela lorsqu'on souhaite créer un « d'attribut dynamique » plutôt qu'avoir un appel de méthode explicite. Regardons un exemple :

```
1  class ADN:
2      def __init__(self):
3          self.sequence = []
4
5      def __repr__(self):
6          return f"La séquence de mon brin d'ADN est {self.sequence}"
7
8      def ajoute_base(self, nom_base):
9          self.sequence.append(nom_base)
10
11     @property
12     def len(self):
13         return len(self.sequence)
```

Voici un dans l'interpréteur :

```
>>> brin_adn = ADN()
>>> brin_adn.ajoute_base("A")
>>> brin_adn.ajoute_base("T")
>>> brin_adn
La séquence de mon brin d'ADN est ['A', 'T']
>>> brin_adn.len
2
>>> brin_adn.ajoute_atome("G")
>>> brin_adn
La séquence de mon brin d'ADN est ['A', 'T', 'G']
>>> brin_adn.len
3
```

Lorsqu'on utilise l'attribut `brin_adn.len`, ceci invoque finalement l'appel de l'objet *property* `len` qui, *in fine*, est une méthode. Ainsi, la valeur renvoyée sera calculée à chaque fois, bien que dans la syntaxe on n'a pas une notation `.methode()`, mais plutôt `.attribut`. Voilà pourquoi nous avons parlé d'attribut dynamique. Cela permet d'alléger la syntaxe quand il n'y a pas spécifiquement d'arguments à passer à la méthode qui se trouve derrière cet attribut.

24.7 Bonnes pratiques pour construire et manipuler ses classes

Nous allons voir dans cette rubrique certaines pratiques que nous vous recommandons lorsque vous construisez vos propres classes.

24.7.1 L'accès aux attributs

On a vu dans la rubrique *Accès et modifications des attributs depuis l'extérieur* que nous avions le moyen de contrôler cet accès avec la classe *property*. Toutefois, cela peut parfois alourdir inutilement le code, ce qui va à l'encontre de certains préceptes de la PEP 20 comme « *Sparse is better than dense* », « *Readability counts* », etc. (voir le chapitre 16 *Bonnes pratiques en programmation Python*).

Conseil

Si on souhaite contrôler ce que fait le client de la classe pour certains attributs « délicats » ou « stratégiques », on peut utiliser la classe *property*. Toutefois, nous vous conseillons de ne l'utiliser que lorsque cela se révèle vraiment nécessaire, donc avec parcimonie. Le but étant de ne pas surcharger le code inutilement. Cela va dans le sens des recommandations des développeurs de Python (comme décrit dans la PEP8).

Les objets *property* ont deux avantages principaux :

- ils permettent de garder une lisibilité du côté client avec une syntaxe `instance.attribut` ;
- même si un jour vous décidez de modifier votre classe et de mettre en place un contrôle d'accès à certains attributs avec des objets *property*, cela ne changera rien du côté client. Ce dernier utilisera toujours `instance.attribut` ou `instance.attribut = valeur`. Tout cela contribuera à une meilleure maintenance du code client utilisant votre classe.

Certains détracteurs disent qu'il est parfois difficile de déterminer qu'un attribut est contrôlé avec un objet *property*. La réponse à cela est simple, dites-le clairement dans la documentation de votre classe via les *docstrings* (voir la rubrique ci-dessous).

24.7.2 Note sur les attributs publics et non publics

Certains langages orientés objet mettent en place des attributs dits *privés* dont l'accès est impossible de l'extérieur de la classe. Ceux-ci existent afin d'éviter qu'un client n'aille perturber ou casser quelque chose dans la classe. Les arguments auxquels l'utilisateur a accès sont dits *publics*.

Attention

En Python, il n'existe pas d'attributs privés comme dans d'autres langages orientés objet. L'utilisateur a accès à tous les attributs quels qu'ils soient, même s'ils contiennent un ou plusieurs caractère(s) *underscore(s)* (voir ci-dessous) !

Au lieu de ça, on parle en Python d'attributs publics et *non publics*.

Définition

En Python les attributs non publics sont des attributs dont le nom commence par un ou deux caractère(s) *underscore*. Par exemple, `_attribut`, ou `__attribut`.

La présence des *underscores* dans les noms d'attributs est un signe clair que le client ne doit pas y toucher. Toutefois, cela n'est qu'une convention, et comme dit ci-dessus le client peut tout de même modifier ces attributs.

Par exemple, reprenons la classe `Citron` de la rubrique précédente dont l'attribut `.masse` est contrôlé avec un objet *property* :

```
>>> citron = Citron()
Coucou je suis dans le set
>>> citron.masse
Coucou je suis dans le get
0
>>> citron.masse = -16
Coucou je suis dans le set
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 11, in set_masse
ValueError: Un citron ne peut pas avoir de masse négative !
>>> citron.masse = 16
Coucou je suis dans le set
>>> citron.masse
Coucou je suis dans le get
16
>>> citron._masse
16
>>> citron._masse = -8364
>>> citron.masse
Coucou je suis dans le get
-8364
>>>
```

Malgré l'objet `property`, nous avons pu modifier l'attribut non public `._masse` directement !

Il existe également des attributs dont le nom commence par deux caractères *underscores*. Nous n'avons encore jamais croisé ce genre d'attribut. Ces derniers mettent en place le *name mangling*.

Définition

Le *name mangling*[14], ou encore substantypage ou déformation de nom en français, correspond à un mécanisme de changement du nom d'un attribut selon si on est à l'intérieur ou à l'extérieur d'une classe.

Regardons un exemple :

14. https://en.wikipedia.org/wiki/Name_mangling

```
class Citron:
    def __init__(self):
        self.__mass = 100

    def get_mass(self):
        return self.__mass


if __name__ == "__main__":
    citron1 = Citron()
    print(citron1.get_mass())
    print(citron1.__mass)
```

Ce code va donner la sortie suivante :

```
100
Traceback (most recent call last):
  File "mangling.py", line 12, in <module>
    print(citron1.__mass)
          ^^^^^^^^^^^^^^
AttributeError: 'Citron' object has no attribute '__mass'
```

La ligne 12 du code a donc conduit à une erreur : Python prétend ne pas connaître l'attribut `.__mass`. On pourrait croire que cela constitue un mécanisme de protection des attributs. En fait il n'en est rien, car on va voir que l'attribut est toujours accessible et modifiable. Si on modifiait le programme principal comme suit :

```
if __name__ == "__main__":
    citron1 = Citron()
    print(citron1.__dict__)
```

On obtiendrait en sortie le dictionnaire `{'_Citron__mass': 100}`.

Le *name mangling* est donc un mécanisme qui transforme le nom `self.__attribut` à l'intérieur de la classe en `instance._NomClasse__attribut` à l'extérieur de la classe. Ce mécanisme a été conçu initialement pour pouvoir retrouver des noms d'attributs identiques lors de l'héritage. Si par exemple une classe mère et une classe fille ont chacune un attribut nommé `__attribut`, le *name mangling* permet d'éviter les conflits de nom. Par exemple :

```
class Fruit:
    def __init__(self):
        self.__mass = 100


class Citron(Fruit):
    def __init__(self):
        Fruit.__init__(self)
        self.__mass = 200

    def print_masse(self):
        print(self._Fruit__mass)
        print(self.__mass)


if __name__ == "__main__":
    citron1 = Citron()
    citron1.print_masse()
```

Ce code affiche `100` puis `200`. La ligne 12 a permis d'accéder à l'attribut `.__mass` de la classe mère `Fruit`, et la ligne 13 a permis d'accéder à l'attribut `.__mass` de la classe `Citron`.

Le *name mangling* n'est donc pas un mécanisme de « protection » d'un attribut, il n'a pas été conçu pour ça. Les concepteurs de Python le disent clairement dans la PEP 8 : « *Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed* ».

Donc en Python, on peut tout détruire, même les attributs délicats contenant des *underscores*. Pourquoi Python permet-il un tel paradoxe ? Selon le concepteur de Python, Guido van Rossum : « *We're all consenting adults here* », nous sommes ici entre adultes, autrement dit nous savons ce que nous faisons !

Conseil

En résumé, n'essayez pas de mettre des barrières inutiles vers vos attributs. Cela va à l'encontre de la philosophie Python. Soignez plutôt la documentation et faites confiance aux utilisateurs de votre classe !

24.7.3 Classes et *docstrings*

Les classes peuvent bien sûr contenir des *docstrings* comme les fonctions et les modules. C'est d'ailleurs une pratique vivement recommandée. Voici un exemple sur notre désormais familière classe `Citron` :

```
class Citron:
    """Voici la classe Citron.

    Il s'agit d'une classe assez impressionnante qui crée des objets
    citrons.
    Par défaut une instance de Citron contient l'attribut de classe
    saveur.
    """
    saveur = "acide"

    def __init__(self, couleur="jaune", taille="standard"):
        """Constructeur de la classe Citron.

        Ce constructeur prend deux arguments par mot-clé
        couleur et taille."""
        self.couleur = couleur
        self.taille = taille

    def __str__(self):
        """Redéfinit le comportement avec print()."""
        return f"saveur: {saveur}, couleur: {couleur}, taille: {taille}"

    def affiche_coucou(self):
        """Méthode inutile qui affiche coucou."""
        print("Coucou !")
```

Si on fait `help(Citron)` dans l'interpréteur, on obtient :

```
Help on class Citron in module __main__:

class Citron(builtins.object)
 |  Citron(couleur='jaune', taille='standard')
 |
 |  Voici la classe Citron.
 |
 |  Il s'agit d'une classe assez impressionnante qui crée des objets
 |  citrons.
 |  Par défaut une instance de Citron contient l'attribut de classe
 |  saveur.
 |
 |  Methods defined here:
 |
 |  __init__(self, couleur='jaune', taille='standard')
 |      Constructeur de la classe Citron.
 |
 |      Ce constructeur prend deux arguments par mot-clé
 |      couleur et taille.
 |
 |  __str__(self)
 |      Redéfinit le comportement avec print().
 |
 |  affiche_coucou(self)
 |      Méthode inutile qui affiche coucou.
 |
[...]
 |
 |  Data and other attributes defined here:
 |
 |  saveur = 'acide'
```

Python formate automatiquement l'aide comme il le fait avec les modules (voir chapitre 15 *Création de modules*). Comme nous l'avons dit dans le chapitre 16 *Bonnes pratiques en programmation Python*, n'oubliez pas que les *docstrings* sont destinées aux utilisateurs de votre classe. Elles doivent donc contenir tout ce dont un utilisateur a besoin pour comprendre ce que fait la classe et comment l'utiliser.

Notez que si on instancie la classe `citron1 = Citron()` et qu'on invoque l'aide sur l'instance `help(citron1)`, on obtient la même page d'aide. Comme pour les modules, si on invoque l'aide pour une méthode de la classe `help(citron1.affiche_coucou)`, on obtient l'aide pour cette méthode seulement.

Toutes les *docstrings* d'une classe sont en fait stockées dans un attribut spécial nommé `instance.__doc__`. Cet attribut est une chaîne de caractères contenant la *docstring* générale de la classe. Ceci est également vrai pour les modules, méthodes et fonctions. Si on reprend notre exemple ci-dessus :

```
>>> citron1 = Citron()
>>> print(citron1.__doc__)
Voici la classe Citron.

    Il s'agit d'une classe assez impressionnante qui crée des objets
    citrons.
    Par défaut une instance de Citron contient l'attribut de classe
    saveur.

>>> print(citron1.affiche_coucou.__doc__)
Méthode inutile qui affiche coucou.
```

L'attribut `.__doc__` est automatiquement créé par Python au moment de la mise en mémoire de la classe (ou module, méthode, fonction, etc.).

24.7.4 Autres bonnes pratiques

Voici quelques points en vrac auxquels nous vous conseillons de faire attention :

- Une classe ne se conçoit pas sans méthode. Si on a besoin d'une structure de données séquentielles ou si on veut donner des noms aux variables (plutôt qu'un indice), utilisez plutôt les dictionnaires. Une bonne alternative peut être les *namedtuples* (voir la rubrique suivante).

- Nous vous déconseillons de mettre comme paramètre par défaut une liste vide (ou tout autre objet séquentiel modifiable) :

```
1 def __init__(self, liste=[]):
2     self.liste = liste
```

  Si vous créez des instances sans passer d'argument lors de l'instanciation, toutes ces instances pointeront vers la même liste. Cela peut avoir des effets désastreux.
- Ne mettez pas non plus une liste vide (ou tout autre objet séquentiel modifiable) comme attribut de classe.

```
1 class Citron:
2     liste = []
```

Ici chaque instance pourra modifier la liste, ce qui n'est pas souhaitable. Souvenez-vous, la modification des attributs de classe doit se faire par une syntaxe `Citron.attribut = valeur` (et non pas via les instances).

- Comme abordé dans la rubrique *Différence entre les attributs de classe et d'instance*, nous vous conseillons de ne jamais modifier les attributs de classe. Vous pouvez néanmoins les utiliser comme constantes.
- Si vous avez besoin d'attributs modifiables, utilisez des attributs d'instance et initialisez-les dans la méthode `.__init__()` (et nulle part ailleurs). Par exemple, si vous avez besoin d'une liste comme attribut, créez la plutôt dans le constructeur :

```
1 class Citron:
2 def __init__(self):
3     self.liste = []
```

Ainsi, vous aurez des listes réellement indépendantes pour chaque instance.

24.7.5 *Namedtuples*

Imaginons que l'on souhaite stocker des éléments dans un conteneur, que l'on puisse retrouver ces éléments avec une syntaxe `conteneur.element` et que ces éléments soient non modifiables. On a vu ci-dessus, les classes ne sont pas faites pour cela, il n'est pas conseillé de les utiliser comme des conteneurs inertes, on les conçoit en général afin d'y créer aussi des méthodes. Dans ce cas, les *namedtuples*[15] sont faits pour vous ! Ce type de conteneur est issu du module *collections* que nous avions évoqué dans le chapitre 14 *Conteneurs*.

```
1  >>> import collections
2  >>> Citron = collections.namedtuple("Citron", "masse couleur saveur forme")
3  >>> Citron
4  <class '__main__.Citron'>
5  >>> citron = Citron(10, "jaune", "acide", "ellipsoide")
6  >>> citron
7  Citron(masse=10, couleur='jaune', saveur='acide', forme='ellipsoide')
8  >>> citron.masse
9  10
10 >>> citron.forme
11 'ellipsoide'
```

Lignes 2 à 4. La fonction `namedtuple()` renvoie une classe qui sert à créer de nouveaux objets citrons. Attention cette classe est différente de celles que l'on a rencontrées jusqu'à maintenant, car elle hérite de la classe `builtins.tuple` (on peut le voir en faisant `help(Citron)`). En ligne 2, on passe en argument le nom de la classe souhaitée (i.e. `Citron`), puis une chaîne de caractères avec des mots séparés par des espaces qui correspondront aux attributs (on pourrait aussi passer une liste `["masse", "couleur", "saveur", "forme"]`).

Ligne 5. On instancie un nouvel objet `citron`.

Lignes 6 à 11. On peut retrouver les différents attributs avec une syntaxe `instance.attribut`.

Mais dans *namedtuple*, il y a *tuple* ! Ainsi, l'instance `citron` hérite de tous les attributs des tuples :

15. https://docs.python.org/fr/3/library/collections.html#collections.namedtuple

```
>>> citron[0]
10
>>> citron[3]
'ellipsoide'
>>> citron.masse = 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> for elt in citron:
...     print(elt)
...
10
jaune
acide
ellipsoide
```

Lignes 1 à 4. On peut retrouver les attributs également par indice.

Lignes 5 à 8. Les attributs / éléments sont non modifiables !

Lignes 9 à 15. Les *namedtuples* sont itérables.

Un *namedtuple* est non modifiable, mais on peut en générer un nouveau avec la méthode `._replace()`, à l'image de la méthode `.replace()` pour les chaînes de caractères :

```
>>> citron._replace(masse=30)
Citron(masse=30, couleur='jaune', saveur='acide', forme='ellipsoide')
>>> citron
Citron(masse=10, couleur='jaune', saveur='acide', forme='ellipsoide')
>>> citron = citron._replace(masse=30)
>>> citron
Citron(masse=30, couleur='jaune', saveur='acide', forme='ellipsoide')
```

Lignes 1 et 2. On crée un nouveau *namedtuples* avec la méthode `._replace()`. Notez qu'il faut passer un (ou plusieurs) argument(s) par mot-clé à cette méthode désignant les attributs à modifier.

Lignes 3 et 4. L'objet initial `citron` est intact puisqu'un *namedtuples* est non modifiable.

Lignes 5 à 7. En ré-affectant ce que renvoie la méthode `._replace()` dans dans un objet de même nom `citron`, on peut faire évoluer son contenu comme on a pu le faire avec les chaînes de caractères.

Enfin, il est possible de convertir un *namedtuple* en dictionnaire (ordonné) avec la méthode `._asdict()` :

```
>>> citron._asdict()
OrderedDict([('masse', 10), ('couleur', 'jaune'), ('saveur', 'acide'), ('forme', 'ellipsoide')])
```

Quand utiliser les *namedtuples* ? Vous souvenez-vous de la différence entre les listes et les dictionnaires ? Ici, c'est un peu la même chose entre les *tuples* et les *namedtuples*. Les *namedtuples* permettent de créer un code plus lisible en remplaçant des numéros d'indice par des noms. Le fait qu'ils soient non modifiables peut aussi avoir un avantage par rapport à l'intégrité des données. Si vous trouvez les *namedtuples* limités, sachez que vous pouvez créer votre propre classe qui hérite d'un *namedtuple* afin de lui ajouter de nouvelles méthodes « maison ».

Pour aller plus loin

Pour aller plus loin, vous pouvez consulter le très bon article[16] de Dan Bader.

24.8 Note finale de sémantique

Jusqu'à présent, lorsque nous avons évoqué les outils pour créer ou convertir des objets Python tels que `int()`, `list()`, `range()`, etc., nous avons toujours parlé de fonctions. Ceci parce-que nous avions une syntaxe `fonction()`, c'est-à-dire `fonction` suivie de parenthèses `()`. Toutefois, vous vous êtes peut-être déjà demandé pourquoi Python indiquait `class` lorsqu'on tapait le nom de ces fonctions dans l'interpréteur (ou en invoquant `help()`) :

16. https://dbader.org/blog/writing-clean-python-with-namedtuples

```
>>> int
<class 'int'>
>>> list
<class 'list'>
>>> range
<class 'range'>
>>> property
<class 'property'>
```

Et bien, c'est parce-que ce sont bel et bien des classes ! Donc, lorsqu'on invoque par exemple `liste1 = list()`, on crée finalement une instance de la classe `list`. Python ne met pas `list` en *CamelCase* car ce sont des classes natives (*built-in classes*). En effet, les auteurs de Python ont décidé que les classes et fonctions natives sont en minuscules, et les exceptions en *CamelCase* (voir ce lien [17]).

Finalement, la création d'une instance à partir d'une classe ou l'appel d'une fonction possède la même syntaxe `mot_clé()` :

```
>>> class Citron:
...     pass
...
>>> Citron()
<__main__.Citron object at 0x7fb776308a10>
>>> def fct():
...     return "Coucou"
...
>>> fct()
'Coucou'
```

On peut le voir aussi quand on invoque l'aide sur un de ces outils, par exemple `help(int)` :

```
Help on class int in module builtins:

class int(object)
 |  int([x]) -> integer
 |  int(x, base=10) -> integer
[...]
```

Il est bien précise que `int` est une classe.

Si on prend des fonctions natives (*built-in functions*) de Python comme `len()` ou `sorted()`, l'interpréteur nous confirme bien qu'il s'agit de fonctions :

```
>>> len
<built-in function len>
>>> sorted
<built-in function sorted>
```

Par conséquent, d'un point de vue purement sémantique nous devrions parler de classe plutôt que de fonction pour les instructions comme `list()`, `range()`, etc. Toutefois, nous avons décidé de garder le nom fonction pour ne pas compliquer les premiers chapitres de ce cours.

24.9 Exercices

Conseil

Pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

24.9.1 Classe molécule

Pour illustrer le mécanisme de la composition en POO, on se propose de créer un programme `molecule.py` qui permettra de décrire une molécule en utilisant les classes. Nous allons créer une classe représentant une molécule (qui sera notre classe *Composite*) et celle-ci contiendra des instances d'une classe décrivant un atome (classe *Component*).

17. https://peps.python.org/pep-0008/#class-names

On se propose de tester cela sur la molécule simple de benzene. Vous aurons besoin du fichier `benzene.pdb`[18] pour réaliser cet exercice.

Après les import nécessaires, le programme contiendra une constante donnant les masses des atomes sous forme de dictionnaire : `ATOM_MASSES = {"C": 12.0, "O": 16.0, "H": 1.0}`.

Créer une classe `Atom` en vous inspirant des exercices du chapitre 23 *Avoir la classe avec les objets*. Cette classe devra instancier des objets contenant les attributs d'instance suivants :

- nom d'atome (par exemple `C1`)
- type d'atome (une seule lettre, déduit du nom d'atome, par exemple `C`)
- coordonnée x
- coordonnée y
- coordonnée z

Le nom d'atome et coordonnées cartésiennes seront passés au constructeur.

Ajouter les méthodes `calc_distance()`, `calc_com()` (*center of mass*). Ajouter une méthode `mute_atom(name)` qui change le nom de l'atome, où `name` est un nouveau nom d'atome (par exemple `O1`). Cette méthode changera également l'attribut d'instance décrivant le type d'atome.

Créer une classe `Molecule` qui construit les attributs d'instance : - Nom de la molécule - Une liste d'atomes (vide à l'instanciation) : `list_atoms` - Une liste indiquant la connectivité (la liste des atomes connectés, vide à l'instanciation) : `list_connectivity`

Le constructeur prendra en argument seulement le nom de la molécule.

Créer une méthode `add_atom(atom)` qui vérifie si l'argument passé est bien une instance de la classe `Atom`, et qui ajoute `atom` dans la liste d'atomes.

Créer une autre méthode `build_mlc_from_pdb(filename)` qui prend en argument un nom de fichier pdb. La méthode lit le fichier pdb, et pour chaque atome lu, crée une instance de la classe `Atom`, et ajouter celle-ci à `list_atoms` .

Ajouter une méthode `calc_mass()` qui calcule et renvoie masse de la molécule.

Créer une méthode `calc_com()` qui cette fois-ci calcule et renvoie le centre de masse de la molécule entière.

Ajouter la méthode `calc_connectivity()` qui calcule et renvoie une liste décrivant la connectivité entre les atomes. Deux atomes sont considérés connectés s'il y a une liaison covalente entre eux, on peut pour cela calculer la distance entre eux qui doit être inférieure à 1.6 Å. La liste de connectivité pourra être construite dans ce style : `[("C1", "H1"), ("C1", "C2"), ...]`.

Chaque paire d'atome doit apparaitre une seule fois (pas de `[("C1", "H1"), [("H1", "C1"), ...]`.

Créer une méthode spéciale affichant les caractéristiques de la molécule lorsqu'on utilise `print()` avec une instance de cette classe `Molecule`, par exemple `print(benzene)`. Cette méthode pourra par exemple afficher avant d'avoir créé la molécule :

```
Molecule benzene
No atom for the moment
No connectivity for the moment
```

Ou bien, lorsque la molécule est créée et la connectivité déterminée, elle s'affichera comme ceci :

```
Molecule benzene
atom C1, type C, mass =  12.0 amu, coor( -2.145,   0.973,  -0.003)
atom H1, type H, mass =   1.0 amu, coor( -3.103,   0.460,  -0.005)
[...]
Connectivity
C1 connected to H1
C1 connected to C2
[...]
```

Pour lancer le programme dans un premier temps, vous pourrez instancier une molécule benzene, puis y ajouter les atomes :

```
1  if __name__ == "__main__":
2      benzene = Molecule("benzene")
3      print(benzene)
4      benzene.build_mlc_from_pdb("benzene.pdb")
5      print(benzene)
```

18. %22https://python.sdv.u-paris.fr/data-files/benzene.pdb%22

Dans un deuxième temps, le programme principal calculera la masse et le centre de masse de benzene et les affichera. Muter ensuite l'atome `H1` en `O1` et recalculer la masse et le centre de masse et les afficher.

Pour aller plus loin, vous pouvez ajouter une méthode qui calcule et affiche un graphe de la molécule avec le module networkx[19]. La page de tutorial[20] pourra vous être utile.

Par exemple :

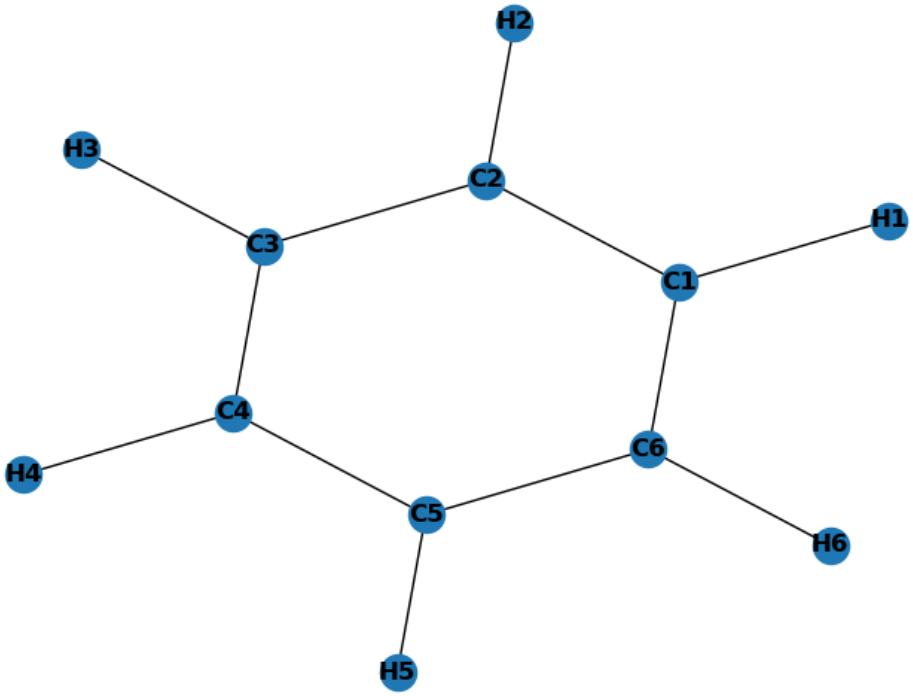


FIGURE 24.4 – Graphe représentant une molécule de benzène.

19. https://networkx.org/
20. https://networkx.org/documentation/latest/tutorial.html#drawing-graphs

CHAPITRE 25

Fenêtres graphiques et *Tkinter*

Conseil

Dans ce chapitre, nous allons utiliser des classes, nous vous conseillons de bien relire les chapitres 23 *Avoir la classe avec les objets* et 24 *Avoir plus la classe avec les objets* (en ligne). Par ailleurs, nous vous conseillons de relire également la rubrique *Arguments positionnels et arguments par mot-clé* du chapitre 10 sur les fonctions.

25.1 Utilité d'une GUI

Dans votre carrière « pythonesque » il se peut que vous soyez amené à vouloir développer une application graphique, on parle encore de *graphical user interface* ou GUI. Jusqu'à maintenant, vous avez fait en sorte qu'un utilisateur interagisse avec votre code via la ligne de commande, par exemple :

```
$ python mon_script.py file.gbk blabla blublu
```

Les arguments passés à la ligne de commande sont tout à fait classiques dans le monde de la bioinformatique. Toutefois, il se peut que vous développiez un programme pour une communauté plus large, qui n'a pas forcément l'habitude d'utiliser un *shell* et la ligne de commande. Une GUI permettra un usage plus large de votre programme, il est donc intéressant de regarder comment s'y prendre. Dans notre exemple ci-dessus on pourrait par exemple développer une interface où l'utilisateur choisirait le nom du fichier d'entrée par l'intermédiaire d'une boîte de dialogue, et de contrôler les options en cliquant sur des boutons, ou des « listes de choix ». Une telle GUI pourrait ressembler à la figure 25.1.

Au delà de l'aspect convivial pour l'utilisateur, vous pourrez, avec une GUI, construire des fenêtres illustrant des éléments que votre programme génère à la volée. Ainsi, vous « verrez » ce qui se passe de manière explicite et en direct ! Par exemple, si on réalise une simulation de particules, on a envie de voir un « film » des particules en mouvement, c'est-à-dire comment ces particules bougent au fur et à mesure que les pas de simulation avancent. Une GUI vous permettra une telle prouesse ! Enfin, sachez que certains logiciels scientifiques ont été développés avec la bibliothèque graphique Tk (par exemple pymol, vmd, etc.). Qui sait, peut-être serez-vous le prochain développeur d'un outil incontournable ?

Il existe beaucoup de modules pour construire des applications graphiques. Par exemple : *Tkinter*[1], wxpython[2], PyQt[3],

1. https://wiki.python.org/moin/TkInter
2. http://www.wxpython.org/
3. https://pyqt.readthedocs.io

FIGURE 25.1 – Exemple de GUI.

PyGObject[4], etc. Nous présentons dans ce chapitre le module *Tkinter* qui est présent de base dans les distributions Python (pas besoin *a priori* de faire d'installation de module externe). *Tkinter* permet de piloter la bibliothèque graphique Tk (*Tool Kit*), *Tkinter* signifiant *tk interface*. On pourra noter que cette bibliothèque Tk peut être également pilotée par d'autres langages (Tcl, perl, etc.).

25.2 Quelques concepts liés à la programmation graphique

Lorsque l'on développe une GUI, nous créons une fenêtre graphique contenant notre application, ainsi que des *widgets* inclus dans la fenêtre.

Définition

Les *widgets* (*window gadget*) sont des objets graphiques permettant à l'utilisateur d'interagir avec votre programme Python de manière conviviale. Par exemple, dans la fenêtre sur la figure 25.1, les boutons, les listes de choix, ou encore la zone de texte sont des *widgets*.

L'utilisation d'une GUI va amener une nouvelle manière d'aborder le déroulement d'un programme, il s'agit de la programmation dite « événementielle ». Jusqu'à maintenant vous avez programmé « linéairement », c'est-à-dire que les instructions du programme principal s'enchaînaient les unes derrière les autres (avec bien sûr de possibles appels à des fonctions). Avec une GUI, l'exécution est décidée par l'utilisateur en fonction de ses interactions avec les différents *widgets*. Comme c'est l'utilisateur qui décide quand et où il clique dans l'interface, il va falloir mettre en place ce qu'on appelle un « gestionnaire d'événements ».

Définition

Le gestionnaire d'événements est une sorte de « boucle infinie » qui est à l'affût de la moindre action de la part de l'utilisateur. C'est lui qui effectuera une action lors de l'interaction de l'utilisateur avec chaque *widget* de la GUI. Ainsi, l'exécution du programme sera réellement guidée par les actions de l'utilisateur.

La bibliothèque Tk que nous piloterons avec le module Python *Tkinter* propose tous les éléments cités ci-dessus (fenêtre graphique, *widgets*, gestionnaire d'événements). Nous aurons cependant besoin d'une dernière notion : les fonctions *callback*.

Définition

4. https://pygobject.readthedocs.io/en/latest/

Une fonction *callback* est une fonction passée en argument d'une autre fonction.

Un exemple de fonction *callback* est présenté dans la rubrique suivante.

25.3 Notion de fonction *callback*

Conseil

Si vous êtes débutant, vous pouvez sauter cette rubrique.

Jusqu'à maintenant nous avons toujours appelé les fonctions ou les méthodes de cette manière :

```
var = fct(arg1, arg2)

obj.methode(arg)
```

où les arguments étaient des objets « classiques » (par exemple une chaîne de caractères, un entier, un *float*, etc.).
Sachez qu'il est possible de passer en argument une fonction à une autre fonction ! Par exemple :

```
def fct_callback(arg):
    print(f"J'aime bien les {arg} !")


def une_fct(ma_callback):
    print("Je suis au début de une_fct(), "
          "et je vais exécuter la fonction callback :")
    ma_callback("fraises")
    print("une_fct() se termine.")

if __name__ == "__main__":
    une_fct(fct_callback)
```

Si on exécute ce code, on obtient :

```
Je suis au début de une_fct() et je vais exécuter la fonction callback :
J'aime bien les fraises !
une_fct() se termine.
```

Vous voyez que dans le programme principal, lors de l'appel de `une_fct()`, on lui passe comme argument une autre fonction mais sans **aucune parenthèses ni argument**, c'est-à-dire `fct_callback` tout court. En d'autres termes, cela est différent de
`une_fct(fct_callback("scoubidous"))`.

Dans une telle construction, `fct_callback("scoubidous")` serait d'abord évaluée, puis ce serait la valeur renvoyée par cet appel qui serait passée à `une_fct()` (n'essayez pas sur notre exemple car cela mènerait à une erreur !). Que se passe-t-il en filigrane lors de l'appel `une_fct(fct_callback)` ? Python passe une référence vers la fonction `fct_callback` (en réalité il s'agit d'un pointeur, mais tout ceci est géré par Python et est transparent pour l'utilisateur). Vous souvenez-vous ce qui se passait avec une liste passée en argument à une fonction (voir le chapitre 13 *Plus sur les fonctions*) ? C'était la même chose, une référence était envoyée plutôt qu'une copie. *Python Tutor*[5] nous confirme cela (cf. figure 25.2).

Lorsqu'on est dans `une_fct()` on pourra utiliser bien sûr des arguments lors de l'appel de notre fonction *callback* si on le souhaite. Notez enfin que dans `une_fct()` la fonction *callback* reçue en argument peut avoir un nom différent (comme pour tout type de variable).

À quoi cela sert-il ? À première vue cette construction peut sembler ardue et inutile. Toutefois, vous verrez que dans le module *Tkinter* les fonctions *callback* sont incontournables. En effet, on utilise cette construction pour lancer une fonction lors de l'interaction de l'utilisateur avec un *widget* : par exemple, lorsque l'utilisateur clique sur un bouton et qu'on souhaite lancer une fonction particulière suite à ce clic. Notez enfin que nous les avons déjà croisées avec :

5. http://pythontutor.com

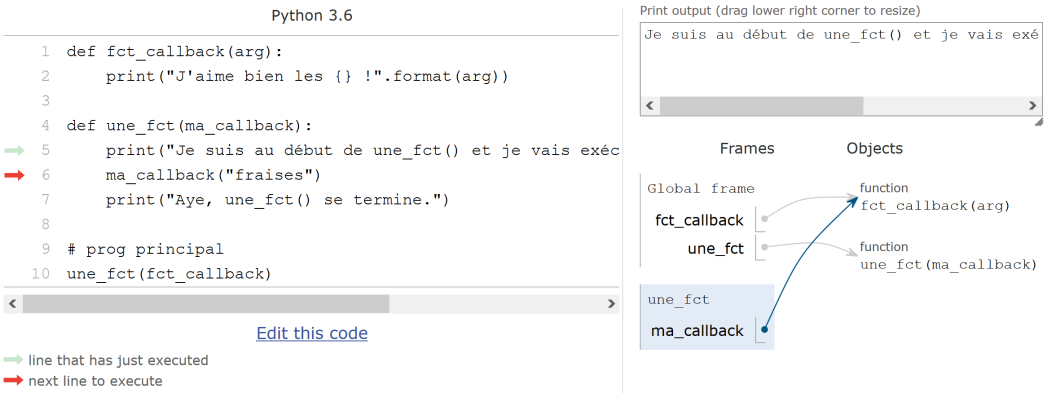


FIGURE 25.2 – Exemple de fonction *callback* dans *Python Tutor*.

- le tri de dictionnaire par valeur avec la syntaxe `sorted(dico, key=dico.get)` (voir le chapitre 8 *Dictionnaires et tuples*) ;
- le tri par longueur de mots avec la syntaxe `sorted(liste, key=len)` (voir chapitre 12 *Plus sur les listes*) ;
- les objets `property` avec la syntaxe `property(fget=get_masse, fset=set_masse)` (voir le chapitre 24 *Avoir plus la classe avec les objets* (en ligne)).

25.4 Prise en main du module *Tkinter*

Le module *Tkinter* est très vaste. Notre but n'est pas de vous faire un cours exhaustif mais plutôt de vous montrer quelques pistes. Pour apprendre à piloter ce module, nous pensons qu'il est intéressant de vous montrer des exemples. Nous allons donc en présenter quelques-uns qui pourraient vous être utiles, à vous ensuite de consulter de la documentation supplémentaire si vous souhaitez aller plus loin (cf. la rubrique *Bibliographie pour aller plus loin*).

25.4.1 Un premier exemple dans l'interpréteur

Commençons par construire un script qui affichera une simple fenêtre avec un message et un bouton. Regardons d'abord comment faire dans l'interpréteur (nous vous conseillons de tester ligne par ligne ce code tout en lisant les commentaires ci-dessous) :

```
1 >>> import tkinter as tk
2 >>> racine = tk.Tk()
3 >>> label = tk.Label(racine, text="J'adore Python !")
4 >>> bouton = tk.Button(racine, text="Quitter", fg="red",
5 ...                    command=racine.destroy)
6 >>> label.pack()
7 >>> bouton.pack()
8 >>>
```

Ligne 2. On crée la fenêtre principale (vous la verrez apparaître !). Pour cela, on crée une instance de la classe `tk.Tk` dans la variable `racine`. Tous les *widgets* que l'on créera ensuite seront des fils de cette fenêtre. On pourra d'ailleurs noter que cette classe `tk.Tk` ne s'instancie en général qu'une seule fois par programme. Vous pouvez, par curiosité, lancer une commande `dir(racine)` ou `help(racine)`, vous verrez ainsi les très nombreuses méthodes et attributs associés à un tel objet Tk.

Ligne 3. On crée un *label*, c'est-à-dire une zone dans la fenêtre principale où on écrit un texte. Pour cela, on a créé une variable `label` qui est une instance de la classe `tk.Label`. Cette variable `label` contient donc notre *widget*, nous la réutiliserons plus tard (par exemple pour placer ce *widget* dans la fenêtre). Notez le premier argument positionnel`racine` passé à la classe `tk.Label`, celui-ci indique la fenêtre parente où doit être dessinée le *label*. Cet argument doit toujours être passé en premier et il est vivement conseillé de le préciser. Nous avons passé un autre argument avec le nom `text` pour indiquer, comme vous l'avez deviné, le texte que nous souhaitons voir dans ce *label*. La classe `tk.Label` peut recevoir de nombreux autres arguments, en voici la liste exhaustive[6]. Dans les fonctions *Tkinter* qui construisent

6. http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/label.html

un *widget*, les arguments possibles pour la mise en forme de celui-ci sont nombreux, si bien qu'ils sont toujours des arguments par mot-clé. Si on ne précise pas un de ces arguments lors de la création du *widget*, l'argument prendra alors une valeur par défaut. Cette liste des arguments par mot-clé est tellement longue qu'en général on ne les précisera pas tous. Heureusement, Python autorise l'utilisation des arguments par mot-clé dans un ordre quelconque. Comme nous l'avons vu dans le chapitre 10 *Fonctions*, souvenez vous que leur utilisation dans le désordre implique qu'il faudra toujours préciser leur nom : par exemple vous écrirez `text="blabla"` et non pas `"blabla"` tout court.

Ligne 4. De même, on crée un bouton « Quitter » qui provoquera la fermeture de la fenêtre et donc l'arrêt de l'application si on clique dessus. À nouveau, on passe la fenêtre parente en premier argument, le texte à écrire dans le bouton, puis la couleur de ce texte. Le dernier argument `command=racine.destroy` va indiquer la fonction / méthode à exécuter lorsque l'utilisateur clique sur le bouton. On pourra noter que l'instance de la fenêtre mère `tk.Tk` (que nous avons nommée `racine`) possède une méthode `.destroy()` qui va détruire le *widget* sur lequel elle s'applique. Comme on tue la fenêtre principale (que l'on peut considérer comme un *widget* contenant d'autres *widgets*), tous les *widgets* fils seront détruits et donc l'application s'arrêtera. Vous voyez par ailleurs que cette méthode `racine.destroy` est passée à l'argument `command=` **sans parenthèses ni arguments** : il s'agit donc d'une fonction *callback* comme expliqué ci-dessus. Dans tous les *widgets Tkinter*, on doit passer à l'argument `command=...` une fonction / méthode *callback*. La liste exhaustive des arguments possibles de la classe `tk.Button` se trouve ici[7].

Lignes 6 et 7. Vous avez noté que lors de la création de ce *label* et de ce bouton, rien ne s'est passé dans la fenêtre. C'est normal, ces deux *widgets* existent bien, mais il faut maintenant les placer à l'intérieur de la fenêtre. On appelle pour ça la méthode `.pack()`, avec une notation objet `widget.pack()` : à ce moment précis, vous verrez votre label apparaître ainsi que la fenêtre qui se redimensionne automatiquement en s'adaptant à la grandeur de votre *label*. L'invocation de la même méthode pour le bouton va faire apparaître celui-ci juste en dessous du *label* et redimensionner la fenêtre. Vous l'aurez compris la méthode `.pack()` place les *widgets* les uns en dessous des autres et ajuste la taille de la fenêtre. On verra plus bas que l'on peut passer des arguments à cette méthode pour placer les *widgets* différemment (en haut, à droite, à gauche).

Au final, vous devez obtenir une fenêtre comme sur la figure 25.3.

25.4.2 Le même exemple dans un script.

Tentons maintenant de faire la même chose dans un script `tk_exemple.py` :

```
1  import tkinter as tk
2
3  racine = tk.Tk()
4  label = tk.Label(racine, text="J'adore Python !")
5  bouton = tk.Button(racine, text="Quitter", command=racine.quit)
6  bouton["fg"] = "red"
7  label.pack()
8  bouton.pack()
9  racine.mainloop()
10 print("C'est fini !")
```

puis lançons ce script depuis un *shell* :

```
$ python tk_exemple.py
```

Vous voyez maintenant la même fenêtre avec les mêmes fonctionnalités par rapport à la version dans l'interpréteur (voir la figure 25.3). Nous commentons ici les différences (dans le désordre) :

Ligne 6. Le bouton a été créé en ligne 5, mais on voit qu'il est possible de préciser une option de rendu du widget après cette création (ici on met le texte en rouge avec l'option `"fg"`). La notation ressemble à celle d'un dictionnaire avec une syntaxe générale `widget["option"] = valeur`.

Ligne 9. L'instruction `racine.mainloop()` va lancer le gestionnaire d'événements que nous avons évoqué ci-dessus. C'est lui qui interceptera la moindre action de l'utilisateur, et qui lancera les portions de code associées à chacune de ses actions. Bien sûr, comme nous développerons dans ce qui va suivre toutes nos applications *Tkinter* dans des scripts (et non pas dans l'interpréteur), cette ligne sera systématiquement présente. Elle sera souvent à la fin du script, puisque, à l'image de ce script, on écrit d'abord le code construisant l'interface, et on lance le gestionnaire d'événements une fois l'interface complètement décrite, ce qui lancera au final l'application.

7. http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/button.html

Ligne 10. Cette ligne ne s'exécute qu'après l'arrêt de l'application (soit en cliquant sur le bouton « Quitter », soit en cliquant sur la croix).

Ligne 5. Pour quitter l'application, on utilise ici la méthode `.quit()`. Celle-ci casse la `.mainloop()` et arrête ainsi le gestionnaire d'événements. Cela mène à l'arrêt de l'application. Dans le premier exemple dans l'interpréteur, on avait utilisé la méthode `.destroy()` sur la fenêtre principale. Comme son nom l'indique, celle-ci détruit la fenêtre principale et mène aussi à l'arrêt de l'application. Cette méthode aurait donc également fonctionné ici. Par contre, la méthode `.quit()` n'aurait pas fonctionné dans l'interpréteur car, comme on l'a vu, la boucle `.mainloop()` n'y est pas présente. Comme nous écrirons systématiquement nos applications *Tkinter* dans des scripts, et que la boucle `.mainloop()` y est obligatoire, vous pourrez utiliser au choix `.quit()` ou `.destroy()` pour quitter l'application.

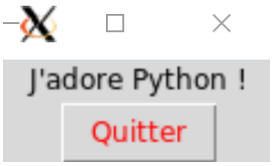


FIGURE 25.3 – Exemple basique de fenêtre *Tkinter*.

25.5 Construire une application *Tkinter* avec une classe

De manière générale, il est vivement conseillé de développer ses applications *Tkinter* en utilisant une classe. Cela présente l'avantage d'encapsuler l'application de manière efficace et d'éviter ainsi l'utilisation de variables globales. Souvenez-vous, elles sont à bannir définitivement ! Une classe crée un espace de noms propre à votre application, et toutes les variables nécessaires seront ainsi des attributs de cette classe. Reprenons notre petit exemple avec un label et un bouton :

```
1  import tkinter as tk
2
3  class Application(tk.Tk):
4      def __init__(self):
5          tk.Tk.__init__(self)
6          self.creer_widgets()
7
8      def creer_widgets(self):
9          self.label = tk.Label(self, text="J'adore Python !")
10         self.bouton = tk.Button(self, text="Quitter", command=self.quit)
11         self.label.pack()
12         self.bouton.pack()
13
14
15 if __name__ == "__main__":
16     app = Application()
17     app.title("Ma Première App :-)")
18     app.mainloop()
```

Ligne 3. On crée notre application en tant que classe. Notez que cette classe porte un nom qui commence par une majuscule (comme recommandé dans les bonnes pratiques de la PEP8[8], voir le chapitre 16 *Bonnes pratiques en programmation Python*). L'argument passé dans les parenthèses indique que notre classe `Application` hérite de la classe `tk.Tk`. Par ce mécanisme, nous héritons ainsi de toutes les méthodes et attributs de cette classe mère, mais nous pouvons en outre en ajouter de nouvelles/nouveaux (on parle aussi de « redéfinition » de la classe `tk.Tk`) !

Ligne 4. On crée un constructeur, c'est-à-dire une méthode qui sera exécutée lors de l'instanciation de notre classe (à la ligne 16).

Ligne 5. On appelle ici le constructeur de la classe mère `tk.Tk.__init__()`. Pourquoi fait-on cela ? On se souvient dans la version linéaire de l'application, on avait utilisé une instanciation classique : `racine = tk.Tk()`. Ici, l'effet de l'appel du constructeur de la classe mère permet d'instancier la fenêtre Tk dans la variable `self` directement. C'est-à-dire que la prochaine fois que l'on aura besoin de cette instance (lors de la création des *widgets* par exemple, cf. lignes 9 et

8. https://www.python.org/dev/peps/pep-0008/

10), on utilisera directement `self` plutôt que `racine` ou tout autre nom donné à l'instance. Comme vu dans le chapitre 23 *Avoir la classe avec les objets*, appeler le constructeur de la classe mère est une pratique classique lorsqu'une classe hérite d'une autre classe.

Ligne 6. On appelle la méthode `self.creer_widgets()` de notre classe `Application`. Pour rappel, le `self` avant le `.creer_widgets()` indique qu'il s'agit d'une méthode de notre classe (et non pas d'une fonction classique).

Ligne 8. La méthode `.creer_widgets()` va créer des *widgets* dans l'application.

Ligne 9. On crée un label en instanciant la classe `tk.Label()`. Notez que le premier argument passé est maintenant `self` (au lieu de `racine` précédemment) indiquant la fenêtre dans laquelle sera construit ce *widget*.

Ligne 10. De même on crée un *widget* bouton en instanciant la classe `tk.Button()`. Là aussi, l'appel à la méthode `.quit()` se fait par `self.quit` puisque la fenêtre est instanciée dans la variable `self`. Par ailleurs, on ne met ni parenthèses ni arguments à `self.quit` car il s'agit d'une fonction *callback* (comme dans la rubrique précédente).

Lignes 11 et 12. On place les deux *widgets* dans la fenêtre avec la méthode `.pack()`.

Ligne 15. Ici on autorise le lancement de notre application *Tkinter* en ligne de commande (`python tk_application.py`), ou bien de réutiliser notre classe en important `tk_application.py` en tant que module (`import tk_application`) (voir le chapitre 15 *Création de modules*).

Ligne 16. On instancie notre application.

Ligne 17. On donne un titre dans la fenêtre de notre application. Comme on utilise de petits *widgets* avec la méthode `pack()`, il se peut que le titre ne soit pas visible lors du lancement de l'application. Toutefois, si on « étire » la fenêtre à la souris, le titre le deviendra. On pourra noter que cette méthode `.title()` est héritée de la classe mère Tk.

Ligne 18. On lance le gestionnaire d'événements.

Au final, vous obtiendrez le même rendu que précédemment (cf. figure 25.3). Alors vous pourrez-vous poser la question, « pourquoi ai-je besoin de toute cette structure alors que le code précédent semblait plus direct ? ». La réponse est simple, lorsqu'un projet de GUI grossit, le code devient très vite illisible s'il n'est pas organisé en classe. De plus, la non-utilisation de classe rend quasi-obligatoire l'utilisation de variables globales, ce qui on l'a vu, est à proscrire définitivement ! Dans la suite du chapitre, nous verrons quelques exemples qui illustrent cela (cf. la rubrique suivante).

25.6 Le *widget canvas*

25.6.1 Un *canvas* simple et le système de coordonnées

Le *widget canvas*[9] de *Tkinter* est très puissant. Il permet de dessiner des formes diverses (lignes, cercles, etc.), et même de les animer !

La classe `tk.Canvas` crée un *widget canvas* (ou encore canevas en français). Cela va créer une zone (*i.e.* le canevas en tant que tel) dans laquelle nous allons dessiner divers objets tels que des ellipses, lignes, polygones, etc., ou encore insérer du texte ou des images. Regardons tout d'abord un code minimal qui construit un *widget canvas*, dans lequel on y dessine un cercle et deux lignes :

```
1  import tkinter as tk
2
3  racine = tk.Tk()
4  canv = tk.Canvas(racine, bg="white", height=200, width=200)
5  canv.pack()
6  canv.create_oval(0, 0, 200, 200, outline="red", width=10)
7  canv.create_line(0, 0, 200, 200, fill="black", width=10)
8  canv.create_line(0, 200, 200, 0, fill="black", width=10)
9  racine.mainloop()
```

Ligne 4. On voit qu'il faut d'abord créer le *widget canvas*, comme d'habitude en lui passant l'instance de la fenêtre principale en tant qu'argument positionnel, puis les options. Notons que nous lui passons comme options la hauteur et la largeur du *canvas*. Même s'il s'agit d'arguments par mot-clé, donc optionnels, c'est une bonne pratique de les préciser. En effet, les valeurs par défaut risqueraient de nous mener à dessiner hors de la zone visible (cela ne génère pas d'erreur mais n'a guère d'intérêt).

Ligne 6 à 8. Nous dessinons maintenant des objets graphiques à l'intérieur du canevas avec les méthodes `.create_oval()` (dessine une ellipse) et `.create_line()` (dessine une ligne). Les arguments positionnels sont les coordonnées de

9. http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/canvas.html

l'ellipse (les deux points englobant l'ellipse, cf. ce lien [10] pour la définition exacte) ou de la ligne. Ensuite, on passe comme d'habitude des arguments par mot-clé (vous commencez à avoir l'habitude !) pour mettre en forme ces objets graphiques.

Le rendu de l'image est montré dans la figure 25.4 ainsi que le système de coordonnées associé au *canvas*. Comme dans la plupart des bibliothèques graphiques, l'origine du repère du *canvas* (*i.e.* la coordonnée $(0,0)$) est en haut à gauche. Les x vont de gauche à droite, et les y vont de haut en bas.

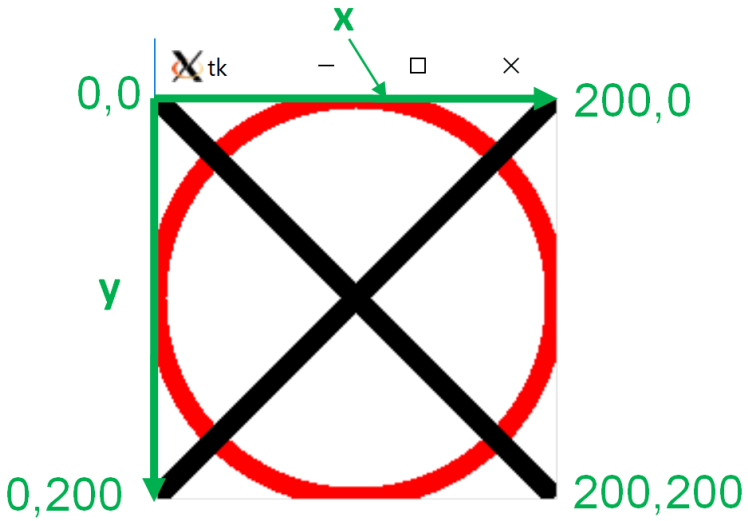


FIGURE 25.4 – Exemple 1 de *canvas* avec le système de coordonnées. Le système de coordonnées est montré en vert et n'apparaît pas sur la vraie fenêtre *Tkinter*.

Attention

L'axe des y est inversé par rapport à ce que l'on représente en mathématique. Si on souhaite représenter une fonction mathématique (ou tout autre objet dans un repère régi par un repère mathématique), il faudra faire un changement de repère.

25.6.2 Un *canvas* encapsulé dans une classe

Voici un exemple un peu plus conséquent d'utilisation du *widget canvas* qui est inclus dans une classe. Il s'agit d'une application dans laquelle il y a une zone de dessin, un bouton dessinant des cercles, un autre des lignes et un dernier bouton qui quitte l'application (figure 25.5).

Le code suivant crée une telle application :

10. http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/create_oval.html

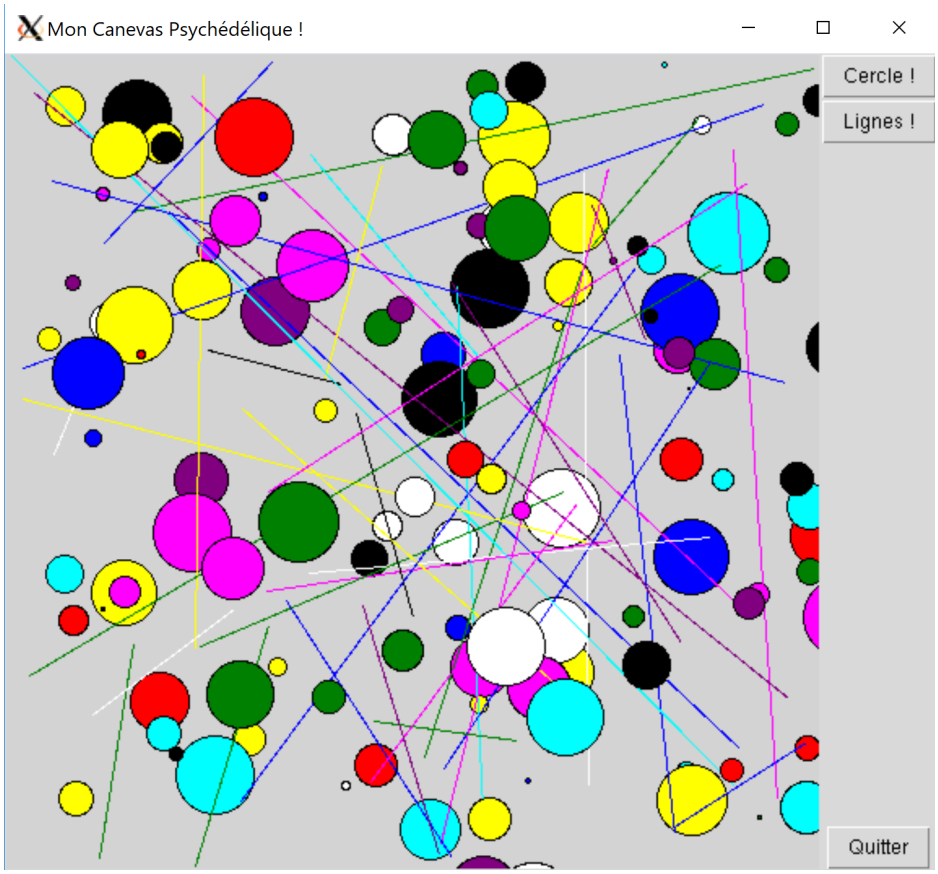


FIGURE 25.5 – Exemple 2 de *canvas*.

```
import tkinter as tk
import random as rd

class AppliCanevas(tk.Tk):
    def __init__(self):
        tk.Tk.__init__(self)
        self.size = 500
        self.creer_widgets()

    def creer_widgets(self):
        # création canevas
        self.canv = tk.Canvas(self, bg="light gray", height=self.size,
                              width=self.size)
        self.canv.pack(side=tk.LEFT)
        # boutons
        self.bouton_cercles = tk.Button(self, text="Cercle !",
                                        command=self.dessine_cercles)
        self.bouton_cercles.pack(side=tk.TOP)
        self.bouton_lignes = tk.Button(self, text="Lignes !",
                                       command=self.dessine_lignes)
        self.bouton_lignes.pack()
        self.bouton_quitter = tk.Button(self, text="Quitter",
                                        command=self.quit)
        self.bouton_quitter.pack(side=tk.BOTTOM)

    def rd_col(self):
        return rd.choice(("black", "red", "green", "blue", "yellow", "magenta",
                          "cyan", "white", "purple"))

    def dessine_cercles(self):
        for i in range(20):
            x, y = [rd.randint(1, self.size) for j in range(2)]
            diameter = rd.randint(1, 50)
            self.canv.create_oval(x, y, x+diameter, y+diameter,
                                  fill=self.rd_col())

    def dessine_lignes(self):
        for i in range(20):
            x, y, x2, y2 = [rd.randint(1, self.size) for j in range(4)]
```

Lignes 4 à 6. Comme montré dans la rubrique *Construire une application Tkinter avec une classe*, notre classe `AppliCanevas` hérite de la classe générale `tk.Tk` et la fenêtre Tk se retrouve dans la variable `self`.

Ligne 7. On crée un attribut de la classe `self.size` qui contiendra la taille (hauteur et largeur) du *canvas*. On rappelle que cet attribut sera visible dans l'ensemble de la classe puisqu'il est « accroché » à celle-ci par le `self`.

Ligne 8. On lance la méthode `.creer_widgets()` (qui est elle aussi « accrochée » à la classe par le `self`).

Lignes 12 à 14. On crée un *widget canvas* en instanciant la classe `tk.Canvas`. On place ensuite le *canvas* dans la fenêtre avec la méthode `.pack()` en lui précisant où le placer avec la variable *Tkinter* `tk.LEFT`.

Lignes 15 à 24. On crée des *widgets* boutons et on les place dans la fenêtre. À noter que chacun de ces *widgets* appelle une méthode différente, dont deux que nous avons créées dans la classe (`.dessine_cercle()` et `.dessine_lignes()`).

Ligne 26 à 28. Cette méthode renvoie une couleur au hasard sous forme de chaîne de caractères.

Lignes 30 à 40. On définit deux méthodes qui vont dessiner des paquets de 20 cercles (cas spécial d'une ellipse) ou 20 lignes aléatoires. Lors de la création de ces cercles et lignes, on ne les récupère pas dans une variable car on ne souhaite ni les réutiliser ni changer leurs propriétés par la suite. Vous pourrez noter ici l'avantage de programmer avec une classe, le *canvas* est directement accessible dans n'importe quelle méthode de la classe avec `self.canv` (pas besoin de le passer en argument ou de créer une variable globale).

25.6.3 Un *canvas* animé dans une classe

Dans ce dernier exemple, nous allons illustrer la puissance du *widget canvas* en vous montrant que l'on peut animer les objets se trouvant à l'intérieur. Nous allons également découvrir une technique intéressante, à savoir, comment « intercepter » des clics de souris générés ou des touches pressées par l'utilisateur. L'application consiste en une « baballe » qui se déplace dans la fenêtre et dont on contrôle les propriétés à la souris (cf. figure 25.6). Vous pouvez télécharger le script ici [11].

11. https://python.sdv.u-paris.fr/data-files/tk_baballe.py

```
"""Super appli baballe !!!

Usage: python tk_baballe.py
- clic gauche: faire grossir la baballe
- clic droit: faire rétrécir la baballe
- clic central: relance la baballe (depuis le  point du clic)
                dans une direction aléatoire
- touche Esc: quitte l'appli baballe
"""

import tkinter as tk
import random as rd

class AppliBaballe(tk.Tk):
    def __init__(self):
        """Constructeur de l'application."""
        tk.Tk.__init__(self)
        # Coord baballe.
        self.x, self.y = 200, 200
        # Rayon baballe.
        self.size = 50
        # Pas de deplacement.
        self.dx, self.dy = 20, 20
        # Création et packing du canvas.
        self.canv = tk.Canvas(self, bg='light gray', height=400, width=400)
        self.canv.pack()
        # Création de la baballe.
        self.baballe = self.canv.create_oval(self.x, self.y,
                                             self.x+self.size,
                                             self.y+self.size,
                                             width=2, fill="blue")
        # Binding des actions.
        self.canv.bind("<Button-1>", self.incr)
        self.canv.bind("<Button-2>", self.boom)
        self.canv.bind("<Button-3>", self.decr)
        self.bind("<Escape>", self.stop)
        # Lancer la baballe.
        self.move()

    def move(self):
        """Déplace la baballe (appelée itérativement avec la méthode after)."""
        # Incrémente coord baballe.
        self.x += self.dx
        self.y += self.dy
        # Vérifier que la baballe ne sort pas du canvas (choc élastique).
        if self.x < 10:
            self.dx = abs(self.dx)
        if self.x > 400-self.size-10:
            self.dx = -abs(self.dx)
        if self.y < 10:
            self.dy = abs(self.dy)
        if self.y > 400-self.size-10:
            self.dy = -abs(self.dy)
        # Mise à jour des coord.
        self.canv.coords(self.baballe, self.x, self.y, self.x+self.size,
                         self.y+self.size)
        # Rappel de move toutes les 50ms.
        self.after(50, self.move)

    def boom(self, mclick):
        """Relance la baballe dans une direction aléatoire au point du clic."""
        self.x = mclick.x
        self.y = mclick.y
        self.canv.create_text(self.x, self.y, text="Boom !", fill="red")
        self.dx = rd.choice([-30, -20, -10, 10, 20, 30])
        self.dy = rd.choice([-30, -20, -10, 10, 20, 30])

    def incr(self, lclick):
        """Augmente la taille de la baballe."""
        self.size += 10
        if self.size > 200:
            self.size = 200

    def decr(self, rclick):
        """Diminue la taille de la baballe."""
        self.size -= 10
        if self.size < 10:
```

Lignes 19 à 23. Les coordonnées de la baballe, ses pas de déplacement, et sa taille sont créés en tant qu'attributs de notre classe. Ainsi ils seront visibles partout dans la classe.

Lignes 25 à 31. Le *canvas* est ensuite créé et placé dans la fenêtre, puis on définit notre fameuse baballe. À noter, les coordonnées `self.x` et `self.y` de la baballe représentent en fait son côté « nord-ouest » (en haut à gauche, voir le point (x_0, y_0) dans la documentation officielle[12]).

Lignes 33 à 35. Jusqu'à maintenant, nous avons utilisé des événements provenant de clics sur des boutons. Ici, on va « intercepter » des événements générés par des clics de souris sur le *canvas* et les lier à une fonction / méthode (comme nous l'avions fait pour les clics sur des boutons avec l'option `command=...`). La méthode pour faire cela est `.bind()`, voilà pourquoi on parle de *event binding* en anglais. Cette méthode prend en argument le type d'événement à capturer en tant que chaîne de caractères avec un format spécial : par exemple `"<Button-1>"` correspond à un clic gauche de la souris (de même `"<Button-2>"` et `"<Button-3>"` correspondent aux clics central et droit respectivement). Le deuxième argument de la méthode `.bind()` est une méthode / fonction *callback* à appeler lors de la survenue de l'événement (comme pour les clics de bouton, vous vous souvenez ? On l'appelle donc sans parenthèses ni arguments). On notera que tous ces événements sont liés à des clics sur le *canvas*, mais il est possible de capturer des événements de souris sur d'autres types de *widgets*.

Ligne 36. De même, on peut « intercepter » un événement lié à l'appui sur une touche, ici la touche Esc.

Ligne 38. La méthode `.move()` est appelée, ainsi l'animation démarrera dès l'exécution du constructeur, donc peu après l'instanciation de notre application (Ligne 86).

Lignes 40 à 58. On définit une méthode `.move()` qui va gérer le déplacement de la baballe avec des chocs élastiques sur les parois (et faire en sorte qu'elle ne sorte pas du *canvas*).

Lignes 55 et 56. On utilise la méthode `.coords()` de la classe `Canvas`, qui « met à jour » les coordonnées de n'importe quel objet dessiné dans le *canvas* (c'est-à-dire que cela déplace l'objet).

Ligne 58. Ici, on utilise une autre méthode spécifique des objets *Tkinter*. La méthode `.after()` rappelle une autre méthode ou fonction (second argument) après un certain laps de temps (ici 50 ms, passé en premier argument). Ainsi la méthode `.move()` se rappelle elle-même, un peu comme une fonction récursive. Toutefois, ce n'est pas une vraie fonction récursive comme celle vue dans le chapitre 13 (exemple du calcul de factorielle), car Python ne conserve pas l'état de la fonction lors de l'appel de `.after()`. C'est comme si on avait un `return`, tout l'espace mémoire alloué à la méthode `.move()` est détruit lorsque Python rencontre la méthode `.after()`. On obtiendrait un résultat similaire avec la boucle suivante :

```
1 import time
2
3 ...
4
5 while True:
6     move()
7     time.sleep(0.05) # attendre 50 ms
```

Le temps de 50 ms donne 20 images (ou clichés) par seconde. Si vous diminuez ce temps, vous aurez plus d'images par secondes et donc un « film » plus fluide.

Ligne 60 à 66. On définit la méthode `.boom()` de notre classe qui on se souvient est appelée lors d'un événement clic central sur le *canvas*. Vous noterez qu'outre le `self`, cette fonction prend un autre argument que nous avons nommé ici `mclick`. Il s'agit d'un objet spécial géré par *Tkinter* qui va nous donner des informations sur l'événement généré par l'utilisateur. Dans les lignes 62 et 63, cet objet `mclick` récupère les coordonnées où le clic a eu lieu grâce aux attributs `mclick.x` et `mclick.y`. Ces coordonnées sont réaffectées à la baballe pour la faire repartir de l'endroit du clic. Nous créons ensuite un petit texte dans le canevas et affectons des valeurs aléatoires aux variables de déplacement pour faire repartir la baballe dans une direction aléatoire.

Lignes 68 à 78. On a ici deux méthodes `.incr()` et `.decr()` appelées lors d'un clic gauche ou droit. Deux choses sont à noter : i) l'attribut `self.size` est modifié dans les deux fonctions, mais le changement de diamètre de la boule ne sera effectif dans le *canvas* que lors de la prochaine exécution de l'instruction `self.canv.coords()` (dans la méthode `.move()`) ; ii) de même que pour la méthode `.boom()`, ces deux méthodes prennent un argument après le `self` (`lclick` ou `rclick`) récupérant ainsi des informations sur l'événement de l'utilisateur. Même si on ne s'en sert pas, cet argument après le `self` est obligatoire car il est imposé par la méthode `.bind()`.

Lignes 80 à 82. Cette méthode quitte l'application lorsque l'utilisateur fait un clic sur la touche *Esc*.

12. http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/create_oval.html

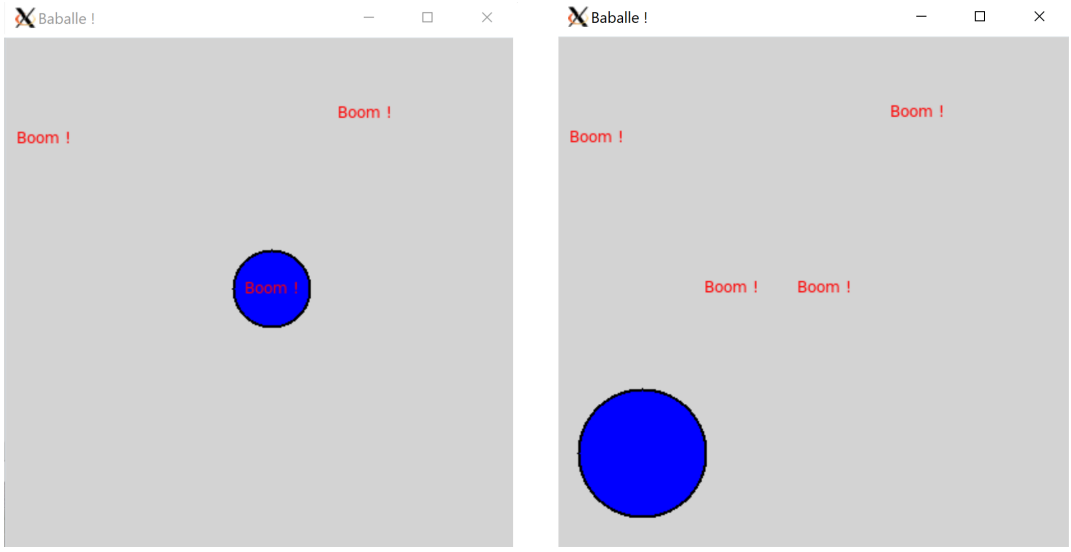


FIGURE 25.6 – Exemple de *canvas* animé à deux instants de l'exécution (panneau de gauche : au moment où on effectue un clic central ; panneau de droite : après avoir effectué plusieurs clics gauches).

Il existe de nombreux autres événements que l'on peut capturer et lier à des méthodes / fonctions *callback*. Vous trouverez une liste complète ici [13].

25.7 Pour aller plus loin

25.7.1 D'autres *widgets*

Jusqu'à maintenant nous avons vu les *widgets Button*, *Canvas*, *Label*, mais il en existe bien d'autres. En voici la liste avec une brève explication pour chacun :

- *Checkbutton* : affiche des cases à cocher.
- *Entry* : demande à l'utilisateur de saisir une valeur / une phrase.
- *Listbox* : affiche une liste d'options à choisir (comme dans la figure 25.1).
- *Radiobutton* : implémente des « boutons radio ».
- *Menubutton* et *Menu* : affiche des menus déroulants.
- *Message* : affiche un message sur plusieurs lignes (extensions du *widget* Label).
- *Scale* : affiche une règle graduée pour que l'utilisateur choisisse parmi une échelle de valeurs.
- *Scrollbar* : affiche des ascenseurs (horizontaux et verticaux).
- *Text* : crée une zone de texte dans lequel l'utilisateur peut saisir un texte sur plusieurs lignes (comme dans la figure 25.1).
- *Spinbox* : sélectionne une valeur parmi une liste de valeurs.
- *tkMessageBox* : affiche une boîte avec un message.

Il existe par ailleurs des *widgets* qui peuvent contenir d'autres widgets et qui organisent le placement de ces derniers :

- *Frame* : *widget* conteneur pouvant contenir d'autres *widgets* classiques, particulièrement utile lorsqu'on réalise une GUI complexe avec de nombreuses zones.
- *LabelFrame* : comme *Frame* mais affiche aussi un *label* sur le bord.
- *Toplevel* : pour créer des fenêtres indépendantes.
- *PanedWindow* : conteneur pour d'autres *widgets*, mais ici l'utilisateur peut réajuster les zones affectées à chaque *widget* fils.

Vous trouverez la documentation exhaustive pour tous ces *widgets* (ainsi que ceux que nous avons décrits dans les rubriques précédentes) sur le site de l'Institut des mines et de technologie du Nouveau Mexique [14] (MNT). Par ailleurs, la

13. http://effbot.org/tkinterbook/tkinter-events-and-bindings.htm
14. http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html

page *Universal widget methods*[15] vous donnera une vue d'ensemble des différentes méthodes associées à chaque widget.

Il existe également une extension de *Tkinter* nommée *ttk*, réimplémentant la plupart des *widgets* de base de *Tkinter* et qui en propose de nouveaux (*Combobox*, *Notebook*, *Progressbar*, *Separator*, *Sizegrip* et *Treeview*). Typiquement, si vous utilisez *ttk*, nous vous conseillons d'utiliser les *widgets ttk* en priorité, et pour ceux qui n'existent pas dans *ttk*, ceux de *Tkinter* (comme *Canvas* qui n'existe que dans *Tkinter*). Vous pouvez importer le sous-module *ttk* de cette manière : `import tkinter.ttk as ttk`.

Vous pourrez alors utiliser les classes de widget de *ttk* (par exemple `ttk.Button`, etc.). Si vous souhaitez importer *ttk* et *Tkinter*, il suffit d'utiliser ces deux lignes :

```
1 import tkinter as tk
2 import tkinter.ttk as ttk
```

Ainsi vous pourrez utiliser des *widgets* de *Tkinter* et de *ttk* en même temps.

Pour plus d'informations, vous pouvez consulter la documentation officielle de Python[16], ainsi que la documentation très complète du site du MNT[17].

25.7.2 Autres pistes à approfondir

Si vous souhaitez aller un peu plus loin en *Tkinter*, voici quelques notions / remarques qui pourraient vous être utiles.

Conseil

Si vous êtes débutant, vous pouvez sauter cette rubrique.

25.7.2.1 Les variables de contrôle

Lorsque vous souhaitez mettre un jour un *widget* avec une certaine valeur (par exemple le texte d'un *label*), vous ne pouvez pas utiliser une variable Python ordinaire, il faudra utiliser une variable *Tkinter* dite de contrôle. Par exemple, si on souhaitait afficher les coordonnées de notre baballe (cf. rubrique précédente) dans un *label*, et que cet affichage se mette à jour au fur et à mesure des mouvements de la baballe, il faudrait utiliser des variables de contrôle. On peut créer de telles variables avec les classes `tk.StringVar` pour les chaînes de caractères, `tk.DoubleVar` pour les *floats*, et `tk.IntVar` pour les entiers. Une fois créée, par exemple avec l'instruction `var = tk.StringVar()`, on peut modifier la valeur d'une variable de contrôle avec la méthode `var.set(nouvelle_valeur)` : ceci mettra à jour tous les *widgets* utilisant cette variable `var`. Il existe aussi la méthode `var.get()` qui récupère la valeur actuelle contenue dans `var`. Enfin, il faudra lors de la création du label utiliser l'option `textvariable=` avec votre variable de contrôle (par exemple `tk.Label(..., textvariable=var, ...)`) pour que cela soit fonctionnel.

À nouveau, vous trouverez une documentation précise sur le site du MNT[18].

25.7.2.2 Autres méthodes de placement des *widgets* dans la fenêtre Tk

Dans les exemples montrés dans ce chapitre, nous avons systématiquement utiliser la méthode `.pack()` pour placer les *widgets*. Cette méthode très simple et directe « empaquette » les *widgets* les uns contre les autres et redimensionne la fenêtre automatiquement. Avec l'option `side=` et les variables `tk.BOTTOM`, `tk.LEFT`, `tk.TOP` et `tk.RIGHT` on place facilement les *widgets* les uns par rapport aux autres. Toutefois, la méthode `.pack()` peut parfois présenter des limites, il existe alors deux autres alternatives.

La méthode `.grid()` permet, grâce à l'utilisation d'une grille, un placement mieux contrôlé des différents *widgets*. La méthode `.place()` place enfin les *widgets* en utilisant les coordonnées de la fenêtre principale. Nous ne développerons pas plus ces méthodes, mais voici de la documentation supplémentaire en accès libre :

- `.pack()`[19] ;

15. https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/universal.html
16. https://docs.python.org/3/library/tkinter.ttk.html
17. http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/ttk.html
18. https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/control-variables.html
19. http://effbot.org/tkinterbook/pack.htm

- `.grid()`[20], [21] ;
- `.place()`[22].

25.7.2.3 Hériter de la classe Frame pour vos applications ?

Comme illustré dans nos exemples, nous vous recommandons pour vos classes applications *Tkinter* d'hériter de la classe mère `tk.Tk` et d'utiliser le constructeur de la classe mère `tk.Tk.__init__()`. Toutefois, il se peut qu'en consultant d'autres ressources certains auteurs utilisent la technique d'héritage de la classe mère `tk.Frame` :

```
1  import tkinter as tk
2
3  class Application(tk.Frame):
4      def __init__(self, racine=None):
5          tk.Frame.__init__(self, racine)
6          self.racine = racine
7          self.create_widgets()
8
9      def create_widgets(self):
10         self.label = tk.Label(self.racine, text="J'adore Python !")
11         self.bouton = tk.Button(self.racine, text="Quitter",
12                                 fg="green", command=self.quit)
13         self.label.pack()
14         self.bouton.pack()
15
16
17 if __name__ == "__main__":
18     racine = tk.Tk()
19     racine.title("Ma Première App :-)")
20     app = Application(racine)
21     racine.mainloop()
```

Lignes 17 à 21. Commentons d'abord le programme principal : ici on crée la fenêtre principale dans l'instance `racine` puis on instancie notre classe en passant `racine` en argument.

Lignes 4 et 5. Ici réside la principale différence par rapport à ce que nous vous avons montré dans ce chapitre : en ligne 4 on passe l'argument `racine` à notre constructeur, puis en ligne 5 on passe ce même argument `racine` lors de l'appel du constructeur de la classe `tk.Frame` (ce qui était inutile lorsqu'on héritait de la classe `Tk`).

Ligne 6. L'argument `racine` passé à la méthode `.__init__()` est finalement une variable locale. Comme il s'agit de l'instance de notre fenêtre principale à passer à tous nos *widgets*, il faut qu'elle soit visible dans toute la classe. La variable `self.racine` est ainsi créée afin d'être réutilisée dans d'autres méthodes.

Vous pourrez vous posez la question : « Pourquoi en ligne 4 l'argument par mot-clé `racine=None` prend la valeur `None` par défaut ? ». Et bien, c'est parce que notre classe Application peut s'appeler sans passer d'instance de fenêtre Tk. Voici un exemple avec les lignes qui changent seulement (tout le reste est identique au code précédent) :

```
1  [...]
2  class Application(tk.Frame):
3      def __init__(self, racine=None):
4          tk.Frame.__init__(self)
5          self.racine = racine
6          [...]
7  [...]
8  if __name__ == "__main__":
9      app = Application()
10     app.mainloop()
```

Dans un tel cas, l'argument `racine` prend la valeur par défaut `None` lorsque la méthode `.__init__()` de notre classe est exécutée. L'appel au constructeur de la classe Frame en ligne 4 instancie automatiquement une fenêtre Tk (car cela est strictement obligatoire). Dans la suite du programme, cette instance de la fenêtre principale sera `self.racine` et il n'y aura pas de changement par rapport à la version précédente. Cette méthode reste toutefois peu intuitive car cette instance de la fenêtre principale `self.racine` vaut finalement `None` !

20. http://effbot.org/tkinterbook/grid.htm
21. https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/grid.html
22. http://effbot.org/tkinterbook/place.htm

Hériter de la classe Frame ou de la classe Tk sont deux manières tout à fait valides pour créer des applications *Tkinter*. Le choix de l'une ou de l'autre relève plus de préférences que l'on acquiert en pratiquant, voire de convictions philosophiques sur la manière de programmer. Toutefois, nous pensons qu'hériter de la classe `tk.Tk` est une manière plus générale et plus compacte : tout ce qui concerne le fenêtrage *Tkinter* se situera dans votre classe Application, et le programme principal n'aura qu'à instancier l'application et à lancer le gestionnaire d'événements (les choses seront ainsi mieux « partitionnées »). C'est donc la méthode que nous vous recommandons.

25.7.2.4 Passage d'arguments avec `*args` et `**kwargs`

Si vous allez chercher de la documentation supplémentaire sur *Tkinter*, il se peut que vous tombiez sur ce style de syntaxe lorsque vous créez votre classe contenant l'application graphique :

```
class MonApplication(tk.Tk):
    def __init__(self, *args, **kwargs):
        tk.Tk.__init__(self, *args, **kwargs)
        # ici débute la construction de votre appli
[...]

# programme principal
if __name__ == "__main__":
    [...]
    app = MonApplication()
    [...]
```

Les arguments `*args` et `**kwargs` récupérent facilement tous les arguments « positionnels » et « par mot-clé ». Pour plus de détails sur comment `*args` et `**kwargs` fonctionnent, reportez-vous au chapitre 26 *Remarques complémentaires* (en ligne).

Dans l'exemple ci-dessus, `*args` et `**kwargs` sont inutiles car lors de l'instanciation de notre application, on ne passe aucun argument : `app = MonApplication()`. Toutefois, on pourrait être intéressé à récupérer des arguments passés au constructeur, par exemple :

```
app = MonApplication(arg1, arg2, option1=val1, option2=val2)
```

Ainsi certains auteurs laissent toujours ces `*args` et `**kwargs` au cas où on en ait besoin dans le futur. Cela est bien utile lorsqu'on distribue notre classe à la communauté et que l'on souhaite que les futurs utilisateurs puissent passer des arguments *Tkinter* au constructeur de notre classe.

Toutefois, même si cela « ne coûte rien », nous vous recommandons de ne pas mettre ces `*args` et `**kwargs` si vous n'en avez pas besoin, comme nous vous l'avons montré dans les exemples de ce chapitre. Rappelons nous de la PEP 20 (voir le chapitre 16 *Bonnes Pratiques en programmation Python*), les assertions « *Simple is better than complex* » ou « *Sparse is better than dense* » nous suggèrent qu'il est inutile d'ajouter des choses dont on ne se sert pas.

25.7.2.5 Toujours préciser l'instance de la fenêtre principale

Tkinter est parfois surprenant. Dans le code suivant, on pourrait penser que celui-ci n'est pas fonctionnel :

```
>>> import tkinter as tk
>>> bouton = tk.Button(text="Quitter")
>>> bouton.pack()
```

Pour autant, cela fonctionne et on voit un bouton apparaître ! En fait, *Tkinter* va automatiquement instancier la fenêtre principale, si bien qu'il n'est pas obligatoire de passer cette instance en argument d'un *widget*. À ce moment, on peut se demander où est passé cette instance. Heureusement, *Tkinter* garde toujours une filiation des *widgets* avec les attributs `.master` et `.children` :

```
>>> racine = bouton.master
>>> racine
<tkinter.Tk object .>
>>> racine.children
{'!button': <tkinter.Button object .!button>}
>>> bouton["command"] = racine.destroy
```

Ligne 1. On « récupère » l'instance de la fenêtre principale dans la variable racine.

Les lignes 4 et 5 montrent que le bouton est un « enfant » de cette dernière.

Enfin, ligne 6, on réassigne la destruction de la fenêtre lorsqu'on clique sur le bouton.

Ces attributs `.master` et `.children` existent pour tous *widgets* et sont bien pratiques lorsqu'on crée de grosses applications graphiques (où on utilise souvent des *widgets* parents contenant d'autres *widgets* enfants). Une autre source d'information sur les *widgets* se trouvent dans les méthodes dont le nom commence par `winfo`. Par exemple, la méthode `.winfo_toplevel()` renvoie la même information que l'attribut `.master` (une référence vers le *widget* parent).

Conseil

Même si cela est possible, nous vous conseillons de systématiquement préciser l'instance de la fenêtre principale lors de la création de vos *widgets*.

25.7.2.6 Passage d'arguments à vos fonctions callback

Comme vu dans nos exemples ci-dessus, les fonctions *callback* ne prennent pas d'arguments ce qui peut se révéler parfois limitant. Il existe toutefois une astuce qui utilise les fonctions *lambda* ; nous expliquons brièvement les fonctions lambda dans le chapitre 26 *Remarques complémentaires* (en ligne). Toutefois, nous ne développons pas leur utilisation avec *Tkinter* et les fonctions *callback* car cela dépasse le cadre de cet ouvrage. Pour de plus amples explications sur cette question, vous pouvez consulter le site pythonprogramming [23] et le livre de Gérard Swinnen [24].

25.7.2.7 Application *Tkinter* avec plusieurs pages

Dans ce chapitre d'introduction, nous vous avons montré des GUI simples avec une seule page. Toutefois, si votre projet se complexifie, il se peut que vous ayez besoin de créer plusieurs fenêtre différentes. Le livre de Gérard Swinnen [25] et le site pythonprogramming [26] sont des bonnes sources pour commencer et voir concrètement comment faire cela.

25.7.3 Bibliographie pour aller plus loin

Voici quelques ressources que vous pouvez utiliser pour continuer votre apprentissage de *Tkinter* :

1. En anglais :
- La Documentation officielle [27] de Python.
- Le manuel [28] de référence sur le site du MNT.
- Le site [29] de Fredrik Lundh est également très complet.
- Pour avoir un exemple [30] rapide de code pour chaque *widget*.
- Le livre [31] de David Love *Learn Tkinter By Example* qui montre des exemples concrets d'applications *Tkinter* de plus en plus complexes (pdf en libre téléchargement).
- Le site [32] très bien fait de Harisson (avec vidéos !) vous guidera dans la construction d'une GUI complète et complexe avec de nombreuses fonctions avancées (comme par exemple mettre des graphes matplotlib qui se mettent à jour dans la GUI !).

2. En français :
- Le site [33] bien complet d'Étienne Florent.
- Le livre [34] de Gérard Swinnen qui montre de nombreux exemples d'applications tkinter (pdf en libre téléchargement).

23. https://pythonprogramming.net/passing-functions-parameters-tkinter-using-lambda/
24. https://inforef.be/swi/python.htm
25. https://inforef.be/swi/python.htm
26. https://pythonprogramming.net/change-show-new-frame-tkinter/
27. https://wiki.python.org/moin/TkInter
28. https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html
29. http://effbot.org/tkinterbook/
30. https://www.tutorialspoint.com/python/python_gui_programming.htm
31. https://github.com/Dvlv/Tkinter-By-Example
32. https://pythonprogramming.net/tkinter-depth-tutorial-making-actual-program/
33. http://tkinter.fdex.eu/index.html
34. https://inforef.be/swi/python.htm

25.8 Exercices

Conseil

Pour ces exercices, créez des scripts puis exécutez-les dans un *shell*. Nous vous recommandons de concevoir une classe pour chaque exercice.

25.8.1 Application de base

Concevez une application qui affiche l'heure dans un *label* (par exemple `09:10:55`) et qui possède un boutton quitter. L'heure affichée sera celle au moment du lancement de l'application. Pour « attraper » l'heure, vous pourrez utiliser la fonction `strftime()` du module `time`.

25.8.2 Horloge

Sur la base de l'application précédente, faites une application qui affiche l'heure dans un *label* en se mettant à jour sur l'heure de l'ordinateur une fois par seconde. Vous concevrez une méthode `.mise_a_jour_heure()` qui met à jour l'heure dans le *label* et qui se rappelle elle-même toutes les secondes (n'oubliez pas la méthode `.after()`, cf. rubrique *Un canvas animé dans une classe* ci-dessus). Pour cette mise à jour, vous pourrez utiliser la méthode `.configure()`, par exemple : `self.label.configure(text=heure)` où `heure` est une chaîne de caractères représentant l'heure actuelle.

25.8.3 Compte à rebours

Créer une application affichant un compte à rebours dans un *label*. L'utilisateur choisira entre 1 et 240 minutes en passant un argument au lancement du script, par exemple : `python tk_compte_a_rebours.py 34` signifiera un compte à rebours de 34' (le programme vérifiera qu'il s'agit d'un entier entre 1 et 240 inclus). Il y aura un bouton « Lancer » pour démarrer le compte à rebours et un boutton « Quitter » au cas où on veuille quitter avant la fin. À la fin du rebours, le programme affichera 10 fois la phrase « C'est fini !!! » dans le *shell* et quittera automatiquement le script. Une image du résultat attendu est montrée dans la figure 25.7.

FIGURE 25.7 – Compte à rebours.

25.8.4 Triangle de Sierpinski

Le triangle de Sierpinski[35] est une fractale classique. On se propose ici de la dessiner avec un algorithme tiré du jeu du chaos[36]. Celui-ci se décompose en pseudo-code de la façon suivante :

35. https://fr.wikipedia.org/wiki/Triangle_de_Sierpi%C5%84ski
36. https://fr.wikipedia.org/wiki/Jeu_du_chaos

```
définir les 3 sommets d'un triangle isocèle ou équilatéral
point <- coordonnées (x, y) du centre du trianle
dessiner(point) # un pixel de large
pour i de 0 à 25000:
    sommet_tmp <- choisir un sommet du triangle au hasard
    point <- calculer(coordonnées(x, y) du centre entre point et sommet_tmp)
    dessiner(point)
```

Le rendu final attendu est montré dans la figure 25.8. On utilisera un canevas de 400x400 pixels. Il y a aura un bouton « Quitter » et un bouton « Launch ! » qui calculera et affichera 10000 points supplémentaires dans le triangle de Sierpinski.

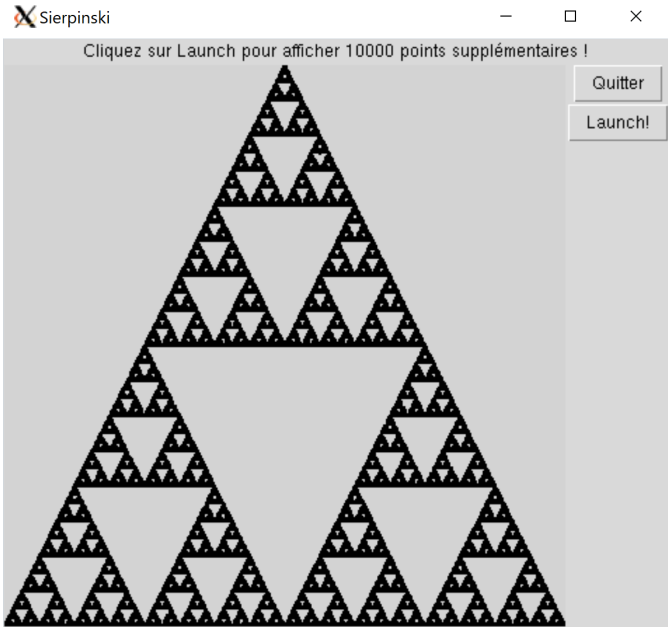


FIGURE 25.8 – Triangle de Sierpinski.

25.8.5 Polygone de Sierpinski (exercice +++)

Améliorer l'application précédente en proposant une liste de choix supplémentaire demandant à l'utilisateur de choisir le nombre de sommets (de 3 à 10). Le programme calculera automatiquement la position des sommets. Pour prendre en main le *widget Listbox*, voici un code minimal qui pourra vous aider. Celui-ci contient une `Listbox` et permet d'afficher dans le terminal l'élément sélectionné. Nous vous conseillons de bien étudier le code ci-dessous et d'avoir résolu l'exercice précédent avant de vous lancer !

```python
import tkinter as tk

class MaListBox(tk.Tk):
    def __init__(self):
        # Instanciation fenêtre Tk.
        tk.Tk.__init__(self)
        self.listbox = tk.Listbox(self, height=10, width=4)
        self.listbox.pack()
        # Ajout des items à la listbox (entiers).
        for i in range(1, 10+1):
            # Utilisation de ma méthode .insert(index, element)
            # Ajout de l'entier i (tk.END signifie en dernier).
            self.listbox.insert(tk.END, i)
        # Selection du premier élément de listbox.
        self.listbox.select_set(0)
        # Liaison d'une méthode quand clic sur listbox.
        self.listbox.bind("<<ListboxSelect>>", self.clic_listbox)

    def clic_listbox(self, event):
        # Récupération du widget à partir de l'objet event.
        widget = event.widget
        # Récupération du choix sélectionné dans la listbox (tuple).
        # Par exemple renvoie `(5,)` si on a cliqué sur `5`.
        selection = widget.curselection()
        # Récupération du nombre sélectionné (déjà un entier).
        choix_select = widget.get(selection[0])
        # Affichage.
        print(f"Le choix sélectionné est {choix_select}, "
              f"son type est {type(choix_select)}")



if __name__ == "__main__":
    app = MaListBox()
    app.title("MaListBox")
    app.mainloop()
```

25.8.6 Projet simulation d'un pendule

Vous souhaitez aller plus loin après ces exercices de « mise en jambe » ? Nous vous conseillons d'aller directement au chapitre 27 *Mini projets* (en ligne). Nous vous proposons de réaliser une application *Tkinter* qui simule le mouvement d'un pendule. En réalisant une application complète de ce genre, un peu plus conséquente, vous serez capable de construire vos propres applications.

CHAPITRE 26

Remarques complémentaires

Dans ce chapitre, nous présentons un certain nombre de points en vrac qui ne rentraient pas forcément dans les autres chapitres ou qui étaient trop avancés au moment où les chapitres étaient abordés. Outre quelques points mineurs, nous abordons les grandes différences entre Python 2 et Python 3, les anciennes méthodes de formatage des chaînes de caractères, les fonctions lambda, les itérateurs, la gestion des exceptions, les passage d'arguments avancés dans les fonctions et les décorateurs. Certains de ces points sont réellement avancés et nécessiteront d'avoir assimilé d'autres notions avant de les aborder.

26.1 Différences Python 2 et Python 3

Python 3 est la version de Python qu'il faut utiliser.

Néanmoins, Python 2 a été employé pendant de nombreuses années par la communauté scientifique et vous serez certainement confrontés à un programme écrit en Python 2. Voici quelques éléments pour vous en sortir :

26.1.1 Le mot-clé `print` / la fonction `print()`

En Python 2 `print` est un mot-clé du langage (en anglais *statement*) au même titre que `for`, `if`, `def`, etc. Il s'utilise ainsi sans parenthèse. Par exemple :

```
>>> print 12
12
>>> print "girafe"
girafe
```

Par contre, en Python 3, `print()` est une fonction. Ainsi, si vous n'utilisez pas de parenthèse, Python vous renverra une erreur :

```
>>> print 12
  File "<stdin>", line 1
    print 12
          ^
SyntaxError: Missing parentheses in call to 'print'
```

26.1.2 Division d'entiers

En Python 3, la division de deux entiers, se fait *naturellement*, c'est-à-dire que l'opérateur `/` renvoie systématiquement un *float*. Par exemple :

```
>>> 3 / 4
0.75
```

Il est également possible de réaliser une division entière avec l'opérateur `//` :

```
>>> 3 // 4
0
```

La division entière renvoie finalement la partie entière du nombre `0.75`, c'est-à-dire `0`.

Attention ! En Python 2, la division de deux entiers avec l'opérateur `/` correspond à la division entière, c'est-à-dire le résultat arrondi à l'entier inférieur. Par exemple :

```
>>> 3 / 5
0
>>> 4 / 3
1
```

Faites très attention à cet aspect si vous programmez encore en Python 2, c'est une source d'erreur récurrente.

26.1.3 La fonction `range()`

En Python 3, la fonction `range()` renvoie un objet de type *range* (voir les chapitres 5 *Boucles et comparaisons* et 14 *Conteneurs*) :

```
>>> range(3)
range(0, 3)
```

Comme on a vu au chapitre 5 *Boucles et comparaisons*, ces objets sont itérables produisant successivement les nombres `0`, puis `1` puis `2` sur notre exemple :

```
>>> for i in range(3):
...     print(i)
...
0
1
2
```

En Python 2, la fonction `range()` renvoie une liste. Par exemple :

```
>>> range(3)
[0, 1, 2]
>>> range(2, 6)
[2, 3, 4, 5]
```

La création de liste avec `range()` était pratique, mais très peu efficace en mémoire lorsque l'argument donné à `range()` était un grand nombre.

D'ailleurs la fonction `xrange()` est disponible en Python 2 pour faire la même chose que la fonction `range()` en Python 3. Attention, ne vous mélangez pas les pinceaux !

```
>>> range(3)
[0, 1, 2]
>>> xrange(3)
xrange(3)
```

Remarque

Pour générer une liste d'entiers avec la fonction `range()` en Python 3, vous avez vu dans le chapitre 4 *Listes* qu'il suffisait de l'associer avec la fonction `list()`. Par exemple :

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Conseil

Pour une comparaison exhaustive entre `xrange()` en Python 2 et `range()` en Python 3, vous pouvez lire ce très bon article[1] tiré du blog de Trey Hunner.

26.1.4 Fonction `zip()`

En Python 2, la fonction `zip()` renvoie une liste de tuples, alors qu'en Python 3 elle renvoie un **itérateur** :

```
>>> # Python2.
>>> zip(range(4), range(10, 14))
[(0, 10), (1, 11), (2, 12), (3, 13)]
```

```
>>> # Python3.
>>> zip(range(4), range(10, 14))
<zip object at 0x7f11423ffd80>
```

Vous pouvez lire la rubrique *Itérables, itérateurs, générateurs et module itertools* un peu plus bas dans ce chapitre pour en savoir plus sur les itérateurs.

26.1.5 Encodage et utf-8

En Python 3, vous pouvez utiliser des caractères accentués dans les commentaires ou dans les chaînes de caractères.

Ce n'est pas le cas en Python 2. Si un caractère accentué est présent dans votre code, cela occasionnera une erreur de ce type lors de l'exécution de votre script :

```
SyntaxError: Non-ASCII character '\xc2' in file xxx on line yyy, but no encoding
declared; see http://python.org/dev/peps/pep-0263/ for details
```

Pour éviter ce genre de désagrément, ajoutez la ligne suivante en tout début de votre script :

```
# coding: utf-8
```

Si vous utilisez un shebang (voir rubrique précédente), il faudra mettre la ligne `# coding: utf-8` sur la deuxième ligne (la position est importante[2]) de votre script :

```
#! /usr/bin/env python
# coding: utf-8
```

Remarque

L'encodage utf-8 peut aussi être déclaré de cette manière :

```
# -*- coding: utf-8 -*-
```

mais c'est un peu plus long à écrire.

26.2 Anciennes méthodes de formatage des chaînes de caractères

Dans les premières versions de Python jusqu'à la 2.6, il fallait utiliser l'opérateur `%`, puis de la version 2.7 jusqu'à la 3.5 il était plutôt conseillé d'utiliser la méthode `.format()`. Même si les *f-strings* sont devenues la manière conseillée pour mettre en place l'écriture formatée, ces deux anciennes manières, sont encore pleinement compatibles avec les versions modernes de Python.

Même si elle fonctionne encore, la première manière avec l'opérateur `%` est maintenant clairement déconseillée pour un certain nombre de raisons[3]. Néanmoins, nous rappelons ci-dessous son fonctionnement, car il se peut que vous tombiez dessus dans d'anciens livres ou si vous lisez de vieux programmes Python.

1. https://treyhunner.com/2018/02/python-3-s-range-better-than-python-2-s-xrange/
2. http://www.python.org/dev/peps/pep-0263/
3. https://docs.python.org/fr/3/library/stdtypes.html?highlight=sprintf#printf-style-string-formatting

La deuxième manière avec la méthode `.format()` est encore utilisée et reste tout à fait valide. Elle est clairement plus puissante et évite un certain nombre de désagréments par rapport à l'opérateur `%`. Vous la croiserez sans doute de temps en temps dans des programmes et ouvrages plus ou moins récents. Heureusement elle a un fonctionnement relativement proche des *f-strings*, donc vous ne serez pas totalement perdus !

Enfin, nous indiquons à la fin de cette rubrique nos conseils sur quelle méthode utiliser.

26.2.1 L'opérateur %

On a vu avec les entiers que l'opérateur `%` ou *modulo* renvoyait le reste d'une division entière. Cet opérateur existe aussi pour les chaînes de caractères mais il met en place l'écriture formatée. En voici un exemple :

```
>>> x = 32
>>> nom = "John"
>>> print("%s a %d ans" % (nom, x))
John a 32 ans
>>> nb_G = 4500
>>> nb_C = 2575
>>> prop_GC = (nb_G + nb_C)/14800
>>> print("On a %d G et %d C -> prop GC = %.2f" % (nb_G, nb_C, prop_GC))
On a 4500 G et 2575 C -> prop GC = 0.48
```

La syntaxe est légèrement différente. Le symbole `%` est d'abord appelé dans la chaîne de caractères (dans l'exemple ci-dessus `%d`, `%d` et `%.2f`) pour :

- Désigner l'endroit où sera placée la variable dans la chaîne de caractères.
- Préciser le type de variable à formater, `d` pour un entier (`i` fonctionne également) ou `f` pour un *float*.
- Éventuellement pour indiquer le format voulu. Ici `.2` signifie une précision de deux décimales.

Le signe `%` est rappelé une seconde fois (`% (nb_G, nb_C, prop_GC)`) pour indiquer les variables à formater.

26.2.2 La méthode `.format()`

Depuis la version 2.7 de Python, la méthode `.format()` a apporté une nette amélioration pour mettre en place l'écriture formatée. Celle-ci fonctionne de la manière suivante :

```
>>> x = 32
>>> nom = "John"
>>> print("{} a {} ans".format(nom, x))
John a 32 ans
>>> nb_G = 4500
>>> nb_C = 2575
>>> prop_GC = (nb_G + nb_C)/14800
>>> print("On a {} G et {} C -> prop GC = {:.2f}".format(nb_G, nb_C, prop_GC))
On a 4500 G et 2575 C -> prop GC = 0.48
```

- Dans la chaîne de caractères, les accolades vides `{}` précisent l'endroit où le contenu de la variable doit être inséré.
- Juste après la chaîne de caractères, l'instruction `.format(nom, x)` fournit la liste des variables à insérer, d'abord la variable `nom` puis la variable `x`.
- On peut éventuellement préciser le formatage en mettant un caractère deux-points `:` puis par exemple ici `.2f` qui signifie deux chiffres après la virgule.
- La méthode `.format()` agit sur la chaîne de caractères à laquelle elle est attachée par le point.

Tout ce que nous avons vu avec les *f-strings* sur la manière de formater l'affichage d'une variable (après les `:` au sein des accolades) est identique avec la méthode `.format()`. Par exemple `{:.2f}`, `{:0>6d}`, `{:.6e}`, etc., fonctionneront de la même manière. La différence notable est qu'on ne met pas directement le nom de la variable au sein des accolades. Comme pour l'opérateur `%`, c'est l'emplacement dans les arguments passés à la méthode `.format()` qui dicte quelle variable doit être remplacée. Par exemple, dans `"{} {} {}".format(bidule, machin, truc)`, les premières accolades remplaceront la variable `bidule`, les deuxièmes la variable `machin`, les troisièmes la variable `truc`.

Le formatage avec la méthode `.format()` se rapproche de la syntaxe des *f-strings* (accolades, deux-points), mais présente l'inconvénient – comme avec l'opérateur `%` – de devoir mettre la liste des variables tout à la fin, alourdissant ainsi la syntaxe. En effet, dans l'exemple avec la proportion de GC, la ligne équivalente avec une *f-string* apparait tout de même plus simple à lire :

```
>>> print(f"On a {nb_G} G et {nb_C} C -> prop GC = {prop_GC:.2f}")
On a 4500 G et 2575 C -> prop GC = 0.48
```

Conseil

Pour conclure, ces deux anciennes façons de formater une chaîne de caractères avec l'opérateur `%` ou la méthode `.format()` vous sont présentées à titre d'information. La première avec l'opérateur `%` est clairement déconseillée. La deuxième avec la méthode `.format()` est encore tout à fait valable. Si vous débutez Python, nous vous conseillons fortement d'apprendre et d'utiliser les *f-strings*. C'est ce que vous rencontrerez dans la suite de ce cours. Si vous connaissez déjà Python et que vous utilisez la méthode `.format()`, nous vous conseillons de passer aux *f-strings*. Depuis que nous les avons découvertes, aucun retour n'est envisageable pour nous tant elles sont puissantes et plus claires à utiliser !

Pour aller plus loin

Enfin, si vous souhaitez aller plus loin, voici deux articles (en anglais) très bien faits sur le site *RealPython* : sur l'écriture formatée[4] et sur les *f-strings*[5]

26.3 Fonctions lambda

26.3.1 Définition

Définition

Une **fonction lambda** est une fonction qui s'écrit sur une ligne. En Python, il s'agit du moyen d'implémenter une fonction anonyme[6] (en anglais *anonymous function*), c'est-à-dire, une fonction qui est la plupart du temps non reliée à un nom (d'où le terme anonyme). Une fonction lambda s'utilise en général à la volée. On parle aussi d'expressions lambda utilisées pour fabriquer des fonctions lambda.

Voici un premier exemple :

```
1  >>> lambda x: x**2
2  <function <lambda> at 0x7fcbd9c58b80>
3  >>> (lambda x: x**2)(4)
4  16
5  >>> (lambda x: x**2)(10)
6  100
```

- **Ligne 1.** On a ici une expression lambda typique définissant une fonction lambda. La syntaxe est (dans l'ordre) : le mot-clé (*statement*) lambda, zero ou un ou plusieurs argument(s), deux-points, une expression utilisant ou pas les arguments.
- **Ligne 2.** Python confirme qu'il s'agit d'une fonction.
- **Lignes 3 à 6.** Pour utiliser la fonction lambda, pour l'instant, on la met entre parenthèses et on utilise un autre jeu de parenthèses pour l'appeler et éventuellement passer des arguments.

Attention

Une fonction lambda ne s'écrit que sur une ligne. Si vous essayez de l'écrire sur plusieurs lignes, Python lèvera une exception `SyntaxError: invalid syntax`.

Comme pour les fonctions classiques, le nombre d'arguments est variable et doit être cohérent avec l'appel :

4. https://realpython.com/python-string-formatting
5. https://realpython.com/python-f-strings/
6. https://en.m.wikipedia.org/wiki/Anonymous_function

```
1  >>> (lambda: 1/2)()
2  0.5
3  >>> (lambda x, y: x + y)(1, 2)
4  3
5  >>> (lambda x, y: x + y)(4, 5)
6  9
7  >>> (lambda: 1/2)(5)
8  Traceback (most recent call last):
9    File "<stdin>", line 1, in <module>
10 TypeError: <lambda>() takes 0 positional arguments but 1 was given
11 >>> (lambda x, y: x + y)(4)
12 Traceback (most recent call last):
13   File "<stdin>", line 1, in <module>
14 TypeError: <lambda>() missing 1 required positional argument: 'y'
```

- **Ligne 1.** Fonction lambda à zéro argument.
- **Lignes 3 et 5.** Fonction lambda à deux arguments.
- **Lignes 7 à 10.** Le nombre d'argument(s) est incorrect et génère une erreur. Dans cet exemple, on passe un argument alors que la fonction lambda créée ici n'en prend pas.
- **Lignes 11 à 14.** Le nombre d'argument(s) est incorrect et génère une erreur. Dans cet exemple, on passe un argument alors que la fonction lambda créée ici en prend deux.

26.3.2 Assignation d'une fonction lambda à un nom ?

Bien que cela soit déconseillé, il est possible d'assigner une fonction lambda à un nom de variable :

```
1  >>> carre = lambda x: x**2
2  >>> carre(3)
3  9
```

L'équivalent avec une fonction classique serait :

```
1  >>> def carre(x):
2  ...     return x**2
3  ...
4  >>> carre(9)
5  81
```

Dans les deux cas l'appel est identique, mais la fonction lambda requière une syntaxe à une ligne lors de sa définition.

Même si on peut le faire, les développeurs déconseillent toutefois d'assigner une fonction lambda à un nom dans la PEP8[7]. Une des raisons est que si une erreur est générée, l'interpréteur ne renvoie pas le numéro de ligne dans la *Traceback* :

```
1  >>> inverse = lambda: 1/0
2  >>> inverse()
3  Traceback (most recent call last):
4    File "<stdin>", line 1, in <module>
5    File "<stdin>", line 1, in <lambda>
6  ZeroDivisionError: division by zero
```

Ligne 5. L'indication de la ligne pour l'erreur dans la fonction lambda (*line 1*) correspond à celle de l'appel et non pas de la définition.

Alors qu'avec une fonction classique :

```
>>> def inverse():
...     return 1/0
...
>>> inverse()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in inverse
ZeroDivisionError: division by zero
```

Ligne 5. Cette fois-ci, la *Traceback* indique bien la bonne ligne (*line 2*) dans la fonction.

7. https://peps.python.org/pep-0008/

Conseil

Pour cette raison, n'assignez pas une fonction lambda à un nom, mais utilisez la seulement à la volée (voir ci-dessous). Une autre raison est que cela peut nuire à la lisibilité. Si une fonction lambda s'écrit en une ligne, c'est bien pour qu'on puisse la lire quand elle est utilisée.

26.3.3 Utilité des fonctions lambda

Jusqu'à maintenant nous avons défini les fonctions lambda et montré ce qu'il ne fallait pas faire. Vous vous posez sans doute la question, mais à quoi servent-elles vraiment ? Nous vous montrons ici deux utilisations principales.

La première est qu'elles sont utiles pour implémenter des concepts de programmation fonctionnelle[8]. Dans ce paradigme de programmation, on cherchera à « emboiter » les fonctions les unes dans les autres. Nous avions déjà croisé cette idée avec la fonction `map()` dans le chapitre 11 *Plus sur les chaînes de caractères*. Celle-ci permet d'appliquer une fonction à tous les éléments d'un objet itérable. Par exemple, convertir en entier les différents éléments d'une chaîne de caractères :

```
1  >>> ligne = "3 5   -10"
2  >>> list(map(int, ligne.split()))
3  [3, 5, -10]
```

Ligne 2. On a converti l'objet *map* en liste pour voir ce qu'il contenait.

L'utilisation impliquant une fonction lambda permet par exemple d'appliquer une opération à tous les éléments d'une liste :

```
1  >>> liste1 = [3, 5, -10]
2  >>> list(map(lambda x: x**2, liste1))
3  [9, 25, 100]
4  >>> list(map(lambda x: 1/x, liste1))
5  [0.3333333333333333, 0.2, -0.1]
```

Lignes 2 et 4. La fonction lambda permet de lire clairement quelle opération on réalise plutôt que de se référer à une fonction classique se trouvant à un autre endroit. Ainsi, cela améliore la lisibilité.

Cela vous rappelle peut-être ce qu'on a rencontré avec les objets *NumPy* et les opérations vectorielles :

```
1  >>> import numpy as np
2  >>> array1 = np.arange(10)
3  >>> array1
4  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
5  >>> array1 * 2
6  array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
7  >>>
8  >>> liste1 = list(range(10))
9  >>> liste1
10 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
11 >>> list(map(lambda x: x*2, liste1))
12 [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Ligne 5. Nativement, l'opération `array1 * 2` se fait vectoriellement (élément par élément) avec un *array NumPy*.

Ligne 11. La fonction `map()` applique l'opération `* 2` de la lambda sur tous les éléments de la liste. Ainsi, on obtient le même effet que sur l'*array NumPy*.

Bien que cela s'avère pratique, nous verrons dans la rubrique suivante sur les itérateurs qu'il existe une syntaxe plus Pythonique avec les listes de compréhensions et les expressions génératrices.

La deuxième grande utilité des fonctions lambda concerne leur utilisation pour faire des tris puissants. Dans le chapitre 14 *Conteneurs*, nous avions vu les tris de dictionnaires par valeurs :

```
1  >>> dico = {"a": 15, "b": 5, "c":20}
2  >>> sorted(dico, key=dico.get)
3  ['b', 'a', 'c']
```

8. https://fr.wikipedia.org/wiki/Programmation_fonctionnelle

Ligne 2. On passe à l'argument par mot-clé `key` la *callback* `dico.get` (cette méthode renvoie initialement les valeurs d'un dictionnaire). Cela permet finalement de trier par ce que renvoie cette méthode, à savoir les valeurs.

Cet argument par mot-clé peut prendre d'autres *callback*, par exemple `len`. Dans l'exemple suivant, on prend 10 mots au hasard dans le dictionnaire et on les trie par leur longueur :

```
>>> mots = ['étudier', 'pie-grièche', 'figurerait', 'retraitait', 'allégerais',
            'distribuent', 'affilierait', 'ramassa', 'galettes', 'connu']
>>> sorted(mots, key=len)
['connu', 'étudier', 'ramassa', 'galettes', 'figurerait', 'retraitait',
'allégerais', 'pie-grièche', 'distribuent', 'affilierait']
```

Bien sûr, on peut utiliser aussi une fonction lambda. Celle-ci va nous permettre de passer une fonction de tri à la volée au moment de l'appel de la fonction `sorted()`. Par exemple, si on reprend le même exemple que le dictionnaire mais sous forme d'une liste de tuples :

```
liste1 = [('a', 15), ('b', 5), ('c', 20)]
```

Comment trier en fonction du deuxième élément de chaque tuple ? Réponse, avec une fonction lambda bien sûr ! Regardez :

```
>>> sorted(liste1, key=lambda x: x[1])
[('b', 5), ('a', 15), ('c', 20)]
```

Autre exemple, on souhaite trier une liste d'entiers aléatoires non pas par leur valeur, mais par le résultat de la fonction `x**2` :

```
>>> liste1 = [-5, 2, 5, 8, 6, 3, -9, 4, -10, 2]
>>> sorted(liste1, key=lambda x: x**2)
[2, 2, 3, 4, -5, 5, 6, 8, -9, -10]
```

Pour comprendre comment le tri est opéré en **ligne 3**, voici la liste initiale et une autre liste avec les carrés :

```
>>> liste1
[-5, 2, 5, 8, 6, 3, -9, 4, -10, 2]
>>> [x**2 for x in liste1]
[25, 4, 25, 64, 36, 9, 81, 16, 100, 4]
```

Le tri de `liste1` ci-dessus est bien effectué en fonction des valeurs montrées en **ligne 4**.

L'agument par mot-clé `key` existe dans d'autres fonctions ou méthodes. Bien sûr il existe dans la méthode `.sort()` qui trie les listes sur place. Mais aussi, dans les fonctions natives `min()` et `max()`. Enfin, on le croise dans la fonction `groupby()` du module itertools (voir rubrique suivante). Dans tous ces cas, on peut utiliser une fonction lambda pour l'argument `key`.

Par exemple, dans le code suivant :

```
>>> liste = ['baccalauréat', 'abaissera', 'barricadé', 'zouave', 'tabac',
             'typographie', 'dactylographes', 'éclipse']
>>> min(liste)
'abaissera'
>>> max(liste)
'éclipse'
>>> min(liste, key=lambda x: x.count("a"))
'éclipse'
>>> max(liste, key=lambda x: x.count("a"))
'baccalauréat'
```

- **Ligne 1.** On prend une liste de mots du dictionnaire.
- **Lignes 2 et 4**. Les fonctions `min()` et `max ()` considèrent l'ordre ASCII par défaut. Elles renvoient le premier et dernier élément de la liste après un tel tri.
- **Lignes 6 et 8**. Comprenez-vous la règle que nous avons utilisée avec la lambda ?

Regardons comment se passe le tri :

```
>>> liste.sort(key=lambda x: x.count("a"))
>>> liste
['éclipse', 'zouave', 'typographie', 'barricadé', 'tabac', 'dactylographes',
'abaissera', 'baccalauréat']
```

Vous l'aurez sans doute compris, avec notre fonction lambda, nous avons trié en fonction du nombre de lettres `a` dans chaque mot !

26.3.4 Conclusion

Nous avons vu que les fonctions lambda permettaient des définitions de fonction rapidement sur une ligne. Il faut absolument éviter de les assigner à un nom. Elles ont toute leur utilité lorsqu'on les utilise avec `map()` pour appliquer une opération à tous les éléments d'un conteneur, ou pour des tris puissants avec `sorted()`.

Conseil

Pour aller plus loin, vous pouvez consulter ces quelques articles : Dataquest [9], Trey Hunner [10], RealPython [11] et Dan Bader [12].

26.4 Itérables, itérateurs, générateurs et module itertools

26.4.1 Itérables et itérateurs

Dans le chapitre 14 *Conteneurs*, nous avons défini le mot **itérable** lorsque nous avions un objet de type conteneur sur lequel on pouvait itérer (comme les listes, tuples, dictionnaires, etc.). En général, nous le faisions avec une boucle `for`. Voyons ce qu'est maintenant un itérateur.

Définition

Un **itérateur** est un objet Python qui permet d'itérer sur une suite de valeurs avec la fonction `next()` jusqu'à temps qu'elles soient épuisées. Si on itère sur une partie des valeurs seulement, l'itérateur garde en mémoire là où il s'est arrêté. Si on le resollicite avec un `next()` il repartira de l'élément suivant. Une règle est toutefois importante : les valeurs ne peuvent être parcourues qu'une seule fois.

On peut générer un itérateur avec la fonction `iter()` à partir de n'importe quel conteneur :

```
>>> animaux = ["chien", "chat", "souris"]
>>> iterateur = iter(animaux)
>>> iterateur
<list_iterator object at 0x7f917e907a30>
```

Une fois l'itérateur généré, on peut accéder à l'élément suivant avec la fonction `next()` :

```
>>> next(iterateur)
'chien'
>>> next(iterateur)
'chat'
>>> next(iterateur)
'souris'
>>> next(iterateur)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Quand il n'y a plus de valeurs sur lesquelles itérer, la fonction `next()` lève une exception `StopIteration`. En général, on n'utilisera pas les itérateurs de cette manière, mais plutôt avec une boucle `for` ce qui évitera cette levée d'exception :

9. https://www.dataquest.io/blog/tutorial-lambda-functions-in-python/
10. https://www.pythonmorsels.com/lambda-expressions/
11. https://realpython.com/python-lambda/
12. https://dbader.org/blog/python-lambda-functions

```
>>> iterateur = iter(animaux)
>>> for elt in iterateur:
...     print(elt)
...
chien
chat
souris
```

On peut transformer un objet de type itérateur en un objet de type séquentiel, par exemple en tuple :

```
>>> iterateur = iter(animaux)
>>> tuple(iterateur)
('chien', 'chat', 'souris')
```

Le point important est qu'une fois toutes les valeurs parcourues, l'itérateur est épuisé et ne renvoie plus rien :

```
>>> iterateur = iter(animaux)
>>> tuple(iterateur)
('chien', 'chat', 'souris')
>>> tuple(iterateur)
()
```

Ainsi, on ne pourra parcourir l'ensemble des valeurs d'un itérateur qu'une fois.

À ce stade, on pourrait se dire que la construction d'un itérateur à partir d'une liste ci-dessus est inutile puisqu'on peut itérer directement sur la liste avec une boucle `for`. Toutefois, lorsqu'on réalise une telle boucle, il y a un itérateur qui est généré implicitement même si on ne s'en rend pas compte. Pour prouver cela, essayons la fonction `next()` avec une liste :

```
>>> next(animaux)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list' object is not an iterator
```

Ceci n'est pas possible car une liste n'est pas un itérateur. Alors pourquoi peut-on itérer dessus avec une boucle `for` ? Et bien, c'est parce que l'objet de type liste possède une méthode *dunder* spéciale nommée `.__iter__()`. Celle-ci génère un itérateur à partir d'elle-même permettant d'itérer sur ses éléments. L'objet itérateur ainsi généré possèdera une autre méthode *dunder* spéciale `.__next__()` permettant de passer à l'élément suivant lorsqu'on itère dessus.

Remarque

Pour rappel, les méthodes *dunder* des classes ont été définies dans la rubrique 24.2.2 *Méthodes magiques ou dunder methods* du chapitre 24 *Avoir plus la classe avec les objets*.

Lorsque vous construirez votre propre objet itérable, il faudra écrire une classe contenant ces deux méthodes *dunder* et l'objet sera *de facto* un itérateur et itérable. Pour vous donnez une première idée, voici une classe minimale créant un objet itérateur sur les lettres de l'alphabet :

```
class Alphabet:
    def __init__(self):
        self.current = 97 # ASCII code for a.

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > 122: # ASCII code for z.
            raise StopIteration
        letter = chr(self.current)
        self.current += 1
        return letter
```

- **Ligne 3**. Dans le constructeur, on crée un attribut d'instance `self.current` qui gardera l'état de l'itérateur. On l'initialise à 97 correspondant au code ASCII de la lettre `a`.

- **Lignes 5 et 6**. La méthode *dunder* `.__iter__()` est très simple à écrire. Elle renvoie `self` correspondant à l'itérateur lui-même. Si cette méthode n'est pas présente, l'objet n'est pas itérable.
- **Lignes 8 à 13**. La méthode *dunder* `.__next__()` s'occupe de passer à l'élément suivant et de garder une mémoire de là où l'itérateur est arrivé. Cela se passe en quatre étapes : i) levée d'une exception `StopIteration` si on est arrivé au bout, ii) détermination de la lettre actuelle, iii) incrémenter le `self.current` de 1 pour l'itération suivante et iv) retourner la lettre actuelle.

Si on sauve cette classe dans un fichier `iterator.py`, voici comment on pourrait l'utiliser :

```
>>> import iterator
>>> iter_alphabet = iterator.Alphabet()
>>> iter_alphabet
<iterator.Alphabet object at 0x7f308edc70b0>
>>> for lettre in iter_alphabet:
...     print(lettre)
...
a
b
[...]
y
z
>>> list(iter_alphabet)
[]
```

À nouveau, une fois l'itérateur épuisé, il ne renvoie plus rien. Bien sûr, cela représente un exemple très simple et la plupart du temps on créera ses propres classes itérateurs en implémentant de nombreuses fonctionnalités et méthodes supplémentaires. Pour créer un itérateur basique comme celui-ci sur l'alphabet, il est plus commode d'utiliser les générateurs (voir rubrique *Générateurs* ci-dessous).

Pour aller plus loin

Pour aller plus loin sur comment fonctionne les itérateurs, vous pouvez lire ces articles de Dan Bader [13], Trey Hunner [14] et du site *RealPython* [15]. Concernant la sémantique, cet article [16] de Trey Hunner explique pourquoi les objets *range* ne sont pas des itérateurs.

26.4.2 Autres fonctions *builtins* renvoyant des itérateurs

Dans les chapitres précédents, nous avons déjà croisé des itérateurs sans le savoir, car nous ne vous l'avons pas toujours précisé explicitement ! Dans le chapitre 5 *Boucles* avec la fonction `enumerate()`, dans le chapitre 11 *Plus sur les chaînes de caractères* avec la fonction `map()` et dans le chapitre 12 *Plus sur les listes* avec la fonction `zip()`. Ces trois fonctions renvoient des itérateurs qui sont épuisés une fois utilisés :

```
>>> animaux = ["chien", "chat", "souris"]
>>> obj_enum = enumerate(animaux)
>>> obj_enum
<enumerate object at 0x7f917ebf93a0>
>>> tuple(obj_enum)
((0, 'chien'), (1, 'chat'), (2, 'souris'))
>>> tuple(obj_enum)
()
```

13. https://dbader.org/blog/python-iterators
14. https://treyhunner.com/2018/06/how-to-make-an-iterator-in-python/
15. https://realpython.com/python-iterators-iterables/
16. https://treyhunner.com/2018/02/python-range-is-not-an-iterator/

```
>>> line = "9 11 25 92 49 98 62 72 63 74"
>>> obj_map = map(int, line.split())
>>> obj_map
<map object at 0x7f029e47b9a0>
>>> min(obj_map)
9
>>> list(obj_map)
[]
```

```
>>> obj_zip = zip(range(5), range(5, 10))
>>> list(obj_zip)
[(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]
>>> list(obj_zip)
[]
```

Lorsque ces fonctions avaient été évoquées, nous n'avions pas vu ce problème d'épuisement car elles étaient utilisées directement dans une boucle. Par exemple :

```
>>> for i, j in zip(range(5), range(5, 10)):
...     print(i, j)
...
0 5
1 6
2 7
3 8
4 9
```

Ainsi, l'itérateur était généré à chaque fois qu'on lançait la boucle et n'était utilisé qu'une seule fois.

Une derniere fonction renvoyant un itérateur qui existe nativement dans les fonctions *builtins* de Python est `reversed()`. Celle-ci prend en argument un objet de type séquence (liste, tuple, chaîne de caractère ou *range*) et renvoie un itérateur parcourant la séquence en sens inverse :

```
>>> reversed(range(5))
<range_iterator object at 0x7f8b34227780>
>>> rev_iterateur = reversed(range(5))
>>> for i in rev_iterateur:
...     print(i)
...
4
3
2
1
0
>>> list(rev_iterateur)
[]
```

Pour finir, examinons les propriétés des itérateurs que nous avions vues pour les conteneurs. Un objet itérateur est bien sûr iterable et ordonné, par contre il n'est pas indexable. Il ne supporte pas la fonction `len()`, supporte l'opérateur `in` et il est hachable.

```
>>> animaux = ["chien", "chat", "souris"]
>>> iterateur = iter(animaux)
>>> len(iterateur)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'list_iterator' has no len()
>>> iterateur[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'list_iterator' object is not subscriptable
>>> "chien" in iterateur
True
>>> hash(iterateur)
8741535406492
```

Attention

L'utilisation de l'opérateur `in` pour un test d'appartenance sur un itérateur épuise ce dernier (au même titre que l'utilisation de l'itérateur dans une boucle où avec la fonction `list()`) :

```
1  >>> line = "9 11 25 92 49 98 62 72 63 74"
2  >>> obj_map = map(int, line.split())
3  >>> 9 in obj_map
4  True
5  >>> 9 in obj_map
6  False
```

Ligne 3, on fait un premier test qui parcourt l'itérateur et renvoie `True`. Même si la valeur `9` était présente initialement, le deuxième test, **ligne 5**, renvoie `False` car l'itérateur est épuisé.

26.4.3 Module itertools

Il existe de nombreuses fonctions générant des itérateurs. Le module `itertools`[17] en est particulièrement riche. Nous n'allons pas faire une liste exhaustive du contenu de ce module, mais nous parlerons de quelques fonctions qui nous paraissent utiles, notamment `product()`[18]. Son fonctionnement fait penser au produit extérieur[19] (*outer product* en anglais) de l'algèbre tensorielle. Nous montrerons également la fonction `groupby()`[20] permettant de faire des regroupements puissants. Enfin, nous évoquerons rapidement les itérateurs infinis comme la fonction `count()` à la fin de la rubrique.

26.4.3.1 Fonction `product()`

La fonction `product()` prend (au moins) deux conteneurs en argument et génère toutes les combinaisons possibles d'association :

```
1  >>> import itertools
2  >>> predateurs = ["lion", "requin", "tigre"]
3  >>> proies = ["souris", "oiseau", "gazelle"]
4  >>> for pred, proie in itertools.product(predateurs, proies):
5  ...     print(pred, proie)
6  ...
7  lion souris
8  lion oiseau
9  lion gazelle
10 requin souris
11 requin oiseau
12 requin gazelle
13 tigre souris
14 tigre oiseau
15 tigre gazelle
```

Il est possible de passer plus de deux conteneurs à la fonction, par exemple :

```
1  >>> ma_liste = [1, 2]
2  >>> list(itertools.product(ma_liste, ma_liste, ma_liste))
3  [(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2), (2, 1, 1), (2, 1, 2), (2, 2, 1), (2, 2, 2)]
```

On a ici toutes les combinaisons possibles entre les trois objets `ma_liste` passés en argument.

Avec deux conteneurs en argument, cette fonction `product()` revient à faire une double boucle sur les deux conteneurs. Elle est donc particulièrement adaptée pour parcourir toutes les éléments d'un tableau. Par exemple, la commande suivante parcourera toutes les cases d'un échiquier :

17. https://docs.python.org/fr/3.12/library/itertools.html
18. https://docs.python.org/fr/3.12/library/itertools.html#itertools.product
19. https://en.wikipedia.org/wiki/Outer_product
20. https://docs.python.org/fr/3.12/library/itertools.html#itertools.groupby

```
>>> parcours_echiquier = itertools.product("abcdefgh", "12345678")
>>> parcours_echiquier
<itertools.product object at 0x7f192e412040>
>>> for col, ligne in parcours_echiquier:
...     print(col, ligne)
...
a 1
a 2
[...]
h 7
h 8
```

Mais attention, la fonction `product()` est un itérateur. Donc quand elle est épuisée, on ne peut plus l'utiliser :

```
>>> list(parcours_echiquier)
[]
```

Une utilisation particulièrement utile de `product()` en bioinformatique peut être de générer toutes les séquences d'ADN possibles (mots) de deux lettres :

```
>>> bases = "atgc"
>>> list(itertools.product(bases, bases))
[('a', 'a'), ('a', 't'), ('a', 'g'), ('a', 'c'), ('t', 'a'), ('t', 't'), ('t', 'g'),
 ('t', 'c'), ('g', 'a'), ('g', 't'), ('g', 'g'), ('g', 'c'), ('c', 'a'), ('c', 't'),
 ('c', 'g'), ('c', 'c')]
```

De même, `itertools.product(bases, bases, bases)` itérera sur tous les mots de trois lettres possibles. Ou encore, si on définit une chaîne de caractères contenant les vingt acides aminés comme suit `aas = "acdefghiklmnpqrstvwy"`, `itertools.product(aas, aas)` produira tous les dipeptides possibles.

26.4.3.2 Fonction `groupby()`

La fonction `groupby()` permet de faire des regroupements puissants. Pour vous montrer son fonctionnement, nous allons prendre un exemple. Nous partons d'une liste de mots que nous triions par longueur avec l'argument `key` auquel on passe la *callback* `len` (voir chapitre 12 *Plus sur les listes*) :

```
>>> mots = ["bar", "babar", "bam", "ba", "babababar", "barre", "bla", "barbare"]
>>> mots.sort(key=len)
>>> mots
['ba', 'bar', 'bam', 'bla', 'babar', 'barre', 'babababar', 'barbare']
```

La fonction `groupby()` crée un itérateur particulier :

```
>>> itertools.groupby(mots, key=len)
<itertools.groupby object at 0x7f467a6d0ca0>
>>> list(itertools.groupby(mots, key=len))
[(2, <itertools._grouper object at 0x7f467a8cf700>),
 (3, <itertools._grouper object at 0x7f467a58c0d0>),
 (5, <itertools._grouper object at 0x7f467a58c100>),
 (7, <itertools._grouper object at 0x7f467a58c040>)]
```

- **Lignes 1 et 3.** Il est important de passer à l'argument `key` la même fonction *callback* que lors du tri initial.
- **Lignes 4 à 7.** En transformant cet itérateur en liste, on voit qu'il génère une liste de tuples. Le premier élément de chaque tuple est un entier correspondant à une longueur de mot, le second élément est un itérateur. Que contient ce dernier ?

```
>>> for longueur, iterateur in itertools.groupby(mots, key=len):
...     print(longueur, list(iterateur))
...
2 ['ba']
3 ['bar', 'bam', 'bla']
5 ['babar', 'barre']
7 ['babababar', 'barbare']
```

Lignes 4 à 7. La conversion de cet itérateur en liste montre qu'il contient tous les mots de même longueur.

Comme vu dans une rubrique précédente, on peut passer une fonction lambda à l'argument `key` :

```
>>> mots.sort(key=lambda chaine: chaine.count("a"))
>>> mots
['ba', 'bar', 'bam', 'bla', 'barre', 'babar', 'barbare', 'bababar']
>>> itertools.groupby(mots, key=lambda chaine: chaine.count("a"))
<itertools.groupby object at 0x7f467a6d0ca0>
>>> list(itertools.groupby(mots, key=lambda chaine: chaine.count("a")))
[(1, <itertools._grouper object at 0x7f467a58c490>),
 (2, <itertools._grouper object at 0x7f467a58c100>),
 (3, <itertools._grouper object at 0x7f467a58c040>)]
>>> for nb_a, iterateur in itertools.groupby(mots, key=lambda chaine: chaine.count("a")):
...     print(nb_a, list(iterateur))
...
1 ['ba', 'bar', 'bam', 'bla', 'barre']
2 ['babar', 'barbare']
3 ['bababar']
```

Ici on a regroupé les mots suivant le nombre de lettres `a` qu'ils contiennent.

Conseil

Avant de faire un regroupement avec`groupby()`, pensez à trier la liste initiale avec `.sort()` ou `sorted()` en utilisant la même fonction (ou fonction lambda) passée à l'argument `key`.

Remarque

Il existe aussi une méthode `.groupby()` qui procède à des regroupements sur les *dataframes* pandas. Son mode de fonctionnement est assez différent par rapport à la fonction `groupby()` du module itertools. Vous pouvez consulter le chapitre 22 *Modules pandas* pour en savoir un peu plus.

26.4.4 Générateurs

Définition

Un **générateur** est un type d'itérateur particulier. On peut créer un générateur très facilement avec le mot-clé `yield` ou avec les expression génératrices (*generator expressions* en anglais) qui ont une syntaxe similaire à celle des listes de compréhension.

La création d'un générateur avec le mot-clé `yield` consiste à créer une fonction utilisant ce mot-clé. À partir de ce moment là, la fonction renvoie un générateur. Avant de voir un exemple, imaginons une fonction qui crée et renvoie une liste :

```
>>> def cree_alphabet():
...     alphabet = []
...     for i in range(97, 123):
...         alphabet.append(chr(i))
...     return alphabet
...
>>> alphabet = cree_alphabet()
>>> alphabet
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q',
 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

Pour créer un générateur équivalent, il suffira de remplacer le `.append()` par un `yield` et d'enlever le `return` :

```
>>> def alphabet_generator():
...     for i in range(97, 123):
...         yield chr(i)
...
>>> gen = alphabet_generator()
>>> gen
<generator object alphabet_generator at 0x7fe1dffe39f0>
>>> for lettre in gen:
...     print(lettre)
...
a
b
c
[...]
y
z
>>>
>>> list(gen)
[]
```

Comme pour tous les itérateurs, une fois tous les éléments parcourus le générateur est épuisé. Notez que le `yield` n'est pas une fonction mais un mot-clé, on n'utilise donc pas de parenthèses. Ce mot-clé `yield` n'a de sens que dans une fonction et ne s'utilise que pour créer des générateurs.

La technique avec une expression génératrice ressemble à la syntaxe des listes de compréhension (voir la rubrique *Listes de compréhension* du chapitre 12 *Plus sur les listes*), mais on l'entoure de parenthèses à la place des crochets :

```
>>> [n**2 for n in range(10)] # Liste de compréhension.
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> (n**2 for n in range(10)) # Expression génératrice.
<generator object <genexpr> at 0x7f917feb39f0>
>>> gen = (n**2 for n in range(10))
>>> for n in gen:
...     print(n)
...
0
[...]
81
```

À nouveau, le générateur est épuisé après avoir itéré dessus :

```
>>> for nb in gen:
...     print(nb)
...
>>>
```

Pour aller plus loin

Un générateur est un itérateur, mais l'inverse n'est pas vrai. Pour comprendre toutes les subtilités liées à cette comparaison, vous pouvez consulter cette page[21] sur le site *Datacamp*.

Conseil

Comme vous le voyez, créer un générateur est extrêmement aisé avec le mot-clé `yield` ou les expressions génératrices par rapport à l'écriture d'une classe itérateur (voir ci-dessus). Ainsi nous vous conseillons d'utiliser plutôt les générateurs lorsque vous souhaitez créer des itérateurs simples.

21. https://www.datacamp.com/tutorial/python-iterators-generators-tutorial

26.4.5 Pourquoi utiliser des itérateurs ?

À ce stade, vous vous posez peut-être la question « Pourquoi utiliser des itérateurs ? ». Nous donnons quelques réponses dans cette rubrique.

26.4.5.1 Consommation de resources optimisée

La première raison fondamentale est la consommation de resources. Lorsque vous créez un itérateur, Python ne va pas construire l'ensemble des éléments dans la mémoire, mais plutôt préparer la « moulinette » qui réalisera les itérations. Résultat, le processsus est très peu consommateur de mémoire même en créant un itérateur itérant sur un très grand nombre d'éléments. Par ailleurs, Python crée les éléments au fur et à mesure et à la demande. C'est pour cela qu'on parle parfois « d'évaluation paresseuse ou fainéante [22] » dans le sens où la valeur suivante d'un itérateur n'est pas pré-calculée mais plutôt évaluée quand on lui demande. Trey Hunner [23] parle ainsi d'objets itérables « paresseux ».

Un itérateur sera par ailleurs très rapide car en interne il fait appel à des routines optimisées en C. Mais aussi, l'utilisation de fonctions Python qui sont elles aussi optimisées (par exemple `sum()`) rend les itérateurs particulièrement efficaces.

Afin de quantifier cela, on propose de mesurer le temps d'exécution de trois petits morceaux de code faisant une somme de tous les entiers de 1 à 100000 (cent mille) avec un générateur, une boucle Python classique et une liste de compréhension. Pour faire une telle mesure, nous utilisons le module timeit [24] qui est particulièrement bien optimisé pour cela. Voici un exemple d'utilisation de `timeit` :

```
1  $ python -m timeit "sum(n**2 for n in range(100000))"
2  50 loops, best of 5: 3.76 msec per loop
```

On peut lancer `timeit` directement à la ligne de commande Unix avec l'option `-m` suivie de l'instruction Python à exécuter entre guillemets. Python va effectuer plusieurs fois l'instruction (ici 50 fois) et donnera une approximation au plus juste du temps d'exécution de celle-ci. Le nombre d'exécutions de l'instruction dépendra du temps pris par celle-ci et sera entièrement déterminé par Python.

En revenant à notre problématique, voici les résultats de notre somme de 1 à 100000 (testé sur un ordinateur portable relativement récent avec la version Python 3.12) :

```
$ python -m timeit "sum(n**2 for n in range(100000))"
50 loops, best of 5: 3.76 msec per loop
$ python -m timeit "somme=0" "for n in range(100000): somme += n**2"
50 loops, best of 5: 3.59 msec per loop
$ python -m timeit "sum([n**2 for n in range(100000)])"
50 loops, best of 5: 4.89 msec per loop
```

- **Ligne 1.** On utilise un générateur et la fonction `sum()` pour calculer cette somme. Notez que lorsqu'un générateur est utilisé dans une fonction, les parenthèses ne sont pas obligatoires. Cela simplifie la syntaxe par rapport à `sum((n**2 for n in range(nb)))`.
- **Ligne 3.** On utilise une boucle Python classique pour calculer cette somme. Notez que pour pouvoir utiliser `timeit` sur une ligne, on est obligé de passer deux arguments entre guillemets (initialisation de la variable `somme` et boucle).
- **Ligne 5.** On utilise une liste de compréhension pour calculer cette somme.

La méthode avec les générateurs est à peu près équivalente à l'utilisation d'une boucle classique où on accumule la somme, preuve que les deux méthodes sont bien optimisées. De manière spectaculaire, la liste de compréhension est bien plus lente (presque 1 ms de plus). Ceci vient du fait qu'il faut créer la liste de tous les éléments en mémoire, ce qui est contre-productif. Le générateur ou la boucle classique se contentent d'itérer et sont bien plus économes.

Dernier point, un test réalisé avec la version Python 3.13 sortie en octobre 2024 conduit aux mêmes observations.

26.4.5.2 Itérateurs infinis

Bien que la taille de la mémoire d'un ordinateur soit finie, il est possible de créer des itérateurs infinis ! Par exemple, la fonction `count()` [25] du module itertools itère de 0 (lorsqu'on l'appelle sans argument) jusqu'à l'infini :

22. https://en.m.wikipedia.org/wiki/Lazy_evaluation
23. https://treyhunner.com/2018/06/how-to-make-an-iterator-in-python/
24. https://docs.python.org/fr/3/library/timeit.html
25. https://docs.python.org/3/library/itertools.html#itertools.count

```
>>> iterateur = itertools.count()
>>> import itertools
>>> iterateur = itertools.count()
>>> for i in iterateur:
...       print(i)
...
0
1
2
3
[Boucle infinie]
```

Attention de ne pas transformer cet itérateur en liste ou tuple sous peine de saturer la mémoire de l'ordinateur et de le faire planter !

Dans le même module les fonctions `cycle()` [26] et `repeat()` [27] sont également des itérateurs infinis.

26.4.5.3 Meilleure lisibilité

De manière générale, l'utilisation d'itérateurs peut améliorer la lisibilité de vos programmes. Cet article [28] fait remarquer que le simple fait de créer un itérateur et de le nommer donne un sens à ce qu'il contient. En reprenant notre exemple sur la somme des carrés :

```
tous_les_carres = (n**2 for n in range(nb))
somme = sum(tous_les_carres)
```

Si on compare à la boucle `for` :

```
somme = 0
for n in range(nb):
    somme += n**2
```

On voit que ce que représente l'objet `tous_les_carres` n'existe tout simplement pas avec la boucle `for` ! Par ailleurs, outre l'avantage de rapidité, l'utilisation de la fonction `sum()` rend la lecture très claire.

Dernier point, les itérateurs et notamment les générateurs, donnent un moyen de faire de la programmation fonctionnelle [29] en Python. Sans rentrer dans les considérations théoriques, nous avons déjà vu l'idée générale lorsque nous avons abordé le *method chaining* sur les chaînes de caractères ou sur les *dataframes* pandas. Initialement, la programmation fonctionnelle en Python utilisait la fonction `map()` (ainsi que les fonctions `filter()` et `reduce()` non abordées ici). Mais depuis l'arrivée des générateurs, on préfère ces derniers qui sont considérés plus Pythoniques. Regardons un exemple où nous transformons une chaîne de caractères en entiers puis nous calculons la somme. D'abord avec un générateur :

```
>>> ligne = "9 11 25 92 49 98 62 72 63 74"
>>> sum(int(nb) for nb in ligne.split())
555
>>>
```

Ensuite avec la fonction `map()` :

```
>>> line = "9 11 25 92 49 98 62 72 63 74"
>>> sum(map(int, line.split()))
555
```

Ne trouvez-vous pas que la version avec le générateur est plus lisible ?

Comme proposé par Dan Bader [30], on peut chainer les générateurs :

26. https://docs.python.org/3/library/itertools.html#itertools.cycle
27. https://docs.python.org/3/library/itertools.html#itertools.repeat
28. https://treyhunner.com/2019/06/loop-better-a-deeper-look-at-iteration-in-python/#How_iterators_can_improve_your_code
29. https://fr.wikipedia.org/wiki/Programmation_fonctionnelle
30. https://dbader.org/blog/python-iterator-chains

```
>>> import math
>>> ligne = "9 11 25 92 49 98 62 72 63 74"
>>> nombres = (int(nb) for nb in ligne.split())
>>> inverses = (nb**-1 for nb in nombres)
>>> cos_inverses = (math.cos(nb) for nb in inverses)
>>> sum(cos_inverse)
9.988141056338993
```

Une chose à noter dans cet exemple est que lorsqu'on crée un générateur à partir d'un autre générateur, le générateur initial n'est pas déclenché. Par exemple, en **Ligne 4** pour `inverses` le générateur `nombres` n'est pas encore déclenché, ou en **Ligne 5** pour `cos_inverses` le générateur `inverses` n'est pas déclenché non plus. Tous les générateurs seront déclenchés en chaine lorsqu'on exécutera la **Ligne 6**.

Conseil

En écrivant un générateur par ligne, le code est bien lisible. Evitez une syntaxe en une ligne qui s'avérera illisible : `(math.cos(nb) for nb in (nb**-1 for nb in (int(nb) for nb in ligne.split())))`

26.5 Gestion des exceptions

Les langages de programmation comme Python contiennent un système de gestion des **exceptions**[31]. Qu'est-ce qu'une exception ? Sur la page anglaise de Wikipedia[32], une exception est définie comme une anomalie de l'exécution d'un programme requérant une action spéciale, en général l'arrêt de l'exécution. Le plus souvent, **une exception correspond à une erreur** que Python rencontre lorsqu'il tente d'exécuter les lignes de code qu'on lui soumet. Par exemple, un problème de syntaxe, une variable ou objet qui prend une valeur aberrante (par exemple diviser par 0, parcourir une liste au-delà du nombre d'éléments, etc.).

Le système de gestion des exceptions évite que votre programme « plante » en prévoyant vous-même les sources d'erreurs éventuelles.

Voici un exemple dans lequel on demande à l'utilisateur d'entrer un nombre entier, puis on affiche ce nombre.

```
>>> nb = int(input("Entrez un nombre entier : "))
Entrez un nombre entier : 23
>>> print(nb)
23
```

La fonction `input()` demande à l'utilisateur de saisir une chaîne de caractères. Cette chaîne de caractères est ensuite transformée en nombre entier avec la fonction `int()`.

Si l'utilisateur ne rentre pas un nombre, voici ce qui se passe :

```
>>> nb = int(input("Entrez un nombre entier : "))
Entrez un nombre entier : ATCG
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'ATCG'
```

L'erreur provient de la fonction `int()` qui n'a pas pu convertir la chaîne de caractères `"ATCG"` en nombre entier, ce qui est parfaitement normal. En termes plus techniques, on dira que « Python a levé une exception de type `ValueError` ». Eh oui il y a de nombreux types d'exceptions différents (voir plus bas) ! Le nom de l'exception apparaît toujours comme le premier mot de la dernière ligne du message d'erreur. Si nous lancions ces lignes de code sous forme de script (du style `python script.py`), cet exemple conduirait à l'arrêt de l'exécution du programme.

Le jeu d'instructions `try` / `except` permet de tester l'exécution d'une commande et d'intervenir en cas de levée d'exception.

31. https://fr.wikipedia.org/wiki/Syst%C3%A8me_de_gestion_d%27exceptions
32. https://en.wikipedia.org/wiki/Exception_handling

```
>>> try:
...     nb = int(input("Entrez un nombre entier : "))
... except:
...     print("Vous n'avez pas entré un nombre entier !")
...
Entrez un nombre entier : ATCG
Vous n'avez pas entré un nombre entier !
```

Dans cet exemple, l'exception levée par la fonction `int()` (qui ne peut pas convertir `"ATCG"` en nombre entier) est interceptée et déclenche l'affichage du message d'avertissement.

On peut ainsi redemander sans cesse un nombre entier à l'utilisateur, jusqu'à ce que celui-ci en rentre bien un.

```
>>> while True:
...     try:
...         nb = int(input("Entrez un nombre entier : "))
...         print("Le nombre est", nb)
...         break
...     except:
...         print("Vous n'avez pas entré un nombre entier !")
...         print("Essayez encore")
...
Entrez un nombre entier : ATCG
Vous n'avez pas entré un nombre entier !
Essayez encore
Entrez un nombre entier : toto
Vous n'avez pas entré un nombre entier !
Essayez encore
Entrez un nombre entier : 3.2
Vous n'avez pas entré un nombre entier !
Essayez encore
Entrez un nombre entier : 55
Le nombre est 55
```

Notez que dans cet exemple, l'instruction `while True` est une boucle infinie car la condition `True` est toujours vérifiée. L'arrêt de cette boucle est alors forcé par la commande `break` lorsque l'utilisateur a effectivement entré un nombre entier.

La gestion des exceptions est très utile dès lors que des données extérieures entrent dans un programme Python, que ce soit directement par l'utilisateur (avec la fonction `input()`) ou par des fichiers. Cela est fondamental si vous distribuez votre code à la communauté : si les utilisateurs ne connaissent pas Python, un message comme `Vous n'avez pas entré un nombre entier !` reste plus clair que `ValueError: invalid literal for int() with base 10: 'ATCG'`.

Vous pouvez par exemple vérifier qu'un fichier a bien été ouvert.

```
>>> nom = "toto.pdb"
>>> try:
...     with open(nom, "r") as fichier:
...         for ligne in fichier:
...             print(ligne)
... except:
...     print("Impossible d'ouvrir le fichier", nom)
```

Si une erreur est déclenchée, c'est sans doute que le fichier n'existe pas à l'emplacement indiqué sur le disque ou que vous n'avez pas les droits pour le lire.

Il est également possible de spécifier le type d'erreur à gérer. Le premier exemple que nous avons étudié peut s'écrire :

```
>>> try:
...     nb = int(input("Entrez un nombre entier : "))
... except ValueError:
...     print("Vous n'avez pas entré un nombre entier !")
...
Entrez un nombre entier : ATCG
Vous n'avez pas entré un nombre entier !
```

Ici, on intercepte une exception de type `ValueError`, ce qui correspond bien à un problème de conversion avec `int()`.

Attention, si vous précisez le type d'exception comme `ValueError`, le `except ValueError` n'empêchera pas la levée d'une autre exception.

```
>>> try:
...     nb = int(variable)
... except ValueError:
...     print("Vous n'avez pas entré un nombre entier !")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'variable' is not defined. Did you mean: 'callable'?
```

Ici l'exception levée est de type `NameError`, car `variable` n'existe pas. Alors que si vous mettez `except` tout court, cela intercepte n'importe quelle exception.

```
>>> try:
...     nb = int(variable)
... except:
...     print("Vous n'avez pas entré un nombre entier !")
...
Vous n'avez pas entré un nombre entier !
>>>
```

Vous voyez qu'ici cela pose un nouveau problème : le message d'erreur ne correspond pas à l'exception levée !

Conseil

- Nous vous conseillons vivement de toujours préciser le type d'exception dans vos `except`. Cela évite d'intercepter une exception que vous n'aviez pas prévue. Il est possible d'intercepter plusieurs types d'exceptions en passant un tuple à `except`, par exemple : `except (Exception1, Exception2)`.
- Par ailleurs, ne mettez pas trop de lignes dans le bloc du `try`. Dans un tel cas, il peut être très pénible de trouver la ligne qui a conduit à l'exécution du `except`. Pire encore, il se peut que des lignes que vous aviez prévues ne soient pas exécutées ! Donc gardez des choses simples dans un premier temps, comme par exemple tester les conversions de type ou vérifier qu'un fichier existe bien et que vous pouvez l'ouvrir.

Il existe de nombreux types d'exception comme `RuntimeError`, `TypeError`, `NameError`, `IOError`, etc. Vous pouvez aller voir la liste complète[33] sur le site de Python. Nous avions déjà croisé des noms d'exception au chapitre 23 (*Avoir la classe avec les objets*) en regardant ce que contient le module `builtins`.

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
[...]
'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError'
[...]
```

Leur présence dans le module `builtins` signifie qu'elles font partie du langage lui même, au même titre que les fonctions de base comme `range()`, `list()`, etc.

Avez-vous aussi remarqué que leur nom commence toujours par une majuscule et qu'il peut en contenir plusieurs à la façon *CamelCase* ? Si vous avez bien lu le chapitre 16 *Bonnes pratiques en programmation Python*, avez-vous deviné pourquoi ? Et bien, c'est parce que **les exceptions sont des classes**. C'est très intéressant car il est ainsi possible d'utiliser l'héritage pour créer ses propres exceptions à partir d'exceptions pré-existantes. Nous ne développerons pas cet aspect, mais en guise d'illustration, regardez ce que renvoit un `help()` de l'exception `OverflowError`.

33. https://docs.python.org/fr/3.12/library/exceptions.html#exceptions.TypeError

```
>>> help(OverflowError)
[...]
class OverflowError(ArithmeticError)
 |  Result too large to be represented.
 |
 |  Method resolution order:
 |      OverflowError
 |      ArithmeticError
 |      Exception
 |      BaseException
 |      object
```

L'exception `OverflowError` hérite de `ArithmeticError`, c'est-à-dire qu'`OverflowError` a été conçue à partir de `ArithmeticError` et en hérite de tous ses attributs.

Un autre aspect très important que nous avons croisé au chapitre 24 *Avoir plus la classe avec les objets* est la possibilité de lever vous-même une exception avec le mot-clé `raise`. Nous avions vu le code suivant :

```
if valeur < 0:
    raise ValueError("Z'avez déjà vu une masse négative ?")
```

La ligne 2 lève une exception `ValueError` lorsque la variable `valeur` est négative. L'instruction `raise` est bien pratique lorsque vous souhaitez stopper l'exécution d'un programme si une variable ne se trouve pas dans le bon intervalle ou ne contient pas la bonne valeur. Vous avez sans doute compris maintenant pourquoi on parlait de « levée » d'exception…

Enfin, on peut aussi être très précis dans le message d'erreur. Observez la fonction `download_page()` qui, avec le module *urllib*, télécharge un fichier sur internet.

```
import urllib.request

def download_page(address):
    error = ""
    page = ""
    try:
        data = urllib.request.urlopen(address)
        page = data.read()
    except IOError as e:
        if hasattr(e, 'reason'):
            error =  "Cannot reach web server: " + str(e.reason)
        if hasattr(e, 'code'):
            error = f"Server failed {e.code:d}"
    return page, error

data, error = download_page("https://files.rcsb.org/download/1BTA.pdb")

if error:
    print(f"Erreur rencontrée : {error}")
else:
    with open("proteine.pdb", "w") as prot:
        prot.write(data.decode("utf-8"))
    print("Protéine enregistrée")
```

La variable `e` est une instance de l'exception `IOError`. Certains de ses attributs sont testés avec la fonction `hasattr()` pour ainsi affiner le message renvoyé (ici contenu dans la variable `error`).

Si tout se passe bien, la page est téléchargée est stockée dans la variable `data`, puis enregistrée sur le disque dur.

26.6 Shebang et /usr/bin/env python3

Lorsque l'on programme sur un système Unix (Mac OS X ou Linux par exemple), on peut exécuter directement un script Python, sans appeler explicitement la commande `python`.

Pour cela, deux opérations sont nécessaires :

Étape 1. Préciser la localisation de l'interpréteur Python en indiquant dans la première ligne du script :

```
#! /usr/bin/env python
```

Par exemple, si le script *test.py* contenait :

```
1  print("Hello World !")
```

il va alors contenir :

```
1  #!/usr/bin/env python
2
3  print("Hello World !")
```

Étape 2.. Rendre le script Python exécutable en lançant l'instruction :

```
$ chmod +x test.py
```

Remarque

La ligne `#! /usr/bin/env python` n'est pas considérée comme un commentaire par Python, ni comme une instruction Python d'ailleurs . Cette ligne a une signification particulière pour le système d'exploitation Unix.

Pour exécuter le script, il suffit alors de taper son nom précédé des deux caractères **./** (afin de préciser au *shell* où se trouve le script) :

```
$ ./test.py
Hello World !
```

Définition

Le **shebang**[34] correspond aux caractères `#!` qui se trouvent au début de la première ligne du script `test`.

Le *shebang* est suivi du chemin complet du programme qui interprète le script ou du programme qui sait où se trouve l'interpréteur Python. Dans l'exemple précédent, c'est le programme `/usr/bin/env` qui indique où se trouve l'interpréteur Python.

26.7 Passage d'arguments avec *args et **kwargs

Avant de lire cette rubrique, nous vous conseillons de bien relire et maîtriser la rubrique *Arguments positionnels et arguments par mot-clé* du chapitre 10 *Fonctions*.

Dans le chapitre 10, nous avons vu qu'il était nécessaire de passer à une fonction tous les arguments positionnels définis dans celle-ci. Il existe toutefois une astuce permettant de passer un nombre arbitraire d'arguments positionnels :

```
1  >>> def fct(*args):
2  ...     print(args)
3  ...
4  >>> fct()
5  ()
6  >>> fct(1)
7  (1,)
8  >>> fct(1, 2, 5, "Python")
9  (1, 2, 5, 'Python')
10 >>> fct(z=1)
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13 TypeError: fct() got an unexpected keyword argument 'z'
```

L'utilisation de la syntaxe `*args` permet d'empaqueter tous les arguments positionnels dans un *tuple* unique `args` récupéré au sein de la fonction. L'avantage est que nous pouvons passer autant d'arguments positionnels que l'on veut. Toutefois, on s'aperçoit en ligne 10 que cette syntaxe ne fonctionne pas avec les arguments par mot-clé.

Il existe un équivalent avec les arguments par mot-clé :

34. http://fr.wikipedia.org/wiki/Shebang

```
>>> def fct(**kwargs):
...     print(kwargs)
...
>>> fct()
{}
>>> fct(z=1, gogo="toto")
{'gogo': 'toto', 'z': 1}
>>> fct(z=1, gogo="toto", y=-67)
{'y': -67, 'gogo': 'toto', 'z': 1}
>>> fct(1, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: fct() takes 0 positional arguments but 2 were given
```

La syntaxe `**kwargs` permet d'empaqueter l'ensemble des arguments par mot-clé, quel que soit leur nombre, dans un dictionnaire unique `kwargs` récupéré dans la fonction. Les clés et valeurs de celui-ci sont les noms des arguments et les valeurs passées à la fonction. Toutefois, on s'aperçoit en ligne 9 que cette syntaxe ne fonctionne pas avec les arguments positionnels.

Si on attend un mélange d'arguments positionnels et par mot-clé, on peut utiliser `*args` et `**kwargs` en même temps :

```
>>> def fct(*args, **kwargs):
...     print(args)
...     print(kwargs)
...
>>> fct()
()
{}
>>> fct(1, 2)
(1, 2)
{}
>>> fct(z=1, y=2)
()
{'y': 2, 'z': 1}
>>> fct(1, 2, 3, z=1, y=2)
(1, 2, 3)
{'y': 2, 'z': 1}
```

Deux contraintes sont toutefois à respecter. Il faut toujours :

- mettre `*args` avant `**kwargs` dans la définition de la fonction ;
- passer les arguments positionnels avant ceux par mot-clé lors de l'appel de la fonction.

Il est possible de combiner des arguments positionnels avec `*args` et des arguments par mot-clé avec `**kwargs`, par exemple :

```
def fct(a, b, *args, **kwargs):
```

Dans un tel cas, il faudra obligatoirement passer les deux arguments `a` et `b` à la fonction, ensuite on pourra mettre un nombre arbitraire d'arguments positionnels (récupérés dans le tuple `args`), puis un nombre arbitraire d'arguments par mot-clé (récupérés dans le dictionnaire `kwargs`).

Conseil

Les noms `*args` et `**kwargs` sont des conventions en Python, ils rappellent les mots *arguments* et *keyword arguments*. Bien qu'on puisse mettre ce que l'on veut, nous vous conseillons de respecter ces conventions pour faciliter la lecture de votre code par d'autres personnes.

L'utilisation de la syntaxe `*args` et `**kwargs` est très classique dans le module *Fenêtres graphiques et Tkinter* présenté dans le chapitre 25 (en ligne).

Il est possible d'utiliser ce mécanisme d'empaquetage / désempaquetage (*packing* / *unpacking*) dans l'autre sens :

```
>>> def fct(a, b, c):
...     print(a,b,c)
...
>>> t = (-5,6,7)
>>>
>>> fct(*t)
-5 6 7
```

Avec la syntaxe `*t` on désempaquette le tuple à la volée lors de l'appel à la fonction. Cela est aussi possible avec un dictionnaire :

```
>>> def fct(x, y, z):
...     print(x, y, z)
...
>>> dico = {'x': -1, 'y': -2, 'z': -3}
>>> fct(**dico)
-1 -2 -3
```

Attention toutefois à bien respecter deux choses :

- la concordance entre le nom des clés du dictionnaire et le nom des arguments dans la fonction (sinon cela renvoie une erreur) ;
- l'utilisation d'une double étoile pour désempaqueter les valeurs du dictionnaire (si vous utilisez une seule étoile, Python désempaquettera les clés !).

Ce mécanisme de désempaquetage est aussi utilisable avec les objets *zip*, on parle de *zip unpacking*. Souvenons-nous, un objet *zip* permettait d'assembler plusieurs listes, éléments par éléments (voir Chapitre 12 *Plus sur les listes*) :

```
>>> animaux = ["poulain", "renard", "python"]
>>> couleurs = ["alezan", "roux", "vert"]
>>> zip(range(3), animaux, couleurs)
<zip object at 0x7f333febc880>
>>> triplets = list(zip(range(3), animaux, couleurs))
>>> triplets
[(0, 'poulain', 'alezan'), (1, 'renard', 'roux'), (2, 'python', 'vert')]
```

Lignes 1 à 4. On crée un objet *zip* avec trois objets de trois éléments.

Lignes 5 à 7. Cet objet *zip* en conjonction avec la fonction `list()` nous permet d'associer les éléments par ordre d'apparition (tous les éléments à la position 1 se retrouve ensemble, idem pour les positions 2 et 3). Au final, l'objet `triplets` est une liste de *tuples* de trois éléments.

L'opérateur `*` en combinaison avec la fonction *zip* va nous permettre de désempaqueter `triplets` pour récupérer les listes initiales (`range(3)`, `animaux` et `couleurs`) :

```
>>> zip(*triplets)
<zip object at 0x7f333fd44980>
>>> list(zip(*triplets))
[(0, 1, 2), ('poulain', 'renard', 'python'), ('alezan', 'roux', 'vert')]
```

Bien sûr, on peut l'utiliser l'affectation multiple :

```
>>> numéros2, animaux2, couleurs2 = zip(*triplets)
>>> numéros2
(0, 1, 2)
>>> animaux2
('poulain', 'renard', 'python')
>>> couleurs2
('alezan', 'roux', 'vert')
```

Au final, on récupère des *tuples* au lieu des listes initiales. Mais à ce stade, vous devriez être capable de les retransformer en liste ;-).

26.8 Décorateurs

Dans le chapitre 24, nous avons rencontré la notion de décorateur pour déclarer des objets de type *property*. Cela permettait de rendre des méthodes accessibles comme des attributs (décorateur `@property`), et plus généralement de

contrôler l'accès, la modification et la destruction d'attributs (décorateurs `@nom_attribut.setter` et `@nom_attribut.deleter`). Il existe d'autres décorateurs prédéfinis en Python (e.g. `@staticmethod`, `@classmethod`, etc.). Nous allons voir dans cette section comment on crée ses propres décorateurs et les mécanismes sous-jacents. Nous vous conseillons de bien relire commment fonctionne les fonctions de rappel, ou fonctions *callback* (chapitre 25 *Tkinter*).

Définition

Un décorateur est une fonction qui modifie le comportement d'une autre fonction.

Ceci étant dit, comme cela fonctionne-t-il ? Commençons par une fonction simple qui affiche de la nourriture :

```
def imprime_victuaille():
    print("tomate / mozza")
```

On souhaite améliorer cette fonction et transformer cette victuaille en sandwich, en affichant une tranche de pain avant et après. La stratégie va être de créer une fonction spéciale, qu'on appelle **décorateur**, modifiant `imprime_victuaille()`.

```
def transforme_en_sandwich(fonction_a_decorer):
    def emballage():
        print("Pain")
        fonction_a_decorer()
        print("Pain")
    return emballage
```

La fonction `transforme_en_sandwich()` est notre décorateur, elle prend en argument la fonction que l'on souhaite décorer sous forme de *callback* (donc sans les parenthèses). On voit qu'à l'intérieur, on définit une sous-fonction `emballage()` qui va littéralement « emballer » (*wrap*) notre fonction à décorer, c'est-à-dire, effectuer une action avant et après l'appel de la fonction à décorer. Enfin, le décorateur renvoie cette sous-fonction `emballage` sous forme de *callback*. Pour que le décorateur soit actif, il faudra « transformer » la fonction à décorer avec notre fonction décoratrice :

```
imprime_victuaille = transforme_en_sandwich(imprime_victuaille)
```

Voici le code complet implémentant la fonction `imprime_victuaille()` décorée :

```
def transforme_en_sandwich(fonction_a_decorer):
    def emballage():
        print("Pain")
        fonction_a_decorer()
        print("Pain")
    return emballage

def imprime_victuaille():
    print("tomate/ mozza")

if __name__ == "__main__":
    print("Fonction non décorée:")
    imprime_victuaille()
    print()
    print("Fonction décorée:")
    imprime_victuaille = transforme_en_sandwich(imprime_victuaille)
    imprime_victuaille()
```

Au final l'idée est d'appeler la fonction décoratrice plutôt que la fonction `imprime_victuaille()` elle-même. Regardons ce que donne l'exécution de la fonction avant et après décoration :

```
Fonction non décorée:
tomate/ mozza

Fonction décorée:
Pain
tomate/ mozza
Pain
```

Le premier appel en ligne 13 exécute la fonction simple, alors que le second en ligne 17 exécute la fonction décorée. Cette construction peut sembler ardue et difficile à comprendre. Heureusement, Python a une notation en « *sucre syntaxique* » (*syntactic sugar*) qui en facilite la lecture. Celle-ci utilise le symbole `@` :

```
1  def transforme_en_sandwich(fonction_a_decorer):
2      def emballage():
3          print("Pain")
4          fonction_a_decorer()
5          print("Pain")
6      return emballage
7
8  @transforme_en_sandwich
9  def imprime_victuaille():
10     print("tomate / mozza")
11
12 if __name__ == "__main__":
13     imprime_victuaille()
```

La ligne 8 transforme irrémédiablement la fonction `imprime_victuaille()` en fonction décorée. Cela parait déjà un peu plus lisible. L'exécution donnera bien sûr :

```
Pain
tomate / mozza
Pain
```

Au final, la notation :

```
1  @decorator
2  def fct():
3      [...]
```

est équivalente à :

```
1  fct = decorator(fct)
```

Cela fonctionne avec n'importe quelle fonction prenant en argument une autre fonction.

Conseil

Nous vous conseillons bien sûr d'utiliser systématiquement la notation `@decorator` qui est plus lisible et intuitive.

Si tout cela vous semble ardu (on vous comprend…), vous devez vous dire « pourquoi utiliser une construction aussi complexe ? ». Et bien, c'est tout simplement parce qu'un décorateur est ré-utilisable dans n'importe quelle fonction. Si on reprend la même fonction décoratrice que ci-dessus :

```
1  @transforme_en_sandwich
2  def imprime_victuaille1():
3      print("tomate / mozza")
4
5  @transforme_en_sandwich
6  def imprime_victuaille2():
7      print("jambon / fromage")
8
9  if __name__ == "__main__":
10     imprime_victuaille1()
11     print()
12     imprime_victuaille2()
```

On a donc un décorateur permettant de transformer en sandwich n'importe quelle fonction imprimant une victuaille ! Ceci renverra :

```
Pain
tomate / mozza
Pain

Pain
jambon / fromage
Pain
```

Un exemple plus concret de décorateur pourrait être la mesure du temps d'exécution d'une fonction :

```
import time

def mesure_temps(fonction_a_decorer):
    def emballage():
        temps1 = time.time()
        fonction_a_decorer()
        temps2 = time.time()
        print(f"Le temps d'éxécution de {fonction_a_decorer.__name__} est "
              f"{temps2 - temps1} s")
    return emballage
```

En ligne 8, l'attribut `.__name__` renvoie le nom de la fonction sous forme de chaîne de caractères. Dans cet exemple, le décorateur `@mesure_temps` mis devant n'importe quelle fonction affichera systématiquement le temps d'exécution de celle-ci.

Pour finir, si on revient sur le décorateur `@property` vu dans le chapitre 24 *Avoir plus la classe avec les objets*, nous avions vu également qu'il existait une fonction `property()`. Donc pour les décorateurs pré-existants que nous avons abordés dans le chapitre 24, il existe des fonctions équivalentes. Comme dans notre exemple, la notation `@decorateur` va finalement appeler la fonction décoratrice. Donc derrière une notation `@quelquechose`, il existe toujours une fonction `quelquechose()` remplissant ce rôle de décorateur.

Pour aller plus loin

Pour aller plus loin, vous pouvez consulter ce très bon article[35] sur le site *RealPython*. Il y est expliqué en outre comment on peut gérer le passage d'arguments quand on utilise des décorateurs, ainsi que l'utilisation de décorateurs multiples.

26.9 Un peu de transformée de Fourier avec *NumPy*

La transformée de Fourier est très utilisée pour l'analyse de signaux, notamment lorsqu'on souhaite extraire des périodicités au sein d'un signal bruité. Le module *NumPy* possède la fonction `fft()` (dans le sous-module *fft*) permettant de calculer des transformées de Fourier.

Voici un petit exemple sur la fonction cosinus de laquelle on souhaite extraire la période à l'aide de la fonction `fft()` :

```
1  import numpy as np
2
3  debut = -2 * np.pi
4  fin = 2 * np.pi
5  pas = 0.1
6  x = np.arange(debut,fin,pas)
7  y = np.cos(x)
8
9  TF = np.fft.fft(y)
10 ABSTF = np.abs(TF)
11 pas_xABSTF = 1/(fin-debut)
12 x_ABSTF = np.arange(0,pas_xABSTF * len(ABSTF),pas_xABSTF)
```

Plusieurs commentaires sur cet exemple :

Ligne 1. On charge le module *NumPy* avec le nom raccourci `np`.

Lignes 3 à 6. On définit l'intervalle (de -2π à 2π radians) pour les valeurs en abscisse ainsi que le pas (0,1 radians).

35. https://realpython.com/primer-on-python-decorators/

Lignes 7. On calcule directement les valeurs en ordonnées avec la fonction cosinus du module *NumPy*. On constate ici que *NumPy* redéfinit certaines fonctions ou constantes mathématiques de base, comme `pi`, `cos()` ou `abs()` (valeur absolue, ou module d'un nombre complexe). Ces fonctions sont directement utilisables avec un objet *array*.

Ligne 9. On calcule la transformée de Fourier avec la fonction `fft()` qui renvoie un vecteur (objet *array* à une dimension) de nombres complexes. Eh oui, le module *NumPy* gère aussi les nombres complexes !

Ligne 10. On extrait le module du résultat précédent avec la fonction `abs()`.

Ligne 11. La variable `x_ABSTFL` représente l'abscisse du spectre (en radian^{-1}).

Ligne 12. La variable `ABSTF` contient le spectre lui même. L'analyse de ce dernier nous donne un pic à 0,15 radian^{-1}, ce qui correspond bien à 2π (c'est plutôt bon signe de retrouver ce résultat).

26.10 Sauvegardez votre historique de commandes

Vous pouvez sauvegarder l'historique des commandes utilisées dans l'interpréteur Python avec le module `readline`.

```
>>> print("hello")
hello
>>> a = 22
>>> a = a + 11
>>> print(a)
33
>>> import readline
>>> readline.write_history_file()
```

Quittez Python. L'historique de toutes vos commandes est dans votre répertoire personnel, dans le fichier `.history`. Relancez l'interpréteur Python.

```
>>> import readline
>>> readline.read_history_file()
```

Vous pouvez accéder aux commandes de la session précédente avec la flèche du haut de votre clavier. D'abord les commandes `readline.read_history_file()` et `import readline` de la session actuelle, puis `print(a)`, `a = a + 11`, `a = 22`…

CHAPITRE 27

Mini-projets

Dans ce chapitre, nous vous proposons quelques scénarios pour développer vos compétences en Python et mettre en œuvre les concepts que vous avez rencontrés dans les chapitres précédents.

27.1 Description des projets

27.1.1 Mots anglais dans le protéome humain

L'objectif de ce premier projet est de découvrir si des mots anglais peuvent se retrouver dans les séquences du protéome humain, c'est-à-dire dans les séquences de l'ensemble des protéines humaines.

Vous aurez à votre disposition :

- Le fichier `english-common-words.txt`[1], qui contient les 3 000 mots anglais les plus fréquents, à raison d'1 mot par ligne.
- Le fichier `human-proteome.fasta`[2] qui contient le protéome humain sous la forme de séquences au format FASTA. Attention, ce fichier est assez gros. Ce fichier provient de la banque de données UniProt à partir de cette page[3].

Conseil

Des explications sur le format FASTA et des exemples de code sont fournis dans l'annexe A *Quelques formats de données en biologie*.

27.1.2 Genbank2fasta

Ce projet consiste à écrire un convertisseur de fichier, du format GenBank au format FASTA.

Pour cela, nous allons utiliser le fichier GenBank du chromosome I de la levure de boulanger *Saccharomyces cerevisiae*. Vous pouvez télécharger ce fichier :

- soit via le lien sur le site du cours `NC_001133.gbk`[4] ;

1. https://python.sdv.u-paris.fr/data-files/english-common-words.txt
2. https://python.sdv.u-paris.fr/data-files/human-proteome.fasta
3. https://www.uniprot.org/help/human_proteome
4. https://python.sdv.u-paris.fr/data-files/NC_001133.gbk

- soit directement sur la page de *Saccharomyces cerevisiae S288c chromosome I, complete sequence*[5] sur le site du NCBI, puis en cliquant sur *Send to*, puis *Complete Record*, puis *Choose Destination : File*, puis *Format : GenBank (full)* et enfin sur le bouton *Create File*.

Vous trouverez des explications sur les formats FASTA et GenBank ainsi que des exemples de code dans l'annexe A *Quelques formats de données en biologie*.

Vous pouvez réaliser ce projet sans ou avec des expressions régulières (abordées dans le chapitre 17).

27.1.3 Simulation d'un pendule

On se propose de réaliser une simulation d'un pendule simple[6] en Tkinter. Un pendule simple est représenté par une masse ponctuelle (la boule du pendule) reliée à un pivot immobile par une tige rigide et sans masse. On néglige les effets de frottement et on considère le champ gravitationnel comme uniforme. Le mouvement du pendule sera calculé en résolvant numériquement l'équation différentielle suivante :

$$a_\theta(t) = \frac{d^2\theta}{dt^2}(t) = -\frac{g}{l} * sin(\theta(t))$$

où θ représente l'angle entre la verticale et la tige du pendule, a_θ l'accélération angulaire, g la gravité, et l la longueur de la tige (note : pour la dérivation d'une telle équation vous pouvez consulter la page wikipedia[7] ou l'accompagnement pas à pas, cf. la rubrique suivante).

Pour trouver la valeur de θ en fonction du temps, on pourra utiliser la méthode semi-implicite d'Euler[8] de résolution d'équation différentielle. La formule ci-dessus donne l'accélération angulaire au temps t : $a_\theta(t) = -\frac{g}{l} \times sin(\theta(t))$. À partir de celle-ci, la méthode propose le calcul de la vitesse angulaire au pas suivant : $v_\theta(t+\delta t) = v_\theta(t) + a_\theta(t) \times \delta t$ (où δt représente le pas de temps entre deux étapes successives de la simulation). Enfin, cette vitesse $v_\theta(t+\delta t)$ donne l'angle θ au pas suivant : $\theta(t+\delta t) = \theta(t) + v_\theta(t+\delta t) \times \delta t$. On prendra un pas de temps $\delta t = 0.05$ s, une accélération gravitationnelle $g = 9.8$ m.s^{-2} et une longueur de tige de $l = 1$ m.

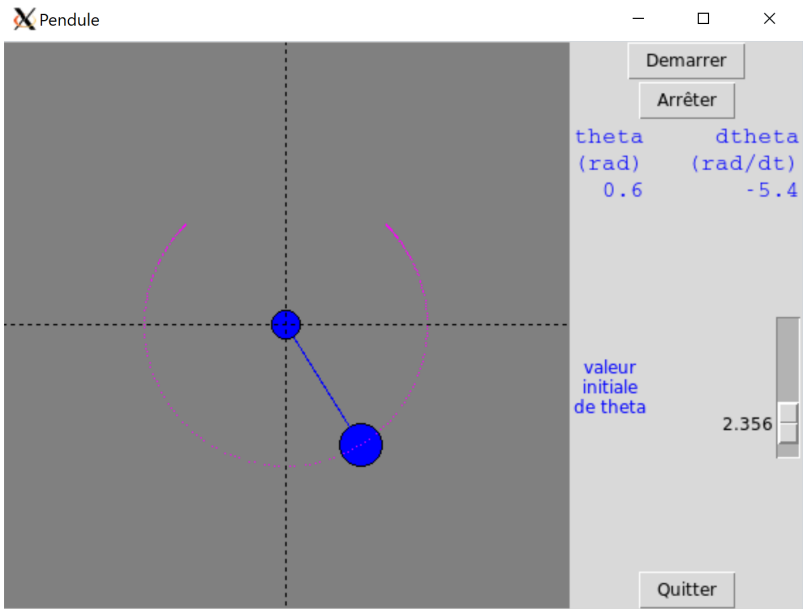


FIGURE 27.1 – Application pendule.

Pour la visualisation, vous pourrez utiliser le *widget canvas* du module *Tkinter* (voir le chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne), rubrique *Un canvas animé dans une classe*). On cherche à obtenir un résultat comme montré dans la figure 27.1.

5. https://www.ncbi.nlm.nih.gov/nuccore/NC_001133
6. https://fr.wikipedia.org/wiki/Pendule_simple
7. https://en.wikipedia.org/wiki/Pendulum_(mathematics)#math_Eq._1
8. https://en.wikipedia.org/wiki/Euler_method

Nous vous conseillons de procéder d'abord à la mise en place du simulateur physique (c'est-à-dire obtenir θ en fonction du temps ou du pas de simulation). Faites par exemple un premier script Python qui produit un fichier à deux colonnes (temps et valeur de θ). Une fois que cela fonctionne bien, il vous faudra construire l'interface *Tkinter* et l'animer. Vous pouvez ajouter un bouton pour démarrer / stopper le pendule et une règle pour modifier sa position initiale.

N'oubliez pas, il faudra mettre dans votre programme final une fonction qui convertit l'angle θ en coordonnées cartésiennes x et y dans le plan du *canvas*. Faites également attention au système de coordonnées du *canvas* où les ordonnées sont inversées par rapport à un repère mathématique. Pour ces deux aspects, reportez-vous à l'exercice *Polygone de Sierpinski* du chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne).

27.2 Accompagnement pas à pas

Vous trouverez ci-après les différentes étapes pour réaliser les mini-projets proposés. Prenez le temps de bien comprendre une étape avant de passer à la suivante.

27.2.1 Mots anglais dans le protéome humain

L'objectif de ce premier projet est de découvrir si des mots anglais peuvent se retrouver dans les séquences du protéome humain, c'est-à-dire dans les séquences de l'ensemble des protéines humaines.

27.2.1.1 Composition aminée

Dans un premier temps, composez 5 mots anglais avec les 20 acides aminés.

27.2.1.2 Des mots

Téléchargez le fichier english-common-words.txt[9]. Ce fichier contient les 3000 mots anglais les plus fréquents, à raison d'1 mot par ligne.

Créez un script `words_in_proteome.py` et écrivez la fonction `read_words()` qui va lire les mots contenus dans le fichier dont le nom est fourni en argument du script et renvoyer une liste contenant les mots convertis en majuscule et composés de 3 caractères ou plus.

Dans le programme principal, affichez le nombre de mots sélectionnés.

27.2.1.3 Des protéines

Téléchargez maintenant le fichier human-proteome.fasta[10]. Attention, ce fichier est assez gros. Ce fichier provient de la banque de données UniProt à partir de cette page[11].

Voici les premières lignes de ce fichier (`[...]` indique une coupure que nous avons faite) :

```
>sp|O95139|NDUB6_HUMAN NADH dehydrogenase [ubiquinone] 1 beta [...]
MTGYTPDEKLRLQQLRELRRRWLKDQELSPREPVLPPQKMGPMEKFWNKFLENKSPWRKM
VHGVYKKSIFVFTHVLVPVWIIHYYMKYHVSEKPYGIVEKKSRIFPGDTILETGEVIPPM
KEFPDQHH
>sp|O75438|NDUB1_HUMAN NADH dehydrogenase [ubiquinone] 1 beta [...]
MVNLLQIVRDHWVHVLVPMGFVIGCYLDRKSDERLTAFRNKSMLFKRELQPSEEVTWK
>sp|Q8N4C6|NIN_HUMAN Ninein OS=Homo sapiens OX=9606 GN=NIN PE=1 SV=4
MDEVEQDQHEARLKELFDSFDTTGTGSLGQEELTDLCHMLSLEEVAPVLQQTLLQDNLLG
RVHFDQFKEALILILSRTLSNEEHFQEPDCSLEAQPKYVRGGKRYGRRSLPEFQESVEEF
PEVTVIEPLDEEARPSHIPAGDCSEHWKTQRSEEYEAEGQLRFWNPDDLNASQSGSSPPQ
```

Toujours dans le script `words_in_proteome.py`, écrivez la fonction `read_sequences()` qui va lire le protéome dans le fichier dont le nom est fourni en second argument du script. Cette fonction va renvoyer un dictionnaire dont les clefs sont les identifiants des protéines (par exemple, `O95139`, `O75438`, `Q8N4C6`) et dont les valeurs associées sont les séquences.

Dans le programme principal, affichez le nombre de séquences lues. À des fins de test, affichez également la séquence associée à la protéine `O95139`.

9. https://python.sdv.u-paris.fr/data-files/english-common-words.txt
10. https://python.sdv.u-paris.fr/data-files/human-proteome.fasta
11. https://www.uniprot.org/help/human_proteome

27.2.1.4 À la pêche aux mots

Écrivez maintenant la fonction `search_words_in_proteome()` qui prend en argument la liste de mots et le dictionnaire contenant les séquences des protéines et qui va compter le nombre de séquences dans lesquelles un mot est présent. Cette fonction renverra un dictionnaire dont les clefs sont les mots et les valeurs le nombre de séquences qui contiennent ces mots. La fonction affichera également le message suivant pour les mots trouvés dans le protéome :

```
ACCESS found in 1 sequences
ACID found in 38 sequences
ACT found in 805 sequences
[...]
```

Cette étape prend quelques minutes. Soyez patient.

27.2.1.5 Et le mot le plus fréquent est...

Pour terminer, écrivez maintenant la fonction `find_most_frequent_word()` qui prend en argument le dictionnaire renvoyé par la précédente fonction `search_words_in_proteome()` et qui affiche le mot trouvé dans le plus de protéines, ainsi que le nombre de séquences dans lesquelles il a été trouvé, sous la forme :

```
=> xxx found in yyy sequences
```

Quel est ce mot ?

Quel pourcentage des séquences du protéome contiennent ce mot ?

27.2.1.6 Pour être plus complet

Jusqu'à présent, nous avions déterminé, pour chaque mot, le nombre de séquences dans lesquelles il apparaissait. Nous pourrions aller plus loin et calculer aussi le nombre de fois que chaque mot apparaît dans les séquences.

Pour cela modifier la fonction `search_words_in_proteome()` de façon à compter le nombre d'occurrences d'un mot dans les séquences. La méthode `.count()` vous sera utile.

Déterminez alors quel mot est le plus fréquent dans le protéome humain.

27.2.2 genbank2fasta (sans expression régulière)

Ce projet consiste à écrire un convertisseur de fichier, du format GenBank au format FASTA. L'annexe A *Quelques formats de données en biologie* rappelle les caractéristiques de ces deux formats de fichiers.

Le jeu de données avec lequel nous allons travailler est le fichier GenBank du chromosome I de la levure du boulanger *Saccharomyces cerevisiae*. Les indications pour le télécharger sont indiqués dans la description du projet.

Dans cette rubrique, nous allons réaliser ce projet **sans expression régulière**.

27.2.2.1 Lecture du fichier

Créez un script `genbank2fasta.py` et créez la fonction `lit_fichier()` qui prend en argument le nom du fichier et qui renvoie le contenu du fichier sous forme d'une liste de lignes, chaque ligne étant elle-même une chaîne de caractères.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk` et affichez le nombre de lignes lues.

27.2.2.2 Extraction du nom de l'organisme

Dans le même script, ajoutez la fonction `extrait_organisme()` qui prend en argument le contenu du fichier précédemment obtenu avec la fonction `lit_fichier()` (sous la forme d'une liste de lignes) et qui renvoie le nom de l'organisme. Pour récupérer la bonne ligne vous pourrez tester si les premiers caractères de la ligne contiennent le mot-clé `ORGANISM`.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk` et affichez le nom de l'organisme.

27.2.2.3 Recherche des gènes

Dans le fichier GenBank, les gènes sens sont notés de cette manière :

```
     gene            58..272
```

ou

```
     gene            <2480..>2707
```

et les gènes antisens (ou encore complémentaires) de cette façon :

```
     gene            complement(55979..56935)
```

ou

```
     gene            complement(<13363..>13743)
```

Les valeurs numériques séparées par `..` indiquent la position du gène dans le génome (numéro de la première base, numéro de la dernière base).

Remarque

Le symbole `<` indique un gène partiel sur l'extrémité 5', c'est-à-dire que le codon START correspondant est incomplet. Respectivement, le symbole `>` désigne un gène partiel sur l'extrémité 3', c'est-à-dire que le codon STOP correspondant est incomplet. Pour plus de détails, consultez la documentation du NCBI sur les délimitations des gènes[12]. Nous vous proposons ici d'ignorer ces symboles `>` et `<`.

Repérez ces différents gènes dans le fichier `NC_001133.gbk`. Pour récupérer ces lignes de gènes il faut tester si la ligne commence par

```
     gene
```

(c'est-à-dire 5 espaces, suivi du mot `gene`, suivi de 12 espaces). Pour savoir s'il s'agit d'un gène sur le brin direct ou complémentaire, il faut tester la présence du mot `complement` dans la ligne lue.

Ensuite si vous souhaitez récupérer la position de début et de fin de gène, nous vous conseillons d'utiliser la fonction `replace()` et de ne garder que les chiffres et les `.` Par exemple

```
     gene            <2480..>2707
```

sera transformé en

```
2480..2707
```

Enfin, avec la méthode `.split()` vous pourrez facilement récupérer les deux entiers de début et de fin de gène.

Dans le même script `genbank2fasta.py`, ajoutez la fonction `recherche_genes()` qui prend en argument le contenu du fichier (sous la forme d'une liste de lignes) et qui renvoie la liste des gènes.

Chaque gène sera lui-même une liste contenant le numéro de la première base, le numéro de la dernière base et une chaîne de caractère `"sens"` pour un gène sens et `"antisens"` pour un gène antisens.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk` et affichez le nombre de gènes trouvés, ainsi que le nombre de gènes sens et antisens.

27.2.2.4 Extraction de la séquence nucléique du génome

La taille du génome est indiqué sur la première ligne d'un fichier GenBank. Trouvez la taille du génome stocké dans le fichier `NC_001133.gbk`.

Dans un fichier GenBank, la séquence du génome se trouve entre les lignes

```
ORIGIN
```

12. https://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html#BaseSpanB

et

```
//
```

Trouvez dans le fichier `NC_001133.gbk` la première et dernière ligne de la séquence du génome.

Pour récupérer les lignes contenant la séquence, nous vous proposons d'utiliser un algorithme avec un drapeau `is_dnaseq` (qui vaudra `True` ou `False`). Voici l'algorithme proposé en pseudo-code :

```
is_dnaseq <- False
Lire chaque ligne du fichier gbk
    si la ligne contient "//"
        is_dnaseq <- False
    si is_dnaseq vaut True
        accumuler la séquence
    si la ligne contient "ORIGIN"
        is_dnaseq <- True
```

Au début ce drapeau aura la valeur `False`. Ensuite, quand il se mettra à True, on pourra lire les lignes contenant la séquence, puis quand il se remettra à False on arrêtera.

Une fois la séquence récupérée, il suffira d'éliminer les chiffres, retours chariots et autres espaces (*Conseil* : calculer la longueur de la séquence et comparer la à celle indiquée dans le fichier gbk).

Toujours dans le même script `genbank2fasta.py`, ajoutez la fonction `extrait_sequence()` qui prend en argument le contenu du fichier (sous la forme de liste de lignes) et qui renvoie la séquence nucléique du génome (dans une chaîne de caractères). La séquence ne devra pas contenir d'espaces, ni de chiffres ni de retours chariots.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk` et affichez le nombre de bases de la séquence extraite. Vérifiez que vous n'avez pas fait d'erreur en comparant la taille de la séquence extraite avec celle que vous avez trouvée dans le fichier GenBank.

27.2.2.5 Construction d'une séquence complémentaire inverse

Toujours dans le même script, ajoutez la fonction `construit_comp_inverse()` qui prend en argument une séquence d'ADN sous forme de chaîne de caractères et qui renvoie la séquence complémentaire inverse (également sous la forme d'une chaîne de caractères).

On rappelle que construire la séquence complémentaire inverse d'une séquence d'ADN consiste à :

- Prendre la séquence complémentaire. C'est-à-dire remplacer la base `a` par la base `t`, `t` par `a`, `c` par `g` et `g` par `c`.
- Prendre l'inverse. C'est-à-dire que la première base de la séquence complémentaire devient la dernière base et réciproquement, la dernière base devient la première.

Pour vous faciliter le travail, ne travaillez que sur des séquences en minuscule.

Testez cette fonction avec les séquences `atcg`, `AATTCCGG` et `gattaca`.

27.2.2.6 Écriture d'un fichier FASTA

Toujours dans le même script, ajoutez la fonction `ecrit_fasta()` qui prend en argument un nom de fichier (sous forme de chaîne de caractères), un commentaire (sous forme de chaîne de caractères) et une séquence (sous forme de chaîne de caractères) et qui écrit un fichier FASTA. La séquence sera à écrire sur des lignes ne dépassant pas 80 caractères.

Pour rappel, un fichier FASTA suit le format suivant :

```
>commentaire
sequence sur une ligne de 80 caractères maxi
suite de la séquence ......................
suite de la séquence ......................
...
```

Testez cette fonction avec :

- nom de fichier : `test.fasta`
- commentaire : `mon commentaire`
- séquence :
  `atcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatc`

27.2.2.7 Extraction des gènes

Toujours dans le même script, ajoutez la fonction `extrait_genes()` qui prend en argument la liste des gènes, la séquence nucléotidique complète (sous forme d'une chaîne de caractères) et le nom de l'organisme (sous forme d'une chaîne de caractères) et qui pour chaque gène :

- extrait la séquence du gène dans la séquence complète ;
- prend la séquence complémentaire inverse (avec la fonction `construit_comp_inverse()` si le gène est antisens ;
- enregistre le gène dans un fichier au format FASTA (avec la fonction `ecrit_fasta()`) ;
- affiche à l'écran le numéro du gène et le nom du fichier FASTA créé.

La première ligne des fichiers FASTA sera de la forme :

```
>nom-organisme|numéro-du-gène|début|fin|sens ou antisens
```

Le numéro du gène sera un numéro consécutif depuis le premier gène jusqu'au dernier. Il n'y aura pas de différence de numérotation entre les gènes sens et les gènes antisens.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk`.

27.2.2.8 Assemblage du script final

Pour terminer, modifiez le script `genbank2fasta.py` de façon à ce que le fichier GenBank à analyser (dans cet exemple `NC_001133.gbk`), soit entré comme argument du script.

Vous afficherez un message d'erreur si :

- le script `genbank2fasta.py` est utilisé sans argument,
- le fichier fourni en argument n'existe pas.

Pour vous aider, n'hésitez pas à jeter un œil aux descriptions des modules *sys* et *pathlib* dans le chapitre 9 *Modules*.

Testez votre script ainsi finalisé.

Bravo, si vous êtes arrivés jusqu'à cette étape.

27.2.3 genbank2fasta (avec expressions régulières)

Ce projet consiste à écrire un convertisseur de fichier, du format GenBank au format FASTA. L'annexe A *Quelques formats de données en biologie* rappelle les caractéristiques de ces deux formats de fichiers.

Le jeu de données avec lequel nous allons travailler est le fichier GenBank du chromosome I de la levure du boulanger *Saccharomyces cerevisiae*. Les indications pour le télécharger sont indiqués dans la description du projet.

Dans cette rubrique, nous allons réaliser ce projet **avec des expressions régulières** en utilisant le module *re*.

27.2.3.1 Lecture du fichier

Créez un script `genbank2fasta.py` et créez la fonction `lit_fichier()` qui prend en argument le nom du fichier et qui renvoie le contenu du fichier sous forme d'une liste de lignes, chaque ligne étant elle-même une chaîne de caractères.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk` et affichez le nombre de lignes lues.

27.2.3.2 Extraction du nom de l'organisme

Dans le même script, ajoutez la fonction `extrait_organisme()` qui prend en argument le contenu du fichier précédemment obtenu avec la fonction `lit_fichier()` (sous la forme d'une liste de lignes) et qui renvoie le nom de l'organisme. Utilisez de préférence une expression régulière.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk` et affichez le nom de l'organisme.

27.2.3.3 Recherche des gènes

Dans le fichier GenBank, les gènes sens sont notés de cette manière :

```
     gene            58..272
```

ou

```
     gene            <2480..>2707
```

et les gènes antisens de cette façon :

```
     gene            complement(55979..56935)
```

ou

```
     gene            complement(<13363..>13743)
```

Les valeurs numériques séparées par `..` indiquent la position du gène dans le génome (numéro de la première base, numéro de la dernière base).

Remarque

Le symbole < indique un gène partiel sur l'extrémité 5', c'est-à-dire que le codon START correspondant est incomplet. Respectivement, le symbole > désigne un gène partiel sur l'extrémité 3', c'est-à-dire que le codon STOP correspondant est incomplet. Pour plus de détails, consultez la documentation du NCBI sur les délimitations des gènes[13].

Repérez ces différents gènes dans le fichier `NC_001133.gbk`. Construisez deux expressions régulières pour extraire du fichier GenBank les gènes sens et les gènes antisens.

Modifiez ces expressions régulières pour que les numéros de la première et de la dernière base puissent être facilement extraits.

Dans le même script `genbank2fasta.py`, ajoutez la fonction `recherche_genes()` qui prend en argument le contenu du fichier (sous la forme d'une liste de lignes) et qui renvoie la liste des gènes.

Chaque gène sera lui-même une liste contenant le numéro de la première base, le numéro de la dernière base et une chaîne de caractère `"sens"` pour un gène sens et `"antisens"` pour un gène antisens.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk` et affichez le nombre de gènes trouvés, ainsi que le nombre de gènes sens et antisens.

27.2.3.4 Extraction de la séquence nucléique du génome

La taille du génome est indiqué sur la première ligne d'un fichier GenBank. Trouvez la taille du génome stocké dans le fichier `NC_001133.gbk`.

Dans un fichier GenBank, la séquence du génome se trouve entre les lignes

```
ORIGIN
```

et

```
//
```

Trouvez dans le fichier `NC_001133.gbk` la première et dernière ligne de la séquence du génome.

Construisez une expression régulière pour extraire du fichier GenBank les lignes correspondantes à la séquence du génome.

Modifiez ces expressions régulières pour que la séquence puisse être facilement extraite.

Toujours dans le même script, ajoutez la fonction `extrait_sequence()` qui prend en argument le contenu du fichier (sous la forme de liste de lignes) et qui renvoie la séquence nucléique du génome (dans une chaîne de caractères). La séquence ne devra pas contenir d'espaces.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk` et affichez le nombre de bases de la séquence extraite. Vérifiez que vous n'avez pas fait d'erreur en comparant la taille de la séquence extraite avec celle que vous avez trouvée dans le fichier GenBank.

27.2.3.5 Construction d'une séquence complémentaire inverse

Toujours dans le même script, ajoutez la fonction `construit_comp_inverse()` qui prend en argument une séquence d'ADN sous forme de chaîne de caractères et qui renvoie la séquence complémentaire inverse (également sous la forme d'une chaîne de caractères).

On rappelle que construire la séquence complémentaire inverse d'une séquence d'ADN consiste à :

13. https://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html#BaseSpanB

- Prendre la séquence complémentaire. C'est-à-dire à remplacer la base `a` par la base `t`, `t` par `a`, `c` par `g` et `g` par `c`.
- Prendre l'inverse. C'est-à-dire à que la première base de la séquence complémentaire devient la dernière base et réciproquement, la dernière base devient la première.

Pour vous faciliter le travail, ne travaillez que sur des séquences en minuscule.

Testez cette fonction avec les séquences `atcg`, `AATTCCGG` et `gattaca`.

27.2.3.6 Écriture d'un fichier FASTA

Toujours dans le même script, ajoutez la fonction `ecrit_fasta()` qui prend en argument un nom de fichier (sous forme de chaîne de caractères), un commentaire (sous forme de chaîne de caractères) et une séquence (sous forme de chaîne de caractères) et qui écrit un fichier FASTA. La séquence sera à écrire sur des lignes ne dépassant pas 80 caractères.

Pour rappel, un fichier FASTA suit le format suivant :

```
>commentaire
sequence sur une ligne de 80 caractères maxi
suite de la séquence ......................
suite de la séquence ......................
...
```

Testez cette fonction avec :

- nom de fichier : `test.fasta`
- commentaire : `mon commentaire`
- séquence :
  `atcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatcgatc`

27.2.3.7 Extraction des gènes

Toujours dans le même script, ajoutez la fonction `extrait_genes()` qui prend en argument la liste des gènes, la séquence nucléotidique complète (sous forme d'une chaîne de caractères) et le nom de l'organisme (sous forme d'une chaîne de caractères) et qui pour chaque gène :

- extrait la séquence du gène dans la séquence complète ;
- prend la séquence complémentaire inverse (avec la fonction `construit_comp_inverse()` si le gène est antisens ;
- enregistre le gène dans un fichier au format FASTA (avec la fonction `ecrit_fasta()`) ;
- affiche à l'écran le numéro du gène et le nom du fichier fasta créé.

La première ligne des fichiers FASTA sera de la forme :

```
>nom-organisme|numéro-du-gène|début|fin|sens ou antisens
```

Le numéro du gène sera un numéro consécutif depuis le premier gène jusqu'au dernier. Il n'y aura pas de différence de numérotation entre les gènes sens et les gènes antisens.

Testez cette fonction avec le fichier GenBank `NC_001133.gbk`.

27.2.3.8 Assemblage du script final

Pour terminer, modifiez le script `genbank2fasta.py` de façon à ce que le fichier GenBank à analyser (dans cet exemple `NC_001133.gbk`), soit entré comme argument du script.

Vous afficherez un message d'erreur si :

- le script `genbank2fasta.py` est utilisé sans argument,
- le fichier fourni en argument n'existe pas.

Pour vous aider, n'hésitez pas à jeter un œil aux descriptions des modules *sys* et *pathlib* dans le chapitre 9 sur les modules.

Testez votre script ainsi finalisé.

27.2.4 Simulation d'un pendule

L'objectif de ce projet est de simuler un pendule simple[14] en deux dimensions, puis de le visualiser à l'aide du module *tkinter*. Le projet peut s'avérer complexe. Tout d'abord, il y a l'aspect physique du projet. Nous allons faire ici tous les rappels de mécanique nécessaires à la réalisation du projet. Ensuite, il y a la partie *tkinter* qui n'est pas évidente au premier abord. Nous conseillons de bien séparer les deux parties. D'abord réaliser la simulation physique et vérifier qu'elle fonctionne (par exemple, en écrivant un fichier de sortie permettant cette vérification). Ensuite passer à la partie graphique *tkinter* **si et seulement si** la première partie est fonctionnelle.

27.2.4.1 Mécanique d'un pendule simple

Nous allons décrire ici ce dont nous avons besoin concernant la mécanique d'un pendule simple. Notamment, nous allons vous montrer comment dériver l'équation différentielle permettant de calculer la position du pendule à tout moment en fonction des conditions initiales. Cette page est largement inspirée de la page Wikipedia en anglais[15]. Dans la suite, une grandeur représentée en gras, par exemple **P**, représente un vecteur avec deux composantes dans le plan 2D (P_x, P_y). Cette notation en gras est équivalente à la notation avec une flèche au dessus de la lettre. La même grandeur représentée en italique, par exemple P, représente le nombre scalaire correspondant. Ce nombre peut être positif ou négatif, et sa valeur absolue vaut la norme du vecteur.

Un pendule simple est représenté par une masse ponctuelle (la boule du pendule) reliée à un axe immobile par une tige rigide et sans masse. Le pendule simple est un système idéal. Ainsi, on néglige les effets de frottement et on considère le champ gravitationnel comme uniforme. La figure 27.2 montre un schéma du système ainsi qu'un bilan des forces agissant sur la masse. Les deux forces agissant sur la boule sont son poids **P** et la tension **T** due à la tige.

La figure 27.3 montre un schéma des différentes grandeurs caractérisant le pendule. La coordonnée naturelle pour définir la position du pendule est l'angle θ. Nous verrons plus tard comment convertir cet angle en coordonnées cartésiennes pour l'affichage dans un *canvas tkinter*. Nous choisissons de fixer $\theta = 0$ lorsque le pendule est à sa position d'équilibre. Il s'agit de la position où la boule est au plus bas. C'est une position à laquelle le pendule ne bougera pas s'il n'a pas une vitesse préexistante. Nous choisissons par ailleurs de considérer θ positif lorsque le pendule se balance à droite, et négatif de l'autre côté. **g** décrit l'accélération due à la gravité, avec $\mathbf{P} = m\mathbf{g}$, ou si on raisonne en scalaire $P = mg$. Les deux vecteurs représentant les composantes tangentielle et orthogonale au mouvement du pendule de **P** sont représentées sur le schéma (les annotations indiquent leur norme).

Si on déplace le pendule de sa position d'équilibre, il sera mû par la force **F** résultant de la tension **T** et de son poids **P** (cf. plus bas). Comme le système est considéré comme parfait (pas de frottement, gravité uniforme, etc.), le pendule ne s'arrêtera jamais. Si on le monte à $\theta = +20$ deg et qu'on le lâche, le pendule redescendra en passant par $\theta = 0$ deg, remontera de l'autre côté à $\theta = -20$ deg, puis continuera de la sorte indéfiniment, grâce à la conservation de l'énergie dans un système fermé (c'est-à-dire sans « fuite » d'énergie).

Ici, on va tenter de simuler ce mouvement en appliquant les lois du mouvement de Newton[16] et en résolvant les équations correspondantes numériquement. D'après la seconde loi de Newton, la force (**F**) agissant sur la boule est égale à sa masse (m) fois son accélération (**a**) :

$$\mathbf{F} = m\mathbf{a}$$

Cette loi est exprimée ici dans le système de coordonnées cartésiennes (le plan à 2 dimensions). La force **F** et l'accélération **a** sont des vecteurs dont les composantes sont respectivement (F_x, F_y) et (a_x, a_y). La force **F** correspond à la somme vectorielle de **T** et **P**. La tige du pendule étant rigide, le mouvement de la boule est restreint sur le cercle de rayon égal à la longueur L de la tige (dessiné en pointillé). Ainsi, seule la composante tangentielle de l'accélération **a** sera prise en compte dans ce mouvement. Comment la calculer ? La force de tension **T** étant orthogonale au mouvement du pendule, celle-ci n'aura pas d'effet. De même, la composante orthogonale $mgcos\theta$ due au poids **P** n'aura pas d'effet non plus. Au final, on ne prendra en compte que la composante tangentielle due au poids, c'est-à-dire $mgsin\theta$ (cf. figure 27.3). Au final, on peut écrire l'expression suivante en raisonnant sur les valeurs scalaires :

$$F = ma = -mgsin\theta$$

14. https://fr.wikipedia.org/wiki/Pendule_simple
15. https://en.wikipedia.org/wiki/Pendulum_(mathematics)
16. https://fr.wikipedia.org/wiki/Lois_du_mouvement_de_Newton

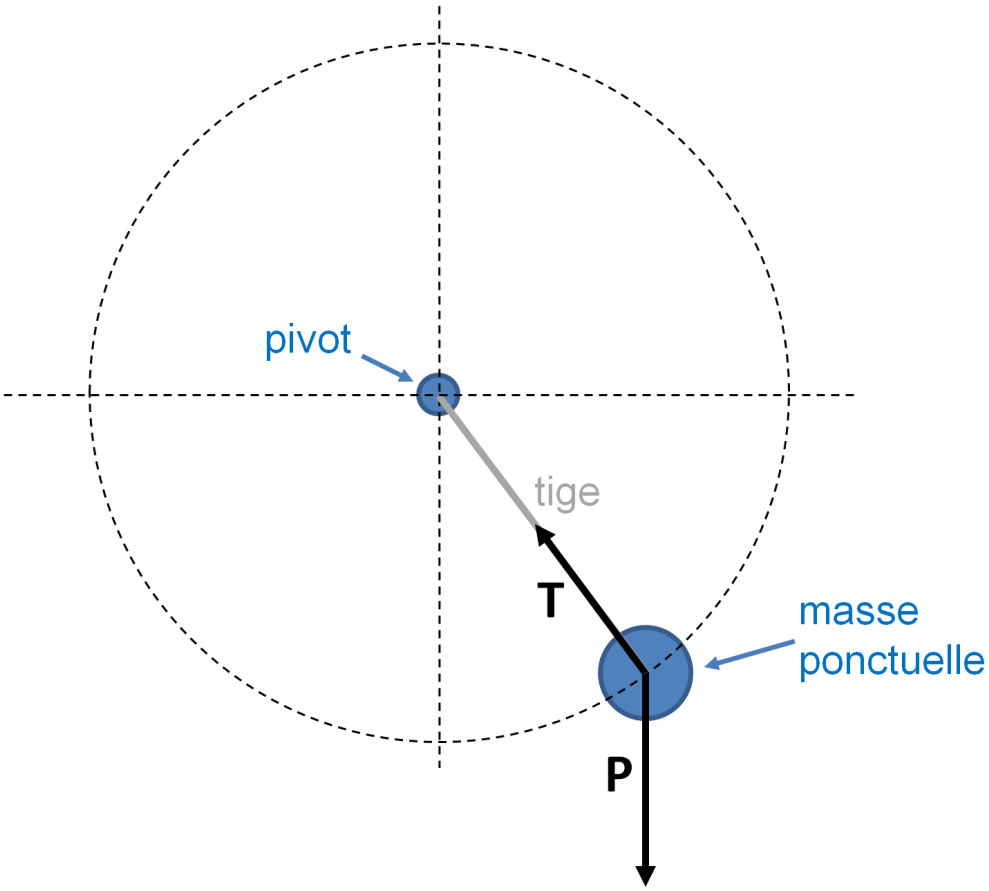


FIGURE 27.2 – Bilan des forces dans un pendule simple.

Le signe $-$ dans cette formule est très important. Il indique que l'accélération s'oppose systématiquement à θ. Si le pendule se balance vers la droite et que θ devient plus positif, l'accélération tendra toujours à faire revenir la boule dans l'autre sens vers sa position d'équilibre à $\theta = 0$. On peut faire un raisonnement équivalent lorsque le pendule se balance vers la gauche et que θ devient plus négatif.

Si on exprime l'accélération en fonction de θ, on trouve ce résultat qui peut sembler peu intuitif au premier abord :

$$a = -gsin\theta$$

Le mouvement du pendule ne dépend pas de sa masse !

Idéalement, nous souhaiterions résoudre cette équation en l'exprimant en fonction de θ seulement. Cela est possible en reliant θ à la longueur effective de l'arc s parcourue par le pendule :

$$s = \theta L$$

Pour bien comprendre cette formule, souvenez-vous de la formule bien connue du cercle $l = 2\pi r$ (où l est la circonférence, et r le rayon) ! Elle relie la valeur de θ à la distance de l'arc entre la position actuelle de la boule et l'origine (à $\theta = 0$). On peut donc exprimer la vitesse du pendule en dérivant s par rapport au temps t :

$$v = \frac{ds}{dt} = L\frac{d\theta}{dt}$$

On peut aussi exprimer l'accélération a en dérivant l'arc s deux fois par rapport à t :

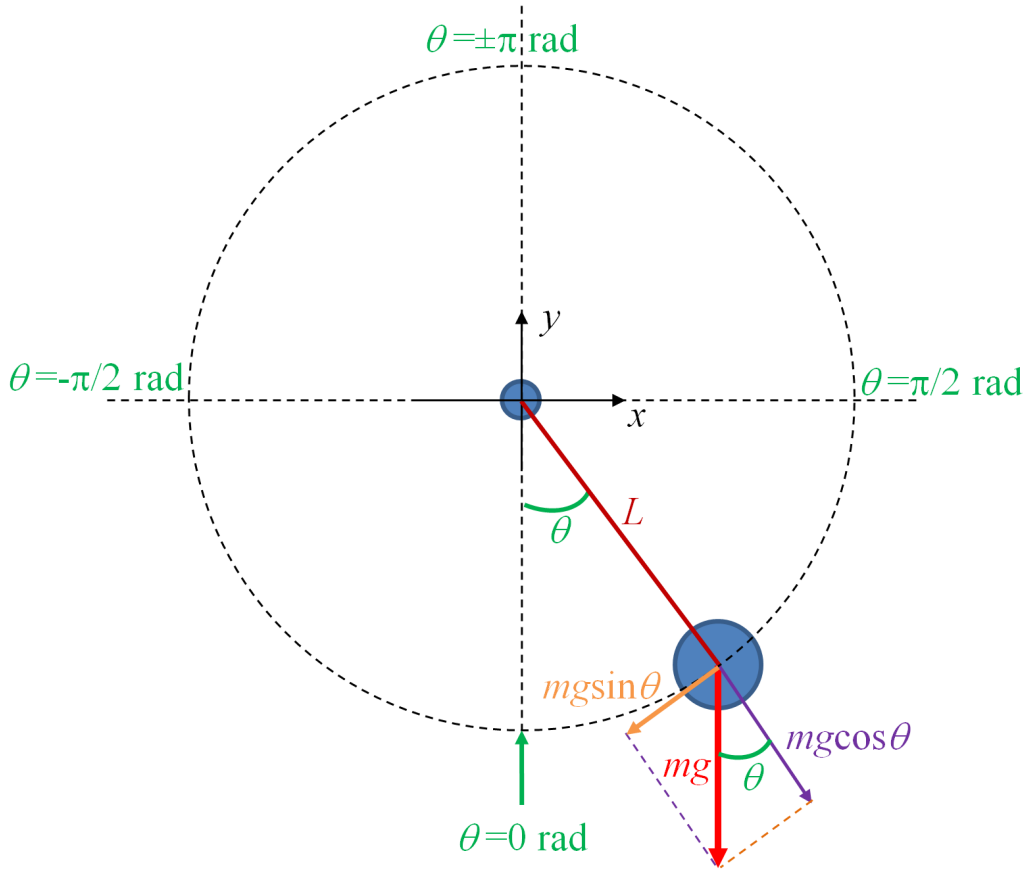


FIGURE 27.3 – Caractérisation géométrique d'un pendule simple.

$$a = \frac{d^2s}{dt^2} = L\frac{d^2\theta}{dt^2}$$

A nouveau, cette dernière formule exprime l'accélération de la boule lorsque le mouvement de celle-ci est restreint sur le cercle pointillé. Si la tige n'était pas rigide, l'expression serait différente.

Si on remplace *a* dans la formule ci-dessus, on trouve :

$$L\frac{d^2\theta}{dt^2} = -gsin\theta$$

Soit en remaniant, on trouve l'équation différentielle en θ décrivant le mouvement du pendule :

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}sin\theta = 0$$

Dans la section suivante, nous allons voir comment résoudre numériquement cette équation différentielle.

27.2.4.2 Résolution de l'équation différentielle du pendule

Il existe de nombreuses méthodes numériques de résolution d'équations différentielles[17]. L'objet ici n'est pas de faire un rappel sur toutes ces méthodes ni de les comparer, mais juste d'expliquer une de ces méthodes fonctionnant efficacement pour simuler notre pendule.

Nous allons utiliser la méthode semi-implicite d'Euler[18]. Celle-ci est relativement intuitive à comprendre.

17. https://en.wikipedia.org/wiki/Numerical_methods_for_ordinary_differential_equations
18. https://en.wikipedia.org/wiki/Semi-implicit_Euler_method

Commençons d'abord par calculer l'accélération angulaire a_θ au temps t en utilisant l'équation différentielle précédemment établie :

$$a_\theta(t) = \frac{d^2\theta}{dt^2}(t) = -\frac{g}{L} sin\theta(t)$$

L'astuce sera de calculer ensuite la vitesse angulaire au pas suivant $t + \delta t$ grâce à la relation :

$$v_\theta(t+\delta t) = \frac{d\theta}{dt}(t+\delta t) \approx v_\theta(t) + a_\theta(t) \times \delta t$$

Cette équation est ni plus ni moins qu'un remaniement de la définition de l'accélération, à savoir, la variation de vitesse par rapport à un temps. Cette vitesse $v_\theta(t+\delta t)$ permettra au final de calculer θ au temps $t+\delta t$ (c'est-à-dire ce que l'on cherche !) :

$$\theta(t+\delta t) \approx \theta(t) + v_\theta(t+\delta t) \times \delta t$$

Dans une réalisation algorithmique, il suffira d'initialiser les variables de notre système puis de faire une boucle sur un nombre de pas de simulation. A chaque pas, on calculera $a_\theta(t)$, puis $v_\theta(t+\delta t)$ et enfin $\theta(t+\delta t)$ à l'aide des formules ci-dessus.

L'initialisation des variables pourra ressembler à cela :

```
L <- 1          # longueur tige en m
g <- 9.8        # accélération gravitationnelle en m/s^2
t <- 0          # temps initial en s
dt <- 0.05      # pas de temps en s
# conditions initiales
theta <- pi / 4 # angle initial en rad
dtheta <- 0     # vitesse angulaire initiale en rad/s

afficher_position_pendule(t, theta) # afficher position de départ
```

L'initialisation des valeurs de `theta` et `dtheta` est très importante, car elle détermine le comportement du pendule. Nous avons choisi ici d'avoir une vitesse angulaire nulle et un angle de départ du pendule $\theta = \pi/4$ rad $= 45$ deg. Le pas `dt` est également très important, c'est lui qui déterminera l'erreur faite sur l'intégration de l'équation différentielle. Plus ce pas est petit, plus on est précis, mais plus le calcul sera long. Ici, on choisit un pas `dt` de 0.05 s qui constitue un bon compromis.

À ce stade, vous avez tous les éléments pour tester votre pendule. Essayez de réaliser un petit programme python `pendule_basic.py` qui utilise les conditions initiales ci-dessus et simule le mouvement du pendule. À la fin de cette rubrique, nous proposons une solution en langage algorithmique. Essayez dans un premier temps de le faire vous-même. À chaque pas, le programme écrira le temps t et l'angle θ dans un fichier `pendule_basic.dat`. Dans les équations, θ doit être exprimé en radian, mais nous vous conseillons de convertir cet angle en degré dans le fichier (plus facile à comprendre pour un humain !). Une fois ce fichier généré, vous pourrez observer le graphe correspondant avec *matplotlib* en utilisant le code suivant :

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # La fonction np.genfromtxt() renvoie un array à 2 dim.
5  array_data = np.genfromtxt("pendule_basic.dat")
6  # col 0: t, col 1: theta
7  t = array_data[:,0]
8  theta = array_data[:,1]
9
10 # Figure.
11 fig, ax = plt.subplots(figsize=(8, 8))
12 mini = min(theta) * 1.2
13 maxi = max(theta) * 1.2
14 ax.set_xlim(0, max(t))
15 ax.set_ylim(mini, maxi)
16 ax.set_xlabel("t (s)")
17 ax.set_ylabel("theta (deg)")
18 ax.plot(t, theta)
19 fig.savefig("pendule_basic.png")
```

Si vous observez une sinusoïde, bravo, vous venez de réaliser votre première simulation de pendule ! Vous avez maintenant le « squelette » de votre « moteur » de simulation. N'hésitez pas à vous amuser avec d'autres conditions initiales. Ensuite vous pourrez passer à la rubrique suivante.

Si vous avez bloqué dans l'écriture de la boucle, voici à quoi elle pourrait ressembler en langage algorithmique :

```
tant qu'on n'arrête pas le pendule:
    # acc angulaire au tps t (en rad/s^2)
    d2theta <- -(g/L) * sin(theta)
    # v angulaire mise à jour de t -> t + dt
    dtheta <- dtheta + d2theta * dt
    # theta mis à jour de t -> t + dt
    theta <- theta + dtheta * dt
    # t mis à jour
    t <- t + dt
    # mettre à jour l'affichage
    afficher_position_pendule(t, theta)
```

27.2.4.3 Constructeur de l'application en *tkinter*

Nous allons maintenant construire l'application *tkinter* en vous guidant pas à pas. Il est bien sûr conseillé de relire le chapitre 25 sur *Fenêtres graphiques et Tkinter* (en ligne) avant de vous lancer dans cette partie.

Comme expliqué largement dans les chapitres 23 *Avoir la classe avec les objets* et 24 *Avoir plus la classe avec les objets* (en ligne), nous allons construire l'application avec une classe. Le programme principal sera donc très allégé et se contentera d'instancier l'application, puis de lancer le gestionnaire d'événements :

```
1  if __name__ == "__main__":
2      """Programme principal (instancie la classe principale, donne un
3      titre et lance le gestionnaire d'événements)
4      """
5      app_pendule = AppliPendule()
6      app_pendule.title("Pendule")
7      app_pendule.mainloop()
```

Ensuite, nous commençons par écrire le constructeur de la classe. Dans ce constructeur, nous aurons une section initialisant toutes les variables utilisées pour simuler le pendule (voir rubrique précédente), puis, une autre partie générant les *widgets* et tous les éléments graphiques. Nous vous conseillons vivement de bien les séparer, et surtout de **mettre des commentaires** pour pouvoir s'y retrouver. Voici un « squelette » pour vous aider :

```
1  class AppliPendule(tk.Tk):
2      def __init__(self):
3          # Instanciation de la classe Tk.
4          tk.Tk.__init__(self)
5          # Ici vous pouvez définir toutes les variables
6          # concernant la physique du pendule.
7          self.theta = np.pi / 4  # valeur intiale theta
8          self.dtheta = 0  # vitesse angulaire initiale
9          [...]
10         self.g = 9.8  # cst gravitationnelle en m/s^2
11         [...]
12         # Oci vous pouvez construire l'application graphique.
13         self.canv = tk.Canvas(self, bg='gray', height=400, width=400)
14         # Création d'un boutton demarrer, arreter, quitter.
15         # Pensez à placer les widgets avec .pack()
16         [...]
```

La figure 27.4 vous montre un aperçu de ce que l'on voudrait obtenir.

Pour le moment, vous pouvez oublier la réglette fixant la valeur initiale de θ, les *labels* affichant la valeur de θ et v_θ ainsi que les points violets « laissés en route » par le pendule. De même, nous dessinerons le pivot, la boule et la tige plus tard. À ce stade, il est fondamental de tout de suite lancer votre application pour vérifier que les *widgets* sont bien placés. N'oubliez pas, un code complexe se teste **au fur et à mesure** lors de son développement.

Conseil

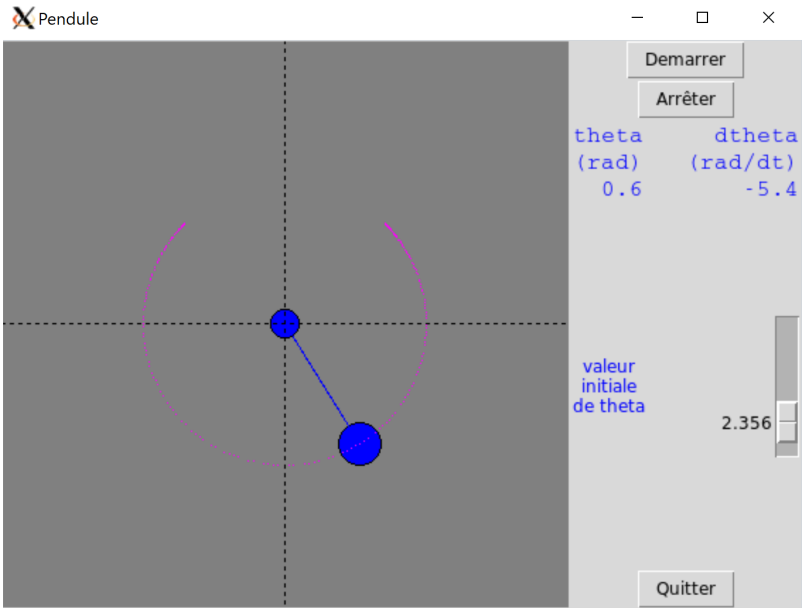


FIGURE 27.4 – Application pendule.

Pour éviter un message d'erreur si toutes les méthodes n'existe pas encore, vous pouvez indiquer `command=self.quit` pour chaque bouton (vous le changerez après).

27.2.4.4 Créations des dessins dans le canvas

Le pivot et la boule pourront être créés avec la méthode `.create_oval()`, la tige le sera avec la méthode `.create_line()`. Pensez à créer des variables pour la tige et la boule lors de l'instanciation car celles-ci bougeront par la suite.

Comment placer ces éléments dans le *canvas* ? Vous avez remarqué que lors de la création de ce dernier, nous avons fixé une dimension de 400 × 400 pixels. Le pivot se trouve au centre, c'est-à-dire au point $(200, 200)$. Pour la tige et la boule, il sera nécessaire de connaître la position de la boule **dans le repère du canvas**. Or, pour l'instant, nous définissons la position de la boule avec l'angle θ. Il va donc nous falloir convertir θ en coordonnées cartésiennes (x, y) dans le repère mathématique défini dans la figure 27.3, puis dans le repère du *canvas* (x_c, y_c) (cf. rubrique suivante).

Conversion de θ en coordonnées (x, y) Cette étape est relativement simple si on considère le pivot comme le centre du repère. Avec les fonctions trigonométriques `sin()` et `cos()`, vous pourrez calculer la position de la boule (voir l'exercice sur la spirale dans le chapitre 7 *Fichiers*). Faites attention toutefois aux deux aspects suivants :

- la trajectoire de la boule suit les coordonnées d'un cercle de rayon L (si on choisit $L = 1$ m, ce sera plus simple) ;
- nous sommes décalés par rapport au cercle trigonométrique classique ; si on considère $L = 1$ m :
  - quand $\theta = 0$, on a le point $(0, -1)$ (pendule en bas) ;
  - quand $\theta = +\pi/2 = 90$ deg, on a $(1, 0)$ (pendule à droite) ;
  - quand $\theta = -\pi/2 = -90$ deg, on a $(-1, 0)$ (pendule à gauche) ;
  - quand $\theta = \pm\pi = \pm 180$ deg, on a $(0, 1)$ (pendule en haut).

La figure 27.3 montre graphiquement les valeurs de θ.

Si vous n'avez pas trouvé, voici la solution :

```
1  self.x = np.sin(self.theta) * self.L
2  self.y = -np.cos(self.theta) * self.L
```

Conversion des coordonnées (x,y) en (x_c,y_c) Il nous faut maintenant convertir les coordonnées naturelles mathématiques du pendule (x,y) en coordonnées dans le *canvas* (x_c,y_c). Plusieurs choses sont importantes pour cela :

- le centre du repère mathématique $(0,0)$ a la coordonnée $(200,200)$ dans le *canvas* ;
- il faut choisir un facteur de conversion : par exemple, si on choisit $L = 1$ m, on peut proposer le facteur 1 m $\rightarrow$ 100 pixels ;
- l'axe des ordonnées dans le *canvas* est **inversé** par rapport au repère mathématique.

Conseil

Dans votre classe, cela peut être une bonne idée d'écrire une méthode qui réalise cette conversion. Celle-ci pourrait s'appeler par exemple `map_realcoor2canvas()`.

Si vous n'avez pas trouvé, voici la solution :

```
self.conv_factor = 100
self.x_c = self.x*self.conv_factor + 200
self.y_c = -self.y*self.conv_factor + 200
```

27.2.4.5 Gestion des boutons

Il reste maintenant à gérer les boutons permettant de démarrer / stopper le pendule. Pour cela il faudra créer trois méthodes dans notre classe :

- La méthode `.start()` : met en mouvement le pendule ; si le pendule n'a jamais été en mouvement, il part de son point de départ ; si le pendule avait déjà été en mouvement, celui-ci repart d'où on l'avait arrêté (avec la vitesse qu'il avait à ce moment-là).
- La méthode `.stop()` : arrête le mouvement du pendule.
- La méthode `.move()` : gère le mouvement du pendule (génère les coordonnées du pendule au pas suivant).

Le bouton « Démarrer » appellera la méthode `.start()`, le bouton « Arrêter » appellera la méthode `.stop()` et le bouton « Quitter » quittera l'application. Pour lier une action au clic d'un bouton, on se souvient qu'il faut donner à l'argument par mot-clé *command* une *callback* (c'est-à-dire le nom d'une fonction ou méthode sans les parenthèses) :

- `btn1 = tk.Button(self, text="Quitter", command=self.quit)`
- `btn2 = tk.Button(self, text="Demarrer", command=self.start)`
- `btn3 = tk.Button(self, text="Arrêter", command=self.stop)`

Ici, `self.start()` et `self.stop()` sont des méthodes que l'on doit créer, `self.quit()` pré-existe lorsque la fenêtre *tkinter* est créée.

Nous vous proposons ici une stratégie inspirée du livre de Gérard Swinnen [19]. Créons d'abord un attribut d'instance `self.is_moving` dans le constructeur. Celui-ci va nous servir de « drapeau » pour définir le mouvement du pendule. Il contiendra un entier positif ou nul. Lorsque ce drapeau sera égal à 0, le pendule sera immobile. Lorsqu'il sera > 0, le pendule sera en mouvement. Ainsi :

- la méthode `.start()` ajoutera 1 à `self.is_moving`. Si `self.is_moving` est égal à 1 alors la méthode `self.move()` sera appelée ;
- la méthode `.stop()` mettra la valeur de `self.is_moving` à 0.

Puisque `.start()` ajoute 1 à `self.is_moving`, le premier clic sur le bouton « Démarrer » appellera la méthode `.move()` car `self.is_moving` vaudra 1. Si l'utilisateur appuie une deuxième fois sur le bouton « Démarrer », `self.is_moving` vaudra 2, mais n'appellera pas `.move()` une deuxième fois ; cela sera vrai pour tout clic ultérieur de l'utilisateur sur ce bouton. Cette astuce évite des appels concurrents de la méthode `.move()`.

27.2.4.6 Le coeur du programme : la méthode `.move()`

Il nous reste maintenant à générer la méthode `.move()` qui meut le pendule. Pour cela vous pouvez vous inspirer de la rubrique *Un canvas animé dans une classe* du chapitre 25 *Fenêtres graphiques et Tkinter* (en ligne).

Cette méthode va réaliser un pas de simulation de t à $t+\delta t$. Il faudra ainsi réaliser dans l'ordre :

19. https://inforef.be/swi/python.htm

- Calculer la nouvelle valeur de θ (`self.theta`) au pas $t + \delta t$ comme nous l'avons fait précédemment avec la méthode semi-implicite d'Euler.
- Convertir la nouvelle valeur de θ (`self.theta`) en coordonnées cartésiennes dans le repère du pendule (`self.x` et `self.y`).
- Convertir ces coordonnées cartésiennes dans le repère du *Canvas* (`self.x_c` et `self.y_c`).
- Mettre à jour le dessin de la baballe et de la tige avec la méthode `self.canv.coords()`.
- Incrémenter le pas de temps.
- Si le drapeau `self.is_moving` est supérieur à 0, la méthode `self.move()` est rappelée après 20 millisecondes (*Conseil* : la méthode `.after()` est votre amie).

27.2.4.7 Ressources complémentaires

Si vous êtes arrivé jusqu'ici, bravo, vous pouvez maintenant admirer votre superbe pendule en mouvement :-) !

Voici quelques indications si vous voulez aller un peu plus loin.

Si vous souhaitez mettre une réglette pour modifier la position de départ du pendule, vous pouvez utiliser la classe `tk.Scale()`. Si vous souhaitez afficher la valeur de θ qui se met à jour au fur et à mesure, il faudra instancier un objet avec la classe `tk.StringVar()`. Cet objet devra être passé à l'argument `textvariable` lors de la création de ce *Label* avec `tk.Label()`. Ensuite, vous pourrez mettre à jour le texte du *Label* avec la méthode `self.instance_StringVar.set()`.

Pour le *fun*, si vous souhaitez laisser une « trace » du passage du pendule avec des points colorés, vous pouvez utiliser tout simplement la méthode `self.canv.create_line()` et créer une ligne d'un pixel de hauteur et de largeur pour dessiner un point. Pour améliorer l'esthétique, vous pouvez faire en sorte que ces points changent de couleur aléatoirement à chaque arrêt / redémarrage du pendule.

27.3 Scripts de correction

Voici les scripts corrigés pour les différents mini-projets.

Remarque

- Prenez le temps de chercher par vous-même avant de télécharger les scripts de correction.
- Nous proposons une correction. D'autres solutions sont possibles.

- Mots anglais dans le protéome humain : words_in_proteome.py [20]
- Genbank2fasta (sans expression régulière) : genbank2fasta_sans_regex.py [21]
- Genbank2fasta (avec expressions régulières) : genbank2fasta_avec_regex.py [22]
- Simulation d'un pendule version simple : tk_pendule.py [23]
- Simulation d'un pendule++ (avec réglette et affichage se mettant à jour) : tk_pendule.py [24]

20. https://python.sdv.u-paris.fr/data-files/words_in_proteome.py
21. https://python.sdv.u-paris.fr/data-files/genbank2fasta_sans_regex.py
22. https://python.sdv.u-paris.fr/data-files/genbank2fasta_avec_regex.py
23. https://python.sdv.u-paris.fr/data-files/tk_pendule_simple.py
24. https://python.sdv.u-paris.fr/data-files/tk_pendule.py

ANNEXE A

Quelques formats de données en biologie

A.1 FASTA

Le format FASTA est utilisé pour stocker une ou plusieurs séquences, d'ADN, d'ARN ou de protéines. Ces séquences sont classiquement représentées sous la forme :

```
>en-tête
séquence avec un nombre maximum de caractères par ligne
séquence avec un nombre maximum de caractères par ligne
séquence avec un nombre maximum de caractères par ligne
séquence avec un nombre maximum de caractères par ligne
séquence avec un nombre max
```

La première ligne débute par le caractère `>` et contient une description de la séquence. On appelle souvent cette ligne « ligne de description » ou « ligne de commentaire ».

Les lignes suivantes contiennent la séquence à proprement dite, mais avec un nombre maximum fixe de caractères par ligne. Ce nombre maximum est généralement fixé à 60, 70 ou 80 caractères. Une séquence de plusieurs centaines de bases ou de résidus est donc répartie sur plusieurs lignes.

Un fichier est dit *multifasta* lorsqu'il contient plusieurs séquences au format FASTA, les unes à la suite des autres.

Les fichiers contenant une ou plusieurs séquences au format FASTA portent la plupart du temps l'extension `.fasta` mais on trouve également `.seq`, `.fas`, `.fna` ou `.faa`.

A.1.1 Exemples

La séquence protéique au format FASTA de l'insuline humaine[1], extraite de la base de données UniProt, est :

```
>sp|P01308|INS_HUMAN Insulin OS=Homo sapiens OX=9606 GN=INS PE=1 SV=1
MALWMRLLPLLALLALWGPDPAAAFVNQHLCGSHLVEALYLVCGERGFFYTPKTRREAED
LQVGQVELGGGPGAGSLQPLALEGSLQKRGIVEQCCTSICSLYQLENYCN
```

La première ligne contient la description de la séquence (*Insulina*), le type de base de données (ici, *sp* qui signifie Swiss-Prot), son identifiant (*P01308*) et son nom (*INS_HUMAN*) dans cette base de données, ainsi que d'autres informations (*OS=Homo sapiens OX=9606 GN=INS PE=1 SV=1B*).

Les lignes suivantes contiennent la séquence sur des lignes ne dépassant pas, ici, 60 caractères. La séquence de l'insuline humaine est composée de 110 acides aminés, soit une ligne de 60 caractères et une seconde de 50 caractères.

1. https://www.uniprot.org/uniprot/P01308

Définition

UniProt[2] est une base de données de séquences de protéines. Ces séquences proviennent elles-mêmes de deux autres bases de données : Swiss-Prot (où les séquences sont annotées manuellement) et TrEMBL (où les séquences sont annotées automatiquement).

Voici maintenant la séquence nucléique (ARN), au format FASTA, de l'insuline humaine[3], extraite de la base de données GenBank[4] :

```
>BT006808.1 Homo sapiens insulin mRNA, complete cds
ATGGCCCTGTGGATGCGCCTCCTGCCCCTGCTGGCGCTGCTGGCCCTCTGGGGACCTGACCCAGCCGCAG
CCTTTGTGAACCAACACCTGTGCGGCTCACACCTGGTGGAAGCTCTCTACCTAGTGTGCGGGGAACGAGG
CTTCTTCTACACACCCAAGACCCGCCGGGAGGCAGAGGACCTGCAGGTGGGGCAGGTGGAGCTGGGCGGG
GGCCCTGGTGCAGGCAGCCTGCAGCCCTTGGCCCTGGAGGGGTCCCTGCAGAAGCGTGGCATTGTGGAAC
AATGCTGTACCAGCATCTGCTCCCTCTACCAGCTGGAGAACTACTGCAACTAG
```

On retrouve sur la première ligne la description de la séquence (*Homo sapiens insulin mRNA*), ainsi que son identifiant (*BT006808.1*) dans la base de données GenBank.

Les lignes suivantes contiennent les 333 bases de la séquence, réparties sur cinq lignes de 70 caractères maximum. Il est curieux de trouver la base T (thymine) dans une séquence d'ARN, qui ne devrait contenir normalement que les bases A, U, G et C. Ici, la représentation d'une séquence d'ARN avec les bases de l'ADN est une convention.

Pour terminer, voici trois séquences protéiques, au format FASTA, qui correspondent à l'insuline humaine (*Homo sapiens*), féline (*Felis catus*) et bovine (*Bos taurus*) :

```
>sp|P01308|INS_HUMAN Insulin OS=Homo sapiens OX=9606 GN=INS PE=1 SV=1
MALWMRLLPLLALLALWGPDPAAAFVNQHLCGSHLVEALYLVCGERGFFYTPKTRREAED
LQVGQVELGGGPGAGSLQPLALEGSLQKRGIVEQCCTSICSLYQLENYCN
>sp|P06306|INS_FELCA Insulin OS=Felis catus OX=9685 GN=INS PE=1 SV=2
MAPWTRLLPLLALLSLWIPAPTRAFVNQHLCGSHLVEALYLVCGERGFFYTPKARREAED
LQGKDAELGEAPGAGGLQPSALEAPLQKRGIVEQCCASVCSLYQLEHYCN
>sp|P01317|INS_BOVIN Insulin OS=Bos taurus OX=9913 GN=INS PE=1 SV=2
MALWTRLRPLLALLALWPPPPARAFVNQHLCGSHLVEALYLVCGERGFFYTPKARREVEG
PQVGALELAGGPGAGGLEGPPQKRGIVEQCCASVCSLYQLENYCN
```

Ces séquences proviennent de la base de données UniProt[5]. Chaque séquence est délimitée par la ligne d'en-tête qui débute par >.

A.1.2 Manipulation avec Python

À partir de l'exemple précédent des 3 séquences d'insuline, voici un exemple de code qui lit un fichier FASTA avec Python :

```python
prot_dict = {}
with open("insulin.fasta", "r") as fasta_file:
    prot_id = ""
    for line in fasta_file:
        if line.startswith(">"):
            prot_id = line[1:].split()[0]
            prot_dict[prot_id] = ""
        else:
            prot_dict[prot_id] += line.strip()
    for id in prot_dict:
        print(id)
        print(prot_dict[id][:30])
```

Pour chaque séquence lue dans le fichier FASTA, on affiche son identifiant et son nom, puis les 30 premiers résidus de sa séquence :

2. https://www.uniprot.org/
3. https://www.ncbi.nlm.nih.gov/nuccore/BT006808.1?report=fasta
4. https://www.ncbi.nlm.nih.gov/nuccore/AY899304.1?report=genbank
5. https://www.uniprot.org/

```
sp|P06306|INS_FELCA
MAPWTRLLPLLALLSLWIPAPTRAFVNQHL
sp|P01317|INS_BOVIN
MALWTRLRPLLALLALWPPPPARAFVNQHL
sp|P01308|INS_HUMAN
MALWMRLLPLLALLALWGPDPAAAFVNQHL
```

Notez que les protéines sont stockées dans un dictionnaire (`prot_dict`) où les clefs sont les identifiants et les valeurs les séquences.

On peut faire la même chose avec le module *Biopython* :

```
1  from Bio import SeqIO
2  with open("insulin.fasta", "r") as fasta_file:
3      for record in SeqIO.parse(fasta_file, "fasta"):
4          print(record.id)
5          print(str(record.seq)[:30])
```

Cela produit le même résultat. L'utilisation de *Biopython* rend le code plus compacte car on utilise ici la fonction `SeqIO.parse()` qui s'occupe de lire le fichier FASTA.

Remarque

L'attribut `.id` renvoie l'identifiant d'une séquence, c'est-à-dire la première partie de l'entête, sans le caractère `>`. Pour obtenir l'entête complet (toujours sans le caractère `>`), il faut utiliser l'attribut `.description`.

A.2 GenBank

GenBank est une banque de séquences nucléiques. Le format de fichier associé contient l'information nécessaire pour décrire un gène ou une portion d'un génome. Les fichiers GenBank portent le plus souvent l'extension `.gbk`.

Le format GenBank est décrit de manière très complète sur le site du NCBI[6]. En voici néanmoins les principaux éléments, avec l'exemple du gène qui code pour la trypsine[7] chez l'Homme.

A.2.1 L'en-tête

```
LOCUS       HUMTRPSGNA              800 bp    mRNA    linear   PRI 14-JAN-1995
DEFINITION  Human pancreatic trypsin 1 (TRY1) mRNA, complete cds.
ACCESSION   M22612
VERSION     M22612.1
KEYWORDS    trypsinogen.
SOURCE      Homo sapiens (human)
  ORGANISM  Homo sapiens
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Euarchontoglires; Primates; Haplorrhini;
            Catarrhini; Hominidae; Homo.
[...]
```

- **Ligne 1** (`LOCUS`) : le nom du locus (*HUMTRPSGNA*), la taille du gène (800 paires de bases), le type de molécule (ARN messager).
- **Ligne 3** (`ACCESSION`) : l'identifiant de la séquence (*M22612*).
- **Ligne 4** (`VERSION`) : la version de la séquence (*M22612.1*). Le nombre qui est séparé de l'identifiant de la séquence par un point est incrémenté pour chaque nouvelle version de la fiche GenBank. Ici, *.1* indique que nous en sommes à la première version.
- **Ligne 6** (`SOURCE`) : la provenance de la séquence (souvent l'organisme d'origine).
- **Ligne 7** (`ORGANISME`) : le nom scientifique de l'organisme, suivi de sa taxonomie (**lignes 8 à 10**).

6. https://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html
7. https://www.ncbi.nlm.nih.gov/nuccore/M22612.1

A.2.2 Les *features*

```
[...]
FEATURES             Location/Qualifiers
     source          1..800
                     /organism="Homo sapiens"
                     /mol_type="mRNA"
                     /db_xref="taxon:9606"
                     /map="7q32-qter"
                     /tissue_type="pancreas"
     gene            1..800
                     /gene="TRY1"
     CDS             7..750
                     /gene="TRY1"
                     /codon_start=1
                     /product="trypsinogen"
                     /protein_id="AAA61231.1"
                     /db_xref="GDB:G00-119-620"
                     /translation="MNPLLILTFVAAALAAPFDDDDKIVGGYNCEENSVPYQVSLNSG
                     YHFCGGSLINEQWVVSAGHCYKSRIQVRLGEHNIEVLEGNEQFINAAKIIRHPQYDRK
                     TLNNDIMLIKLSSRAVINARVSTISLPTAPPATGTKCLISGWGNTASSGADYPDELQC
                     LDAPVLSQAKCEASYPGKITSNMFCVGFLEGGKDSCQGDSGGPVVCNGQLQGVVSWGD
                     GCAQKNKPGVYTKVYNYVKWIKNTIAANS"
     sig_peptide     7..51
                     /gene="TRY1"
                     /note="G00-119-620"
[...]
```

- **Ligne 9** (`gene 1..800`) : la délimitation du gène. Ici, de la base 1 à la base 800. Par ailleurs, la notation `<x..y` indique que la séquence est partielle sur l'extrémité 5'. Réciproquement, `x..y>` indique que la séquence est partielle sur l'extrémité 3'. Enfin, pour les séquences d'ADN, la notation `complement(x..y)` indique que le gène se trouve de la base *x* à la base *y*, mais sur le brin complémentaire.
- **Ligne 10** (`/gene="TRY1"`) : le nom du gène.
- **Ligne 11** (`CDS 7..750`) : la délimitation de la séquence codante.
- **Ligne 14** (`/product="trypsinogen"`) : le nom de la protéine produite.
- **Lignes 17 à 20** (`/translation="MNPLLIL...`) : la séquence protéique issue de la traduction de la séquence codante.
- **Ligne 22** (`sig_peptide 7..51`) : la délimitation du peptide signal.

A.2.3 La séquence

```
[...]
ORIGIN
        1 accaccatga atccactcct gatccttacc tttgtggcag ctgctcttgc tgcccccttt
       61 gatgatgatg acaagatcgt tgggggctac aactgtgagg agaattctgt cccctaccag
      121 gtgtccctga attctggcta ccacttctgt ggtggctccc tcatcaacga acagtgggtg
      181 gtatcagcag gccactgcta caagtcccgc atccaggtga gactgggaga gcacaacatc
      241 gaagtcctgg aggggaatga gcagttcatc aatgcagcca agatcatccg ccacccccaa
      301 tacgacagga agactctgaa caatgacatc atgttaatca agctctcctc acgtgcagta
      361 atcaacgccc gcgtgtccac catctctctg cccaccgccc ctccagccac tggcacgaag
      421 tgcctcatct ctggctgggg caacactgcg agctctggcg ccgactaccc agacgagctg
      481 cagtgcctgg atgctcctgt gctgagccag gctaagtgtg aagcctccta ccctggaaag
      541 attaccagca acatgttctg tgtgggcttc cttgagggag gcaaggattc atgtcagggt
      601 gattctggtg gccctgtggt ctgcaatgga cagctccaag gagttgtctc ctggggtgat
      661 ggctgtgccc agaagaacaa gcctggagtc tacaccaagg tctacaacta cgtgaaatgg
      721 attaagaaca ccatagctgc caatagctaa agcccccagt atctcttcag tctctatacc
      781 aataaagtga ccctgttctc
//
```

La séquence est contenue entre les balises `ORIGIN` (ligne 2) et `//` (ligne 17).

Chaque ligne est composée d'une série d'espaces, puis du numéro du premier nucléotide de la ligne, puis d'au plus 6 blocs de 10 nucléotides. Chaque bloc est précédé d'un espace. Par exemple, ligne 10, le premier nucléotide de la ligne (`t`) est le numéro 421 dans la séquence.

A.2.4 Manipulation avec Python

À partir de l'exemple précédent, voici comment lire un fichier GenBank avec Python et le module *Biopython* :

```
from Bio import SeqIO
with open("M22612.gbk", "r") as gbk_file:
    record = SeqIO.read(gbk_file, "genbank")
    print(record.id)
    print(record.description)
    print(record.seq[:60])
```

Pour la séquence lue dans le fichier GenBank, on affiche son identifiant, sa description et les 60 premiers résidus :

```
M22612.1
Human pancreatic trypsin 1 (TRY1) mRNA, complete cds.
ACCACCATGAATCCACTCCTGATCCTTACCTTTGTGGCAGCTGCTCTTGCTGCCCCCTTT
```

Il est également possible de lire un fichier GenBank sans le module *Biopython*. Une activité dédiée est proposée dans le chapitre 27 *Mini-projets* (en ligne).

A.3 PDB

La *Protein Data Bank*[8] (PDB) est une banque de données qui contient les structures de biomacromolécules (protéines, ADN, ARN, virus…). Historiquement, le format de fichier qui y est associé est le PDB, dont une documentation détaillée est disponible sur le site éponyme[9]. Les extensions de fichier pour ce format de données sont `.ent` et surtout `.pdb`.

Un fichier PDB est constitué de deux parties principales : l'en-tête et les coordonnées.

- L'**en-tête** est lisible et utilisable par un être humain (comme par une machine).
- À l'inverse, les **coordonnées** sont surtout utilisables par un programme pour calculer certaines propriétés de la structure ou simplement la représenter sur l'écran d'un ordinateur. Bien sûr, un utilisateur expérimenté peut parfaitement jeter un œil à cette seconde partie.

Examinons ces deux parties avec la trypsine bovine[10].

A.3.1 En-tête

Pour la trypsine bovine, l'en-tête compte 510 lignes. En voici quelques unes :

8. https://www.rcsb.org/
9. http://www.wwpdb.org/documentation/file-format-content/format33/v3.3.html
10. https://www.rcsb.org/structure/2PTN

```
HEADER    HYDROLASE (SERINE PROTEINASE)           26-OCT-81   2PTN
TITLE     ON THE DISORDERED ACTIVATION DOMAIN IN TRYPSINOGEN.
TITLE    2 CHEMICAL LABELLING AND LOW-TEMPERATURE CRYSTALLOGRAPHY
COMPND    MOL_ID: 1;
COMPND   2 MOLECULE: TRYPSIN;
COMPND   3 CHAIN: A;
[...]
SOURCE   2 ORGANISM_SCIENTIFIC: BOS TAURUS;
[...]
EXPDTA    X-RAY DIFFRACTION
[...]
REMARK   2 RESOLUTION.    1.55 ANGSTROMS.
[...]
DBREF  2PTN A   16   245  UNP    P00760   TRY1_BOVIN      21    243
SEQRES   1 A  223  ILE VAL GLY GLY TYR THR CYS GLY ALA ASN THR VAL PRO
SEQRES   2 A  223  TYR GLN VAL SER LEU ASN SER GLY TYR HIS PHE CYS GLY
SEQRES   3 A  223  GLY SER LEU ILE ASN SER GLN TRP VAL VAL SER ALA ALA
SEQRES   4 A  223  HIS CYS TYR LYS SER GLY ILE GLN VAL ARG LEU GLY GLU
[...]
HELIX    1  H1 SER A  164  ILE A  176  1SNGL ALPHA TURN,REST IRREG.       13
HELIX    2  H2 LYS A  230  VAL A  235  5CONTIGUOUS WITH H3                 6
HELIX    3  H3 SER A  236  ASN A  245  1CONTIGUOUS WITH H2                10
SHEET    1   A 7 TYR A  20  THR A  21  0
SHEET    2   A 7 LYS A 156  PRO A 161 -1  N  CYS A 157   O  TYR A  20
[...]
SSBOND   1 CYS A   22    CYS A  157                          1555   1555  2.04
SSBOND   2 CYS A   42    CYS A   58                          1555   1555  2.02
[...]
```

- **Ligne 1.** Cette ligne `HEADER` contient :
  - le nom de la protéine : *HYDROLASE (SERINE PROTEINASE)*,
  - la date de dépôt de cette structure dans la banque de données : *26 octobre 1981*
  - et l'identifiant de la structure dans la PDB, on parle souvent de « code PDB » : *2PTN*.
- **Ligne 2.** `TITLE` correspond au titre de l'article scientifique dans lequel a été publié cette structure.
- **Lignes 4-6.** `COMPND` indique que la trypsine est composée d'une seule chaîne peptidique, appelée ici `A`.
- **Ligne 8.** `SOURCE` indique le nom scientifique de l'organisme dont provient cette protéine (ici, le bœuf).
- **Ligne 10.** `EXPDTA` précise la technique expérimentale employée pour déterminer cette structure. Ici, la cristallographie aux rayons X. Mais on peut également trouver *SOLUTION NMR* pour de la résonance magnétique nucléaire en solution, *ELECTRON MICROSCOPY* pour de la microscopie électronique.
- **Ligne 12.** `REMARK 2` précise, dans le cas d'une détermination par cristallographie aux rayons X, la résolution obtenue, ici 1,55 Angströms.
- **Ligne 14.** `DBREF` indique les liens éventuels vers d'autres banques de données. Ici, l'identifiant correspondant à cette protéine dans UniProt (*UNP*) est P00760[11].
- **Ligne 15-18.** `SEQRES` donnent à la séquence de la protéine. Les résidus sont représentés par leur code à trois lettres.
- **Lignes 20-22** et **23-24.** `HELIX` et `SHEET` correspondent aux structures secondaires hélices α et brin β de cette protéine. Ici, *H1 SER A 164 ILE A 176* indique qu'il y a une première hélice α (*H1*), comprise entre les résidus Ser164 et Ile176 de la chaîne A.
- **Lignes 26-27.** `SSBOND` indique les ponts disulfures. Ici, entre les résidus Cys22 et Cys157 et entre les résidus Cys42 et Cys58.

A.3.2 Coordonnées

Avec la même protéine, la partie coordonnées représente plus de 1 700 lignes. En voici quelques unes correspondantes au résidu leucine 99 :

11. https://www.uniprot.org/uniprot/P00760

```
[...]
ATOM    601  N   LEU A  99      10.007  19.687  17.536  1.00 12.25           N
ATOM    602  CA  LEU A  99       9.599  18.429  18.188  1.00 12.25           C
ATOM    603  C   LEU A  99      10.565  17.281  17.914  1.00 12.25           C
ATOM    604  O   LEU A  99      10.256  16.101  18.215  1.00 12.25           O
ATOM    605  CB  LEU A  99       8.149  18.040  17.853  1.00 12.25           C
ATOM    606  CG  LEU A  99       7.125  19.029  18.438  1.00 18.18           C
ATOM    607  CD1 LEU A  99       5.695  18.554  18.168  1.00 18.18           C
ATOM    608  CD2 LEU A  99       7.323  19.236  19.952  1.00 18.18           C
[...]
```

Chaque ligne correspond à un atome et débute par `ATOM` ou `HETATM`. `ATOM` désigne un atome de la structure de la biomolécule. `HETATM` est utilisé pour les atomes qui ne sont pas une biomolécule, comme les ions ou les molécules d'eau.

Toutes les lignes de coordonnées ont sensiblement le même format. Par exemple, pour la première ligne :

- `ATOM` (ou `HETATM`).
- `601` : le numéro de l'atome.
- `N` : le nom de l'atome. Ici, un atome d'azote du squelette peptidique. La structure complète du résidu leucine est représentée figure A.1.
- `LEU` : le résidu dont fait partie l'atome. Ici, une leucine.
- `A` : le nom de la chaîne peptidique.
- `99` : le numéro du résidu dans la protéine.
- `10.007` : la coordonnée x de l'atome.
- `19.687` : la coordonnée y de l'atome.
- `17.536` : la coordonnée z de l'atome.
- `1.00` : le facteur d'occupation, c'est-à-dire la probabilité de trouver l'atome à cette position dans l'espace en moyenne. Cette probabilité est inférieure à 1 lorsque, expérimentalement, on n'a pas pu déterminer avec une totale certitude la position de l'atome. Par exemple, dans le cas d'un atome très mobile dans une structure, qui est déterminé comme étant à deux positions possibles, chaque position aura alors la probabilité `0.50`.
- `12.25` : le facteur de température, qui est proportionnel à la mobilité de l'atome dans l'espace. Les atomes situés en périphérie d'une structure sont souvent plus mobiles que ceux situés au coeur de la structure.
- `N` : l'élément chimique de l'atome. Ici, l'azote.

Une documentation plus complète des différents champs qui constituent une ligne de coordonnées atomiques se trouve sur le site de la PDB [12].

Les résidus sont ensuite décrits les uns après les autres, atome par atome. Voici par exemple les premiers résidus de la trypsine bovine :

```
[...]
ATOM      1  N   ILE A  16      -8.155   9.648  20.365  1.00 10.68           N
ATOM      2  CA  ILE A  16      -8.150   8.766  19.179  1.00 10.68           C
ATOM      3  C   ILE A  16      -9.405   9.018  18.348  1.00 10.68           C
ATOM      4  O   ILE A  16     -10.533   8.888  18.870  1.00 10.68           O
ATOM      5  CB  ILE A  16      -8.091   7.261  19.602  1.00 10.68           C
ATOM      6  CG1 ILE A  16      -6.898   6.882  20.508  1.00  7.42           C
ATOM      7  CG2 ILE A  16      -8.178   6.281  18.408  1.00  7.42           C
ATOM      8  CD1 ILE A  16      -5.555   6.893  19.773  1.00  7.42           C
ATOM      9  N   VAL A  17      -9.224   9.305  17.090  1.00  9.63           N
ATOM     10  CA  VAL A  17     -10.351   9.448  16.157  1.00  9.63           C
ATOM     11  C   VAL A  17     -10.500   8.184  15.315  1.00  9.63           C
ATOM     12  O   VAL A  17      -9.496   7.688  14.748  1.00  9.63           O
ATOM     13  CB  VAL A  17     -10.123  10.665  15.222  1.00  9.63           C
ATOM     14  CG1 VAL A  17     -11.319  10.915  14.278  1.00 11.95           C
ATOM     15  CG2 VAL A  17      -9.737  11.970  15.970  1.00 11.95           C
[...]
```

Vous remarquerez que le numéro du premier résidu est 16 et non pas 1. Cela s'explique par la technique expérimentale utilisée qui n'a pas permis de déterminer la structure des 15 premiers résidus.

La structure de la trypsine bovine n'est constituée que d'une seule chaîne peptidique (notée `A`). Lorsqu'une structure est composée de plusieurs chaînes, comme dans le cas de la structure du récepteur GABAB 1 et 2 chez la drosophile

12. http://www.wwpdb.org/documentation/file-format-content/format33/sect9.html

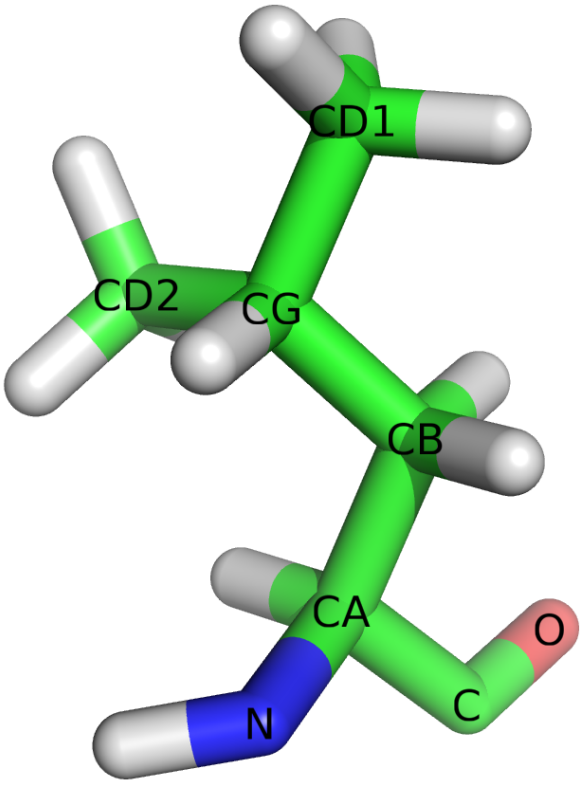


FIGURE A.1 – Structure tridimensionnelle d'un résidu leucine. Les noms des atomes sont indiqués en noir.

(code PDB 5X9X[13]) :

```
[...]
ATOM    762  HB1 ALA A  44      37.162  -2.955   2.220  1.00  0.00           H
ATOM    763  HB2 ALA A  44      38.306  -2.353   3.417  1.00  0.00           H
ATOM    764  HB3 ALA A  44      38.243  -1.621   1.814  1.00  0.00           H
TER     765      ALA A  44
ATOM    766  N   GLY B  95     -18.564   3.009  13.772  1.00  0.00           N
ATOM    767  CA  GLY B  95     -19.166   3.646  12.621  1.00  0.00           C
ATOM    768  C   GLY B  95     -20.207   2.755  11.976  1.00  0.00           C
[...]
```

La première chaîne est notée `A` et la seconde `B`. La séparation entre les deux chaînes est marquée par la ligne :

```
TER     765      ALA A  44
```

Dans un fichier PDB, chaque structure porte un nom de chaîne différent (par exemple : A, B, C', etc.).

Enfin, lorsque la structure est déterminée par RMN, il est possible que plusieurs structures soient présentes dans le même fichier PDB. Toutes ces structures, ou « modèles », sont des solutions possibles du jeu de contraintes mesurées expérimentalement en RMN. Voici un exemple, toujours pour la structure du récepteur GABAB 1 et 2 chez la drosophile :

13. http://www.rcsb.org/structure/5X9X

```
[...]
MODEL        1
ATOM      1  N   MET A   1     -27.283  -9.772   5.388  1.00  0.00           N
ATOM      2  CA  MET A   1     -28.233  -8.680   5.682  1.00  0.00           C
[...]
ATOM   1499  HG2 GLU B 139      36.113  -5.242   2.536  1.00  0.00           H
ATOM   1500  HG3 GLU B 139      37.475  -4.132   2.428  1.00  0.00           H
TER    1501      GLU B 139
ENDMDL
MODEL        2
ATOM      1  N   MET A   1     -29.736 -10.759   4.394  1.00  0.00           N
ATOM      2  CA  MET A   1     -28.372 -10.225   4.603  1.00  0.00           C
[...]
ATOM   1499  HG2 GLU B 139      36.113  -5.242   2.536  1.00  0.00           H
ATOM   1500  HG3 GLU B 139      37.475  -4.132   2.428  1.00  0.00           H
TER    1501      GLU B 139
ENDMDL
MODEL        2
ATOM      1  N   MET A   1     -29.736 -10.759   4.394  1.00  0.00           N
ATOM      2  CA  MET A   1     -28.372 -10.225   4.603  1.00  0.00           C
[...]
```

Chaque structure est encadrée par les lignes :

```
MODEL        n
```

et :

```
ENDMDL
```

où *n* est le numéro du modèle. Pour la structure du récepteur GABAB 1 et 2, il y a 20 modèles de décrits dans le fichier PDB.

A.3.3 Manipulation avec Python

Le module *Biopython* peut également lire un fichier PDB.

Voici comment charger la structure de la trypsine bovine :

```
1  from Bio.PDB import PDBParser
2  parser = PDBParser()
3  prot_id = "2PTN"
4  prot_file = "2PTN.pdb"
5  structure = parser.get_structure(prot_id, prot_file)
```

Remarque

Les fichiers PDB sont parfois (très) mal formatés. Si *Biopython* ne parvient pas à lire un tel fichier, remplacez alors **la ligne 2** par `parser = PDBParser(PERMISSIVE=1)`. Soyez néanmoins très prudent quant aux résultats obtenus.

Affichage du nom de la structure et de la technique expérimentale utilisée pour déterminer la structure :

```
1  print(structure.header["head"])
2  print(structure.header["structure_method"])
```

ce qui produit :

```
1  hydrolase (serine proteinase)
2  x-ray diffraction
```

Extraction des coordonnées de l'atome `N` du résidu Ile16 et de l'atome `CA` du résidu Val17 :

```
model = structure[0]
chain = model["A"]
res1 = chain[16]
res2 = chain[17]
print(res1.resname, res1["N"].coord)
print(res2.resname, res2["CA"].coord)
```

ce qui produit :

```
ILE [ -8.15499973   9.64799976  20.36499977]
VAL [-10.35099983   9.44799995  16.15699959]
```

L'objet `res1["N"].coord` est un *array* de *NumPy* (voir le chapitre 20 *Module NumPy*). On peut alors obtenir simplement les coordonnées x, y et z d'un atome :

```
print(res1["N"].coord[0], res1["N"].coord[1], res1["N"].coord[2])
```

ce qui produit :

```
-8.155 9.648 20.365
```

Remarque

Biopython utilise la hiérarchie suivante :
`structure > model > chain > residue > atom`
même lorsque la structure ne contient qu'un seul modèle. C'est d'ailleurs le cas ici, puisque la structure a été obtenue par cristallographie aux rayons X.

Enfin, pour afficher les coordonnées des carbones α (notés `CA`) des 10 premiers résidus (à partir du résidu 16, car c'est le premier résidu dont on connaît la structure) :

```
res_start = 16
model = structure[0]
chain = model["A"]
for i in range(10):
    idx = res_start + i
    print(chain[idx].resname, idx, chain[idx]["CA"].coord)
```

avec pour résultat :

```
ILE 16 [ -8.14999962   8.76599979  19.17900085]
VAL 17 [-10.35099983   9.44799995  16.15699959]
GLY 18 [-12.02099991   6.63000011  14.25899982]
GLY 19 [-10.90200043   3.89899993  16.68400002]
TYR 20 [-12.65100002   1.44200003  19.01600075]
THR 21 [-13.01799965   0.93800002  22.76000023]
CYS 22 [-10.02000046  -1.16299999  23.76000023]
GLY 23 [-11.68299961  -2.86500001  26.7140007 ]
ALA 24 [-10.64799976  -2.62700009  30.36100006]
ASN 25 [ -6.96999979  -3.43700004  31.02000046]
```

Il est aussi très intéressant (et formateur) d'écrire son propre *parser* de fichier PDB, c'est-à-dire un programme qui lit un fichier PDB (sans le module *Biopython*). Dans ce cas, la figure A.2 vous aidera à déterminer comment extraire les différentes informations d'une ligne de coordonnées `ATOM` ou `HETATM`.

Exemple : pour extraire le nom du résidu, il faut isoler le contenu des colonnes 18 à 20 du fichier PDB, ce qui correspond aux index de 17 à 19 pour une chaîne de caractères en Python (soit la tranche de chaîne de caractères `[17:20]`, car la première borne est incluse et la seconde exclue).

Pour lire le fichier PDB de la trypsine bovine (`2PTN.pdb`) et extraire (encore) les coordonnées des carbones α des 10 premiers résidus, nous pouvons utiliser le code suivant :

```
with open("2PTN.pdb", "r") as pdb_file:
    res_count = 0
    for line in pdb_file:
        if line.startswith("ATOM"):
            atom_name = line[12:16].strip()
            res_name = line[17:20].strip()
            res_num = int(line[22:26])
            if atom_name == "CA":
                res_count += 1
                x = float(line[30:38])
                y = float(line[38:46])
                z = float(line[46:54])
                print(res_name, res_num, x, y, z)
            if res_count >= 10:
                break
```

ce qui donne :

```
ILE 16 -8.15 8.766 19.179
VAL 17 -10.351 9.448 16.157
GLY 18 -12.021 6.63 14.259
GLY 19 -10.902 3.899 16.684
TYR 20 -12.651 1.442 19.016
THR 21 -13.018 0.938 22.76
CYS 22 -10.02 -1.163 23.76
GLY 23 -11.683 -2.865 26.714
ALA 24 -10.648 -2.627 30.361
ASN 25 -6.97 -3.437 31.02
```

Remarque

Pour extraire des valeurs numériques, comme des numéros de résidus ou des coordonnées atomiques, il ne faudra pas oublier de les convertir en entiers ou en *floats*.

A.4 Format XML, CSV et TSV

Les formats XML, CSV et TSV dont des formats de fichiers très largement utilisés en informatique. Ils sont également très utilisés en biologie. En voici quelques exemples :

A.4.1 XML

Le format XML est un format de fichier à balises qui permet de stocker quasiment n'importe quel type d'information de façon structurée et hiérarchisée. L'acronyme XML signifie *Extensible Markup Language* qui pourrait se traduire en français par « Langage de balisage extensible[14] ». Les balises dont il est question servent à délimiter du contenu :

```
<balise>contenu</balise>
```

La balise `<balise>` est une balise ouvrante. La balise `</balise>` est une balise fermante. Notez le caractère `/` qui marque la différence entre une balise ouvrante et une balise fermante.

Il existe également des balises vides, qui sont à la fois ouvrantes et fermantes :

```
<balise />
```

Une balise peut avoir certaines propriétés, appelées *attributs*, qui sont définies, dans la balise ouvrante. Par exemple :

```
<balise propriété1=valeur1 propriété2=valeur2>contenu</balise>
```

Un attribut est un couple nom et valeur (par exemple `propriété1` est un nom et `valeur1` est la valeur associée).

Enfin, les balises peuvent être imbriquées les unes dans les autres :

14. https://fr.wikipedia.org/wiki/Extensible_Markup_Language

```
<protein>
<element>élément 1</element>
<element>élément 2</element>
<element>élément 3</element>
</protein>
```

Dans cet exemple, nous avons trois balises `element` qui sont contenues dans une balise `protein`.

Voici un autre exemple avec l'enzyme trypsine[15] humaine (code P07477[16]), telle qu'on peut la trouver décrite dans la base de données UniProt :

```
<?xml version='1.0' encoding='UTF-8'?>
<uniprot xmlns="http://uniprot.org/uniprot" xmlns:xsi=[...]>
<entry dataset="Swiss-Prot" created="1988-04-01" modified="2018-09-12" [...]>
<accession>P07477</accession>
<accession>A1A509</accession>
[...]
<gene>
<name type="primary">PRSS1</name>
<name type="synonym">TRP1</name>
<name type="synonym">TRY1</name>
</gene>
[...]
<sequence length="247" mass="26558" checksum="DD49A487B8062813" [...]>
MNPLLILTFVAAALAAPFDDDDKIVGGYNCEENSVPYQVSLNSGYHFCGGSLINEQWVVS
AGHCYKSRIQVRLGEHNIEVLEGNEQFINAAKIIRHPQYDRKTLNNDIMLIKLSSRAVIN
ARVSTISLPTAPPATGTKCLISGWGNTASSGADYPDELQCLDAPVLSQAKCEASYPGKIT
SNMFCVGFLEGGKDSCQGDSGGPVVCNGQLQGVVSWGDGCAQKNKPGVYTKVYNYVKWIK
NTIAANS
</sequence>
</entry>
[...]
</uniprot>
```

- La **ligne 1** indique que nous avons bien un fichier au format XML.
- La **ligne 3** indique que nous avons une entrée UniProt. Il s'afit d'une balise ouvrante avec plusieurs attributs (`dataset="Swiss-Prot"`, `created="1988-04-01"`, etc.).
- Les **lignes 4 et 5** précisent les numéros d'accession dans la base de données UniProt qui font référence à cette même protéine.
- Les **lignes 8-10** listent les quatre gènes correspondants à cette protéine. Le premier gène porte l'attribut `type ="primary"` et indique qu'il s'agit du nom officiel du gène de la trypsine. L'attribut `type="synonym"` pour les autres gènes indique qu'il s'agit bien de noms synonymes pour le gène `PRSS1`.
- Les **lignes 13-18** contiennent la séquence de la trypsine. Dans les attributs de la balise `<sequence>`, on retrouve, par exemple, la taille de la protéine (`length="247"`).

Voici un exemple de code Python pour manipuler le fichier XML de la trypsine humaine :

```
1  from lxml import etree
2  import re
3
4  with open("P07477.xml") as xml_file:
5      xml_content = xml_file.read()
6
7  xml_content = re.sub("<uniprot [^>]+>", "<uniprot>", xml_content)
8
9  root = etree.fromstring(xml_content.encode("utf-8"))
10
11 for gene in root.xpath("/uniprot/entry/gene/name"):
12     print(f"gene : {gene.text} ({gene.get('type')})")
13
14 sequence = root.xpath("/uniprot/entry/sequence")[0]
15 print(f"sequence: {sequence.text.strip()}")
16 print(f"length: {sequence.get('length')}")
```

- **Ligne 1.** On utilise le sous-module `etree` du module *lxml* pour lire le fichier XML.

15. https://www.uniprot.org/uniprot/P07477
16. https://www.uniprot.org/uniprot/P07477.xml

- **Ligne 2.** On utilise le module d'expressions régulières *re* pour supprimer tous les attributs de la balise `uniprot` (ligne 7). Nous ne rentrerons pas dans les détails, mais ces attributs rendent plus complexe la lecture du fichier XML.
- **Ligne 9.** La variable `root` contient le fichier XML prêt à être manipulé.
- **Ligne 11.** On recherche les noms des gènes (balises `<name></name>`) associés à la trypsine. Pour cela, on utilise la méthode `.xpath()`, avec comme argument l'enchaînement des différentes balises qui conduisent aux noms des gènes.
- **Ligne 12.** Pour chaque nom de gène, on va afficher son contenu (`gene.text`) et la valeur associée à l'attribut `type` avec la méthode `.get("type")`.
- **Ligne 14.** On stocke dans la variable `sequence` la balise associée à la séquence de la protéine. Comme `root.xpath("/uniprot/entry/sequence")` renvoie un itérateur et qu'il n'y a qu'une seule balise séquence, on prend ici le seul et unique élément `root.xpath("/uniprot/entry/sequence")[0]`.
- **Ligne 15.** On affiche le contenu de la séquence `sequence.text`, nettoyé d'éventuels retours chariots ou espaces `sequence.text.strip()`.
- **Ligne 16.** On affiche la taille de la séquence en récupérant la valeur de l'attribut `length` (toujours de la balise `<sequence></sequence>`).

Le résultat obtenu est le suivant :

```
gene : PRSS1 (primary)
gene : TRP1 (synonym)
gene : TRY1 (synonym)
gene : TRYP1 (synonym)
sequence: MNPLLILTFVAAALAAPFDDDDKIVGGYNCEENSVPYQVSLNSGYHFCGGSLINEQWVVS
AGHCYKSRIQVRLGEHNIEVLEGNEQFINAAKIIRHPQYDRKTLNNDIMLIKLSSRAVIN
ARVSTISLPTAPPATGTKCLISGWGNTASSGADYPDELQCLDAPVLSQAKCEASYPGKIT
SNMFCVGFLEGGKDSCQGDSGGPVVCNGQLQGVVSWGDGCAQKNKPGVYTKVYNYVKWIK
NTIAANS
length: 247
```

A.4.2 CSV et TSV

A.4.2.1 Définition des formats

L'acronyme CSV signifie « *Comma-Separated values* », qu'on peut traduire littéralement par « valeurs séparées par des virgules ». De façon similaire, TSV signifie « *Tabulation-Separated Values* », soit des « valeurs séparées par des tabulations ».

Ces deux formats sont utiles pour stocker des données structurées sous forme de tableau, comme vous pourriez l'avoir dans un tableur.

À titre d'exemple, le tableau ci-dessous liste les structures associées à la transferrine, protéine présente dans le plasma sanguin et impliquée dans la régulation du fer. Ces données proviennent de la *Protein Data Bank* (PDB). Pour chaque protéine (*PDB ID*) est indiqué le nom de l'organisme associé (*Source*), la date à laquelle cette structure a été déposée dans la PDB (*Deposit Date*), le nombre d'acides aminés de la protéine et sa masse moléculaire (*MW*).

PDB ID	Source	Deposit Date	Length	MW
1A8E	Homo sapiens	1998-03-24	329	36408.40
1A8F	Homo sapiens	1998-03-25	329	36408.40
1AIV	Gallus gallus	1997-04-28	686	75929.00
1AOV	Anas platyrhynchos	1996-12-11	686	75731.80
[...]	[...]	[...]	[...]	[...]

Voici maintenant l'équivalent en CSV[17] :

17. https://python.sdv.u-paris.fr/data-files/transferrin_report.csv

```
PDB ID,Source,Deposit Date,Length,MW
1A8E,Homo sapiens,1998-03-24,329,36408.40
1A8F,Homo sapiens,1998-03-25,329,36408.40
1AIV,Gallus gallus,1997-04-28,686,75929.00
1AOV,Anas platyrhynchos,1996-12-11,686,75731.80
[...]
```

Sur chaque ligne, les différentes valeurs sont séparées par une virgule. La première ligne contient le nom des colonnes et est appelée ligne d'en-tête.

L'équivalent en TSV[18] est :

```
PDB ID  Source  Deposit Date    Length  MW
1A8E    Homo sapiens    1998-03-24  329 36408.40
1A8F    Homo sapiens    1998-03-25  329 36408.40
1AIV    Gallus gallus   1997-04-28  686 75929.00
1AOV    Anas platyrhynchos  1996-12-11  686 75731.80
[...]
```

Sur chaque ligne, les différentes valeurs sont séparées par une tabulation.

Attention

Le caractère tabulation est un caractère invisible « élastique », c'est-à-dire qu'il a une largeur variable suivant l'éditeur de texte utilisé. Par exemple, dans la ligne d'en-tête, l'espace entre *PDB ID* et *Source* apparaît comme différent de l'espace entre *Deposit Date* et *Length* alors qu'il y a pourtant une seule tabulation à chaque fois.

A.4.2.2 Lecture

En Python, le module *csv* de la bibliothèque standard est très pratique pour lire et écrire des fichiers au format CSV et TSV. Nous vous conseillons de lire la documentation très complète sur ce module[19].

Voici un exemple :

```
1  import csv
2
3  with open("transferrin_report.csv") as f_in:
4      f_reader = csv.DictReader(f_in)
5      for row in f_reader:
6          print(row["PDB ID"], row["Deposit Date"], row["Length"])
```

- **Ligne 1.** Chargement du module *csv*.
- **Ligne 3.** Ouverture du fichier.
- **Ligne 4.** Utilisation du module *csv* pour lire le fichier CSV comme un dictionnaire (fonction `DictReader()`). La ligne d'en-tête est utilisée automatiquement pour définir les clés du dictionnaire.
- **Ligne 5.** Parcours de toutes les lignes du fichier CSV.
- **Ligne 6.** Affichage des champs correspondants à *PDB ID*, *Deposit Date*, *Length*.

Le résultat obtenu est :

```
1A8E 1998-03-24 329
1A8F 1998-03-25 329
1AIV 1997-04-28 686
1AOV 1996-12-11 686
[...]
```

Il suffit de modifier légèrement le script précédent pour lire un fichier TSV :

18. https://python.sdv.u-paris.fr/data-files/transferrin_report.tsv
19. https://docs.python.org/fr/3.7/library/csv.html

```
1  import csv
2
3  with open("transferrin_PDB_report.tsv") as f_in:
4      f_reader = csv.DictReader(f_in, delimiter="\t")
5      for row in f_reader:
6          print(row["PDB ID"], row["Deposit Date"], row["Length"])
```

- **Ligne 3.** Modification du nom du fichier lu.
- **Ligne 4.** Utilisation de l'argument `delimiter="\t"`, qui indique que les champs sont séparés par des tabulations.

Le résultat obtenu est strictement identique au précédent.

A.4.2.3 Écriture

Voici un exemple d'écriture de fichier CSV :

```
1  import csv
2
3  with open("test.csv", "w") as f_out:
4      fields = ["Name", "Quantity"]
5      f_writer = csv.DictWriter(f_out, fieldnames=fields)
6      f_writer.writeheader()
7      f_writer.writerow({"Name": "girafe", "Quantity":5})
8      f_writer.writerow({"Name": "tigre", "Quantity":3})
9      f_writer.writerow({"Name": "singe", "Quantity":8})
```

- **Ligne 3.** Ouverture du fichier `test.csv` en lecture.
- **Ligne 4.** Définition du nom des colonnes (*Name* et *Quantity*).
- **Ligne 5.** Utilisation du module *csv* pour écrire un fichier CSV à partir d'un dictionnaire.
- **Ligne 6**. Écriture des noms des colonnes.
- **Ligne 7-9.** Écriture de trois lignes. Pour chaque ligne, un dictionnaire dont les clefs sont les noms des colonnes est fourni comme argument à la méthode `.writerow()`.

Le contenu du fichier `test.csv` est alors :

```
Name,Quantity
girafe,5
tigre,3
singe,8
```

De façon très similaire, l'écriture d'un fichier TSV est réalisée avec le code suivant :

```
1  import csv
2
3  with open("test.tsv", "w") as f_out:
4      fields = ["Name", "Quantity"]
5      f_writer = csv.DictWriter(f_out, fieldnames=fields, delimiter="\t")
6      f_writer.writeheader()
7      f_writer.writerow({"Name": "girafe", "Quantity":5})
8      f_writer.writerow({"Name": "tigre", "Quantity":3})
9      f_writer.writerow({"Name": "singe", "Quantity":8})
```

- **Ligne 3.** Modification du nom du fichier en écriture.
- **Ligne 5.** Utilisation de l'argument `delimiter="\t"`, qui indique que les champs sont séparés par des tabulations.

Le contenu du fichier `test.tsv` est :

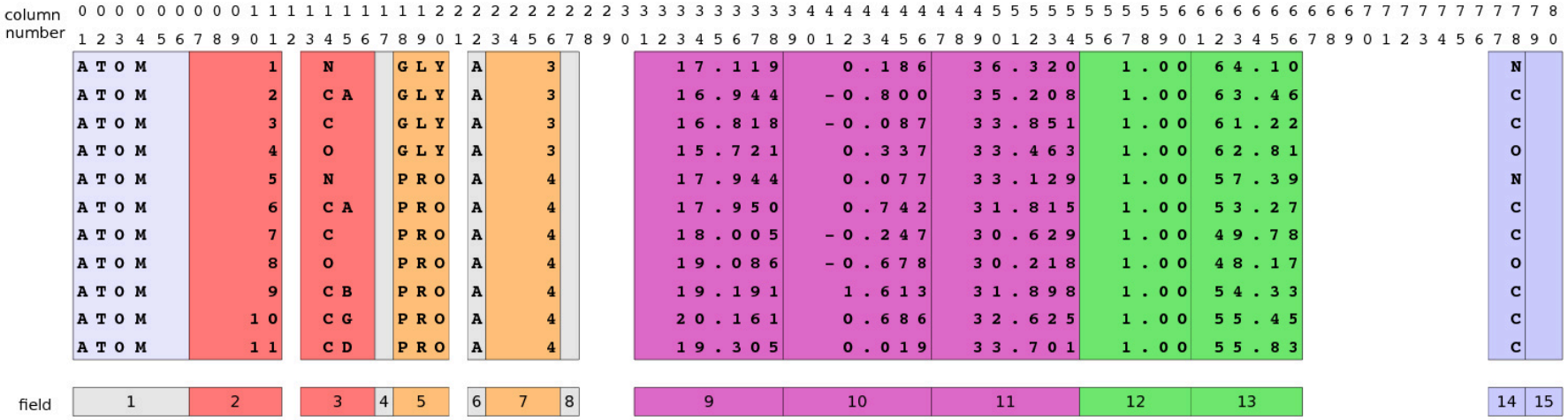
```
Name    Quantity
girafe  5
tigre   3
singe   8
```

Vous êtes désormais capables de lire et écrire des fichiers aux formats CSV et TSV. Les codes que nous vous avons proposés ne sont que des exemples. À vous de poursuivre l'exploration du module *csv*.

Remarque

Le module *pandas* décrit dans le chapitre 22 *Module Pandas* est tout à fait capable de lire et écrire des fichiers CSV et TSV. Nous vous conseillons de l'utiliser si vous analysez des données avec ces types de fichiers.

PDB file format 3.3



field	definition	length	format	range	Python extraction
1	'ATOM ' or 'HETATM'	6	{:6s}	01—06	[0:6]
2	atom serial number	5	{:5d}	07—11	[6:11]
		1			
3	atom name	4	{:^4s}	13—16	[12:16]
4	alternate location indicator	1	{:1s}	17	[16:17]
5	residue name	3	{:3s}	18—20	[17:20]
		1			
6	chain identifier	1	{:1s}	22	[21:22]
7	residue sequence number	4	{:4d}	23—26	[22:26]
8	code for insertion of residues	1	{:1s}	27	[26:27]
		3			
9	orthogonal coordinates for X in Angstrom	8	{:8.3f}	31—38	[30:38]
10	orthogonal coordinates for Y in Angstrom	8	{:8.3f}	39—46	[38:46]
11	orthogonal coordinates for Z in Angstrom	8	{:8.3f}	47—54	[46:54]
12	occupancy	6	{:6.2f}	55—60	[54:60]
13	temperature factor	6	{:6.2f}	61—66	[60:66]
		10			
14	element symbol	2	{:>2s}	77—78	[76:78]
15	charge on the atom	2	{:2s}	79—80	[78:80]

Python formated string (old): `'%-6s%5d %4s%1s%3s %1s%4d%1s %8.3f%8.3f%8.3f%6.2f%6.2f %2s%2s'`

Python formated string (Python 2.7 and Python 3.x): `'{:6s}{:5d} {:^4s}{:1s}{:3s} {:1s}{:4d}{:1s} {:8.3f}{:8.3f}{:8.3f}{:6.2f}{:6.2f} {:>2s}{:2s}'`

PDB File Format - Contents Guide Version 3.30 (Nov. 21, 2012)

reference taken from http://www.wwpdb.org/documentation/file-format-content/format33/sect9.html#ATOM

PP 04/09/2016

FIGURE A.2 – Format PDB et les différents champs de coordonnées.

ANNEXE B

Installation de Python

Attention

La procédure d'installation ci-dessous a été testée avec la version Miniconda `Latest - Conda 24.5.0 Python 3.12.4 released Jun 26, 2024`.

Python est déjà présent sous Linux ou Mac OS X et s'installe très facilement sous Windows. Toutefois, nous décrivons dans cet ouvrage l'utilisation de modules supplémentaires qui sont très utiles en bioinformatique (*NumPy*, *scipy*, *matplotlib*, *pandas*, *Biopython*), mais également les *notebooks* Jupyter.

On va donc utiliser un gestionnaire de paquets qui va installer ces modules supplémentaires. On souhaite également que ce gestionnaire de paquets soit disponible pour Windows, Mac OS X et Linux. Fin 2018, il y a deux grandes alternatives :

1. **Anaconda** et **Miniconda** : Anaconda[1] est une distribution complète de Python qui contient un gestionnaire de paquets très puissant nommé *conda*. Anaconda installe de très nombreux paquets et outils mais nécessite un espace disque de plusieurs gigaoctets. Miniconda[2] est une version allégée d'Anaconda, donc plus rapide à installer et occupant peu d'espace sur le disque dur. Le gestionnaire de paquet *conda* est aussi présent dans Miniconda.
2. **Pip** : pip[3] est le gestionnaire de paquets de Python et qui est systématiquement présent depuis la version 3.4.

B.1 Que recommande-t-on pour l'installation de Python ?

Quel que soit le système d'exploitation, nous recommandons l'utilisation de Miniconda dont la procédure d'installation est détaillée ci-dessous pour Windows, Mac OS X et Linux. Le gestionnaire de paquets *conda* est très efficace. Il gère la version de Python et les paquets compatibles avec cette dernière de manière optimale.

Par ailleurs, nous vous recommandons vivement la lecture de la rubrique sur les éditeurs de texte. Il est en effet fondamental d'utiliser un éditeur robuste et de savoir le configurer pour « pythonner » efficacement.

Enfin, dans tout ce qui suit, nous partons du principe que vous installerez Miniconda **en tant qu'utilisateur**, et non pas en tant qu'administrateur. Autrement dit, vous n'aurez pas besoin de droits spéciaux pour pouvoir installer Miniconda et les autres modules nécessaires. La procédure proposée a été testée avec succès sous Windows 10 et 11, Mac OS X, Ubuntu 22.04 et 24.04).

1. https://www.anaconda.com/
2. https://conda.io/miniconda.html
3. https://pip.pypa.io/en/stable/

Depuis quelques années, Windows 10 (et 11) propose le WSL [4] (Windows Subsystem for Linux). Le WSL permet de lancer un terminal Linux au sein de Windows et propose (quasiment) toutes les fonctionnalités disponibles sous un vrai système Linux. Nous ne détaillons par comment l'installer, mais vous pouvez vous référer à la page d'installation sur le site de Microsoft [5]. Si vous avez installé WSL sur votre ordinateur, nous vous recommandons de suivre la procédure ci-dessous comme si vous étiez sous Linux (rubrique *Installation de Python avec Miniconda pour Linux*), plutôt que d'installer la version Windows.

B.2 Installation de Python avec Miniconda

Nous vous conseillons l'installation de la distribution Miniconda [6] qui présente l'avantage d'installer Python et un puissant gestionnaire de paquets appelé *conda*. Dans toute la suite de cette annexe, l'indication avec le $ et un espace comme suit :

```
$
```

signifie l'invite d'un *shell* quel qu'il soit (PowerShell sous Windows, bash sous Mac OS X et Linux).

B.2.1 Installation de Python avec Miniconda pour Linux

Dans un navigateur internet, ouvrez la page du site Miniconda https://conda.io/miniconda.html puis cliquez sur le lien *Miniconda3 Linux 64-bit* correspondant à Linuxet et Python 3.12.

Vous allez télécharger un fichier dont le nom ressemble à quelque chose du type :
`Miniconda3-latest-Linux-x86_64.sh`.

Dans un *shell*, lancez l'installation de Miniconda avec la commande :

```
$ bash Miniconda3-latest-Linux-x86_64.sh
```

Dans un premier temps, validez la lecture de la licence d'utilisation :

```
Welcome to Miniconda3 py312_24.5.0-0

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>>
```

Comme demandé, appuyez sur la touche *Entrée*. Faites ensuite défiler la licence d'utilisation avec la touche *Espace*. Tapez `yes` puis appuyez sur la touche *Entrée* pour valider :

```
Do you accept the license terms? [yes|no]
[no] >>> yes
```

Le programme d'installation vous propose ensuite d'installer Miniconda dans le répertoire `miniconda3` dans votre répertoire personnel. Par exemple, dans le répertoire `/home/pierre/miniconda3` si votre nom d'utilisateur est `pierre`. Validez cette proposition en appuyant sur la touche *Entrée* :

```
Miniconda3 will now be installed into this location:
/home/pierre/miniconda3

  - Press ENTER to confirm the location
  - Press CTRL-C to abort the installation
  - Or specify a different location below

[/home/pierre/miniconda3] >>>
```

Le programme d'installation va alors installer Python et le gestionnaire de paquets *conda*.

Cette étape terminée, le programme d'installation vous propose d'initialiser *conda* pour que celui-ci soit accessible à chaque fois que vous ouvrez un *shell*. Nous vous conseillons d'accepter en tapant `yes` puis en appuyant sur la touche *Entrée*.

4. https://fr.wikipedia.org/wiki/Windows_Subsystem_for_Linux
5. https://learn.microsoft.com/fr-fr/windows/wsl/install
6. https://conda.io/miniconda.html

```
Do you wish the installer to initialize Miniconda3
by running conda init? [yes|no]
[no] >>> yes
```

L'installation de Miniconda est terminée. L'espace utilisé par Miniconda sur votre disque dur est d'environ 450 Mo.

B.2.1.1 Test de l'interpréteur Python

Ouvrez un nouveau *shell*. Vous devriez voir dans votre invite la chaîne `(base)` indiquant que l'environnement conda de base est activé. À partir de maintenant, lorsque vous taperez la commande `python`, c'est le Python 3 de Miniconda qui sera lancé :

```
$ python
Python 3.12.4 | packaged by Anaconda, Inc. | (main, Jun 18 2024, 15:12:24) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Quittez Python en tapant la commande `exit()` puis appuyant sur la touche *Entrée*.

B.2.1.2 Test du gestionnaire de paquets *conda*

De retour dans le *shell*, testez si le gestionnaire de paquets *conda* est fonctionnel. Tapez la commande `conda` dans le *shell*, vous devriez avoir la sortie suivante :

```
$ conda
usage: conda [-h] [-v] [--no-plugins] [-V] COMMAND ...

conda is a tool for managing and deploying applications, environments and packages.

options:
  -h, --help            Show this help message and exit.
[...]
```

Si c'est bien le cas, bravo, *conda* et bien installé et vous pouvez passez à la suite (rendez-vous à la rubrique Installation des modules supplémentaires) !

B.2.1.3 Désinstallation de Miniconda

Si vous souhaitez supprimer Miniconda, rien de plus simple, il suffit de suivre ces deux étapes :

Étape 1. Supprimer le répertoire de Miniconda. Par exemple pour l'utilisateur `pierre` :

```
$ rm -rf /home/pierre/miniconda3
```

Étape 2. Dans le fichier de configuration du *shell Bash*, supprimer les lignes comprises entre

```
# >>> conda initialize >>>
```

et

```
# <<< conda initialize <<<
```

B.2.2 Installation de Python avec Miniconda pour Mac OS X

Dans un navigateur internet, ouvrez la page du site Miniconda https://conda.io/miniconda.html puis cliquez sur le lien *Miniconda3 macOS Intel x86 64-bit bash* correspondant à Mac OS X et Python 3.12.

Vous allez télécharger un fichier dont le nom ressemble à quelque chose du type :

`Miniconda3-latest-MacOSX-x86_64.sh`.

Le système d'exploitation Mac OS X étant basé sur Unix, la suite de la procédure est en tout point identique à la procédure détaillée à la rubrique précédente pour Linux.

Donc, lancez la commande :

```
$ bash Miniconda3-latest-MacOSX-x86_64.sh
```

puis suivez les mêmes instructions que dans la rubrique précédente (la seule petite subtilité est pour le chemin, choisissez `/User/votre_nom_utilisateur/miniconda3` sous Mac au lieu de `/home/votre_nom_utilisateur/miniconda3` sous Linux).

B.2.3 Installation de Python avec Miniconda pour Windows 10 et 11

Dans cette rubrique, nous détaillons l'installation de Miniconda sous Windows.

Attention

Nous partons du principe qu'aucune version d'Anaconda, Miniconda, ou encore de Python « classique » (obtenue sur le site officiel de Python [7]) n'est installée sur votre ordinateur. Si tel est le cas, nous vous recommandons vivement de la désinstaller pour éviter des conflits de version.

- Dans un navigateur internet, ouvrez la page du site Miniconda `https://conda.io/miniconda.html` puis cliquez sur le lien *Miniconda3 Windows 64-bit* correspondant à Windows et Python 3.12. Vous allez télécharger un fichier dont le nom ressemble à quelque chose du type : `Miniconda3-latest-Windows-x86_64.exe`.
- Une fois téléchargé, double-cliquez sur ce fichier, cela lancera l'installateur de Miniconda :

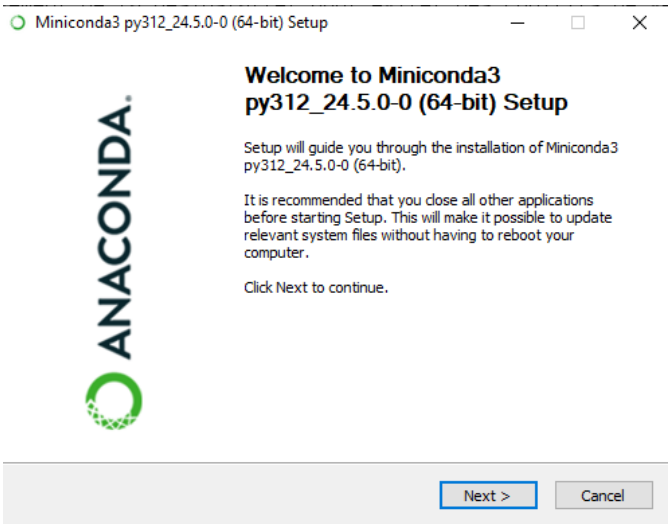


FIGURE B.1 – Installation Miniconda étape 1.

- Cliquez sur *Next*, vous arrivez alors sur l'écran suivant :
- Lisez la licence et (si vous êtes d'accord) cliquez sur *I agree*. Vous aurez ensuite :
- Gardez le choix de l'installation seulement pour vous (case cochée à *Just me (recommended)*), puis cliquez sur *Next*. Vous aurez ensuite :
- L'installateur vous demande où installer Miniconda, nous vous recommandons de laisser le choix par défaut (ressemblant à `C:\Users\votre_nom_utilisateur\Miniconda3`). Cliquez sur *Next*, vous arriverez sur :
- Gardez la case *Register Anaconda as my default Python 3.12* cochée et ne cochez pas la case *Add Anaconda to my PATH environment variable*. Vous pouvez garder la case *Create Shortcuts* cochée. Cliquez ensuite sur *Install*, l'installation se lance et durera quelques minutes :
- À la fin, vous obtiendrez :
- Décochez les cases *Learn more about Anaconda Cloud* et *Learn how to get started with Anaconda* et cliquez sur *Finish*. Miniconda est maintenant installé.

7. `https://www.python.org/downloads/`

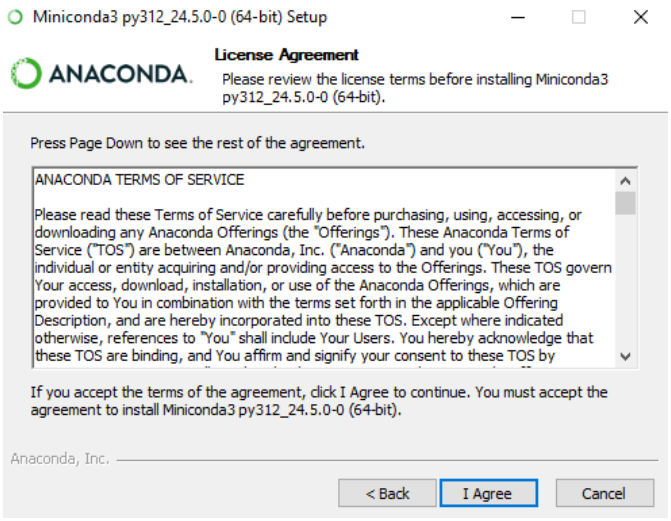


FIGURE B.2 – Installation Miniconda étape 2.

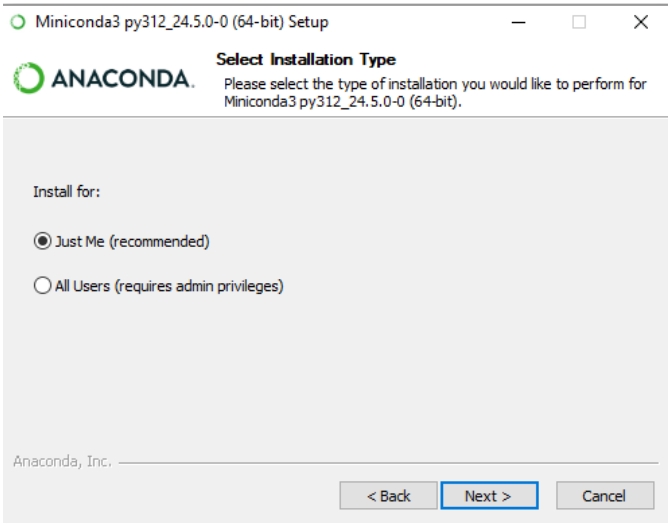


FIGURE B.3 – Installation Miniconda étape 3.

B.2.3.1 Initialisation de conda

Il nous faut maintenant initialiser *conda*. Cette manipulation va permettre de le rendre visible dans n'importe quel *shell* Powershell.

L'installateur a en principe ajouté des nouveaux raccourcis dans le Menu Démarrer contenant le mot `Anaconda` :

- `Anaconda Powershell Prompt (Miniconda3)` : pour lancer un *shell* Powershell (shell standard de Windows équivalent du bash sous Linux) avec *conda* qui est activé correctement ;
- `Anaconda Prompt (Miniconda3)` : même chose mais avec le *shell* nommé cmd ; ce vieux shell est limité et nous vous en déconseillons l'utilisation.

Nous allons maintenant initialiser *conda* « à la main ». Cliquez sur `Anaconda Powershell Prompt (Miniconda3)` qui va lancer un Powershell avec *conda* activé, puis tapez la commande `conda init` :

Lorsque vous presserez la touche Entrée vous obtiendrez une sortie de ce style :

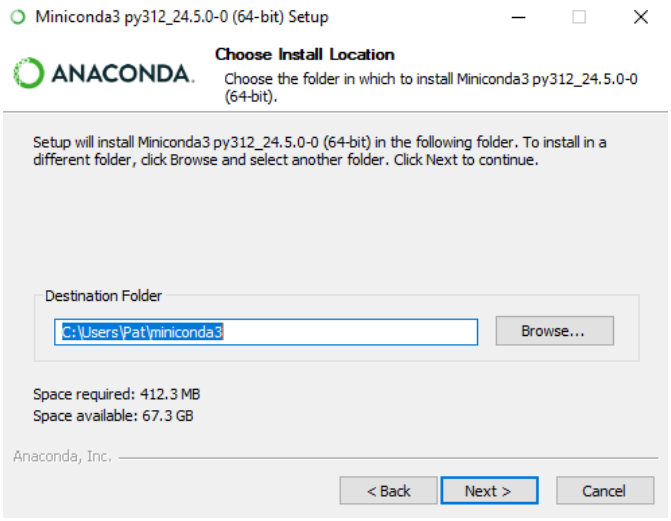


FIGURE B.4 – Installation Miniconda étape 4.

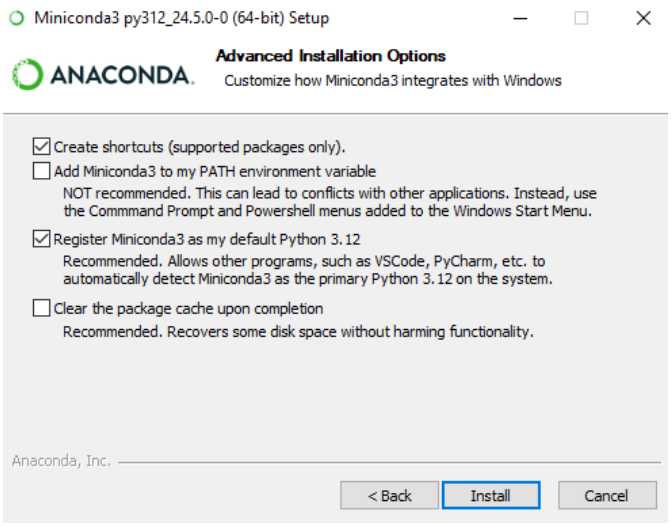


FIGURE B.5 – Installation Miniconda étape 5.

```
$ conda init
no change     C:\Users\Pat\miniconda3\Scripts\conda.exe
no change     C:\Users\Pat\miniconda3\Scripts\conda-env.exe
no change     C:\Users\Pat\miniconda3\Scripts\conda-script.py
no change     C:\Users\Pat\miniconda3\Scripts\conda-env-script.py
no change     C:\Users\Pat\miniconda3\condabin\conda.bat
no change     C:\Users\Pat\miniconda3\Library\bin\conda.bat
no change     C:\Users\Pat\miniconda3\condabin\_conda_activate.bat
no change     C:\Users\Pat\miniconda3\condabin\rename_tmp.bat
no change     C:\Users\Pat\miniconda3\condabin\conda_auto_activate.bat
no change     C:\Users\Pat\miniconda3\condabin\conda_hook.bat
no change     C:\Users\Pat\miniconda3\Scripts\activate.bat
no change     C:\Users\Pat\miniconda3\condabin\activate.bat
no change     C:\Users\Pat\miniconda3\condabin\deactivate.bat
modified      C:\Users\Pat\miniconda3\Scripts\activate
modified      C:\Users\Pat\miniconda3\Scripts\deactivate
modified      C:\Users\Pat\miniconda3\etc\profile.d\conda.sh
modified      C:\Users\Pat\miniconda3\etc\fish\conf.d\conda.fish
no change     C:\Users\Pat\miniconda3\shell\condabin\Conda.psm1
modified      C:\Users\Pat\miniconda3\shell\condabin\conda-hook.ps1
no change     C:\Users\Pat\miniconda3\Lib\site-packages\xontrib\conda.xsh
modified      C:\Users\Pat\miniconda3\etc\profile.d\conda.csh
modified      C:\Users\Pat\Documents\WindowsPowerShell\profile.ps1
modified      HKEY_CURRENT_USER\Software\Microsoft\Command Processor\AutoRun

==> For changes to take effect, close and re-open your current shell. <==
$ conda init
no change     C:\Users\Pat\Miniconda3\Scripts\conda.exe
no change     C:\Users\Pat\Miniconda3\Scripts\conda-env.exe
```

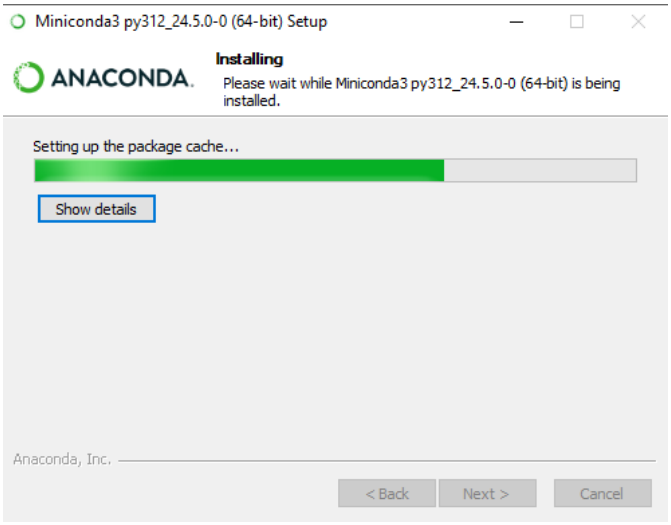


FIGURE B.6 – Installation Miniconda étape 6.

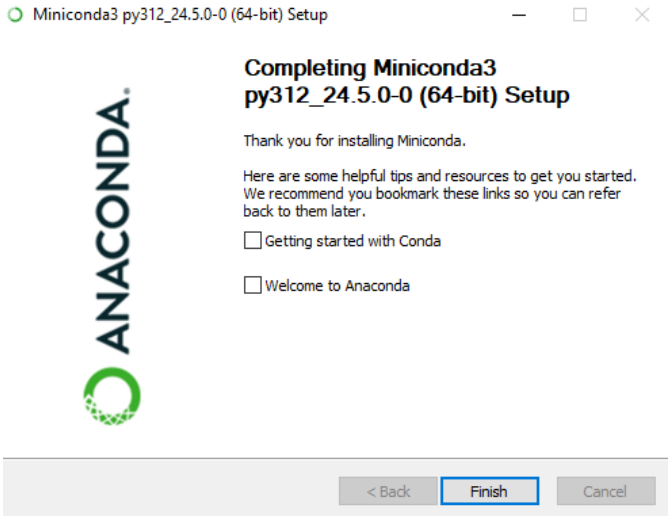


FIGURE B.7 – Installation Miniconda étape 7.

Notez que cette manipulation créera automatiquement un fichier `C:\Users\nom_utilisateur\Documents\WindowsPowerShell\profile.ps1`.

Ce fichier sera exécuté à chaque lancement d'un Powershell (équivalent du `.bashrc` sous bash) et fera en sorte que *conda* soit bien activé.

B.2.3.2 Test de l'interpréteur Python

Nous sommes maintenant prêts à tester l'interpréteur Python. En premier lieu, il faut lancer un *shell* PowerShell. Pour cela, cliquez sur le bouton Windows et tapez `powershell`. Vous devriez voir apparaitre le menu suivant :

Cliquez sur l'icône `Windows PowerShell`, cela va lancer un *shell* PowerShell avec un fond bleu (couleur que l'on peut bien sûr modifier en cliquant sur la petite icône représentant un terminal dans la barre de titre). En principe, l'invite du shell doit ressembler à `(base) PS C:\Users\Pat>`. La partie `(base)` indique que conda a bien été activé suite à l'initialisation faite si dessus (plus exactement c'est son environnement de base qui est activé, mais ça ne nous importe pas pour l'instant). Pour tester si Python est bien installé, il suffit alors de lancer l'interpréteur Python en tapant la commande `python` :

Si tout s'est bien passé, vous devriez avoir un affichage de ce style :

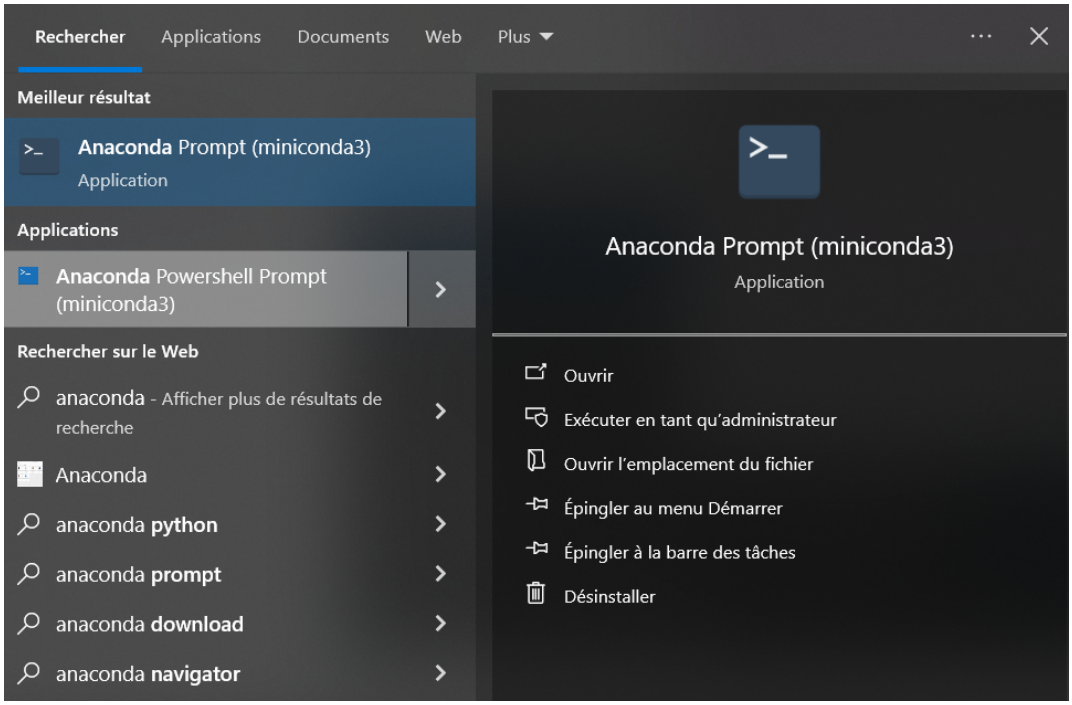


FIGURE B.8 – Menu Anaconda Powershell Prompt

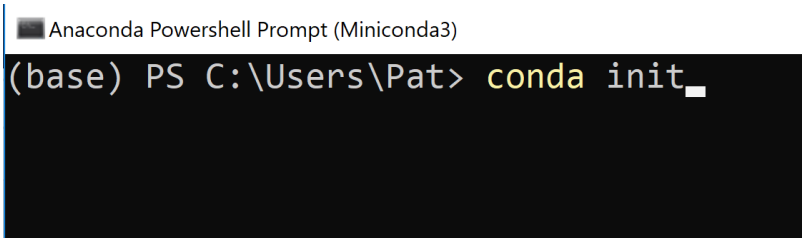


FIGURE B.9 – Initialisation de conda

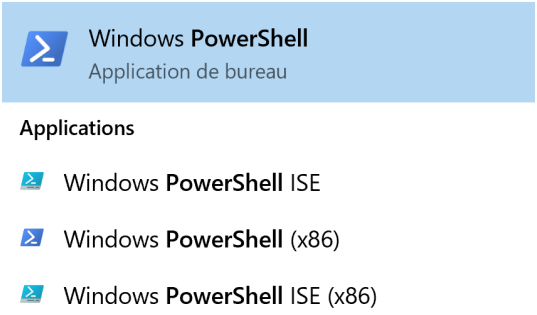


FIGURE B.10 – Menu pour lancer un PowerShell.

```
(base) PS C:\Users\Pat> python
Python 3.12.4 | packaged by Anaconda, Inc. | (main, Jun 18 2024, 15:03:56) [MSC v.1929 64 bit (AMD64)] on
    win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Cela signifie que vous êtes bien dans l'interpréteur Python. À partir de là vous pouvez taper `exit()` puis appuyer sur

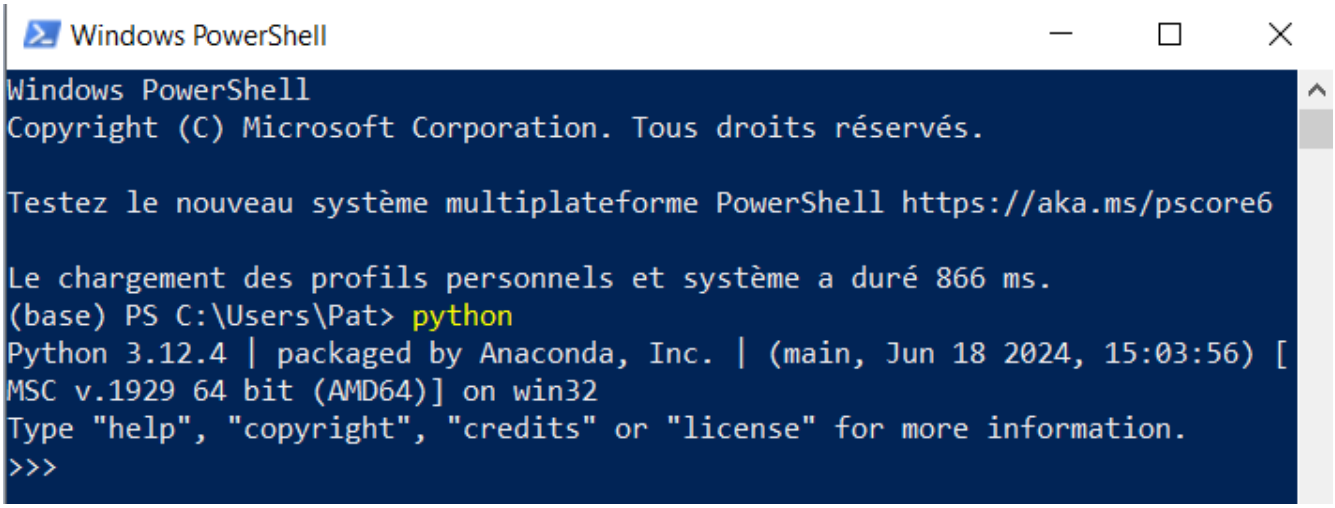


FIGURE B.11 – Lancement de l'interpréteur Python dans un PowerShell.

la touche *Entrée* pour sortir de l'interpréteur Python.

B.2.3.3 Test du gestionnaire de paquets *conda*

Une fois revenu dans le *shell*, tapez la commande `conda`, vous devriez obtenir :

```
usage: conda-script.py [-h] [-v] [--no-plugins] [-V] COMMAND ...

conda is a tool for managing and deploying applications, environments and packages.

options:
  -h, --help            Show this help message and exit.
  -v, --verbose         Can be used multiple times. Once for detailed output, twice for INFO logging, thrice
     for DEBUG
                        logging, four times for TRACE logging.
  --no-plugins          Disable all plugins that are not built into conda.
  -V, --version         Show the conda version number and exit.
[...]
```

Si c'est le cas, bravo, *conda* est bien installé et vous pouvez passez à la suite (rendez-vous à la rubrique Installation des modules supplémentaires) !

B.2.3.4 Désinstallation de Miniconda

Si vous souhaitez désinstaller Miniconda, rien de plus simple. Dans le menu Windows, tapez *Anaconda* puis *Désinstaller*. Cela vous emmènera dans le panneau de configuration. Faites alors un clic droit sur `Miniconda3 py312...`, puis cliquez sur *Désinstaller*. Cela devrait ouvrir la fenêtre suivante :

Cliquez sur *Next*. Vous aurez alors l'écran suivant :

Cliquez sur *Uninstall*, puis à l'écran suivant confirmez que vous souhaitez désintaller Miniconda :

Le désinstallateur se lancera alors (cela peut prendre quelques minutes) :

Une fois la désinstallation terminée, cliquez sur *Next* :

Puis enfin sur *Finish* :

À ce point, Miniconda est bien désinstallé. Il reste toutefois une dernière manipulation que l'installateur n'a pas effectué : il faut détruire à la main le fichier

`C:\Users\nom_utilisateur\Documents\WindowsPowerShell\profile.ps1`

(bien sûr, remplacez `nom_utilisateur` par votre propre nom d'utilisateur). Si vous ne le faites pas, cela affichera un message d'erreur à chaque fois que vous lancerez un Powershell.

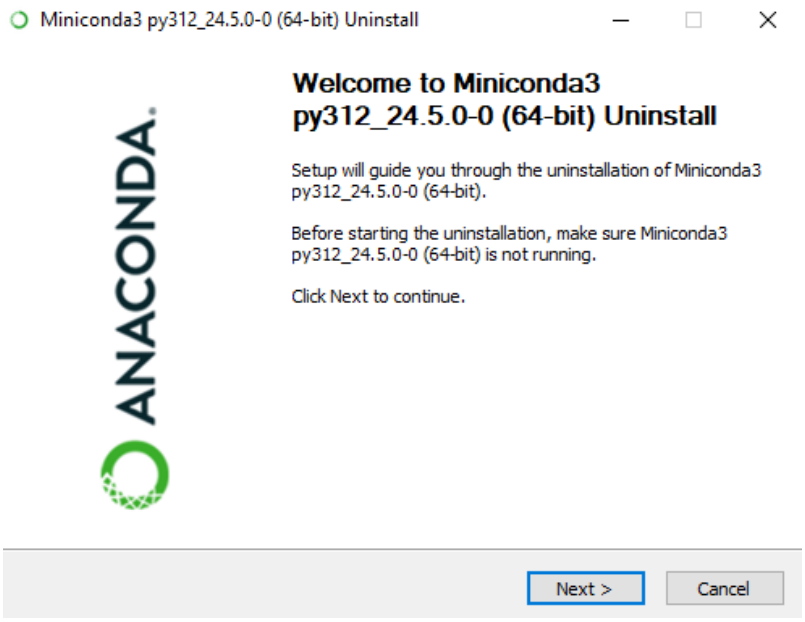


FIGURE B.12 – Désinstallation de Miniconda (étape 1).

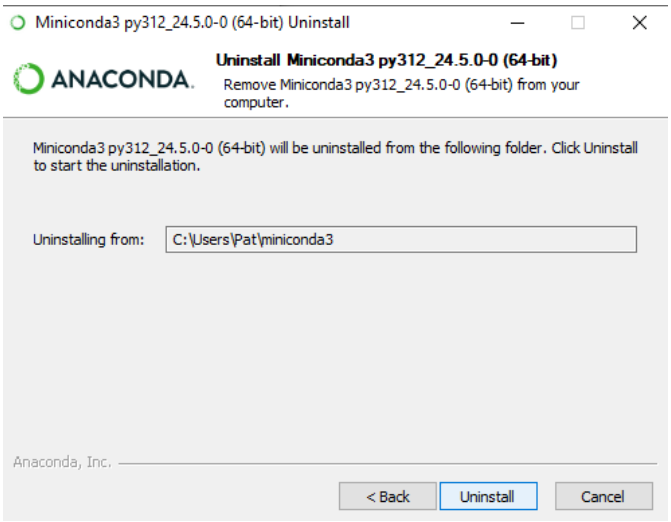


FIGURE B.13 – Désinstallation de Miniconda (étape 2).

B.3 Utilisation de conda pour installer des modules complémentaires

B.3.1 Installation des modules supplémentaires

Cette étape sera commune pour les trois systèmes d'exploitation. À nouveau, lancez un *shell* (c'est-à-dire PowerShell sous Windows ou un terminal pour Mac OS X ou Linux).

Dans le *shell*, tapez la ligne suivante puis appuyez sur la touche *Entrée* :

```
$ conda install numpy pandas matplotlib scipy biopython jupyterlab
```

Cette commande va lancer l'installation des modules externes *NumPy*, *pandas*, *matplotlib*, *scipy*, *Biopython* et *Jupyter lab*. Ces modules vont être téléchargés depuis internet par *conda*, il faut bien sûr que votre connexion internent soit fonctionnelle. Au début, *conda* va déterminer les versions des paquets à télécharger en fonction de la version de Python ainsi que d'autres paramètres (cela prend une à deux minutes). Cela devrait donner la sortie suivante :

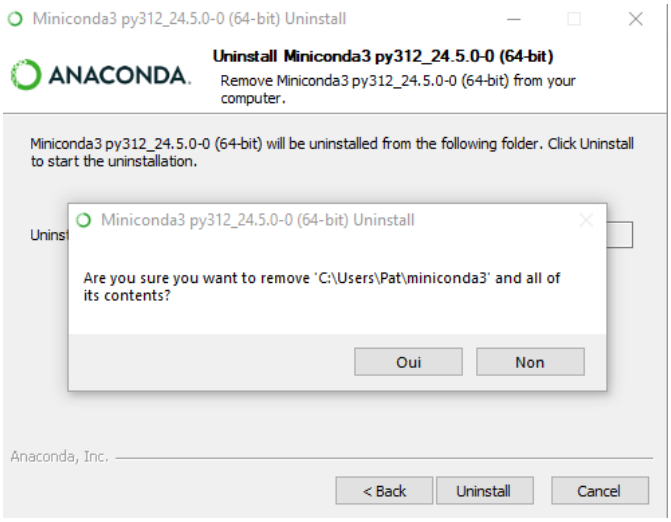


FIGURE B.14 – Désinstallation de Miniconda (étape 3).

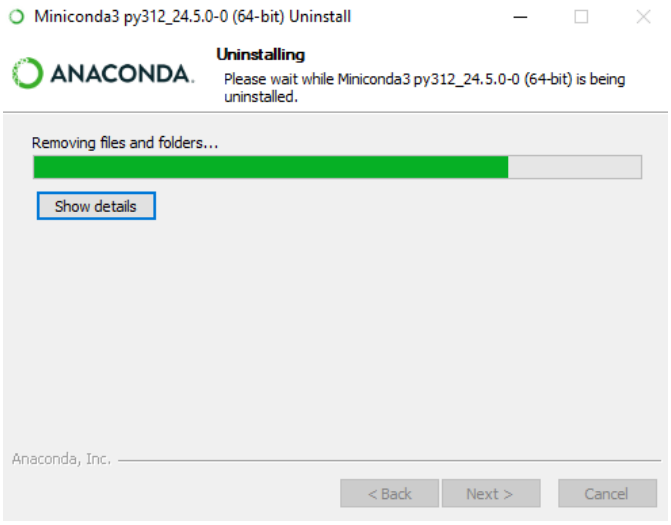


FIGURE B.15 – Désinstallation de Miniconda (étape 4).

```
Channels:
 - defaults
Platform: linux-64
Collecting package metadata (repodata.json): done
Solving environment: done

## Package Plan ##

  environment location: /home/fuchs/miniconda3

  added / updated specs:
    - biopython
    - jupyterlab
    - matplotlib
    - numpy
    - pandas
    - scipy


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    anyio-4.2.0                |  py312h06a4308_0         238 KB
    argon2-cffi-21.3.0         |     pyhd3eb1b0_0          15 KB
    argon2-cffi-bindings-21.2.0|  py312h5eee18b_2          33 KB
    asttokens-2.0.5            |     pyhd3eb1b0_0          20 KB
[...]
```

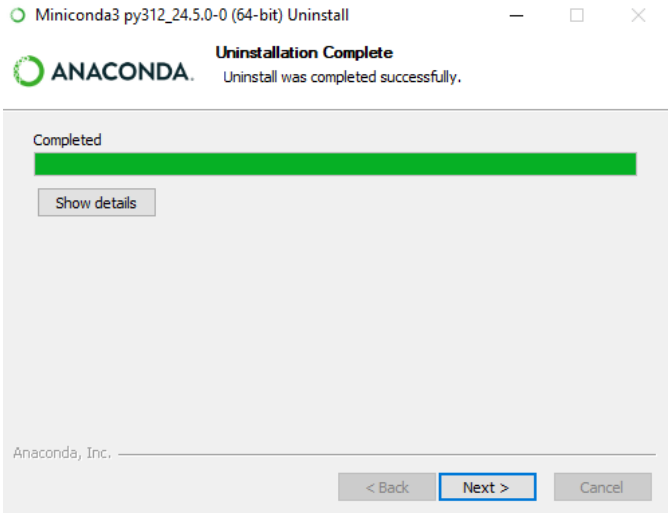


FIGURE B.16 – Désinstallation de Miniconda (étape 5).

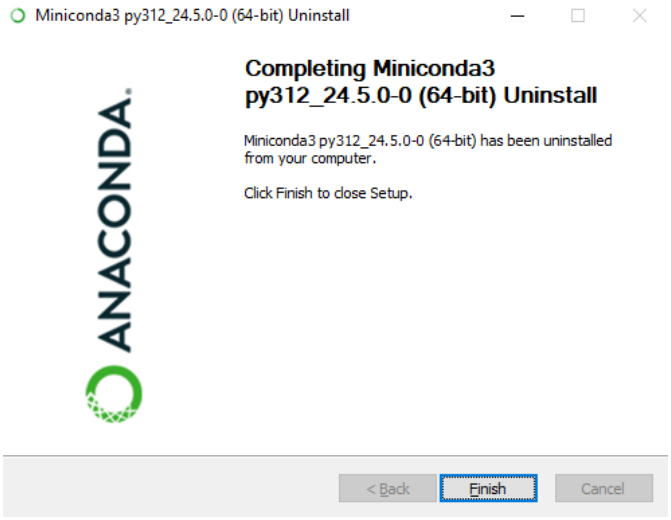


FIGURE B.17 – Désinstallation de Miniconda (étape 6).

Une fois que les versions des paquets ont été déterminées, *conda* vous demande confirmation avant de démarrer le téléchargement. Tapez `y` puis appuyez sur la touche *Entrée* pour confirmer. S'en suit alors le téléchargement et l'installation de tous les paquets (cela prendra quelques minutes) :

Une fois que tout cela est terminé, vous récupérez la main dans le *shell* :

```
[...]
Downloading and Extracting Packages:
mkl-2023.1.0         | 171.5 MB  | #
    ##################################################################################################################################
                     |  92%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
$
```

B.3.2 Test des modules supplémentaires

Pour tester la bonne installation des modules, lancez l'interpréteur Python :

```
$ python
```

Puis tapez les lignes suivantes :

```
import numpy
import scipy
import Bio
import matplotlib
import pandas
```

Vous devriez obtenir la sortie suivante :

```
>>> import numpy
>>> import scipy
>>> import Bio
>>> import matplotlib
>>> import pandas
>>>
```

Si aucune erreur ne s'affiche et que vous récupérez la main dans l'interpréteur, bravo, ces modules sont bien installés. Quittez l'interpréteur Python en tapant la commande `exit()` puis en appuyant sur la touche *Entrée*.

Vous êtes de nouveau dans le *shell*. Nous allons maintenant pouvoir tester Jupyter. Tapez dans le *shell* :

```
$ jupyter lab
```

Cette commande devrait ouvrir votre navigateur internet par défaut et lancer Jupyter :

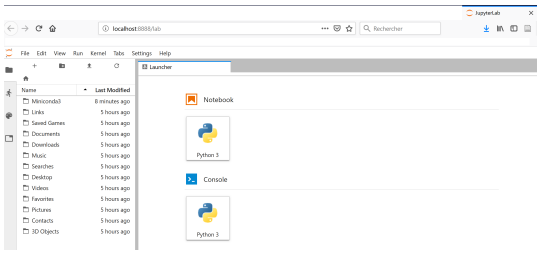


FIGURE B.18 – Test de Jupyter : ouverture dans un navigateur.

Pour quitter Jupyter, allez dans le menu *File* puis sélectionnez *Quit*. Vous pourrez alors fermer l'onglet de Jupyter. Pendant ces manipulations dans le navigateur, de nombreuses lignes ont été affichées dans l'interpréteur :

```
(base) PS C:\Users\Pat> jupyter lab
[I 18:26:05.544 LabApp] JupyterLab extension loaded from C:\Users\Pat\Miniconda3\lib\site-packages\
    jupyterlab
[I 18:26:05.544 LabApp] JupyterLab application directory is C:\Users\Pat\Miniconda3\share\jupyter\lab
[...]
[I 18:27:20.645 LabApp] Interrupted...
[I 18:27:32.986 LabApp] Shutting down 0 kernels
(base) PS C:\Users\Pat>
```

Il s'agit d'un comportement normal. Quand Jupyter est actif, vous n'avez plus la main dans l'interpréteur et tous ces messages s'affichent. Une fois que vous quittez Jupyter, vous devriez récupérer la main dans l'interpréteur. Si ce n'est pas le cas, pressez deux fois la combinaison de touches *Ctrl* + *C*

Si tous ces tests ont bien fonctionné, bravo, vous avez installé correctement Python avec Miniconda ainsi que tous les modules qui seront utilisés pour ce cours. Vous pouvez quitter le *shell* en tapant `exit` puis en appuyant sur la touche *Entrée* et aller faire une pause !

B.3.3 Un mot sur pip pour installer des modules complémentaires

Conseil

Si vous êtes débutant, vous pouvez sauter cette rubrique.

Comme indiqué au début de ce chapitre, pip[8] est un gestionnaire de paquets pour Python et permet d'installer des modules externes. *Pip* est également présent dans Miniconda, donc utilisable et parfaitement fonctionnel. Vous pouvez vous poser la question « Pourquoi utiliser le gestionnaire de paquets *pip* si le gestionnaire de paquets `conda` est déjà présent ? ». La réponse est simple, certains modules ne sont présents que sur les dépôts *pip*. Si vous souhaitez les installer il faudra impérativement utiliser *pip*. Inversement, certains modules ne sont présents que dans les dépôts de *conda*. Toutefois, pour les modules classiques (comme *NumPy*, *scipy*, etc), tout est gérable avec *conda*.

Sauf cas exceptionnel, nous vous conseillons l'utilisation de *conda* pour gérer l'installation de modules supplémentaires.

Si vous souhaitez installer un paquet qui n'est pas présent sur un dépôt *conda* avec *pip*, assurez vous d'abord que votre environnement *conda* est bien activé (avec `conda activate` ou `conda activate nom_environnement`). La syntaxe est ensuite très simple :

```
$ pip install nom_du_paquet
```

Si votre environnement *conda* était bien activé lors de l'appel de cette commande, celle-ci aura installé votre paquet dans l'environnement *conda*. Tout est donc bien encapsulé dans l'environnement *conda*, et l'ajout de tous ces paquets ne risque pas d'interférer avec le Python du système d'exploitation, rendant ainsi les choses bien « propres ».

B.4 Choisir un bon éditeur de texte

La programmation nécessite d'écrire des lignes de code en utilisant un éditeur de texte. Le choix de cet éditeur est donc fondamental, celui-ci doit nous aider à repérer rapidement certaines zones du programme afin d'être efficace. Outre les fonctions de manipulation / remplacement / recherche de texte, un bon éditeur doit absolument posséder la **coloration syntaxique** (*syntax highlighting* en anglais). Celle-ci change la couleur et / ou la police de certaines zones du code comme les mot-clés du langage, les zones entre guillemets, les commentaires, etc. Dans ce qui suit, nous vous montrons des éditeurs faciles à prendre en main par les débutants pour Linux, Windows et Mac OS X.

B.4.1 Installation et réglage de gedit sous Linux

Pour Linux, on vous recommande l'utilisation de l'éditeur de texte *gedit* qui a les avantages d'être simple à utiliser et présent dans la plupart des distributions Linux.

Si *gedit* n'est pas installé, vous pouvez l'installer avec la commande :

```
$ sudo apt install -y gedit
```

Il faudra entrer votre mot de passe utilisateur puis valider en appuyant sur la touche *Entrée*.

Pour lancer cet éditeur, tapez la commande `gedit` dans un *shell* ou cherchez *gedit* dans le lanceur d'applications. Vous devriez obtenir une fenêtre similaire à celle-ci :

On configure ensuite *gedit* pour que l'appui sur la touche *Tab* corresponde à une indentation de 4 espaces, comme recommandée par la PEP 8 (chapitre 15 *Bonnes pratiques en programmation Python*). Pour cela, cliquez sur l'icône en forme de 3 petites barres horizontales en haut à droite de la fenêtre de *gedit*, puis sélectionnez *Préférences*. Dans la nouvelle fenêtre qui s'ouvre, sélectionnez l'onglet *Éditeur* puis fixez la largeur des tabulations à 4 et cochez la case *Insérer des espaces au lieu des tabulations* :

Si vous le souhaitez, vous pouvez également cochez la case *Activer l'indentation automatique* qui indentera automatiquement votre code quand vous êtes dans un bloc d'instructions. Fermez la fenêtre de paramètres une fois la configuration terminée.

B.4.2 Installation et réglage de Notepad++ sous Windows

Sous Windows, nous vous recommandons l'excellent éditeur Notepad++[9]. Une fois cet éditeur installé, il est important de le régler correctement. En suivant le menu `Paramètres`, `Préférences`, vous arriverez sur un panneau vous permettant de configurer Notepad++.

En premier on va configurer l'appui sur la touche *Tab* afin qu'il corresponde à une indentation de 4 espaces, comme recommandé par la PEP 8 (chapitre 15 *Bonnes pratiques en programmation Python*). Dans la liste sur la gauche, cliquez

8. https://pip.pypa.io/en/stable/
9. https://notepad-plus-plus.org/download

FIGURE B.19 – Éditeur de texte *gedit*.

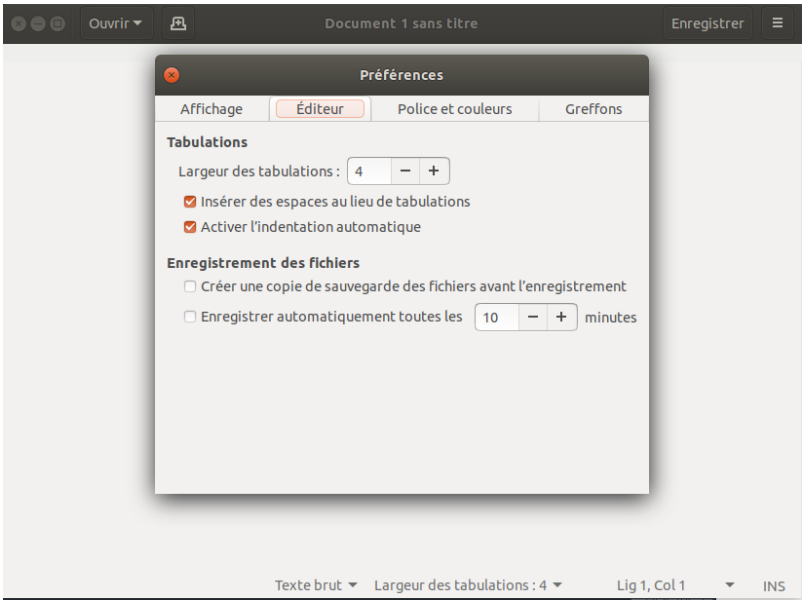


FIGURE B.20 – Configuration de *gedit*.

sur `Langage`, puis à droite dans le carré `Tabulations` cochez la case `Insérer des espaces` en réglant sur 4 espaces comme indiqué ci-dessous :

Ensuite, il est important de faire en sorte que Notepad++ affiche les numéros de ligne sur la gauche (très pratique lorsque l'interpréteur nous indique qu'il y a une erreur, par exemple, à la ligne 47). Toujours dans la fenêtre `Préférences`, dans la liste sur la gauche cliquez sur `Zones d'édition`, puis sur la droite cochez la case `Afficher la numérotation des lignes` comme indiqué ici :

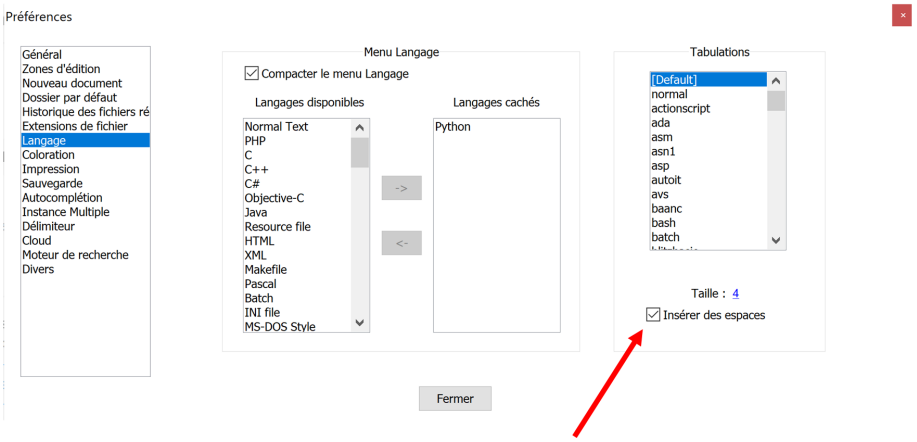


FIGURE B.21 – Configuration de *Notepad++* : indentation avec des espaces.

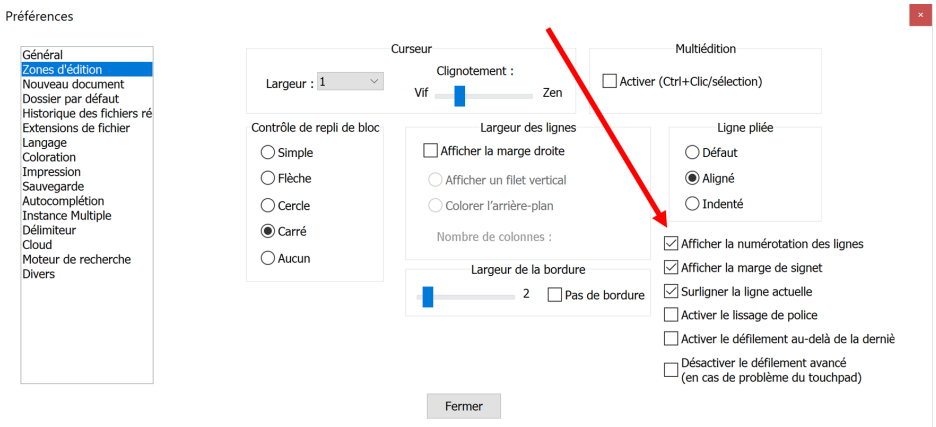


FIGURE B.22 – Configuration de *Notepad++* : numéro de ligne.

B.4.3 Installation et réglage de TextWrangler/BBedit sous Mac OS X

Sur les anciennes versions de Mac OS X (< 10.14), TextWrangler[10] était un éditeur de texte simple, intuitif et efficace. Toutefois son développement a été arrêté car il fonctionnait en 32-bits. Il a été remplacé par BBedit[11] qui possède de nombreuses fonctionnalités supplémentaires mais qui doit en principe être acheté. Toutefois, ce dernier est utilisable gratuitement avec les mêmes fonctionnalités que TextWrangler, sans les nouvelles fonctionnalités étendues. Ne possédant pas de Mac, nous nous contentons ici de vous donner quelques liens utiles :

- La page de téléchargement[12] ;
- La page vers de nombreuses ressources[13] utiles ;
- Le manuel d'utilisation[14] (avec toutes les instructions pour son installation au chapitre 2) ;
- Une page sur Stackoverflow[15] qui vous montre comment faire en sorte que l'appui sur la touche *Tab* affiche 4 espaces plutôt qu'une tabulation.

10. http://www.barebones.com/products/textwrangler/
11. https://www.barebones.com/products/bbedit/
12. http://www.barebones.com/products/bbedit/download.html
13. https://www.barebones.com/support/bbedit/
14. https://s3.amazonaws.com/BBSW-download/BBEdit_12.6.6_User_Manual.pdf
15. https://stackoverflow.com/questions/5750361/auto-convert-tab-to-4-spaces-in-textwrangler

B.4.4 Pour aller plus loin

Jusque là, nous vous avons montré des éditeurs de texte simples qui sont, selon nous, idéaux pour apprendre un langage de programmation. Ainsi, on se concentre sur le langage Python plutôt que toutes les options de l'éditeur. Toutefois, pour les utilisateurs plus avancés, nous vous conseillons des plateformes de développement [16] ou IDE (*integrated development environment*) qui, au-delà de l'édition, permettent par exemple d'exécuter le code et de le *debugger* (c'est-à-dire, y chasser les erreurs). On peut citer par exemple les IDE libres Visual Studio code [17] et Spyder [18].

B.5 Comment se mettre dans le bon répertoire dans le shell

Pour apprendre Python, nous allons devoir écrire des scripts, les enregistrer dans un répertoire, puis les exécuter avec l'interpréteur Python. Il faut pour cela être capable d'ouvrir un *shell* et de se mettre dans le répertoire où se trouve ce script.

Notre livre n'est pas un cours d'Unix, mais il convient au moins de savoir se déplacer dans l'arborescence avant de lancer Python. Sous Linux et sous Mac il est donc fondamental de connaître les commandes Unix `cd`, `pwd`, `ls` et la signification de `..` (point point).

Sous Linux, il existe une astuce très pratique. Si vous utilisez l'explorateur de fichiers Nautilus, quand vous êtes dans un répertoire, faites un clic droit et choisissez dans le menu *Ouvrir dans un terminal*. Vous vous retrouverez automatiquement dans le bon répertoire (vous pouvez vous en assurer avec la commande Unix `pwd`).

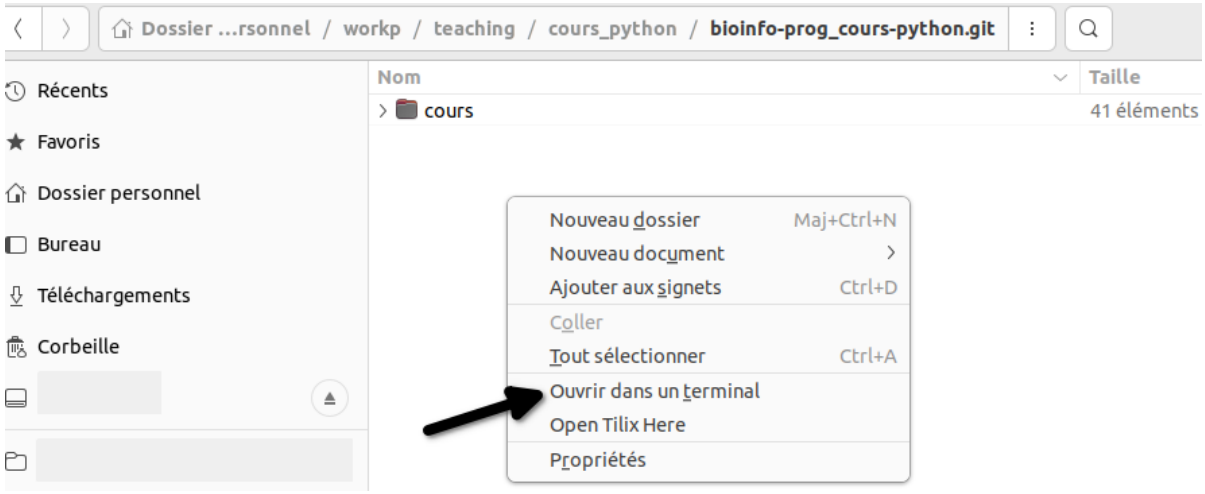


FIGURE B.23 – Lancement d'un terminal depuis un répertoire donné avec Nautilus).

De façon similaire sous Windows, il existe deux astuces très pratiques. Lorsqu'on utilise l'explorateur Windows et que l'on est dans un répertoire donné :

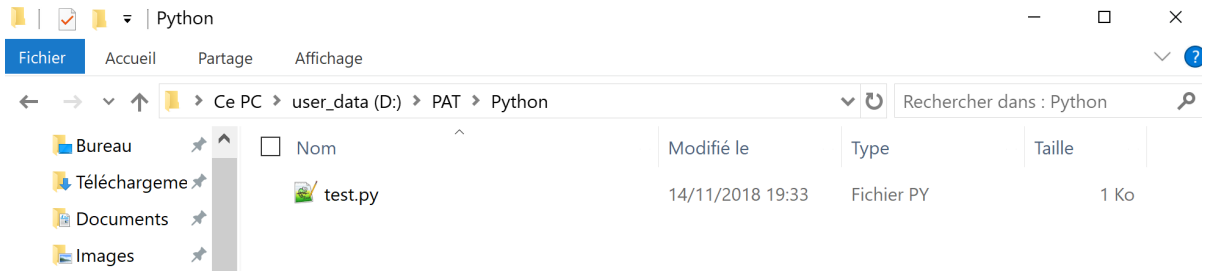


FIGURE B.24 – Lancement d'un *powershell* depuis un répertoire donné (étape 1).

16. https://fr.wikipedia.org/wiki/Environnement_de_d%C3%A9veloppement
17. https://code.visualstudio.com/
18. https://www.spyder-ide.org/

Il est possible d'ouvrir un PowerShell directement dans ce répertoire :

Première astuce

Il suffit de taper `powershell` dans la barre qui indique le chemin :

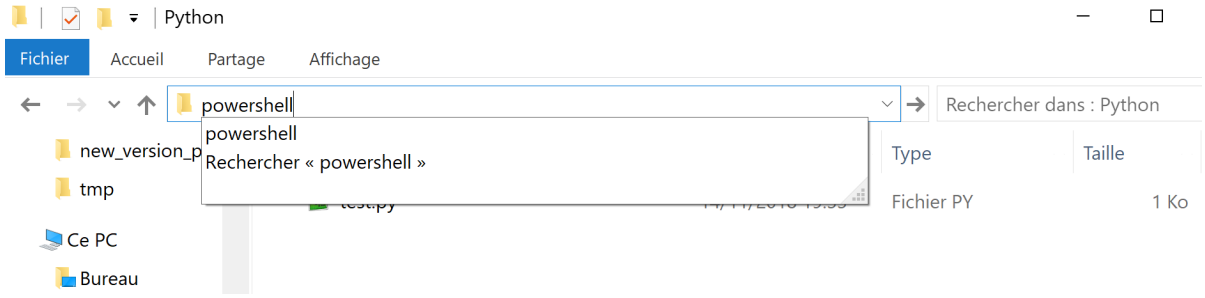


FIGURE B.25 – Lancement d'un *powershell* depuis un répertoire donné (étape 2).

puis on appuie sur entrée et le PowerShell se lance en étant directement dans le bon répertoire !

Deuxième astuce

En pressant la touche Shift et en faisant un clic droit dans un endroit de l'explorateur qui ne contient pas de fichier (attention, ne pas faire de clic droit sur un fichier !). Vous verrez alors s'afficher le menu contextuel suivant :

Cliquez sur *Ouvrir la fenêtre PowerShell ici*, à nouveau votre Powershell sera directement dans le bon répertoire !

Vérification

La figure suivante montre le PowerShell, ouvert de la première ou la deuxième façon, dans lequel nous avons lancé la commande `ls` qui affiche le nom du répertoire courant (celui dans lequel on se trouve, dans notre exemple `D:\PAT\Python`) ainsi que les fichiers s'y trouvant (ici il n'y a qu'un fichier : `test.py`). Ensuite nous avons lancé l'exécution de ce fichier `test.py` en tapant `python test.py`.

À votre tour !

Pour tester si vous avez bien compris, ouvrez votre éditeur favori, tapez les lignes suivantes puis enregistrez ce fichier avec le nom `test.py` dans le répertoire de votre choix.

```
1  import tkinter as tk
2
3  racine = tk.Tk()
4  label = tk.Label(racine, text="J'adore Python !")
5  bouton = tk.Button(racine, text="Quitter", command=racine.quit)
6  bouton["fg"] = "red"
7  label.pack()
8  bouton.pack()
9  racine.mainloop()
10 print("C'est fini !")
```

Comme nous vous l'avons montré ci-dessus, ouvrez un *shell* et déplacez-vous dans le répertoire où se trouve `test.py`. Lancez le script avec l'interpréteur Python :

```
$ python test.py
```

Si vous avez fait les choses correctement, cela devrait afficher une petite fenêtre avec un message « J'adore Python ! » et un bouton *Quitter*.

B.6 Python web et mobile

Si vous ne pouvez ou ne souhaitez pas installer Python sur votre ordinateur (quel dommage !), des solutions alternatives s'offrent à vous.

Des sites internet vous proposent l'équivalent d'un interpréteur Python utilisable depuis votre navigateur web :

- repl.it [19] ;
- Tutorials Point [20] ;

19. https://repl.it/languages/python3
20. https://www.tutorialspoint.com/execute_python3_online.php

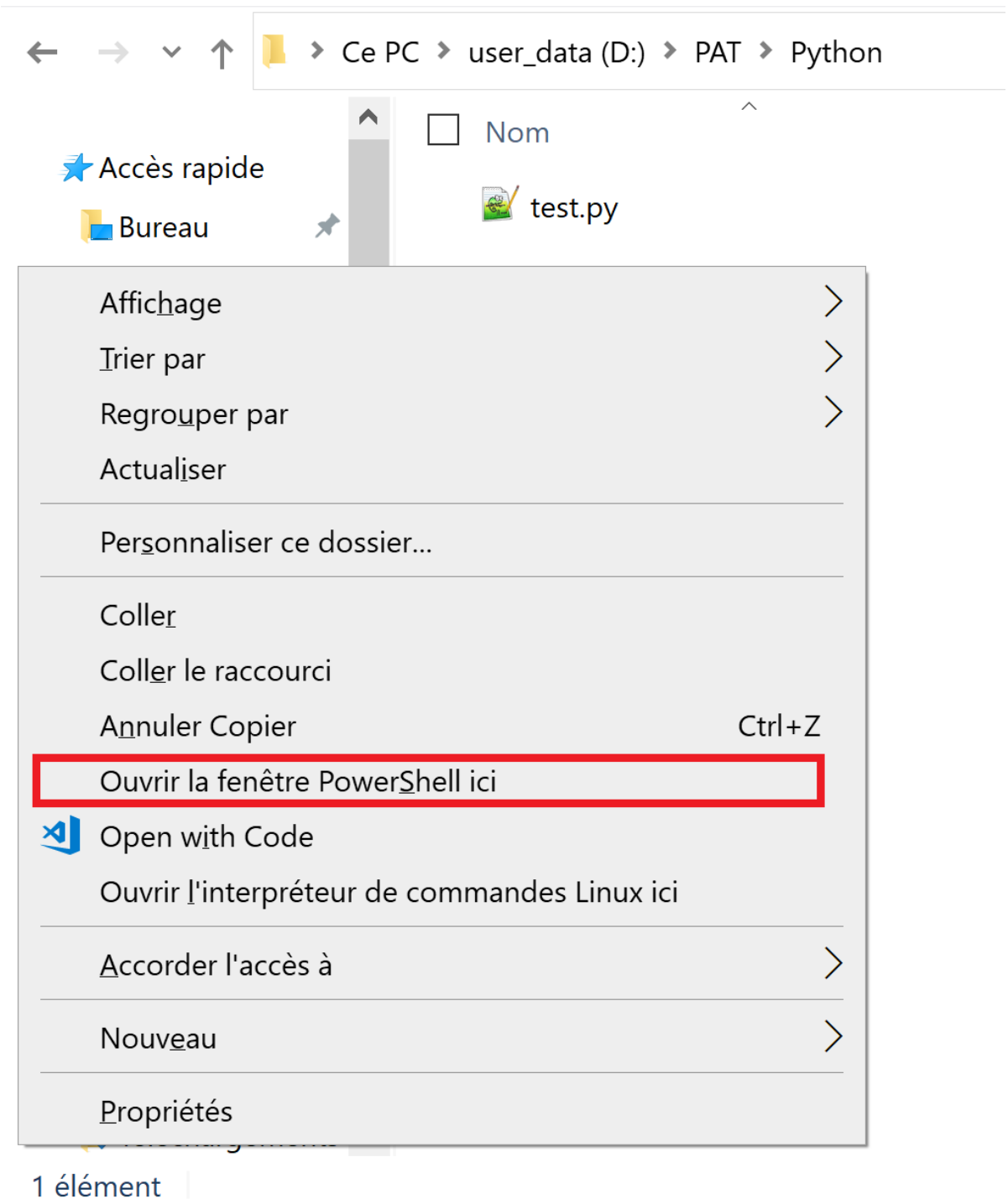


FIGURE B.26 – Lancement d'un *powershell* depuis un répertoire donné (étape 2bis).

- et bien sur l'incontournable Python Tutor[21].

Des applications mobiles vous permettent aussi de « pythonner » avec votre smartphone :

- Pydroid 3[22] pour Android ;

21. http://pythontutor.com/visualize.html#mode=edit
22. https://play.google.com/store/apps/details?id=ru.iiec.pydroid3

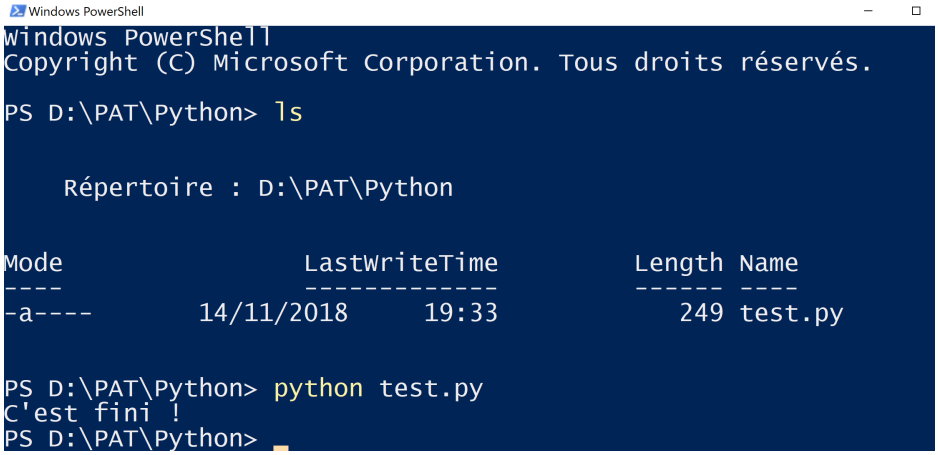


FIGURE B.27 – Lancement d'un *powershell* depuis un répertoire donné (étape 3).

- Pythonista 3[23] pour iOS (payant).

Soyez néanmoins conscient que ces applications web ou mobiles peuvent être limitées, notamment sur leur capacité à installer des modules supplémentaires et à gérer les fichiers.

23. https://itunes.apple.com/us/app/pythonista-3/id1085978097