

Cours de Python



<https://python.sdv.univ-paris-diderot.fr/>

Patrick Fuchs et Pierre Poulain

prénom [point] nom [arobase] univ-paris-diderot [point] fr

version du 9 mai 2019

Université Paris Diderot-Paris 7, Paris, France

Table des matières

Avant-propos	6
Quelques mots sur l'origine de ce cours	6
Remerciements	6
1 Introduction	7
1.1 C'est quoi Python ?	7
1.2 Conseils pour installer et configurer Python	7
1.3 Notations utilisées	8
1.4 Introduction au <i>shell</i>	8
1.5 Premier contact avec Python	9
1.6 Premier programme	10
1.7 Commentaires	10
1.8 Notion de bloc d'instructions et d'indentation	10
1.9 Autres ressources	11
2 Variables	12
2.1 Définition	12
2.2 Les types de variables	12
2.3 Nommage	13
2.4 Opérations	13
2.5 La fonction <code>type()</code>	15
2.6 Conversion de types	15
2.7 Note sur la division de deux nombres entiers	15
2.8 Note sur le vocabulaire et la syntaxe	15
2.9 Exercices	16
3 Affichage	17
3.1 La fonction <code>print()</code>	17
3.2 Écriture formatée	18
3.3 Ancienne méthode de formatage des chaînes de caractères	20
3.4 Note sur le vocabulaire et la syntaxe	20
3.5 Exercices	21
4 Listes	22
4.1 Définition	22
4.2 Utilisation	22
4.3 Opération sur les listes	22
4.4 Indice négatif	23
4.5 Tranches	23
4.6 Fonction <code>len()</code>	24
4.7 Les fonctions <code>range()</code> et <code>list()</code>	24
4.8 Listes de listes	25
4.9 Exercices	25

5 Boucles et comparaisons	27
5.1 Boucles <code>for</code>	27
5.2 Comparaisons	30
5.3 Boucles <code>while</code>	31
5.4 Exercices	31
6 Tests	36
6.1 Définition	36
6.2 Tests à plusieurs cas	36
6.3 Importance de l'indentation	37
6.4 Tests multiples	37
6.5 Instructions <code>break</code> et <code>continue</code>	38
6.6 Tests de valeur sur des <i>floats</i>	39
6.7 Exercices	39
7 Fichiers	44
7.1 Lecture dans un fichier	44
7.2 Écriture dans un fichier	46
7.3 Ouvrir deux fichiers avec l'instruction <code>with</code>	47
7.4 Note sur les retours à la ligne sous Unix et sous Windows	47
7.5 Importance des conversions de types avec les fichiers	48
7.6 Du respect des formats de données et de fichiers	48
7.7 Exercices	48
8 Modules	51
8.1 Définition	51
8.2 Importation de modules	51
8.3 Obtenir de l'aide sur les modules importés	53
8.4 Quelques modules courants	54
8.5 Module <code>sys</code> : passage d'arguments	54
8.6 Module <code>os</code> : interaction avec le système d'exploitation	55
8.7 Exercices	56
9 Fonctions	59
9.1 Principe et généralités	59
9.2 Définition	60
9.3 Passage d'arguments	61
9.4 Renvoi de résultats	61
9.5 Arguments positionnels et arguments par mot-clé	61
9.6 Variables locales et variables globales	63
9.7 Exercices	66
10 Plus sur les chaînes de caractères	70
10.1 Préambule	70
10.2 Chaînes de caractères et listes	70
10.3 Caractères spéciaux	70
10.4 Méthodes associées aux chaînes de caractères	71
10.5 Extraction de valeurs numériques d'une chaîne de caractères	73
10.6 Conversion d'une liste de chaînes de caractères en une chaîne de caractères	73
10.7 Exercices	74

11 Plus sur les listes	78
11.1 Méthodes associées aux listes	78
11.2 Construction d'une liste par itération	79
11.3 Test d'appartenance	80
11.4 Copie de listes	80
11.5 Exercices	82
12 Plus sur les fonctions	84
12.1 Appel d'une fonction dans une fonction	84
12.2 Fonctions récursives	85
12.3 Portée des variables	86
12.4 Portée des listes	87
12.5 Règle LGI	88
12.6 Recommandations	88
12.7 Exercices	89
13 Dictionnaires et tuples	90
13.1 Dictionnaires	90
13.2 Tuples	91
13.3 Exercices	92
14 Création de modules	95
14.1 Pourquoi créer ses propres modules ?	95
14.2 Création d'un module	95
14.3 Utilisation de son propre module	96
14.4 Les <i>docstrings</i>	96
14.5 Visibilité des fonctions dans un module	97
14.6 Module ou script ?	97
14.7 Exercice	98
15 Bonnes pratiques en programmation Python	99
15.1 De la bonne syntaxe avec la PEP 8	99
15.2 Les <i>docstrings</i> et la PEP 257	103
15.3 Outils de contrôle qualité du code	105
15.4 Organisation du code	107
15.5 Conseils sur la conception d'un script	108
15.6 Pour terminer : la PEP 20	108
16 Expressions régulières et parsing	110
16.1 Définition et syntaxe	110
16.2 Quelques ressources en ligne	112
16.3 Le module <i>re</i>	112
16.4 Exercices	114
17 Quelques modules d'intérêt en bioinformatique	117
17.1 Module <i>NumPy</i>	117
17.2 Module <i>Biopython</i>	122
17.3 Module <i>matplotlib</i>	124
17.4 Module <i>pandas</i>	128
17.5 Un exemple plus complet	132
17.6 Exercices	137

18 Jupyter et ses notebooks	140
18.1 Installation	140
18.2 Lancement de Jupyter et création d'un notebook	140
18.3 Le format Markdown	143
18.4 Des graphiques dans les notebooks	145
18.5 Les <i>magic commands</i>	148
18.6 JupyterLab	148
19 Avoir la classe avec les objets	152
19.1 Construction d'une classe	152
19.2 Espace de noms	163
19.3 Polymorphisme	166
19.4 Héritage	168
19.5 Accès et modifications des attributs depuis l'extérieur	173
19.6 Bonnes pratiques pour construire et manipuler ses classes	177
19.7 Exercices	181
20 Fenêtres graphiques et Tkinter	182
20.1 Utilité d'une GUI	182
20.2 Quelques concepts liés à la programmation graphique	183
20.3 Notion de fonction <i>callback</i>	183
20.4 Prise en main du module <i>Tkinter</i>	184
20.5 Construire une application <i>Tkinter</i> avec une classe	186
20.6 Le <i>widget canvas</i>	187
20.7 Pour aller plus loin	193
20.8 Exercices	197
21 Remarques complémentaires	200
21.1 Différences Python 2 et Python 3	200
21.2 Liste de compréhension	202
21.3 Gestion des erreurs	202
21.4 Shebang et /usr/bin/env python3	204
21.5 Passage d'arguments avec <i>*args</i> et <i>**kwargs</i>	205
21.6 Un peu de transformée de Fourier avec <i>NumPy</i>	206
21.7 Sauvegardez votre historique de commandes	207
22 Mini-projets	208
22.1 Description des projets	208
22.2 Accompagnement pas à pas	209
A Quelques formats de données rencontrés en biologie	223
A.1 FASTA	223
A.2 GenBank	225
A.3 PDB	226
A.4 Format XML, CSV et TSV	232
B Installation de Python	237
B.1 Que recommande-t-on pour l'installation de Python ?	237
B.2 Installation de Python avec Miniconda	237
B.3 Utilisation de conda pour installer des modules complémentaires	246
B.4 Choisir un bon éditeur de texte	252
B.5 Comment se mettre dans le bon répertoire dans le shell	255
B.6 Python web et mobile	256

Avant-propos

Quelques mots sur l'origine de ce cours

Ce cours 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 7¹; et plus spécialement pour les étudiants du master Biologie Informatique.

Ce cours est basé sur la version 3 de Python, version recommandée par la communauté scientifique. Des références à l'ancienne version, Python 2, seront néanmoins régulièrement apportées.

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

Le cours est disponible en version HTML² et PDF³.

Remerciements

Un grand merci à Sander⁴ du *Centre for Molecular and Biomolecular Informatic* de Nijmegen aux Pays-Bas pour la toute première version⁵ de ce cours qui remonte à l'année 2003.

Nous remercions le professeur Philip Guo⁶ de l'UC San Diego, pour nous avoir autorisé à utiliser des copies d'écran de son excellent site *Python Tutor*⁷.

Merci également à tous les contributeurs, occasionnels ou réguliers : Jennifer Becq, Virginie Martiny, Romain Laurent, Benoist Laurent, Benjamin Boyer, Hubert Santuz, Catherine Lesourd, Philippe Label, Rémi Cuchillo, Cédric Gageat, Philibert Malbranche, Mikaël Naveau, Amélie Bacle, Alexandra Moine-Fanel.

Nous remercions aussi Denis Mestivier de qui nous nous sommes inspirés pour certains exercices.

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.

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.

-
1. <http://www.univ-paris-diderot.fr/>
 2. <https://python.sdv.univ-paris-diderot.fr/index.html>
 3. <https://python.sdv.univ-paris-diderot.fr/cours-python.pdf>
 4. <http://sander.nabuurs.org/>
 5. <http://www.cmbi.ru.nl/pythoncourse/spy/index.spy?site=python&action=Home>
 6. <http://pgbovine.net/>
 7. <http://pythontutor.com/>

Chapitre 1

Introduction

1.1 C'est quoi 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 3.7 a été publiée en juin 2018. La version 2 de Python est désormais obsolète et cessera d'être maintenue après le 1er janvier 2020. Dans la mesure du possible 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 cellule, une protéine, un atome, etc.) avec un certain nombre de règles de fonctionnement et d'interactions.
- Il est relativement *simple* à prendre en main².
- Enfin, il est très utilisé en bioinformatique et plus généralement en analyse de données.

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

1.2 Conseils pour installer et configurer Python

Pour apprendre la programmation Python, il va falloir que vous pratiquiez et pour cela 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 dans cette rubrique un résumé des points importants concernant cette installation. Tous les détails et la marche à suivre pas à pas sont donnés à l'adresse <https://python.sdv.univ-paris-diderot.fr/livre-dunod>.

1.2.1 Python 2 ou Python 3 ?

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

1. <https://www.python.org/psf/>

2. Nous sommes d'accord, cette notion est très relative.

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 21 *Remarques complémentaires* vous apportera plus de précisions.

1.2.2 Miniconda

Nous vous conseillons d'installer Miniconda³, 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 les *notebooks* Jupyter. Vous trouverez en ligne⁴ une documentation pas à pas pour installer Miniconda, Python 3 et les modules supplémentaires qui seront utilisés dans ce cours.

1.2.3 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 ligne⁵. Bien sur, si vous préférez d'autres éditeurs comme *Atom*, *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.

1.3 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.

1.4 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* Unix⁶ fonctionnant sous Mac OS X et Linux ;
- du *shell* PowerShell⁷ 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 systématiquement par le symbole dollar \$, 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 $.
```

3. <https://conda.io/miniconda.html>

4. <https://python.sdv.univ-paris-diderot.fr/livre-dunod>

5. <https://python.sdv.univ-paris-diderot.fr/livre-dunod>

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

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

1.5 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 :

```
1 PS C:\Users\pierre> python
2 Python 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)] [...]
3 Type "help", "copyright", "credits" or "license" for more information.
4 >>>
```

pour Mac OS X :

```
1 iMac-de-pierre:Downloads$ python
2 Python 3.7.1 (default, Dec 14 2018, 19:28:38)
3 [Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
4 Type "help", "copyright", "credits" or "license" for more information.
5 >>>
```

ou pour Linux :

```
1 pierre@jeera:~$ python
2 Python 3.7.1 (default, Dec 14 2018, 19:28:38)
3 [GCC 7.3.0] :: Anaconda, Inc. on linux
4 Type "help", "copyright", "credits" or "license" for more information.
5 >>>
```

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*. 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 de l'interpréteur Python (*prompt* en anglais). 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 votre prochaine instruction en affichant l'invite de l'interpréteur Python (>>>). En résumé, voici ce qui a dû apparaître sur votre écran :

```
1 >>> print("Hello world!")
2 Hello world!
3 >>>
```

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

```
1 >>> 1+1
2 2
3 >>> 6*3
4 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.

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 que Perl⁸ ou R⁹. 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 !

8. <http://www.perl.org>

9. <http://www.r-project.org>

1.6 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*¹⁰) 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`.

Pour exécuter votre script, ouvrez un *shell* et entrez la commande :

```
python test.py
```

Vous devriez obtenir un résultat similaire à ceci :

```
1 | $ python test.py
2 | Hello world!
```

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

1.7 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 15 *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 « croisillon¹¹ ».

1.8 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.

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).

10. <https://python.sdv.univ-paris-diderot.fr/livre-dunod>

11. [https://fr.wikipedia.org/wiki/Croisillon_\(signe\)](https://fr.wikipedia.org/wiki/Croisillon_(signe))

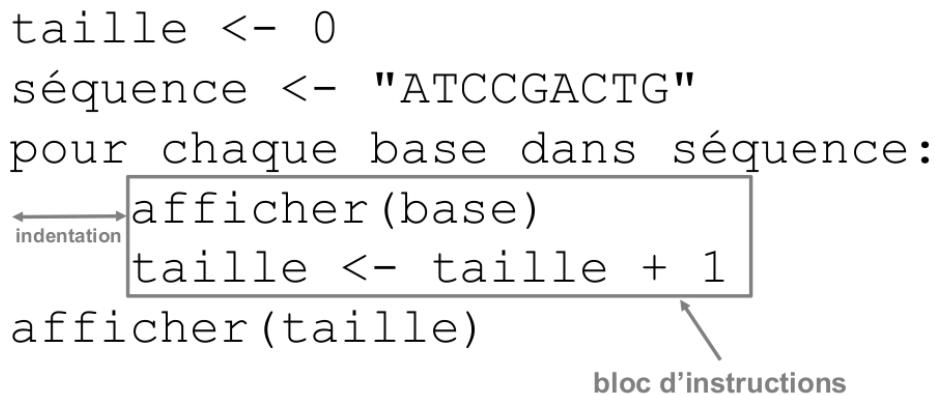


FIGURE 1.1 – Notion d’indentation et de bloc d’instructions.

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 15 *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.

1.9 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 Swinnen¹². 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 Pointal¹³.
- Le livre *Apprenez à programmer en Python* de Vincent Legoff¹⁴ que vous trouverez sur le site *OpenClassrooms*.

Et pour terminer, une ressource incontournable en langue anglaise :

- Le site www.python.org¹⁵. Il contient énormément d’informations et de liens sur Python. La page d’index des modules¹⁶ est particulièrement utile (et traduite en français).

12. <http://www.inforef.be/swi/python.htm>

13. <https://perso.limsi.fr/pointal/python:courspython3>

14. <https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python>

15. <http://www.python.org>

16. <https://docs.python.org/fr/3/py-modindex.html>

Chapitre 2

Variables

2.1 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 3 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. Par contre, 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 (pour autant, cette instruction reste valide et ne générera pas d'erreur).

Sachez par ailleurs que l'opérateur d'affectation `=` s'utilise dans un certain sens. Par exemple, l'instruction `x = 2` signifie qu'on attribue la valeur située à droite de l'opérateur `=` (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`.

Enfin, dans l'instruction `x = y - 3`, l'opération `y - 3` est d'abord évaluée et ensuite le résultat de cette opération est affecté à la variable `x`.

2.2 Les types de variables

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 ici¹.

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 variable que nous verrons par la suite :

1. <https://docs.python.org/fr/3.7/library/stdtypes.html>

```

1 | >>> y = 3.14
2 | >>> y
3 | 3.14
4 | >>> a = "bonjour"
5 | >>> a
6 | 'bonjour'
7 | >>> b = 'salut'
8 | >>> b
9 | 'salut'
10 | >>> c = """girafe"""
11 | >>> c
12 | 'girafe'
13 | >>> d = '''lion'''
14 | >>> d
15 | 'lion'

```

Remarque

Python reconnaît certains types de variable 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.

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.).

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

2.4 Opérations

2.4.1 Opérations sur les types numériques

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

```

1 | >>> x = 45
2 | >>> x + 2
3 | 47
4 | >>> x - 2
5 | 43
6 | >>> x * 3
7 | 135
8 | >>> y = 2.5
9 | >>> x - y
10 | 42.5
11 | >>> (x * 10) + y
12 | 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* :

```

1 | >>> 3 / 4
2 | 0.75
3 | >>> 2.5 / 2
4 | 1.25

```

L'opérateur puissance utilise les symboles ** :

```

1 | >>> 2**3
2 | 8
3 | >>> 2**4
4 | 16

```

Pour obtenir le quotient et le reste d'une division entière (voir ici² pour un petit rappel sur la division entière), on utilise respectivement les symboles `//` et modulo `%` :

```

1 | >>> 5 // 4
2 | 1
3 | >>> 5 % 4
4 | 1
5 | >>> 8 // 4
6 | 2
7 | >>> 8 % 4
8 | 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 :

```

1 | >>> i = 0
2 | >>> i = i + 1
3 | >>> i
4 | 1
5 | >>> i += 1
6 | >>> i
7 | 2
8 | >>> i += 2
9 | >>> i
10 | 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.

2.4.2 Opérations sur les chaînes de caractères

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

```

1 | >>> chaine = "Salut"
2 | >>> chaine
3 | 'Salut'
4 | >>> chaine + " Python"
5 | 'Salut Python'
6 | >>> chaine * 3
7 | 'SalutSalutSalut'

```

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

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

Attention

Vous observez que les opérateurs `+` et `*` se comportent différemment selon qu'il s'agisse d'entiers ou de chaînes de caractères : `2 + 2` est une addition alors que `"2" + "2"` est une concaténation. On appelle ce comportement **redéfinition des opérateurs**. Nous serons amenés à revoir cette notion dans le chapitre 19 *Avoir la classe avec les objets*.

2.4.3 Opérations illicites

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

```

1 | >>> "toto" * 1.3
2 | Traceback (most recent call last):
3 |   File "<stdin>", line 1, in <module>
4 | TypeError: can't multiply sequence by non-int of type 'float'
5 | >>> "toto" + 2
6 | Traceback (most recent call last):
7 |   File "<stdin>", line 1, in <module>
8 | 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 https://fr.wikipedia.org/wiki/Division_euclidienne

2.5 La fonction `type()`

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

```

1 | >>> x = 2
2 | >>> type(x)
3 | <class 'int'>
4 | >>> y = 2.0
5 | >>> type(y)
6 | <class 'float'>
7 | >>> z = '2'
8 | >>> type(z)
9 | <class 'str'>
```

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).

2.6 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 :

```

1 | >>> i = 3
2 | >>> str(i)
3 | '3'
4 | >>> i = '456'
5 | >>> int(i)
6 | 456
7 | >>> float(i)
8 | 456.0
9 | >>> i = '3.1416'
10 | >>> float(i)
11 | 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.

2.7 Note sur la division de deux nombres entiers

Notez bien qu'en Python 3, la division de deux nombres entiers renvoie par défaut un *float* :

```

1 | >>> x = 3 / 4
2 | >>> x
3 | 0.75
4 | >>> type(x)
5 | <class 'float'>
```

Remarque

Ceci n'était pas le cas en Python 2. Pour en savoir plus sur ce point, vous pouvez consulter le chapitre 21 *Remarques complémentaires*.

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 19 *Avoir la classe avec les objets* 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 argument, 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 Exercices

Conseil : utilisez l'interpréteur Python pour les exercices suivants.

2.9.1 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
```

2.9.2 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, 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 9 *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 :

```
1 | >>> x = 32
2 | >>> nom = "John"
3 | >>> print(nom, "a" , x , "ans")
4 | 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` :

```
1 | >>> x = 32
2 | >>> nom = "John"
3 | >>> print(nom, "a" , x , "ans", sep="")
```

```

4| Johna32ans
5| >>> print(nom , "a" , x , "ans" , sep="-")
6| 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 :

```

1| >>> ani1 = "chat"
2| >>> ani2 = "souris"
3| >>> print(ani1, ani2)
4| chat souris
5| >>> print(ani1 + ani2)
6| chatsouris
7| >>> print(ani1, ani2, sep="")
8| chatsouris

```

3.2 Écriture formatée

La méthode `.format()` permet une meilleure organisation de l'affichage des variables (nous expliquerons à la fin de ce chapitre ce que le terme « méthode » signifie en Python).

Si on reprend l'exemple précédent :

```

1| >>> x = 32
2| >>> nom = "John"
3| >>> print("{} a {} ans".format(nom, x))
4| John a 32 ans

```

- 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`. La méthode `.format()` agit sur la chaîne de caractères à laquelle elle est attachée par le point.

Remarque

Il est possible d'indiquer entre les accolades `{}` dans quel ordre afficher les variables, avec 0 pour la variable à afficher en premier, 1 pour la variable à afficher en second, etc. (attention, Python commence à compter à 0). Cela permet de modifier l'ordre dans lequel sont affichées les variables.

```

1| >>> x = 32
2| >>> nom = "John"
3| >>> print("{} a {} ans".format(nom, x))
4| John a 32 ans
5| >>> print("{} a {} ans".format(x, nom))
6| 32 a John ans

```

Imaginez maintenant que vous voulez 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, 4500 bases G et 2575 bases C, pour un total de 14800 bases, vous pourriez procéder comme suit (notez bien l'utilisation des parenthèses pour gérer les priorités des opérateurs) :

```

1| >>> prop_GC = (4500 + 2575) / 14800
2| >>> print("La proportion de GC est", prop_GC)
3| La proportion de GC est 0.4780405405405

```

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 :

```

1| >>> print("La proportion de GC est {:.2f}".format(prop_GC))
2| Le proportion de GC est 0.48
3| >>> print("La proportion de GC est {:.3f}".format(prop_GC))
4| La proportion de GC est 0.478

```

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

- Les deux points : indiquent qu'on veut préciser le 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.

Il est par ailleurs possible de combiner le formatage (à droite des 2 points) ainsi que l'emplacement des variables à substituer (à gauche des 2 points), par exemple :

```
1 | >>> print("prop GC(2 déci.) = {:.2f}, prop GC(3 déci.) = {:.3f}".format(prop_GC))
2 | prop GC(2 déci.) = 0.48, prop GC(3 déci.) = 0.478
```

Vous remarquerez qu'on utilise ici la même variable (prop_GC) à deux endroits différents.

Vous pouvez aussi formater des entiers avec la lettre d :

```
1 | >>> nb_G = 4500
2 | >>> print("Ce génome contient {:d} guanines".format(nb_G))
3 | Ce génome contient 4500 guanines
```

ou mettre plusieurs nombres dans une même chaîne de caractères.

```
1 | >>> nb_G = 4500
2 | >>> nb_C = 2575
3 | >>> print("Ce génome contient {:d} G et {:d} C, soit une prop de GC de {:.2f}" \
4 | ... .format(nb_G,nb_C,prop_GC))
5 | Ce génome contient 4500 G et 2575 C, soit une prop de GC de 0.48
6 | >>> perc_GC = prop_GC * 100
7 | >>> print "Ce génome contient {:d} G et {:d} C, soit un %GC de {:.2f} %" \
8 | ... .format(nb_G,nb_C,perc_GC)
9 | Ce génome contient 4500 G et 2575 C, soit un %GC de 47.80 %
```

Remarque

Le signe \ en fin de ligne permet de poursuivre la commande sur la ligne suivante. Cette syntaxe est pratique lorsque vous voulez taper une commande longue.

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 centré). Dans la portion de code suivante, le caractère ; sert de séparateur entre les instructions sur une même ligne :

```
1 | >>> print(10) ; print(1000)
2 | 10
3 | 1000
4 | >>> print("{:>6d}".format(10)) ; print("{:>6d}".format(1000))
5 |    10
6 |   1000
7 | >>> print("{:<6d}".format(10)) ; print("{:<6d}".format(1000))
8 |    10
9 |   1000
10 | >>> print("{:^6d}".format(10)) ; print("{:^6d}".format(1000))
11 |    10
12 |   1000
13 | >>> print("{:.*^6d}".format(10)) ; print("{:.*^6d}".format(1000))
14 | **10**
15 | *1000*
16 | >>> print("{:0>6d}".format(10)) ; print("{:0>6d}".format(1000))
17 | 000010
18 | 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, notées s (comme string) :

```
1 | >>> print("atom HN") ; print("atom HDE1")
2 | atom HN
3 | atom HDE1
4 | >>> print("atom {:>4s}".format("HN")) ; print("atom {:>4s}".format("HDE1"))
5 | atom   HN
6 | 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 rencontrés en biologie*).

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

```
1 | >>> print("{:7.3f}".format(perc_GC))
2 | 47.804
3 | >>> print("{:10.3f}".format(perc_GC))
4 |     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.

Enfin, si on veut afficher des accolades littérales et utiliser la méthode `.format()` en même temps, il faut doubler les accolades pour échapper au formatage :

```
1 | >>> print("Accolades littérales {} et pour le formatage {}".format(10))
2 | Accolades littérales {} et pour le formatage 10
```

La méthode `.format()` agit sur la chaîne de caractères à laquelle elle est *attachée* par un point et n'a rien à voir avec la fonction `print()`. Si on donne une chaîne de caractères suivie d'un `.format()` à la fonction `print()`, Python évalue d'abord le formatage 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 :

```
1 | >>> "{}:10.3f".format(perc_GC)
2 |      47.804'
3 | >>> type("{}:10.3f".format(perc_GC))
4 | <class 'str'>
```

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

3.3 Ancienne méthode de formatage des chaînes de caractères

Conseil : Pour les débutants, vous pouvez passer cette rubrique.

Dans d'anciens livres ou programmes Python, il se peut que vous rencontriez l'écriture formatée avec le style suivant :

```
1 | >>> x = 32
2 | >>> nom = "John"
3 | >>> print("%s a %d ans" % (nom, x))
4 | John a 32 ans
5 | >>> nb_G = 4500
6 | >>> nb_C = 2575
7 | >>> prop_GC = (nb_G + nb_C)/14800
8 | >>> print("On a %d G et %d C -> prop GC = %.2f" % (nb_G, nb_C, prop_GC))
9 | 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.

Cette ancienne façon de formater une chaîne de caractères vous est présentée à titre d'information. Ne l'utilisez pas dans vos programmes.

3.4 Note sur le vocabulaire et la syntaxe

Revenons quelques instants sur la notion de **méthode** abordée dans ce chapitre avec `.format()`. 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 :

```
1 | >>> "Joe a {} ans".format(20)
2 | 'Joe a 20 ans'
```

la méthode `.format()` est liée à `"Joe a {} ans"` qui est un objet de type chaîne de caractères. La méthode renvoie une nouvelle chaîne de caractères avec le bon formatage (ici, `'Joe a 20 ans'`).

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

3.5 Exercices

Conseil : utilisez l'interpréteur Python pour les exercices 2 à 5.

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*.

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 2 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 `.format()`. On souhaite que le programme affiche la sortie suivante :

```
1 | Le pourcentage de GC est 48      %
2 | Le pourcentage de GC est 47.8    %
3 | Le pourcentage de GC est 47.80   %
4 | Le pourcentage de GC est 47.804 %
```

Chapitre 4

Listes

4.1 Définition

Une **liste** est une structure de données qui contient une série de valeurs. 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 :

```
1 | >>> animaux = ['girafe', 'tigre', 'singe', 'souris']
2 | >>> tailles = [5, 2.5, 1.75, 0.15]
3 | >>> mixte = ['girafe', 5, 'souris', 0.15]
4 | >>> animaux
5 | ['girafe', 'tigre', 'singe', 'souris']
6 | >>> tailles
7 | [5, 2.5, 1.75, 0.15]
8 | >>> mixte
9 | ['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 pouvez appeler ses éléments par leur position. Ce numéro est appelé **indice** (ou **index**) de la liste.

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

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

```
1 | >>> animaux = ['girafe', 'tigre', 'singe', 'souris']
2 | >>> animaux[0]
3 | 'girafe'
4 | >>> animaux[1]
5 | 'tigre'
6 | >>> animaux[3]
7 | 'souris'
```

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

```
1 | >>> animaux[4]
2 | Traceback (innermost last):
3 |   File "<stdin>", line 1, in ?
4 | 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 :

```

1 | >>> ani1 = ['girafe', 'tigre']
2 | >>> ani2 = ['singe', 'souris']
3 | >>> ani1 + ani2
4 | ['girafe', 'tigre', 'singe', 'souris']
5 | >>> ani1 * 3
6 | ['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.

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

```

1 | >>> a = []
2 | >>> a
3 | []

```

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

```

1 | >>> a = a + [15]
2 | >>> a
3 | [15]
4 | >>> a = a + [-5]
5 | >>> a
6 | [15, -5]

```

puis avec la méthode `.append()` :

```

1 | >>> a.append(13)
2 | >>> a
3 | [15, -5, 13]
4 | >>> a.append(-3)
5 | >>> a
6 | [15, -5, 13, -3]

```

Dans l'exemple ci-dessus, nous ajoutons des éléments à une liste en utilisant l'opérateur de concaténation `+` ou la méthode `.append()`. 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 11 *Plus sur les listes*.

4.4 Indication négatif

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

```

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

```

ou encore :

```

1 | liste : ['A', 'B', 'C', 'D', 'E', 'F']
2 | indice positif : 0 1 2 3 4 5
3 | 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.

```

1 | >>> animaux = ['girafe', 'tigre', 'singe', 'souris']
2 | >>> animaux[-1]
3 | 'souris'
4 | >>> animaux[-2]
5 | 'singe'

```

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

```

1 | >>> animaux[-4]
2 | '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 indicage construit sur le modèle `[m:n+1]` pour récupérer tous les éléments, du `m`ème au `n`ème (de l'élément `m` inclus à l'élément `n+1` exclu). On dit alors qu'on récupère une **tranche** de la liste, par exemple :

```

1 | >>> animaux = ['girafe', 'tigre', 'singe', 'souris']
2 | >>> animaux[0:2]
3 | ['girafe', 'tigre']
4 | >>> animaux[0:3]
5 | ['girafe', 'tigre', 'singe']
6 | >>> animaux[0:]
7 | ['girafe', 'tigre', 'singe', 'souris']
8 | >>> animaux[:]
9 | ['girafe', 'tigre', 'singe', 'souris']
10 | >>> animaux[1:]
11 | ['tigre', 'singe', 'souris']
12 | >>> animaux[1:-1]
13 | ['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.

```

1 | >>> animaux = ['girafe', 'tigre', 'singe', 'souris']
2 | >>> animaux[0:3:2]
3 | ['girafe', 'singe']
4 | >>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
5 | >>> x
6 | [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
7 | >>> x[::-1]
8 | [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
9 | >>> x[::-2]
10 | [0, 2, 4, 6, 8]
11 | >>> x[::-3]
12 | [0, 3, 6, 9]
13 | >>> x[1:6:3]
14 | [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 :

```

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

```

1 | >>> list(range(10))
2 | [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 :

```

1 | >>> list(range(0,5))
2 | [0, 1, 2, 3, 4]
3 | >>> list(range(15,20))
4 | [15, 16, 17, 18, 19]
5 | >>> list(range(0,1000,200))
6 | [0, 200, 400, 600, 800]
7 | >>> list(range(2,-2,-1))
8 | [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`) :

```
1 | >>> list(range(10,0))
2 | []
```

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 :

```
1 | >>> list(range(10,0,-1))
2 | [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 :

```
1 | >>> enclos1 = ['girafe', 4]
2 | >>> enclos2 = ['tigre', 2]
3 | >>> enclos3 = ['singe', 5]
4 | >>> zoo = [enclos1, enclos2, enclos3]
5 | >>> zoo
6 | [[['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'indication habituel :

```
1 | >>> zoo[1]
2 | ['tigre', 2]
```

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

```
1 | >>> zoo[1][0]
2 | 'tigre'
3 | >>> zoo[1][1]
4 | 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 Exercices

Conseil : utilisez l'interpréteur Python.

4.9.1 Jours de la semaine

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

1. À partir de cette liste, comment récupérez-vous seulement les 5 premiers jours de la semaine d'une part, et ceux du week-end d'autre part ? Utilisez pour cela l'indication.
2. Cherchez un autre moyen pour arriver au même résultat (*en utilisant un autre indiquage*).
3. Trouvez deux manières pour accéder au dernier jour de la semaine.
4. Inversez les jours de la semaine en une commande.

4.9.2 Saisons

Créez 4 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][1]`. Comment expliquez-vous ce dernier résultat ?

4.9.3 Table de multiplication par 9

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

4.9.4 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 de manière compacte et efficace.

Imaginez par exemple que vous souhaitez 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 1000 ! 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 itération de la 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` ne sera pas détruite et contiendra ainsi 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, et `animal` est une **chaîne de caractères** car chaque élément de la liste 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 toujours sur un objet dit **séquentiel** (c'est-à-dire un objet constitué d'autres objets) tel qu'une liste. 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 avait été abordées rapidement dans le chapitre 1 *Introduction*.

Dans l'exemple suivant, le corps de la boucle contient deux instructions : `print(animal)` et `print(animal*2)` 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 `print("C'est fini")` 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 15 *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_
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).

```
1 >>> for i in [1,2,3]:
2 ...     print(i)
3 ...
4 1
5 2
6 3
```

5.1.2 Fonction `range()`

Python possède la fonction `range()` que nous avons rencontrée précédemment dans le chapitre 4 sur les *Listes* et qui est aussi bien commode pour faire une boucle sur une liste d'entiers de manière automatique :

```
1 >>> for i in range(4):
2 ...     print(i)
3 ...
4 0
5 1
6 2
7 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-ce possible ? Et bien `range()` est une fonction qui a été spécialement conçue pour cela¹, 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, mettez un nom explicite pour la variable d'itération. Par exemple :

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

5.1.4 Itération sur les indices

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

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

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 2 méthodes ? La plus efficace est celle qui réalise **les itérations directement sur les éléments** :

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

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

```
1 | >>> animaux = ['girafe','tigre','singe','souris']
2 | >>> for i in range(len(animaux)):
3 | ...     print("L'animal {} est un(e) {}".format(i, animaux[i]))
4 ...
5 | L'animal 0 est un(e) girafe
6 | L'animal 1 est un(e) tigre
7 | L'animal 2 est un(e) singe
8 | L'animal 3 est un(e) souris
```

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

```
1 | >>> animaux = ['girafe','tigre','singe','souris']
2 | >>> for i, animal in enumerate(animaux):
3 | ...     print("L'animal {} est un(e) {}".format(i, animal))
4 ...
```

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

```

5 L'animal 0 est un(e) girafe
6 L'animal 1 est un(e) tigre
7 L'animal 2 est un(e) singe
8 L'animal 3 est un(e) souris

```

5.2 Comparaisons

Avant de passer à une autre sorte de boucles (les boucles `while`), nous abordons tout de suite les **comparaisons**. Celles-ci seront reprises dans le chapitre 6 sur les *Tests*.

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

Syntaxe Python	Signification
<code>==</code>	égal à
<code>!=</code>	différent de
<code>></code>	supérieur à
<code>>=</code>	supérieur ou égal à
<code><</code>	inférieur à
<code><=</code>	inférieur ou égal à

Observez les exemples suivants avec des nombres entiers.

```

1 | >>> x = 5
2 | >>> x == 5
3 | True
4 | >>> x > 10
5 | False
6 | >>> x < 10
7 | True

```

Python renvoie la valeur `True` si la comparaison est vraie et `False` si elle est fausse. `True` et `False` sont des booléens (un nouveau type de variable).

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.

```

1 | >>> animal = "tigre"
2 | >>> animal == "tig"
3 | False
4 | >>> animal != "tig"
5 | True
6 | >>> animal == 'tigre'
7 | 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 :

```

1 | >>> "a" < "b"
2 | 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 ASCII² 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 :

```

1 | >>> "ali" < "alo"
2 | True
3 | >>> "abb" < "ada"
4 | 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.

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

5.3 Boucles while

Une autre alternative à l'instruction `for` couramment utilisée en informatique est la boucle `while`. Le principe est simple. Une série d'instructions est exécutée tant qu'une condition est vraie. Par exemple :

```

1 | >>> i = 1
2 | >>> while i <= 4:
3 | ...     print(i)
4 | ...     i = i + 1
5 |
6 | 1
7 | 2
8 | 3
9 | 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).

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

1. Initialisation de la variable de test avant la boucle (ligne 1).
2. Test de la variable associée à l'instruction `while` (ligne 2).
3. Mise à jour de la variable de test 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. Il faut ensuite convertir cette dernière en entier (avec la fonction `int()`).

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 manières différentes (deux avec `for` et une avec `while`).

5.4.2 Boucle 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 : n'hésitez pas 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

Avez 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 :

```
1 | 8
2 | 24
3 | 48
4 | [...]
```

5.4.7 Triangle

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

```
1 | *
2 | **
3 | ***
4 | ****
5 | *****
6 | ******
7 | ******
8 | *********
9 | *********
10 | *********
```

5.4.8 Triangle inversé

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

```
1 | *********
2 | *********
3 | *********
4 | *****
5 | *****
6 | *****
7 | ****
8 | ***
9 | **
10 | *
```

5.4.9 Triangle gauche

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

```
1 | *
2 | **
3 | ***
4 | ****
5 | *****
6 | *****
7 | *****
8 | *****
9 | *****
10 | *****
```

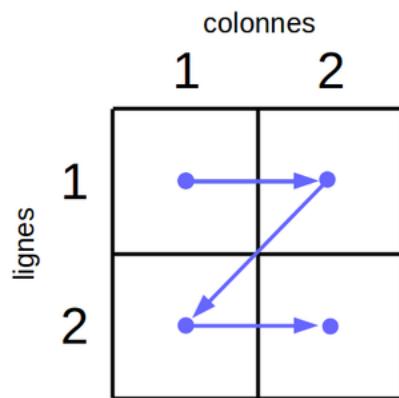


FIGURE 5.1 – Parcours d'une matrice.

5.4.10 Pyramide

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

```

1   *
2   ***
3   *****
4   ******
5   **** ***
6   **** ****
7   **** *****
8   **** **** ****
9   **** **** ****
10  **** **** ****

```

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

```

1| ligne  colonne
2|     1      1
3|     1      2
4|     2      1
5|     2      2

```

Attention à bien respecter l'alignement des chiffres qui doit être justifié à droite sur 4 caractères. Testez pour une matrice 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 ($N=4$) :

	1	2	3	4
1				
2				
3				
4				

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

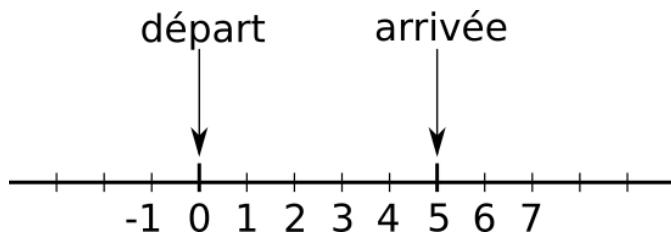


FIGURE 5.3 – Sauts de puce.

```

1 ligne   colonne
2     1     2
3     1     3
4     1     4
5     2     3
6     2     4
7     3     4
8 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`, simuler 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 : vous utiliserez l'instruction `random.choice([-1, 1])` qui renvoie au hasard les valeurs -1 ou 1 avec la même probabilité. Avant d'utiliser cette instruction vous mettrez au tout début de votre script la ligne

```
import random
```

Nous reverrons la signification de cette syntaxe particulière dans le chapitre 8 *Modules*.

5.4.14 Suite de Fibonacci (exercice ++)

La suite de Fibonacci³ 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

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

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 ?

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 :

```
1 | >>> x = 2
2 | >>> if x == 2:
3 | ...     print("Le test est vrai !")
4 | ...
5 | Le test est vrai !
```

et un second :

```
1 | >>> x = "souris"
2 | >>> if x == "tigre":
3 | ...     print("Le test est vrai !")
4 | ...
```

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

- Dans le premier exemple, le test étant vrai, l'instruction `print("Le test est vrai !")` est exécutée. Dans le second exemple, le test est faux et rien n'est affiché.
- Les blocs d'instruction 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.
- Et 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` :

```
1 | >>> x = 2
2 | >>> if x == 2:
3 | ...     print("Le test est vrai !")
4 | ... else:
5 | ...     print("Le test est faux !")
6 | ...
7 | Le test est vrai !
8 | >>> x = 3
9 | >>> if x == 2:
10 | ...    print("Le test est vrai !")
11 | ... else:
12 | ...    print("Le test est faux !")
13 | ...
14 | 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 8 *Modules*, admettez pour le moment qu'elle est nécessaire.

```

1| >>> import random
2| >>> base = random.choice(["a", "t", "c", "g"])
3| >>> if base == "a":
4| ...     print("choix d'une adénine")
5| ... elif base == "t":
6| ...     print("choix d'une thymine")
7| ... elif base == "c":
8| ...     print("choix d'une cytosine")
9| ... elif base == "g":
10| ...    print("choix d'une guanine")
11| ...
12| 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`.

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

```

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

```

Résultat :

```

1| Le test est vrai
2| 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("car la variable nb vaut {}".format(nb))

```

Résultat :

```

1| car la variable nb vaut 4
2| Le test est vrai
3| car la variable nb vaut 5
4| 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 appartient au bloc d'instructions de la boucle `for`, 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 le **OU** et le **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

```

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 permettent de modifier le comportement d'une boucle (**for** ou **while**) avec un test. L'instruction **break** stoppe la boucle.

```

1 | >>> for i in range(5):
2 | ...     if i > 2:
3 | ...         break
4 | ...     print(i)
5 | ...
6 | 0
7 | 1
8 | 2

```

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

```

1 | >>> for i in range(5):
2 | ...     if i == 2:
3 | ...         continue
4 | ...     print(i)
5 | ...
6 | 0
7 | 1
8 | 3
9 | 4

```

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 limitations¹. 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 `False` 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 de s'en rendre compte est d'utiliser l'écriture formatée en demandant l'affichage d'un grand nombre de décimales :

```
1 | >>> 0.3
2 | 0.3
3 | >>> "{:.5f}".format(0.3)
4 | '0.30000'
5 | >>> "{:.60f}".format(0.3)
6 | '0.2999999999999988897769753748434595763683319091796875000000'
7 | >>> "{:.60f}".format(3.0 - 2.7)
8 | '0.29999999999999822364316059974953532218933105468750000000000'
```

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 donc essentiel d'avoir cela en tête lorsque l'on effectue un test .

Conseil

Pour les raisons évoquées ci-dessus, il ne faut surtout pas tester si un *float* est égal à une certaine valeur. 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 :

```
1 | >>> delta = 0.0001
2 | >>> var = 3.0 - 2.7
3 | >>> 0.3 - delta < var < 0.3 + delta
4 | True
5 | >>> abs(var - 0.3) < delta
6 | True
```

Ici on teste si `var` est compris dans l'intervalle $0.3 \pm \text{delta}$. Les deux méthodes mènent à un résultat strictement équivalent :

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

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.

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

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 :

```
["A", "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 :

```
1 | >>> 4 % 3
2 | 1
3 | >>> 5 % 3
4 | 2
5 | >>> 4 % 2
6 | 0
7 | >>> 5 % 2
8 | 1
9 | >>> 6 % 2
10 | 0
11 | >>> 7 % 2
12 | 1
```

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 Syracuse² 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. Si 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.

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...

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

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 ?

Remarque

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 ± 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 1TFE³ :

```

1 | [[48.6, 53.4], [-124.9, 156.7], [-66.2, -30.8], \
2 | [-58.8, -43.1], [-73.9, -40.6], [-53.7, -37.5], \
3 | [-80.6, -26.0], [-68.5, 135.0], [-64.9, -23.5], \
4 | [-66.9, -45.5], [-69.6, -41.0], [-62.7, -37.5], \
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 3 premiers acides aminés :

```

1 | [48.6, 53.4] n'est pas en hélice
2 | [-124.9, 156.7] n'est pas en hélice
3 | [-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 protéines⁴.

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 wikipédia⁵.

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 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.

3. <https://www.rcsb.org/structure/1TFE>

4. https://fr.wikipedia.org/wiki/Structure_des_prot%C3%A9ines#Angles_d%C3%A9fis_and_structures_secondaires

5. http://fr.wikipedia.org/wiki/Nombre_premier

Méthode 1 (peu optimale mais assez intuitive)

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

Méthode 2 (plus optimale et plus rapide, mais un peu plus compliquée)

Parcourez tous les nombres de 2 à 100 et vérifiez si ceux-ci sont composés, c'est-à-dire qu'ils sont le produit de deux nombres premiers. Pratiquement, cela consiste à vérifier que le reste de la division entière (opérateur modulo %) entre le nombre considéré et chaque nombre premier déterminé jusqu'à maintenant est nul. Le cas échéant, ce nombre n'est pas premier. Attention, pour cette méthode, il faudra initialiser la liste de nombres premiers avec le premier nombre premier (donc 2!).

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

La recherche par dichotomie⁶ 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 ($75 + (100-75)/2$).
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 :

```
1 | >>> lettre = input("Entrez une lettre : ")
2 | Entrez une lettre : P
3 | >>> print(lettre)
4 | P
```

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

```
1 | Pensez à un nombre entre 1 et 100.
2 | Est-ce votre nombre est plus grand, plus petit ou égal à 50 ? [+/-/=] +
3 | Est-ce votre nombre est plus grand, plus petit ou égal à 75 ? [+/-/=] +
4 | Est-ce votre nombre est plus grand, plus petit ou égal à 87 ? [+/-/=] -
5 | Est-ce votre nombre est plus grand, plus petit ou égal à 81 ? [+/-/=] -
```

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

- 6| Est-ce votre nombre est plus grand, plus petit ou égal à 78 ? [+/-/=] +
7| Est-ce votre nombre est plus grand, plus petit ou égal à 79 ? [+/-/=] =
8| 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 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.

Avant de passer à un exemple concret, créez un fichier dans un éditeur de texte que vous enregistrerez dans votre répertoire courant avec le nom `zoo.txt` et le contenu suivant :

```
1 girafe
2 tigre
3 singe
4 souris
```

Ensuite, testez le code suivant dans l'interpréteur Python :

```
1 >>> filin = open("zoo.txt", "r")
2 >>> filin
3 <_io.TextIOWrapper name='zoo.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. L'instruction `open()` ouvre le fichier `zoo.txt`. Ce fichier est ouvert en lecture seule, comme l'indique le second argument `r` (pour *read*) de la fonction `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("zoo.txt", "r")` suppose que le fichier `zoo.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/zoo.txt` pour Linux ou Mac OS X ou `C:\Users\pierre\zoo.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 correspondrait 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("zoo.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. 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 >>> with open("zoo.txt", 'r') as filin:
2 ...     lignes = filin.readlines()
3 ...     for ligne in lignes:
4 ...         print(ligne)
5 ...
6 girafe
7
8 tigre
9
10 singe
11
12 souris
13
14 >>>

```

Remarque

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

7.1.1 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.

```

1 >>> with open("zoo.txt", "r") as filin:
2 ...     filin.read()
3 ...
4 'girafe\n\tigre\n\tsinge\n\souris\n'
5 >>>

```

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

7.1.2 Méthode .readline()

La méthode `.readline()` (sans `s` à la fin) lit une ligne d'un fichier et la renvoie sous forme d'une 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.

```

1 | >>> with open("zoo.txt", "r") as filin:
2 | ...     ligne = filin.readline()
3 | ...     while ligne != "":
4 | ...         print(ligne)
5 | ...         ligne = filin.readline()
6 ...
7 | girafe
8 |
9 | tigre
10 |
11 | singe
12 |
13 | souris
14 |
15 | >>>

```

7.1.3 Itérations directe 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.

```

1 | >>> with open("zoo.txt", "r") as filin:
2 | ...     for ligne in filin:
3 | ...         print(ligne)
4 ...
5 | girafe
6 |
7 | tigre
8 |
9 | singe
10 |
11 | souris
12 |
13 | >>>

```

La boucle `for` va demander à Python d'aller lire le fichier ligne par ligne. 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 différencier 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("zoo2.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 de liste de chaînes de caractères `animaux2`.

Ligne 2. Ouverture du fichier `zoo2.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 `zoo2.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 :

```
1 | >>> animaux2 = ["poisson", "abeille", "chat"]
2 | >>> with open("zoo2.txt", "w") as filout:
3 | ...     for animal in animaux2:
4 | ...         filout.write("{}\n".format(animal))
5 |
6 | 8
7 | 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 `zoo2.txt` est alors :

```
1 | poisson
2 | abeille
3 | chat
```

Vous voyez qu'il est extrêmement 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 :

```
1 | with open("zoo.txt", "r") as fichier1, open("zoo2.txt", "w") as fichier2:
2 |     for ligne in fichier1:
3 |         fichier2.write("* " + ligne)
```

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

```
1 | souris
2 | girafe
3 | lion
4 | singe
```

alors le contenu de `zoo2.txt` sera :

```
1 | * souris
2 | * girafe
3 | * lion
4 | * 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 est utilisable dans d'autres contextes².

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

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>

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 conversions 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 particulier, 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 rencontrés 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 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

Le fichier `notes.txt`⁵ 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 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 3 premières lignes attendues pour le fichier `notes2.txt` :

```
1 | 13.5  admis
2 | 17.0  admis
3 | 9.5   recalé
```

7.7.3 Spirale (exercice +++)

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

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 :

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

4. <https://python.sdv.univ-paris-diderot.fr/data-files/notes.txt>

5. <https://python.sdv.univ-paris-diderot.fr/data-files/notes.txt>

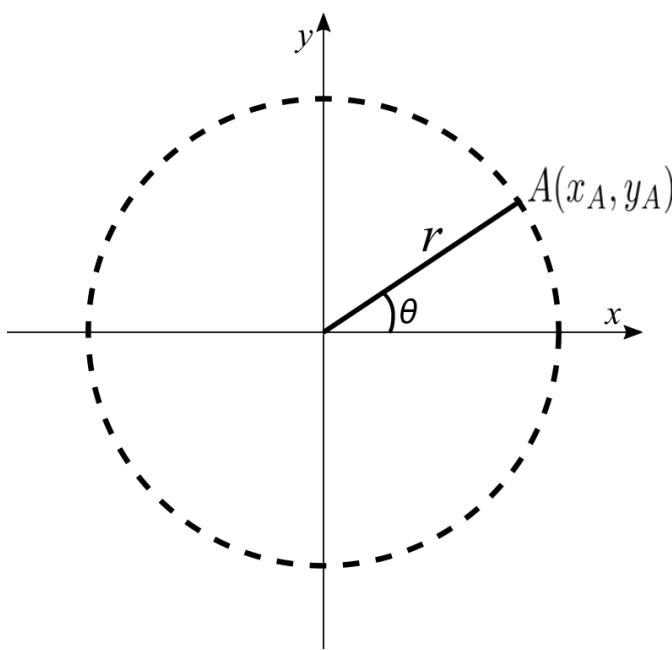


FIGURE 7.1 – Point A sur le cercle de rayon r.

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

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 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 8 *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`.

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 ;
- 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 :

```
1 0.50000 0.00000
2 0.59700 0.05990
3 0.68605 0.13907
4 0.76427 0.23642
5 0.82895 0.35048
6 0.87758 0.47943
7 [...] [...]
```

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 plt.figure(figsize=(8,8))
```

```
12| mini = min(x+y) * 1.2
13| maxi = max(x+y) * 1.2
14| plt.xlim(mini, maxi)
15| plt.ylim(mini, maxi)
16| plt.plot(x, y)
17| plt.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 17 *Quelques modules d'intérêt en bioinformatique*.

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

Chapitre 8

Modules

8.1 Définition

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

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

La plupart de ces modules sont déjà installés dans les versions standards de Python. Vous pouvez accéder à une documentation exhaustive¹ sur le site de Python. N'hésitez pas à explorer un peu ce site, la quantité de modules disponibles est impressionnante (plus de 300).

8.2 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.
- Ensuite, 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 chapitre 7 *Fichiers*). Ce module nous a donné accès aux fonctions trigonométriques 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
```

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.

1. <https://docs.python.org/3/py-modindex.html>

2. <https://docs.python.org/fr/3/library/random.html#module-random>

Par exemple, la méthode `.format()` dans l'instruction

```
"{}".format(3.14)
```

utilise l'objet chaîne de caractères "`{}`" (auquel elle est liée) pour finalement renvoyer une autre chaîne de caractères "`3.14`".

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 Python³ précise bien que dans ce cas `.cos()` est une fonction. Dans cet ouvrage, nous utiliserons systématiquement le mot **fonction** lorsqu'on évoquera des fonctions issues de modules.

Si cela vous paraît encore ardu, ne vous inquiétez pas, c'est à force de pratiquer et de lire que vous vous approprierez le vocabulaire. Ici, 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 :

```
1 | >>> from random import randint
2 | >>> randint(0,10)
3 | 7
```

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

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

```
1 | >>> from random import *
2 | >>> x = [1, 2, 3, 4]
3 | >>> shuffle(x)
4 | >>> x
5 | [2, 3, 1, 4]
6 | >>> shuffle(x)
7 | >>> x
8 | [4, 2, 1, 3]
9 | >>> randint(0,50)
10 | 46
11 | >>> uniform(0,2.5)
12 | 0.64943174760727951
```

L'instruction `from random import *` importe toutes les fonctions du module `random`. On peut ainsi utiliser toutes ses fonctions directement, comme par exemple `shuffle()` qui permute une liste aléatoirement.

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

```
1 | from random import *
```

nous vous conseillons de charger le module seul :

```
1 | import random
```

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

```
1 | random.randint(0,2)
```

Il est également possible de définir un alias (un nom plus court) pour un module :

```
1 | >>> import random as rand
2 | >>> rand.randint(1, 10)
3 | 6
4 | >>> rand.uniform(1, 3)
5 | 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 | >>> del random
5 | >>> random.randint(0,10)
6 | Traceback (most recent call last):
7 |   File "<stdin>", line 1, in ?
8 | NameError: name 'random' is not defined
```

On constate alors qu'un rappel d'une fonction du module `random` après l'avoir vidé de la mémoire retourne un message d'erreur.

3. <https://docs.python.org/3/tutorial/modules.html>

8.3 Obténir 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()` :

```

1 | >>> import random
2 | >>> help(random)
3 | ...

```

Ce qui renvoie quelque chose du type :

```

1 | Help on module random:
2 |
3 | NAME
4 |     random - Random variable generators.
5 |
6 | MODULE REFERENCE
7 |     https://docs.python.org/3.7/library/random
8 |
9 |     The following documentation is automatically generated from the Python
10 |     source files. It may be incomplete, incorrect or include features that
11 |     are considered implementation detail and may vary between Python
12 |     implementations. When in doubt, consult the module reference at the
13 |     location listed above.
14 |
15 | DESCRIPTION
16 |     integers
17 |     -----
18 |         uniform within range
19 |
20 |     sequences
21 |     -----
22 |         pick random element
23 |         pick random sample

```

Remarque

- Pour vous déplacer dans l'aide, utilisez les flèches du haut et du bas pour parcourir les lignes les unes après les autres, 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 / 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 obtenir de l'aide sur une fonction particulière d'un module de la manière suivante :
`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.

```

1 | >>> t = [1, 2, 3]
2 | >>> help(t)
3 | Help on list object:
4 |
5 | class list(object)
6 |     list() -> new list
7 |     list(sequence) -> new list initialized from sequence's items
8 |
9 |     Methods defined here:
10 |
11 |     __add__(...)
12 |         x.__add__(y) <==> x+y
13 |
14 | ...

```

Enfin, pour connaître d'un seul coup d'œil toutes les méthodes ou variables associées à un objet, utilisez la fonction `dir()` :

```

1 | >>> import random
2 | >>> dir(random)
3 | ['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
4 | 'SystemRandom', 'TWOPI', 'WichmannHill', '_BuiltInMethodType', '_MethodT
5 | ype', '__all__', '__builtins__', '__doc__', '__file__', '__name__', '__ac
6 | os', '__ceil__', '__cos__', '__e__', '__exp__', '__hexlify__', '__inst
7 | ', '__log__', '__pi__', '__random__', '__sin__', '__sqrt__', '__test
8 | ', 'choice', 'expovariate', 'gammavariate', 'gauss', 'g
9 | etrandbits', 'getstate', 'jumpahead', 'lognormvariate', 'normalvariate
10 | ', 'paretovariate', 'randint', 'random', 'randrange', 'sample', 'seed', 's
11 | etestate', 'shuffle', 'uniform', 'vonmisesvariate', 'weibullvariate']
12 | >>>

```

8.4 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 modules⁴ sur le site de Python :

- *math*⁵ : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- *sys*⁶ : interaction avec l'interpréteur Python, passage d'arguments.
- *os*⁷ : dialogue avec le système d'exploitation.
- *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.
- *re*¹² : gestion des expressions régulières.

Nous vous conseillons d'aller explorer les pages de ces modules pour découvrir toutes leurs potentialités.

Vous verrez dans le chapitre 14 *Création de module* 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. Citons-en quelques-uns : *NumPy* (manipulations de vecteurs, matrices, algèbre linéaire), *Biopython* (recherche dans les banques de données biologiques, manipulation de séquences ou de structures de biomolécules), *matplotlib* (construction de graphiques), *pandas* (analyse de données)... Ces modules vous seront présentés dans le chapitre 17 *Quelques modules d'intérêt en bioinformatique*.

8.5 Module sys : passage d'arguments

Le module *sys*¹³ 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 court script suivant que l'on enregistrera sous le nom *test.py* :

```
1| import sys
2| print(sys.argv)
```

Ensuite, dans un *shell*, exécutons le script *test.py* suivi de plusieurs arguments. Par exemple :

```
1| $ python test.py salut girafe 42
2| ['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 lui-même qu'on retrouve comme premier élément de cette liste dans *sys.argv[0]*. On peut donc accéder à chacun des arguments du script avec *sys.argv[1]*, *sys.argv[2]*...

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 :

```
1| import sys
2|
3| if len(sys.argv) != 2:
4|     sys.exit("ERREUR : il faut exactement un argument .")
5|
6| print("Argument vaut : {}".format(sys.argv[1]))
```

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>
 13. <https://docs.python.org/fr/3/library/sys.html#module-sys>

Puis on l'exécute sans argument :

```
1 | $ python test.py
2 | ERREUR : il faut exactement un argument.
```

et avec un argument :

```
1 | $ python test.py 42
2 | Argument vaut : 42
```

Notez qu'ici on vérifie 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 à titre d'exemple le script `compte_lignes.py` qui va prendre comme argument le nom d'un fichier puis afficher 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("{} contient {}".format(nom_fichier, taille))
```

Supposons que dans le même répertoire, nous ayons le fichier `zoo1.txt` dont voici le contenu :

```
1 | girafe
2 | tigre
3 | singe
4 | souris
```

et le fichier `zoo2.txt` qui contient :

```
1 | poisson
2 | abeille
3 | chat
```

Utilisons maintenant notre script `compte_lignes.py` :

```
1 | $ python compte_lignes.py
2 | ERREUR : il faut exactement un argument.
3 | $ python compte_lignes.py zoo1.txt
4 | zoo1.txt contient 4 lignes.
5 | $ python compte_lignes.py zoo2.txt
6 | zoo2.txt contient 3 lignes.
```

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 donné en argument, de compter puis d'afficher le nombre de lignes.

Par contre, le script ne vérifie pas si le fichier existe bien :

```
1 | $ python compte_lignes.py zoo3.txt
2 | Traceback (most recent call last):
3 |   File "compte_lignes.py", line 8, in <module>
4 |     with open(nom_fichier, "r") as f_in:
5 |   FileNotFoundError: [Errno 2] No such file or directory: 'zoo3.txt'
```

La lecture de la partie suivante va nous permettre d'améliorer notre script `compte_lignes.py`.

8.6 Module *os* : interaction avec le système d'exploitation

Le module *os*¹⁴ gère l'interface avec le système d'exploitation.

La fonction `os.path.exists()` est une fonction pratique de ce module qui vérifie la présence d'un fichier sur le disque dur.

14. <https://docs.python.org/fr/3/library/os.html#module-os>

```

1| >>> import sys
2| >>> import os
3| >>> if os.path.exists("toto.pdb"):
4| ...     print("le fichier est présent")
5| ... else:
6| ...     sys.exit("le fichier est absent")
7| ...
8| le fichier est absent

```

Dans cet exemple, si le fichier n'existe pas sur le disque, on quitte le programme avec la fonction `exit()` du module `sys` que nous venons de voir.

La fonction `os.getcwd()` renvoie le répertoire (sous forme de chemin complet) depuis lequel est lancé Python :

```

1| >>> import os
2| >>> os.getcwd()
3| '/home/pierre'

```

Enfin, la fonction `os.listdir()` renvoie le contenu du répertoire depuis lequel est lancé Python :

```

1| >>> import os
2| >>> os.listdir()
3| ['1BTA.pdb', 'demo.py', 'tests']

```

Le résultat est renvoyé sous forme d'une liste contenant à la fois le nom des fichiers et des répertoires.

8.7 Exercices

Conseils : pour les trois premiers exercices, utilisez l'interpréteur Python. Pour les exercices suivants, créez des scripts puis exécutez-les dans un *shell*.

8.7.1 Racine carrée

Affichez sur la même ligne les nombres de 10 à 20 (inclus) ainsi que leur racine carrée avec 3 décimales. Utilisez pour cela le module `math` avec la fonction `sqrt()`. Exemple :

```

1| 10 3.162
2| 11 3.317
3| 12 3.464
4| 13 3.606
5| [...]

```

Documentation de la fonction `math.sqrt()` :

<https://docs.python.org/3/library/math.html#math.sqrt>

8.7.2 Cosinus

Calculez le cosinus de $\pi/2$ en utilisant le module `math` avec la fonction `cos()` et la constante `pi`.

Documentation de la fonction `math.cos()` :

<https://docs.python.org/fr/3/library/math.html#math.cos>

Documentation de la constante `math.pi` :

<https://docs.python.org/fr/3/library/math.html#math.pi>

8.7.3 Nom et contenu du répertoire courant

Affichez le nom 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.

Documentation de la fonction `os.getcwd()` :

<https://docs.python.org/fr/3/library/os.html#os.getcwd>

Documentation de la fonction `os.listdir()` :

<https://docs.python.org/fr/3/library/os.html#os.listdir>

8.7.4 Affichage temporisé

Affichez les nombres de 1 à 10 avec 1 seconde d'intervalle. Utilisez pour cela le module `time` et sa fonction `sleep()`.

Documentation de la fonction `time.sleep()` :

<https://docs.python.org/fr/3/library/time.html#time.sleep>

8.7.5 Séquences aléatoires de chiffres

Générez une séquence aléatoire de 6 chiffres, ceux-ci étant des entiers tirés entre 1 et 4. Utilisez le module `random` avec la fonction `randint()`.

Documentation de la fonction `random.randint()` :

<https://docs.python.org/fr/3/library/random.html#random.randint>

8.7.6 Séquences aléatoires d'ADN

Générez une séquence aléatoire d'ADN de 20 bases de deux manières différentes. Utilisez le module `random` avec la fonction `randint()` ou `choice()`.

Documentation de la fonction `random.randint()` :

<https://docs.python.org/fr/3/library/random.html#random.randint>

Documentation de la fonction `random.choice()` :

<https://docs.python.org/fr/3/library/random.html#random.choice>

8.7.7 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, vous utiliserez, au choix, la fonction `random.randint()` ou `random.choice()` abordées dans l'exercice précédent.

8.7.8 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 `zoo1.txt` et `zoo2.txt` sont bien dans le répertoire courant, mais pas `zoo3.txt` :

```
1 | $ python compte_lignes.py zoo1.txt
2 | zoo1.txt contient 4 lignes.
3 | $ python compte_lignes.py zoo2.txt
4 | zoo2.txt contient 3 lignes.
5 | $ python compte_lignes.py zoo3.txt
6 | ERREUR : zoo3.txt n'existe pas.
```

8.7.9 Détermination du nombre pi par la méthode Monte Carlo (exercice ++)

Soit un cercle de rayon 1 (en trait plein sur la figure 8.1) 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 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 :

- Choisissez aléatoirement les coordonnées x et y d'un point entre -1 et 1. Utilisez la fonction `uniform()` du module `random`.
- Calculez la distance entre le centre du cercle et ce point.
- Déterminez si cette distance est inférieure au rayon du cercle, c'est-à-dire si le point est dans le cercle ou pas.

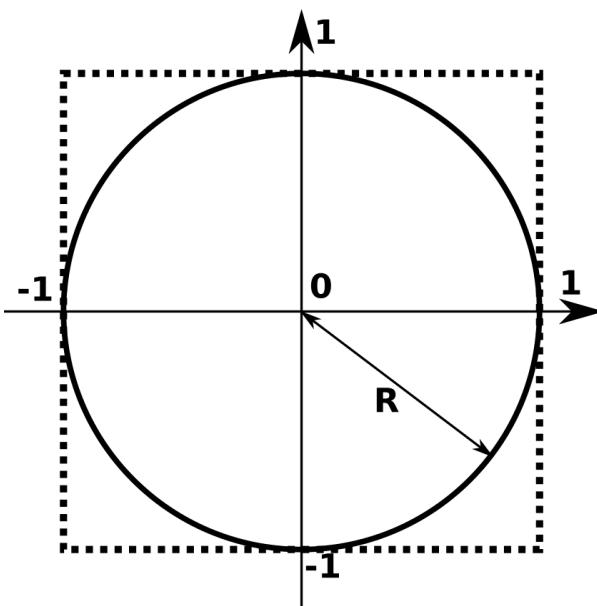


FIGURE 8.1 – Cercle de rayon 1 inscrit dans un carré de côté 2.

— 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 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 :

$$d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$$

Documentation de la fonction `random.uniform()` :

<https://docs.python.org/3/library/random.html#random.uniform>

Chapitre 9

Fonctions

9.1 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 » (voir figure 9.1) :

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. Et qui renvoie un objet Python ou rien du tout.

Par exemple, si vous appelez la fonction `len()` de la manière suivante :

```
1 | >>> len([0, 1, 2])
2 | 3
```

voici ce qui se passe :

1. vousappelez `len()` en lui passant une liste en argument (ici `[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 vousappelez 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) :

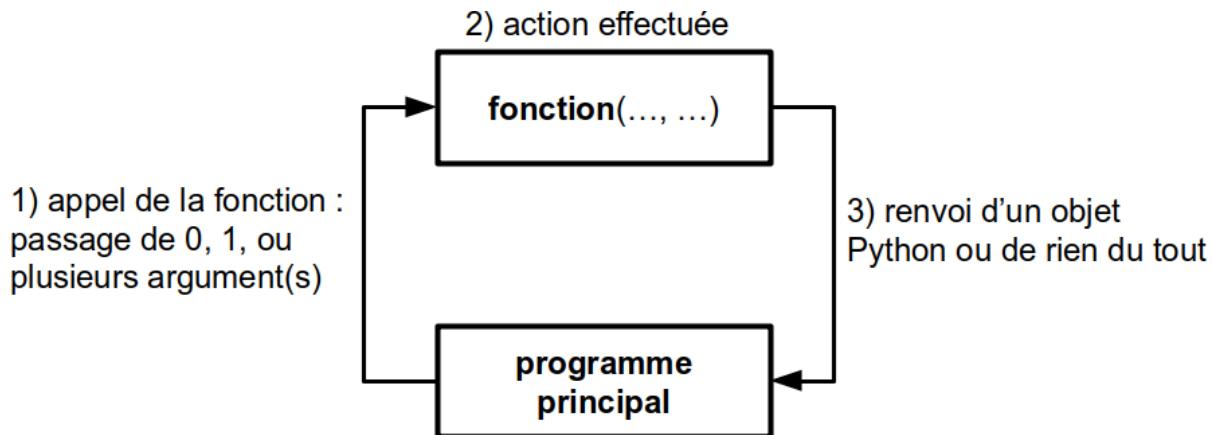


FIGURE 9.1 – Fonctionnement schématique d'une fonction.

```
1 | >>> ma_liste.append(5)
2 | 
3 | 1. Vous passez l'entier 5 en argument;
4 | 2. la méthode append() ajoute l'entier 5 à l'objet ma_liste;
5 | 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 9.1 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 12 *Plus sur les fonctions*.

9.2 Définition

Pour définir une fonction, Python utilise le mot-clé `def` et si on veut que celle-ci renvoie quelque chose, il faut utiliser le mot-clé `return`. Par exemple :

```
1 | >>> def carre(x):
2 | ...     return x**2
3 | ...
4 | >>> print(carre(2))
5 | 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 :

```
1 | >>> res = carre(2)
2 | >>> print(res)
3 | 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 :

```
1 | >>> def hello():
2 | ...     print("bonjour")
3 | ...
4 | >>> hello()
5 | 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 :

```
1 | >>> var = hello()
2 | bonjour
3 | >>> print(var)
4 | 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.

9.3 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 0 ou 1 argument. Dans les chapitres précédents, vous avez rencontré des fonctions internes à Python qui prenaient au moins 2 arguments. Souvenez-vous par exemple de `range(1, 10)` ou encore `range(1, 10, 2)`. Le nombre d'argument 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 :

```

1 | >>> def fois(x, y):
2 | ...     return x*y
3 | ...
4 | >>> fois(2, 3)
5 | 6
6 | >>> fois(3.1415, 5.23)
7 | 16.430045000000003
8 | >>> fois('to', 2)
9 | 'toto'
```

L'opérateur `*` reconnaît plusieurs types (entiers, *floats*, chaînes de caractères). 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.

9.4 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 :

```

1 | >>> def carre_cube(x):
2 | ...     return x**2, x**3
3 | ...
4 | >>> carre_cube(2)
5 | (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 verrons dans le chapitre 13 *Dictionnaires et tuples* (mais *grossso modo*, il s'agit d'une sorte de liste avec des propriétés différentes). Notre fonction pourrait tout autant renvoyer une liste :

```

1 | >>> def carre_cube2(x):
2 | ...     return [x**2, x**3]
3 | ...
4 | >>> carre_cube2(3)
5 | [9, 27]
```

Renvoyer un *tuple* ou une liste de deux éléments (ou plus) est très pratique en conjonction avec l'**affectation multiple**, par exemple :

```

1 | >>> z1, z2 = carre_cube2(3)
2 | >>> z1
3 | 9
4 | >>> z2
5 | 27
```

Cela permet de récupérer plusieurs valeurs renvoyées par une fonction et de les affecter à la volée à des variables différentes.

9.5 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 ?

```

1 | >>> def fois(x, y):
2 | ...     return x*y
3 | ...
4 | >>> fois(2, 3)
5 | 6
6 | >>> fois(2)
7 | Traceback (most recent call last):
8 |   File "<stdin>", line 1, in <module>
9 |     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 :

```

1 | >>> def fct(x=1):
2 | ...     return x
3 | ...
4 | >>> fct()
5 | 1
6 | >>> fct(10)
7 | 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 :

```

1 | >>> def fct(x=0, y=0, z=0):
2 | ...     return x, y, z
3 | ...
4 | >>> fct()
5 | (0, 0, 0)
6 | >>> fct(10)
7 | (10, 0, 0)
8 | >>> fct(10, 8)
9 | (10, 8, 0)
10 | >>> fct(10, 8, 3)
11 | (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 pourrions-nous faire si 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 :

```

1 | >>> fct(z=10)
2 | (0, 0, 10)

```

Python permet même de rentrer les arguments par mot-clé dans un ordre arbitraire :

```

1 | >>> fct(z=10, x=3, y=80)
2 | (3, 80, 10)
3 | >>> fct(z=10, y=80)
4 | (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 :

```

1 | >>> def fct(a, b, x=0, y=0, z=0):
2 | ...     return a, b, x, y, z
3 | ...
4 | >>> fct(1, 1)

```

```

5| (1, 1, 0, 0, 0)
6| >>> fct(1, 1, z=5)
7| (1, 1, 0, 0, 5)
8| >>> fct(1, 1, z=5, y=32)
9| (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.

```

1| >>> fct(z=0)
2| Traceback (most recent call last):
3|   File "<stdin>", line 1, in <module>
4| 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` :

```

1| >>> print("Message ", end="")
2| Message >>>

```

Nous verrons, dans le chapitre 20 *Fenêtres graphiques et Tkinter*, que l'utilisation d'arguments par mot-clé est systématique lorsqu'on crée un objet graphique (une fenêtre, un bouton, etc.).

9.6 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 :

```

1| # définition d'une fonction carre()
2| def carre(x):
3|     y = x**2
4|     return y
5|
6| # programme principal
7| z = 5
8| resultat = carre(z)
9| 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>

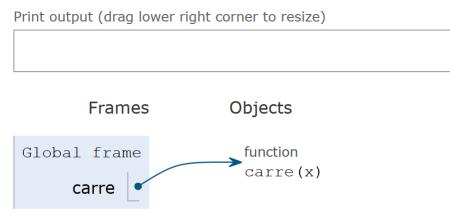
```
Python 3.6
1 # définition d'une fonction carre()
→ 2 def carre(x):
3     y = x**2
4     return y
5
6 # programme principal
7 z = 5
8 resultat = carre(z)
9 print(resultat)
```



Frames Objects

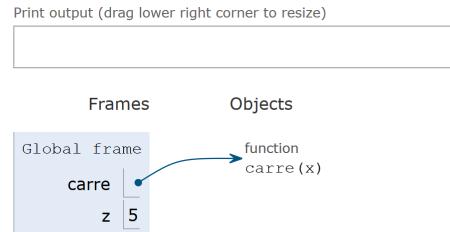
- Étape 2 : Python met en mémoire la fonction `carre()`. Notez qu'il ne l'exécute pas ! La fonction est mise dans un espace de la mémoire nommé *Global frame*, il s'agit de l'espace du programme principal. Dans cet espace, seront stockées toutes les variables *globales* créées dans le programme. Python est maintenant prêt à exécuter le programme principal.

```
Python 3.6
1 # définition d'une fonction carre()
→ 2 def carre(x):
3     y = x**2
4     return y
5
6 # programme principal
→ 7 z = 5
8 resultat = carre(z)
9 print(resultat)
```

Frames Objects
Global frame
carre
z 5

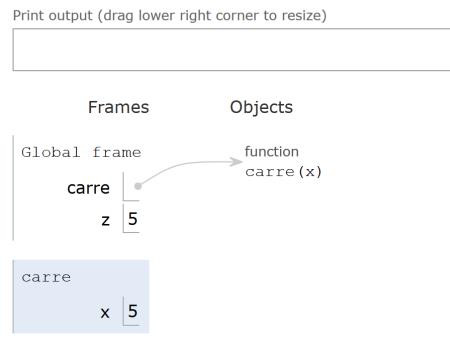
- Étape 3 : Python lit et met en mémoire la variable `z`. Celle-ci étant créée dans le programme principal, il s'agira d'une variable *globale*. Ainsi, elle sera également stockée dans le *Global frame*.

```
Python 3.6
1 # définition d'une fonction carre()
2 def carre(x):
3     y = x**2
4     return y
5
6 # programme principal
→ 7 z = 5
8 resultat = carre(z)
9 print(resultat)
```

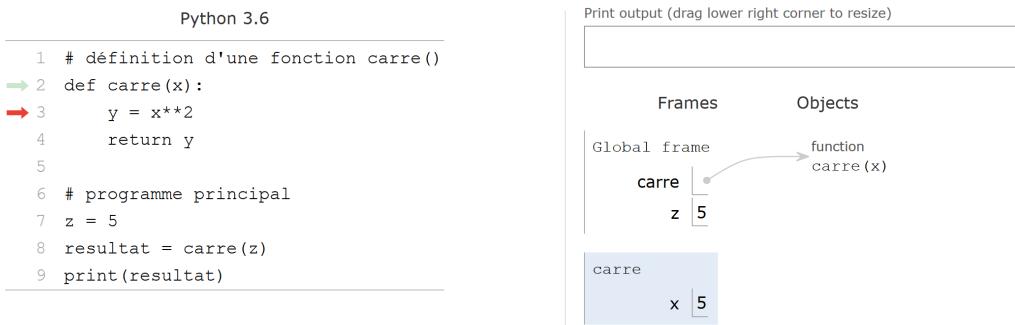
Frames Objects
Global frame
carre
z 5

- Étape 4 : La fonction `carre()` est appelée et on lui passe en argument l'entier `z`. 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.

```
Python 3.6
1 # définition d'une fonction carre()
→ 2 def carre(x):
3     y = x**2
4     return y
5
6 # programme principal
7 z = 5
→ 8 resultat = carre(z)
9 print(resultat)
```

carre
x 5

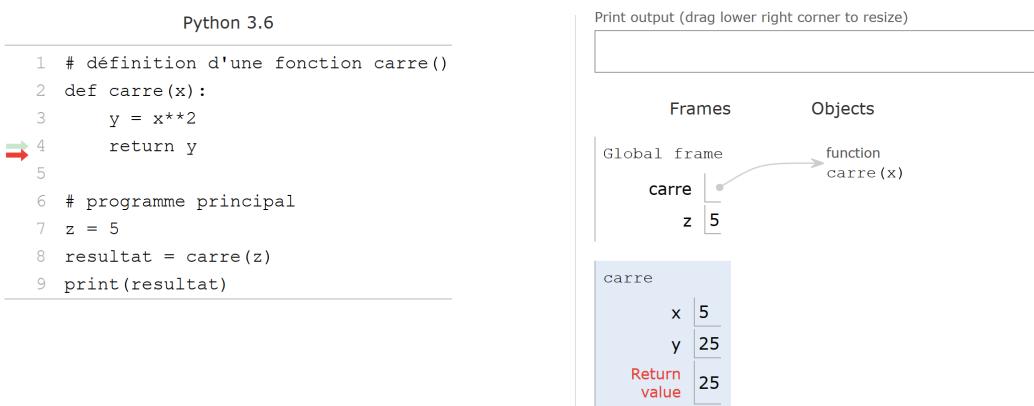
- Étape 5 : Python est maintenant prêt à exécuter chaque ligne de code de la fonction.



- Étape 6 : La variable `y` est créée dans la fonction. Celle-ci est donc stockée en tant que variable *locale* à la fonction.

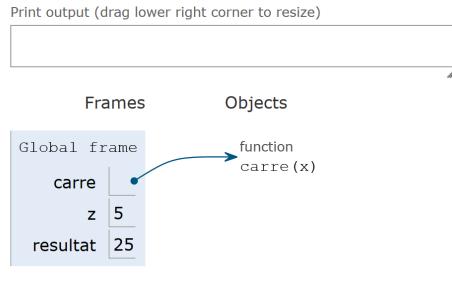


- Étape 7 : Python s'apprête à renvoyer la variable *locale* `y` au programme principal. *Python Tutor* nous indique le contenu de la valeur renvoyée.



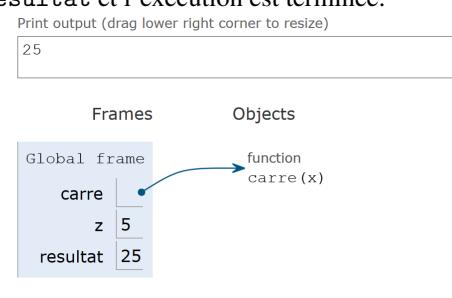
- Étape 8 : Python quitte la fonction et la valeur renvoyée par celle-ci est affectée à la variable *globale* `resultat`. Notez bien que lorsque Python quitte la fonction, **l'espace des variables alloué à la fonction est détruit**. Ainsi, toutes les variables créées dans la fonction n'existent plus. On comprend pourquoi elles portent le nom de *locales* puisqu'elles n'existent que lorsque la fonction est exécutée.

```
Python 3.6
1 # définition d'une fonction carre()
2 def carre(x):
3     y = x**2
4     return y
5
6 # programme principal
7 z = 5
8 resultat = carre(z)
9 print(resultat)
```



— Étape 9 : Python affiche le contenu de la variable `resultat` et l'exécution est terminée.

```
Python 3.6
1 # définition d'une fonction carre()
2 def carre(x):
3     y = x**2
4     return y
5
6 # programme principal
7 z = 5
8 resultat = carre(z)
9 print(resultat)
```



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 12. 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.

9.7 Exercices

Conseil : pour le premier exercice, utilisez *Python Tutor*. Pour les exercices suivants, créez des scripts puis exécutez-les dans un *shell*.

9.7.1 Carré et factorielle

Reprenez l'exemple précédent à l'aide du site *Python Tutor*² :

```
1 # définition d'une fonction carre()
2 def carre(x):
3     y = x**2
4     return y
5
6 # programme principal
7 z = 5
8 resultat = carre(z)
9 print(resultat)
```

Analysez ensuite le code suivant et tentez de prédire sa sortie :

```
1 def calc_factorielle(n):
2     fact = 1
3     for i in range(2,n+1):
4         fact = fact * i
5     return fact
6
7
8 # programme principal
9 nb = 4
10 factorielle_nb = calc_factorielle(nb)
11 print("{}! = {}".format(nb,factorielle_nb))
```

2. <http://www.pythontutor.com>

```

12| nb2 = 10
13| print("{}! = {}".format(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 ?

9.7.2 Puissance

Créez une fonction `calc_puissance(x,y)` qui renvoie x^y en utilisant l'opérateur `**`. Pour rappel :

```

1| >>> 2**2
2| 4
3| >>> 2**3
4| 8
5| >>> 2**4
6| 16

```

Dans le programme principal, calculez et affichez à l'écran 2^i avec i variant de 0 à 20 inclus. On veut que le résultat soit présenté avec le formatage suivant :

```

1| 2^ 0 =
2| 2^ 1 =
3| 2^ 2 =
4| [...]
5| 2^20 = 1048576

```

9.7.3 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.

9.7.4 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 :

```

1| 2 est premier
2| 3 est premier
3| 4 n'est pas premier
4| [...]
5| 100 n'est pas premier

```

9.7.5 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.

9.7.6 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 :

$$d = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2 + (z_B - z_A)^2}$$

9.7.7 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 exemple d'utilisation :

```

1| >>> import random
2| >>> random.uniform(1, 10)
3| 8.199672607202174
4| >>> random.uniform(1, 10)
5| 2.607528561528022
6| >>> random.uniform(1, 10)
7| 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 $[0, 10]$.

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 (min, max, et moyenne) avec deux chiffres après la virgule :

```

1| Liste 1 : min = 0.17 ; max = 99.72 ; moyenne = 57.38
2| Liste 2 : min = 1.25 ; max = 99.99 ; moyenne = 47.41
3| [...]
4| Liste 19 : min = 1.05 ; max = 99.36 ; moyenne = 49.43
5| 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 ?

9.7.8 Distance à l'origine (exercice ++)

En reprenant votre fonction de calcul de distance euclidienne en 3D `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 9.2 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 $(0, 0)$.

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 :

```

1| # création d'une image pour la visualisation du résultat
2| import matplotlib.pyplot as plt
3|
4| x = []
5| y = []
6| with open("sin2ori.dat", "r") as f_in:
7|     for line in f_in:
8|         coords = line.split()
9|         x.append(float(coords[0]))
10|        y.append(float(coords[1]))
11| plt.figure(figsize=(8,8))
12| plt.plot(x, y)
13| plt.xlabel("x")
14| plt.ylabel("Distance de sin(x) à l'origine")
15| plt.savefig("sin2ori.png")

```

Ouvrez l'image `sin2ori.png`.

Remarque

Le module `matplotlib` sera expliqué en détail dans le chapitre 17 *Quelques modules d'intérêt en bioinformatique*.

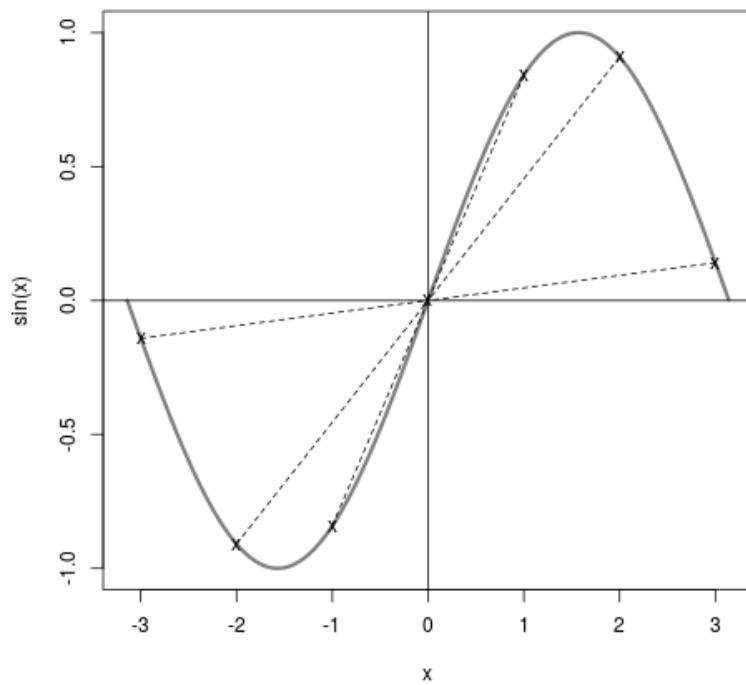


FIGURE 9.2 – Illustration de la distance à l'origine.

Chapitre 10

Plus sur les chaînes de caractères

10.1 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 caractères¹.

10.2 Chaînes de caractères et listes

Les chaînes de caractères peuvent être considérées comme des listes (de caractères) un peu particulières :

```
1 | >>> animaux = "girafe tigre"
2 | >>> animaux
3 | 'girafe tigre'
4 | >>> len(animaux)
5 | 12
6 | >>> animaux[3]
7 | 'a'
```

Nous pouvons donc utiliser certaines propriétés des listes comme les tranches :

```
1 | >>> animaux = "girafe tigre"
2 | >>> animaux[0:4]
3 | 'gira'
4 | >>> animaux[9:]
5 | 'gre'
6 | >>> animaux[:-2]
7 | 'girafe tig'
```

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 | >>> animaux = "girafe tigre"
2 | >>> animaux[4]
3 | 'f'
4 | >>> animaux[4] = "F"
5 | Traceback (most recent call last):
6 |   File "<stdin>", line 1, in <module>
7 | 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.

10.3 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

1. <https://docs.python.org/fr/3.7/library/string.html>

guillemets de déclaration de la chaîne de caractères, vous pouvez utiliser `\' ou `\" ou utiliser respectivement des guillemets doubles ou simples pour déclarer votre chaîne de caractères.

```

1 | >>> print("Un retour à la ligne\npuis une tabulation\t puis un guillemet\"")
2 | Un retour à la ligne
3 | puis une tabulation      puis un guillemet"
4 | >>> print('J\'affiche un guillemet simple')
5 | J'affiche un guillemet simple
6 | >>> print("Un brin d'ADN")
7 | Un brin d'ADN
8 | >>> print('Python est un "super" langage de programmation')
9 | 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) :

```

1 | >>> x = """souris
2 | ...   chat
3 | ...   abeille"""
4 | >>> x
5 | 'souris\nchat\nabeille'
6 | >>> print(x)
7 | souris
8 | chat
9 | abeille

```

10.4 Méthodes associées aux chaînes de caractères

Voici quelques méthodes² spécifiques aux objets de type str :

```

1 | >>> x = "girafe"
2 | >>> x.upper()
3 | 'GIRAFE'
4 | >>> x
5 | 'girafe'
6 | >>> 'TIGRE'.lower()
7 | 'tigre'

```

Les méthodes .lower() et .upper() renvoient un texte en minuscule et en majuscule respectivement. On remarque que l'utilisation de ces fonctions 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 :

```

1 | >>> x[0].upper() + x[1:]
2 | 'Girafe'

```

ou encore plus simple avec la méthode adéquate :

```

1 | >>> x.capitalize()
2 | 'Girafe'

```

Il existe une méthode associée aux chaînes de caractères qui est particulièrement pratique, la méthode .split() :

```

1 | >>> animaux = "girafe tigre singe souris"
2 | >>> animaux.split()
3 | ['girafe', 'tigre', 'singe', 'souris']
4 | >>> for animal in animaux.split():
5 | ...     print(animal)
6 |
7 | girafe
8 | tigre
9 | singe
10 | 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 blanc³ (*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 :

2. <https://docs.python.org/fr/3/library/string.html>
 3. https://en.wikipedia.org/wiki/Whitespace_character

```

1 | >>> animaux = "girafe:tigre:singe::souris"
2 | >>> animaux.split(":")
3 | ['girafe', 'tigre', 'singe', '', 'souris']

```

Attention, dans cet exemple, le séparateur est un seul caractères : (et non pas une combinaison de un ou plusieurs :) menant 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()` :

```

1 | >>> animaux = "girafe tigre singe souris"
2 | >>> animaux.split(maxsplit=1)
3 | ['girafe', 'tigre singe souris']
4 | >>> animaux.split(maxsplit=2)
5 | ['girafe', 'tigre', 'singe souris']

```

La méthode `.find()`, quant à elle, recherche une chaîne de caractères passée en argument :

```

1 | >>> animal = "girafe"
2 | >>> animal.find("i")
3 | 1
4 | >>> animal.find("afe")
5 | 3
6 | >>> animal.find("z")
7 | -1
8 | >>> animal.find("tig")
9 | -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 :

```

1 | >>> animaux = "girafe tigre"
2 | >>> animaux.find("i")
3 | 1

```

On trouve aussi la méthode `.replace()` qui substitue une chaîne de caractères par une autre :

```

1 | >>> animaux = "girafe tigre"
2 | >>> animaux.replace("tigre", "singe")
3 | 'girafe singe'
4 | >>> animaux.replace("i", "o")
5 | 'gorafe togre'

```

La méthode `.count()` compte le nombre d'occurrences d'une chaîne de caractères passée en argument :

```

1 | >>> animaux = "girafe tigre"
2 | >>> animaux.count("i")
3 | 2
4 | >>> animaux.count("z")
5 | 0
6 | >>> animaux.count("tigre")
7 | 1

```

La méthode `.startswith()` vérifie si une chaîne de caractères commence par une autre chaîne de caractères :

```

1 | >>> chaine = "Bonjour monsieur le capitaine !"
2 | >>> chaine.startswith("Bonjour")
3 | True
4 | >>> chaine.startswith("Au revoir")
5 | 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 :

```

1 | >>> chaine = " Comment enlever les espaces au début et à la fin ? "
2 | >>> chaine.strip()
3 | '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 :

```

1 | >>> chaine = " \tfonctionne avec les tabulations et les retours à la ligne\n"
2 | >>> chaine.strip()
3 | 'fonctionne avec les tabulations et les retours à la ligne'

```

La méthode `.strip()` est très pratique quand on lit un fichier et qu'on veut se débarrasser des retours à la ligne.

10.5 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'extraire des valeurs de cette chaîne de caractères puis ensuite de les manipuler.

On considère par exemple la chaîne de caractères val :

```
1 | >>> val = "3.4 17.2 atom"
```

On souhaite extraire les valeurs 3.4 et 17.2 pour ensuite les additionner.

Dans un premier temps, on découpe la chaîne de caractères avec l'instruction `.split()` :

```
1 | >>> val2 = val.split()
2 | >>> val2
3 | ['3.4', '17.2', 'atom']
```

On obtient alors une liste de chaînes de caractères. On transforme ensuite les deux premiers éléments de cette liste en *floats* (avec la fonction `float()`) pour pouvoir les additionner :

```
1 | >>> float(val2[0]) + float(val2[1])
2 | 20.59999999999998
```

10.6 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 un peu particulière puisqu'elle fait appelle à la méthode `.join()`.

```
1 | >>> seq = ["A", "T", "G", "A", "T"]
2 | >>> seq
3 | ['A', 'T', 'G', 'A', 'T']
4 | >>> "-".join(seq)
5 | 'A-T-G-A-T'
6 | >>> " ".join(seq)
7 | 'A T G A T'
8 | >>> "".join(seq)
9 | '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.

Attention, la méthode `.join()` ne s'applique qu'à une liste de chaînes de caractères.

```
1 | >>> maliste = ["A", 5, "G"]
2 | >>> ".join(maliste)
3 | Traceback (most recent call last):
4 |   File "<stdin>", line 1, in <module>
5 | TypeError: sequence item 1: expected string, 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()`.

```
1 | >>> animaux = "girafe tigre"
2 | >>> dir(animaux)
3 | ['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
4 | '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
5 | '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',
6 | '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
7 | '__rmul__', '__setattro__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
8 | 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'for-
9 | mat_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'is-
10 | identifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'is-
11 | title', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
12 | 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
13 | 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
14 | 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Pour l'instant, vous pouvez ignorer les méthodes qui commencent et qui se terminent par deux tirets bas (*underscores*) `__`.

Vous pouvez également accéder à l'aide et à la documentation d'une méthode particulière avec `help()`, par exemple pour la méthode `.split()` :

```

1 | >>> help(animaux.split)
2 | Help on built-in function split:
3 |
4 | split(...)
5 |     S.split([sep [,maxsplit]]) -> list of strings
6 |
7 |     Return a list of the words in the string S, using sep as the
8 |     delimiter string. If maxsplit is given, at most maxsplit
9 |     splits are done. If sep is not specified or is None, any
10 |    whitespace string is a separator.
11 | (END)

```

Attention à ne pas mettre les parenthèses à la suite du nom de la méthode. L'instruction correcte est `help(animaux.split)` et pas `help(animaux.split())`.

10.7 Exercices

Conseil : pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

10.7.1 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).

10.7.2 Lecture d'une séquence à partir d'un fichier FASTA

Le fichier `UBI4_SCerevisiae.fasta`⁴ 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 base 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 :

```

1 | UBI4_SCerevisiae.fasta
2 | La séquence contient WWW bases
3 | La longueur de la séquence est un multiple de 3 nucléotides
4 | La séquence possède XXX codons
5 | 10 premières bases : YYYYYYYYYY
6 | 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 rencontrés en biologie*.

10.7.3 Fréquence des bases dans une séquence d'ADN

Soit la séquence d'ADN `ATATACGGATCGGCTGTTGCCTGCGTAGCTAGCGT`. 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.

4. https://python.sdv.univ-paris-diderot.fr/data-files/UBI4_SCerevisiae.fasta

10.7.4 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 1 lettre.

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
Alanine	Ala	A
Arginine	Arg	R
Asparagine	Asn	N
Aspartate	Asp	D
Cystéine	Cys	C
Glutamate	Glu	E
Glutamine	Gln	Q
Glycine	Gly	G
Histidine	His	H
Isoleucine	Ile	I
Leucine	Leu	L
Lysine	Lys	K
Méthionine	Met	M
Phénylalanine	Phe	F
Proline	Pro	P
Sérine	Ser	S
Thrénanine	Thr	T
Tryptophane	Trp	W
Tyrosine	Tyr	Y
Valine	Val	V

10.7.5 Distance de Hamming

La distance de Hamming⁵ 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

AGWPSSGGASAGLAIL et IGWPSAGASAGLWIL

puis entre les séquences

ATTCCATACGTTACGATT et ATACTTACGTAACCATT.

10.7.6 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 `test_palindrome()` qui prend en argument une chaîne de caractères et qui affiche `xxx est un palindrome` si la chaîne de caractères `xxx` passée en argument est un palindrome ou `xxx n'est pas un palindrome` sinon. Pensez à vous débarrasser au préalable des majuscules 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

5. http://en.wikipedia.org/wiki/Hamming_distance

10.7.7 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 ».

Écrivez la fonction `test_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 affiche `Le mot xxx est composable à partir de yyy` si le mot (xxx) est composable à partir de la séquence de lettres (yyy) ou `Le mot xxx n'est pas composable à partir de yyy` sinon.

Testez cette fonction avec les mots et les séquences suivantes :

Mot	Séquence
python	aophrtkny
python	aeiouyhpq
coucou	uocuoceokzezh
fonction	nhwfnitvkloco

10.7.8 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 pangramme⁶ 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 `pangramme()` qui utilise la fonction `get_alphabet()` précédente, qui prend en argument une chaîne de caractères (xxx) et qui renvoie `xxx est un pangramme` si cette chaîne de caractères est un pangramme ou `xxx n'est pas un pangramme` sinon. 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

10.7.9 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.gbk⁷.

Le format GenBank est présenté en détails dans l'annexe A *Quelques formats de données rencontrés 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 `//` :

```

1 ORIGIN
2      1 ccacaccaca cccacacacc cacacaccac accacacacacc cacacacaca
3          61 catcctaaca ctaccctaa acagccctaa tctaaccctg gccaacctgt ctctcaactt
4 [...]
5      230101 ttttagtgtt agtattaggg tgtggtgtgt gggtgtggg tgggtgtggg tgtgggtgt
6      230161 ggtgtgggtg tgggtgtgt gtgggtgtgt ggtgtgggtg ggggtgtggg tgtgtggg
7 //

```

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.

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 :

6. <http://fr.wikipedia.org/wiki/Pangramme>

7. https://python.sdv.univ-paris-diderot.fr/data-files/NC_001133.gbk

```

1| NC_001133.gbk
2| La séquence contient XXX bases
3| 10 premières bases : YYYYYYYYYY
4| 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 :

```

1| drapeau <- Faux
2| seq <- chaîne de caractères vide
3| Lire toutes les lignes du fichier:
4|   si la ligne contient //:
5|     drapeau <- Faux
6|   si drapeau est Vrai:
7|     on ajoute à seq la ligne (sans espaces, chiffres et retour à la ligne)
8|   si la ligne contient ORIGIN
9|     drapeau <- Vrai

```

10.7.10 Affichage des carbones alpha d'une structure de protéine

Téléchargez le fichier 1bta.pdb⁸ qui correspond à la structure tridimensionnelle de la protéine barstar⁹ 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 rencontrés en biologie*.

10.7.11 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 2 caractères. La distance sera affichée avec deux chiffres après la virgule. Voici un exemple avec les premiers carbones alpha :

```

1| 1 2 3.80
2| 2 3 3.80
3| 3 4 3.83
4| 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 rencontrés en biologie*.

8. <https://files.rcsb.org/download/1BTA.pdb>
 9. <http://www.rcsb.org/pdb/explore.do?structureId=1BTA>

Chapitre 11

Plus sur les listes

11.1 Méthodes associées aux listes

Comme pour les chaînes de caractères, les listes possèdent de nombreuses **méthodes** qui leur sont propres et qui peuvent se révéler très pratiques. On rappelle qu'une méthode est une fonction qui agit sur l'objet auquel elle est attachée par un point.

11.1.1 .append()

La méthode `.append()`, que l'on a déjà vu au chapitre 4 *Listes*, ajoute un élément à la fin d'une liste :

```
1 | >>> a = [1, 2, 3]
2 | >>> a.append(5)
3 | >>> a
4 | [1, 2, 3, 5]
```

qui est équivalent à :

```
1 | >>> a = [1, 2, 3]
2 | >>> a = a + [5]
3 | >>> a
4 | [1, 2, 3, 5]
```

Conseil : préférez la version avec `.append()` qui est plus compacte et facile à lire.

11.1.2 .insert()

La méthode `.insert()` insère un objet dans une liste avec un indice déterminé :

```
1 | >>> a.insert(2, -15)
2 | >>> a
3 | [1, 2, -15, 3, 5]
```

11.1.3 del

L'instruction `del` supprime un élément d'une liste à un indice déterminé :

```
1 | >>> del a[1]
2 | >>> a
3 | [1, -15, 3, 5]
```

Remarque

Contrairement aux autres méthodes associées aux listes, `del` est une instruction générale de Python, utilisable pour d'autres objets que des listes. Celle-ci ne prend pas de parenthèse.

11.1.4 .remove()

La méthode `.remove()` supprime un élément d'une liste à partir de sa valeur :

```
1 | >>> a.remove(5)
2 | >>> a
3 | [1, -15, 3]
```

11.1.5 .sort()

La méthode `.sort()` trie une liste :

```
1 | >>> a.sort()
2 | >>> a
3 | [-15, 1, 3]
```

11.1.6 .reverse()

La méthode `.reverse()` inverse une liste :

```
1 | >>> a.reverse()
2 | >>> a
3 | [3, 1, -15]
```

11.1.7 .count()

La méthode `.count()` compte le nombre d'éléments (passés en argument) dans une liste :

```
1 | >>> a=[1, 2, 4, 3, 1, 1]
2 | >>> a.count(1)
3 | 3
4 | >>> a.count(4)
5 | 1
6 | >>> a.count(23)
7 | 0
```

11.1.8 Actions sur les listes

De nombreux méthodes ci-dessus (`.append()`, `.sort()`, etc.) modifient la liste mais ne renvoient rien, c'est-à-dire qu'elles 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.

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(ma_liste)` (`ma_liste` étant une liste).

11.2 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 de la forme `maliste = []`. Voici un exemple où une chaîne de caractères est convertie en liste :

```
1 | >>> seq = "CAAAGGTAAACGC"
2 | >>> seq_list = []
3 | >>> seq_list
4 | []
5 | >>> for base in seq:
6 | ...     seq_list.append(base)
7 | ...
8 | >>> seq_list
9 | ['C', 'A', 'A', 'A', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
```

Remarquez que dans cet exemple, vous pouvez directement utiliser la fonction `list()` qui prend n'importe quel objet séquentiel (liste, chaîne de caractères, etc.) et qui renvoie une liste :



FIGURE 11.1 – Copie de liste.

```

1 | >>> seq = "CAAAGGTAAACGC"
2 | >>> list(seq)
3 | ['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. On rappelle 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`.

11.3 Test d'appartenance

L'opérateur `in` teste si un élément fait partie d'une liste.

```

1 | liste = [1, 3, 5, 7, 9]
2 | >>> 3 in liste
3 | True
4 | >>> 4 in liste
5 | False
6 | >>> 3 not in liste
7 | False
8 | >>> 4 not in liste
9 | True

```

La variation avec `not` permet, *a contrario*, de vérifier qu'un élément n'est pas dans une liste.

11.4 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** :

```

1 | >>> x = [1, 2, 3]
2 | >>> y = x
3 | >>> y
4 | [1, 2, 3]
5 | >>> x[1] = -15
6 | >>> x
7 | [1, -15, 3]
8 | >>> y
9 | [1, -15, 3]

```

Vous voyez que la modification de `x` modifie `y` aussi ! Pour comprendre ce qui se passe nous allons de nouveau utiliser le site *Python Tutor* avec cet exemple (Figure 11.1) :

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 `x` et `y` et qui pointent vers la même liste. Donc, si on modifie la liste `x`, la liste `y` 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 :

```

1 | >>> x = [1, 2, 3]
2 | >>> y = x[:]
3 | >>> x[1] = -15

```

Python 3.6

```

1 x = [1, 2, 3]
2 y = x[:]
3 z = list(x)
4 x[1] = -15
5 print(y)
6 print(z)

```

[Edit code](#) | [Live programming](#)

Breakpoint; use the Back and Forward buttons to jump there.

< Back Step 5 of 6 Forward > Last >>

Print output (drag lower right corner to resize)

Frames Objects

Global frame

x	list	0	1	-15	3
y	list	0	1	2	3
z	list	0	1	2	3

FIGURE 11.2 – Copie de liste avec une tranche [:] et la fonction list().

```

4 >>> y
5 [1, 2, 3]

```

L'instruction `x[:]` a créé une copie « à la volée » de la liste `x`. Vous pouvez utiliser aussi la fonction `list()` qui renvoie explicitement une liste :

```

1 >>> x = [1, 2, 3]
2 >>> y = list(x)
3 >>> x[1] = -15
4 >>> y
5 [1, 2, 3]

```

Si on regarde à nouveau dans *Python Tutor* (Figure 11.2), on voit clairement que l'utilisation des tranches `[:]` ou de la fonction `list()` crée des copies explicites. Chaque flèche pointe vers une liste différente, indépendante des autres.

Attention, les deux techniques 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 :

```

1 >>> x = [[1, 2], [3, 4]]
2 >>> x
3 [[1, 2], [3, 4]]
4 >>> y = x[:]
5 >>> x[1][1] = 55
6 >>> x
7 [[1, 2], [3, 55]]
8 >>> y
9 [[1, 2], [3, 55]]

```

et

```

1 >>> y = list(x)
2 >>> x[1][1] = 77
3 >>> x
4 [[1, 2], [3, 77]]
5 >>> y
6 [[1, 2], [3, 77]]

```

La méthode de copie qui **fonctionne à tous les coups** consiste à appeler la fonction `deepcopy()` du module `copy`.

```

1 >>> import copy
2 >>> x = [[1, 2], [3, 4]]
3 >>> x
4 [[1, 2], [3, 4]]
5 >>> y = copy.deepcopy(x)
6 >>> x[1][1] = 99
7 >>> x
8 [[1, 2], [3, 99]]
9 >>> y
10 [[1, 2], [3, 4]]

```

11.5 Exercices

Conseil : pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

11.5.1 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.

11.5.2 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 liste de bases. Utilisez la méthode `.append()` pour ajouter les différentes bases à la liste et la fonction `random.choice()` du module `random` pour choisir une base parmi les 4 possibles.

Utilisez cette fonction pour générer aléatoirement une séquence d'ADN de 15 bases.

11.5.3 Séquence d'ADN complémentaire inverse

Créez une fonction `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.

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.

11.5.4 Doublons

Soit la liste de nombres [5, 1, 1, 2, 5, 6, 3, 4, 4, 4, 2].

Retirez les doublons de cette liste, triez-la et affichez-la.

11.5.5 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 4 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 liste.

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 : la fonction `random.shuffle()` du module `random` vous sera utile.

11.5.6 Le nombre mystère

Trouvez le nombre mystère qui répond aux conditions suivantes :

- Il est composé de 3 chiffres.
- Il est strictement inférieur à 300.
- Il est pair.
- Deux de ses chiffres sont identiques.
- 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.

11.5.7 Triangle de Pascal (exercice +++)

Voici le début du triangle de Pascal :

```
1 | 1
2 | 1 1
3 | 1 2 1
4 | 1 3 3 1
5 | 1 4 6 4 1
6 | 1 5 10 10 5 1
7 | [...]
```

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 12

Plus sur les fonctions

Avant d'aborder ce chapitre, nous vous conseillons de relire le chapitre 9 *Fonctions* et de bien en assimiler toutes les notions (et aussi d'en faire les exercices). Nous avons vu dans ce chapitre 9 le concept puissant et 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.

12.1 Appel d'une fonction dans une fonction

Dans le chapitre 9 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 polynome(x):
3 |     return (x**2 - 2*x + 1)
4 |
5 | def calc_vals(debut, fin):
6 |     liste_vals = []
7 |     for x in range(debut, fin + 1):
8 |         liste_vals.append(polynome(x))
9 |
10|    return liste_vals
11|
12| # programme principal
12| print(calc_vals(-5,5))
```

Nous appelons depuis le programme principal la fonction `calc_vals()`, puis à l'intérieur de celle-ci nous appelons l'autre fonction `polynome()`. Regardons ce que *Python Tutor* nous montre lorsque la fonction `polynome()` est exécutée (Figure 12.1) :

L'espace mémoire alloué à `polynome()` est grisé, indiquant que cette fonction est en cours d'exécution. La fonction appelante `calc_vals()` 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 `polynome()`, et on pourra ainsi noter que ses variables *locales* (`debut`, `fin`, `liste_vals` et `x`) 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_vals()` ne seront détruites que lorsque la boucle sera terminée et que la liste `liste_vals` sera retournée au programme principal. Enfin, notez bien que la fonction `calc_vals()` appelle la fonction `polynome()` 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 !

1. <http://www.pythontutor.com/>

The screenshot shows the Python Tutor interface. On the left, a code editor displays a Python script with a recursive factorial function:

```

1 def polynome(x):
2     return (x**2 - 2*x + 1)
3
4 def calc_vals(debut, fin):
5     liste_vals = []
6     for x in range(debut, fin + 1):
7         liste_vals.append(polynome(x))
8     return liste_vals
9
10 # programme principal
11 print(calc_vals(-5, 5))

```

Below the code editor is a message: "at has just executed". A toolbar below includes buttons for "Edit this code", "Step 15 of 63", and navigation arrows. To the right is a detailed call graph and variable state visualization.

Frames (Global frame):

- polynome: function
- calc_vals: function

Objects:

- calc_vals: list [0, 36]
- debut: -5
- fin: 5
- liste_vals: list [-4]
- x: -4
- polynome: x: -4, Return value: 25

FIGURE 12.1 – Appel d'une fonction dans une fonction.

12.2 Fonctions récursives

Conseil : pour les débutants, vous pouvez passer 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 rapide² (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 factorielle s'écrit avec ! et produit les résultats suivants :

$$\begin{aligned} 3! &= 3 \times 2 \times 1 = 6 \\ 4! &= 4 \times 3 \times 2 \times 1 = 30 \\ n! &= n \times n-1 \times \dots \times 2 \times 1 \end{aligned}$$

Si nous essayons de coder cette fonction mathématique en Python, voici ce que nous pourrions écrire :

```

1 def calc_factorielle(nb):
2     if nb == 1:
3         return 1
4     else:
5         return nb * calc_factorielle(nb - 1)
6
7 # prog principal
8 print(calc_factorielle(4))

```

Pas très facile à comprendre, n'est-ce pas ? À nouveau, demandons l'aide à *Python Tutor* pour visualiser ce qui se passe (nous vous conseillons bien sûr de tester vous-même cet exemple) :

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 `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 12.2). 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.

2. https://fr.wikipedia.org/wiki/Tri_rapide

The screenshot shows a Python 3.6 debugger interface. On the left, the code for a recursive factorial function is displayed:

```

1 def calc_factorielle(nb):
2     if nb == 1:
3         return 1
4     else:
5         return nb * calc_factorielle(nb - 1)
6
7 # prog principal
8 print(calc_factorielle(4))

```

Below the code, there is a link to "Edit this code". A note says "that has just executed" followed by "the line to execute". It also says "Line of code to set a breakpoint; use the Back and Forward buttons to jump there." Below the code are navigation buttons: "<< First", "< Back", "Step 15 of 18", "Forward >", and "Last >>".

On the right, a "Print output" window is shown with a single line of text: "1". Above it, a "Frames" section shows the call stack:

- Global frame: calc_factorielle (nb=4)
- calc_factorielle (nb=3)
- calc_factorielle (nb=2)
- calc_factorielle (nb=1, Return value=1)

Below the frames, there are two buttons: "I just cleared up a misunderstanding!" and "I just fixed a bug in my code!".

FIGURE 12.2 – Fonction récursive : factorielle.

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.

12.3 Portée des variables

Il est très important, lorsque l'on manipule des fonctions, de connaître la portée des variables. 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 :

```

1 >>> def ma_fonction():
2 ...     x = 2
3 ...     print('x vaut {} dans la fonction'.format(x))
4 ...
5 >>> ma_fonction()
6 x vaut 2 dans la fonction
7 >>> print(x)
8 Traceback (most recent call last):
9   File "<stdin>", line 1, in <module>
10  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 :

```

1 >>> def ma_fonction(x):
2 ...     print('x vaut {} dans la fonction'.format(x))
3 ...
4 >>> ma_fonction(2)
5 x vaut 2 dans la fonction
6 >>> print(x)
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in ?
9  NameError: name 'x' is not defined

```

Lorsqu'une variable est déclarée à la racine du module (c'est aussi comme cela que l'on appelle un programme Python), elle est visible dans tout le module. On a vu qu'on parlait de variable **globale** :

```

1 | >>> def ma_fonction():
2 | ...     print(x)
3 | ...
4 | >>> x = 3
5 | >>> ma_fonction()
6 | 3
7 | >>> print(x)
8 | 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 :

```

1 | >>> def ma_fonction():
2 | ...     x = x + 1
3 | ...
4 | >>> x = 1
5 | >>> ma_fonction()
6 | Traceback (most recent call last):
7 | File "<stdin>", line 1, in <module>
8 |   File "<stdin>", line 2, in fct
9 | UnboundLocalError: local variable 'x' referenced before assignment

```

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` :

```

1 | >>> def ma_fonction():
2 | ...     global x
3 | ...     x = x + 1
4 | ...
5 | >>> x = 1
6 | >>> ma_fonction()
7 | >>> x
8 | 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.

12.4 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 :

```

1 | >>> def ma_fonction():
2 | ...     liste[1] = -127
3 | ...
4 | >>> liste = [1,2,3]
5 | >>> ma_fonction()
6 | >>> liste
7 | [1, -127, 3]

```

De même, si vous passez une liste en argument, elle est modifiable au sein de la fonction :

```

1 | >>> def ma_fonction(x):
2 | ...     x[1] = -15
3 | ...
4 | >>> y = [1,2,3]
5 | >>> ma_fonction(y)
6 | >>> y
7 | [1, -15, 3]

```

Si vous voulez éviter ce problème, utilisez des tuples (ils seront présentés dans le chapitre 13 *Dictionnaires et tuples*), Python renverra une erreur puisque 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.

```

1 | >>> def ma_fonction(x):
2 | ...     x[1] = -15
3 | ...

```

```

4| >>> y = [1, 2, 3]
5| >>> ma_fonction(y[:])
6| >>> y
7| [1, 2, 3]
8| >>> ma_fonction(list(y))
9| >>> y
10| [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.

12.5 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 cette règle la règle **LGI** pour locale, globale, interne. En voici un exemple :

```

1| >>> def ma_fonction():
2| ...     x = 4
3| ...     print('Dans la fonction x vaut {}'.format(x))
4| ...
5| >>> x = -15
6| >>> ma_fonction()
7| Dans la fonction x vaut 4
8| >>> print('Dans le module principal x vaut {}'.format(x))
9| 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 peut reconnaître une variable ayant le même nom que ses propres fonctions ou variables internes, évitez de les utiliser 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 ». Nous en parlerons plus longuement dans le chapitre 19 *Avoir la classe avec les objets*.

12.6 Recommandations

Dans ce chapitre nous avons *joué* avec les fonctions (et les listes) afin de vous montrer comment Python réagit. 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 raisons³ 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 19 *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 :

Plutôt que d'utiliser des variables globales, passez vos variables explicitement aux fonctions comme des argument(s).

Vous connaissez maintenant les fonctions sous tous leurs angles. Comme indiqué en introduction du chapitre 9, 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 également la décomposer en d'autres fonctions.

3. <http://wiki.c2.com/?GlobalVariablesAreBad>

- Au risque de nous répéter, forcez-vous à utiliser des fonctions en permanence. Pratiquez, pratiquez... et pratiquez encore !

12.7 Exercices

Conseil : pour le second exercice, créez un script puis exécutez-le dans un *shell*.

12.7.1 Prédire la sortie

Prédez le comportement des codes suivant, sans les recopier dans un script ni dans l’interpréteur Python :

Code 1

```

1 def hello(prenom):
2     print("Bonjour {}".format(prenom))
3
4
5 # Programme principal.
6 hello("Patrick")
7 print(x)

```

Code 2

```

1 def hello(prenom):
2     print("Bonjour {}".format(prenom))
3
4
5 # Programme principal.
6 x = 10
7 hello("Patrick")
8 print(x)

```

Code 3

```

1 def hello(prenom):
2     print("Bonjour {}".format(prenom))
3     print(x)
4
5
6 # Programme principal.
7 x = 10
8 hello("Patrick")
9 print(x)

```

Code 4

```

1 def hello(prenom):
2     x = 42
3     print("Bonjour {}".format(prenom))
4     print(x)
5
6
7 # Programme principal.
8 x = 10
9 hello("Patrick")
10 print(x)

```

12.7.2 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, on effectuera 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 13

Dictionnaires et tuples

13.1 Dictionnaires

Les **dictionnaires** se révèlent très pratiques lorsque vous devez manipuler des structures complexes à décrire et que les listes présentent leurs limites. Les dictionnaires sont des collections non ordonnées d'objets, c'est-à-dire qu'il n'y a pas de notion d'ordre (*i.e.* pas d'indice). On accède aux **valeurs** d'un dictionnaire par des **clés**. Ceci semble un peu confus ? Regardez l'exemple suivant :

```
1 | >>> ani1 = {}
2 | >>> ani1["nom"] = "girafe"
3 | >>> ani1["taille"] = 5.0
4 | >>> ani1["poids"] = 1100
5 | >>> ani1
6 | {'nom': 'girafe', 'taille': 5.0, 'poids': 1100}
```

En premier, on définit un dictionnaire vide avec les accolades {} (tout comme on peut le faire pour les listes avec []). Ensuite, on remplit le dictionnaire avec différentes clés ("nom", "taille", "poids") auxquelles on affecte des valeurs ("girafe", 5.0, 1100). Vous pouvez mettre autant de clés que vous voulez dans un dictionnaire (tout comme vous pouvez ajouter autant d'éléments que vous voulez dans une liste).

Remarque

Un dictionnaire est affiché sans ordre particulier.

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

```
1 | >>> ani2 = {"nom": "singe", "poids": 70, "taille": 1.75}
```

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

```
1 | >>> ani2["age"] = 15
```

Pour récupérer la valeur associée à une clé donnée, il suffit d'utiliser la syntaxe suivante `dictionnaire["cle"]`. Par exemple :

```
1 | >>> ani1["taille"]
2 | 5.0
```

Remarque

Toutes les clés de dictionnaire utilisées jusqu'à présent étaient des chaînes de caractères. Rien n'empêche d'utiliser d'autres types d'objets comme des entiers ou des *floats*, cela peut parfois s'avérer très utile.

Néanmoins, nous vous conseillons, autant que possible, d'utiliser systématiquement des chaînes de caractères pour vos clés de dictionnaire.

13.1.1 Itération sur les clés pour obtenir les valeurs

Il est possible d'obtenir toutes les valeurs d'un dictionnaire à partir de ses clés :

```

1 | >>> ani2 = {'nom':'singe', 'poids':70, 'taille':1.75}
2 | >>> for key in ani2:
3 | ...     print(key, ani2[key])
4 |
5 | poids 70
6 | nom singe
7 | taille 1.75

```

13.1.2 Méthodes .keys() et .values()

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

```

1 | >>> ani2.keys()
2 | dict_keys(['poids', 'nom', 'taille'])
3 | >>> ani2.values()
4 | dict_values([70, 'singe', 1.75])

```

Les mentions `dict_keys` et `dict_values` indiquent que nous avons affaire à des objets un peu particuliers. Si besoin, nous pouvons les transformer en liste avec la fonction `list()` :

```

1 | >>> ani2.values()
2 | dict_values(['singe', 70, 1.75])
3 | >>> list(ani2.values())
4 | ['singe', 70, 1.75]

```

13.1.3 Existence d'une clé

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

```

1 | >>> if "poids" in ani2:
2 | ...     print("La clé 'poids' existe pour ani2")
3 |
4 | La clé 'poids' existe pour ani2
5 | >>> if "age" in ani2:
6 | ...     print("La clé 'age' existe pour ani2")
7 |
8 | >>>

```

Dans le second test (lignes 5 à 7), le message n'est pas affiché car la clé `age` n'est pas présente dans le dictionnaire `ani2`.

13.1.4 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 :

```

1 | >>> animaux = [ani1, ani2]
2 | >>> animaux
3 | [{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}, {'nom': 'singe',
4 | 'poids': 70, 'taille': 1.75}]
5 | >>>
6 | >>> for ani in animaux:
7 | ...     print(ani['nom'])
8 |
9 | girafe
10 | singe

```

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

13.2 Tuples

Les **tuples** correspondent aux listes à la différence qu'ils sont **non modifiables**. On a vu dans le chapitre 11 *Plus sur les listes* que les listes pouvaient être modifiées par références, notamment lors de la copie de listes. Les tuples s'affranchissent de ce problème puisqu'ils sont non modifiables. Pratiquement, ils utilisent les parenthèses au lieu des crochets :

```

1 | >>> x = (1, 2, 3)
2 | >>> x
3 | (1, 2, 3)
4 | >>> x[2]
5 | 3
6 | >>> x[0:2]
7 | (1, 2)
8 | >>> x[2] = 15
9 | Traceback (innermost last):
10| File "<stdin>", line 1, in ?
11| TypeError: object doesn't support item assignment

```

L'affectation et l'indication fonctionnent comme avec les listes. Mais si on essaie de modifier un des éléments du tuple, Python renvoie un message d'erreur. Si vous voulez ajouter un élément (ou le modifier), vous devez créer un autre tuple :

```

1 | >>> x = (1, 2, 3)
2 | >>> x + (2,)
3 | (1, 2, 3, 2)

```

Remarque

Pour utiliser un tuple d'un seul élément, vous devez utiliser une syntaxe avec une virgule (`element,`), ceci pour éviter une ambiguïté avec une simple expression.

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

```

1 | >>> x = (1, 2, 3)
2 | >>> x
3 | (1, 2, 3)
4 | >>> x = 1, 2, 3
5 | >>> x
6 | (1, 2, 3)

```

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

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 séquentiel et renvoie le tuple correspondant (opération de *casting*) :

```

1 | >>> tuple([1, 2, 3])
2 | (1, 2, 3)
3 | >>> tuple("ATGCCGGCAT")
4 | ('A', 'T', 'G', 'C', 'C', 'G', 'C', 'G', 'A', 'T')

```

Remarque

Les listes, les dictionnaires et les tuples sont des objets qui peuvent contenir des collections d'autres objets. On peut donc construire des listes qui contiennent des dictionnaires, des tuples ou d'autres listes, mais aussi des dictionnaires contenant des tuples, des listes, etc.

Pratiquement, nous avons déjà croisé les tuples avec la fonction `enumerate()` dans le chapitre 5 *Boucles et comparaisons* qui renvoie l'indice de l'élément et l'élément d'une liste, ainsi dans le chapitre 9 *Fonctions* lorsqu'on voulait qu'une fonction renvoie plusieurs valeurs (par exemple dans l'instruction `return x, y`, le couple `x, y` est un tuple). Cela revenait à faire une affectation multiple du type `x, y = 1, 2`.

13.3 Exercices

Conseil : pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

13.3.1 Composition en acides aminés

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

13.3.2 Mots de 2 et 3 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 2 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 3 lettres.

Utilisez ces fonctions pour affichez les mots de 2 et 3 lettres et leurs occurrences trouvés dans la séquence d'ADN :

```
ACCTAGCCATGTAGAATGCCCTAGGCTTAGCTAGCTAGCTAGCTG
```

Voici un exemple de sortie attendue :

```
1 Mots de 2 lettres
2 AC : 1
3 CC : 3
4 CT : 8
5 [...]
6 Mots de 3 lettres
7 ACC : 1
8 CCT : 2
9 CTA : 5
10 [...]
```

13.3.3 Mots de 2 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. N'hésitez pas à vous inspirer 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 2 lettres et leurs occurrences dans la séquence du chromosome I de la levure du boulanger *Saccharomyces cerevisiae* (fichier NC_001133.fna¹).

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 rencontrés en biologie*.

13.3.4 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.fna²)
- le génome de la bactérie *Escherichia coli* (fichier NC_000913.fna³)

Les deux fichiers sont au format FASTA.

Cette méthode vous paraît-elle efficace sur un génome assez gros comme celui d'*Escherichia coli* ?

13.3.5 Atomes carbone alpha d'un fichier PDB

Téléchargez le fichier 1bta.pdb⁴ qui correspond à la structure tridimensionnelle de la protéine barstar⁵ 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 :

1. https://python.sdv.univ-paris-diderot.fr/data-files/NC_001133.fna
 2. https://python.sdv.univ-paris-diderot.fr/data-files/NC_001133.fna
 3. https://python.sdv.univ-paris-diderot.fr/data-files/NC_000913.fna
 4. <https://files.rcsb.org/download/1BTA.pdb>
 5. <http://www.rcsb.org/pdb/explore.do?structureId=1BTA>

- 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 : 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 rencontrés en biologie*.

13.3.6 Barycentre d'une protéine (exercice ++)

Téléchargez le fichier `1bta.pdb`⁶ qui correspond à la structure tridimensionnelle de la protéine barstar⁷ sur le site de la *Protein Data Bank* (PDB).

Un atome de 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 :

$$G_x = \frac{1}{n} \sum_{i=1}^n \text{CA}_{i,x}$$

$$G_y = \frac{1}{n} \sum_{i=1}^n \text{CA}_{i,y}$$

$$G_z = \frac{1}{n} \sum_{i=1}^n \text{CA}_{i,z}$$

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.

6. <https://files.rcsb.org/download/1BTA.pdb>

7. <http://www.rcsb.org/pdb/explore.do?structureId=1BTA>

Chapitre 14

Création de modules

14.1 Pourquoi créer ses propres modules ?

Dans le chapitre 8 *Modules*, nous avons découvert quelques modules existants dans Python comme *random*, *math*, etc. Nous avons vu par ailleurs dans les chapitres 9 *Fonctions* et 12 *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 bien écrite pourrait être judicieusement réutilisée dans un autre programme Python. C'est justement l'intérêt de créer un module. On y met 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.

14.2 Crédit d'un module

En Python, la création d'un module est très simple. Il suffit d'écrire un ensemble de fonctions (et/ou 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` :

```
1 """Module inutile qui affiche des messages :-)."""
2 DATE = 16092008
3
4
5 def bonjour(nom):
6     """Dit Bonjour."""
7     return "Bonjour " + nom
8
9
10 def ciao(nom):
11     """Dit Ciao."""
12     return "Ciao " + nom
13
14
15 def hello(nom):
16     """Dit Hello."""
17     return "Hello " + nom
18
```

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).

14.3 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`. 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` et pourtant on ne la précise pas lorsqu'on importe le module. Ensuite, on peut utiliser les fonctions comme avec un module classique.

```
1 >>> import message
2 >>> message.hello("Joe")
3 'Hello Joe'
4 >>> message.ciao("Bill")
5 'Ciao Bill'
6 >>> message.bonjour("Monsieur")
7 'Bonjour Monsieur'
8 >>> message.DATE
9 16092008
```

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 bytecode¹, c'est-à-dire le code précompilé du module.

14.4 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*. Ces *docstrings* permettent notamment de fournir de l'aide lorsqu'on invoque la commande `help()` :

```
1 >>> help(message)
2 Help on module message:
3 
4 NAME
5     message - Module inutile qui affiche des messages :-).
6 
7 FUNCTIONS
8     bonjour(nom)
9         Dit Bonjour.
10 
11    ciao(nom)
12        Dit Ciao.
13 
14    hello(nom)
15        Dit Hello.
16 
17 DATA
18     DATE = 16092008
```

1. <https://docs.python.org/fr/3/glossary.html>

```

20 | FILE
21 | /home/pierre/message.py
22 |

```

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 :

```

1 | >>> help(message.ciao)
2 | Help on function ciao in module message:
3 | 
4 |     ciao(nom)
5 |         Dit Ciao.
6 |

```

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 15 *Bonnes pratiques en programmation Python*.

14.5 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 19 *Avoir la classe avec les objets*.

14.6 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 ne renverrait rien :

```

1 | $ python message.py
2 | $

```

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 alors 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 "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 (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 "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 :

```

1 | $ python message2.py
2 | 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 | >>>

```

À nouveau, ce comportement est possible grâce à la gestion des espaces de noms par Python (pour plus détails, consultez le chapitre 19 *Avoir la classe avec les objets*).

Au delà de la commodité de pouvoir utiliser votre script en tant que programme ou en tant que module, cela présente l'avantage de bien voir où se situe le programme principal quand on lit le code. Ainsi, plus besoin d'ajouter un commentaire `# programme principal` comme nous vous l'avions suggéré dans les chapitres 9 *Fonctions* et 12 *Plus sur les fonctions*. L'utilisation de la ligne `if __name__ == "__main__":` est une bonne pratique que nous vous recommandons !

14.7 Exercice

14.7.1 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 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 ($A \rightarrow T$, $T \rightarrow C$, $G \rightarrow C$ et $C \rightarrow G$). 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.

Conseils :

- 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 11 *Plus sur les listes*.
- Voici un exemple de fichier FASTA `adn.fasta`² pour tester la fonction `lit_fasta()`.

2. <https://python.sdv.univ-paris-diderot.fr/data-files/adn.fasta>

Chapitre 15

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 1 mois, 6 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 19 *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.

15.1 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.

15.1.1 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. Dans la PEP 8, la recommandation pour la syntaxe de chaque niveau d'indentation est très simple : 4 espaces. N'utilisez pas autre chose, c'est le meilleur compromis.

1. <https://www.python.org/dev/peps/>

2. <https://www.python.org/dev/peps/pep-0008/>

Attention

Afin de toujours utiliser cette règle des 4 espaces pour l'indentation, il est essentiel de régler correctement votre éditeur de texte. Consultez pour cela l'annexe *Installation de Python* disponible en ligne³). Avant d'écrire la moindre ligne de code, faites en sorte que lorsque vous pressez la touche tabulation, cela ajoute 4 espaces (et non pas un caractère tabulation).

15.1.2 Importation des modules

Comme on l'a vu au chapitre 8 *Modules*, le chargement d'un module se fait 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 à 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 » (voir chapitre 19 *Avoir la classe avec les objets*) spécifique au module.

Dans un script Python, on met en général 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).

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 :

```

1 import module_interne_1
2 import module_interne_2
3 from module_interne_3 import fonction_spécifique
4 from module_interne_4 import constante_1, fonction_1, fonction_2
5
6 import module_externe_1
7 import module_externe_2
8 import module_externe_3_qui_a_un_nom_long as mod3

```

15.1.3 Règles de nommage

Les noms de variables, de fonctions et de modules doivent être de la forme :

```

1 ma_variable
2 fonction_test_27()
3 mon_module

```

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 :

```

1 MA_CONSTANTE
2 VITESSE_LUMIERE

```

Les noms de classes (chapitre 19) et les exceptions (chapitre 21) sont de la forme :

```

1 MaClasse
2 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.

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 boucles et les indices :

³. <https://python.sdv.univ-paris-diderot.fr/livre-dunod>

```

1| >>> ma_liste = [1, 3, 5, 7, 9, 11]
2| >>> for i in range(len(ma_liste)):
3| ...     print(ma_liste[i])

```

Bien sûr, une écriture plus « pythonique » de l'exemple précédent permet de se débarrasser de l'indice *i* :

```

1| >>> ma_liste = [1, 3, 5, 7, 9, 11]
2| >>> for entier in ma_liste:
3| ...     print(entier)
4| ...

```

Enfin, des noms de variable à une lettre peuvent être utilisées lorsque cela a un sens mathématique (par exemple, les noms *x*, *y* et *z* évoquent des coordonnées cartésiennes).

15.1.4 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 :

```

1| # code recommandé :
2| ma_variable = 3 + 7
3| mon_texte = "souris"
4| mon_texte == ma_variable
5| # code non recommandé :
6| ma_variable=3+7
7| mon_texte="souris"
8| mon_texte== ma_variable

```

Il n'y a, par contre, pas d'espace à l'intérieur de crochets, d'accolades et de parenthèses :

```

1| # code recommandé :
2| ma_liste[1]
3| mon_dico{"clé"}
4| ma_fonction(argument)
5| # code non recommandé :
6| ma_liste[ 1 ]
7| mon_dico{"clé" }
8| ma_fonction( argument )

```

Ni juste avant la parenthèse ouvrante d'une fonction ou le crochet ouvrant d'une liste ou d'un dictionnaire :

```

1| # code recommandé :
2| ma_liste[1]
3| mon_dico{"clé"}
4| ma_fonction(argument)
5| # code non recommandé :
6| ma_liste [1]
7| mon_dico {"clé"}
8| ma_fonction (argument)

```

On met un espace après les caractères : et , (mais pas avant) :

```

1| # code recommandé :
2| ma_liste = [1, 2, 3]
3| mon_dico = {"clé1": "valeur1", "clé2": "valeur2"}
4| ma_fonction(argument1, argument2)
5| # code non recommandé :
6| ma_liste = [1 , 2 , 3]
7| mon_dico = {"clé1": "valeur1", "clé2": "valeur2"}
8| ma_fonction(argument1 ,argument2)

```

Par contre, pour les tranches de listes, on ne met pas d'espace autour du :

```

1| ma_liste = [1, 3, 5, 7, 9, 1]
2| # code recommandé :
3| ma_liste[1:3]
4| ma_liste[1:4:2]
5| ma_liste[::-2]
6| # code non recommandé :
7| ma_liste[1 : 3]
8| ma_liste[1: 4:2 ]
9| ma_liste[ : :2]

```

Enfin, on n'ajoute pas plusieurs espaces autour du = ou des autres opérateurs pour faire joli :

```

1| # code recommandé :
2| x1 = 1
3| x2 = 3
4| x_old = 5

```

```

5 | # code non recommandé :
6 | x1      = 1
7 | x2      = 3
8 | x_old   = 5

```

15.1.5 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 au chapitre 3 *Affichage* que le caractère \ permet de couper des lignes trop longues. Par exemple :

```

1 | >>> ma_variable = 3
2 | >>> if ma_variable > 1 and ma_variable < 10 \
3 | ... and ma_variable % 2 == 1 and ma_variable % 3 == 0:
4 | ...     print("ma variable vaut {}".format(ma_variable))
5 | ...
6 | ma variable vaut 3

```

À l'intérieur d'une parenthèse, 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 :

```

1 | >>> def ma_fonction(argument_1, argument_2,
2 | ...                 argument_3, argument_4):
3 | ...     return argument_1 + argument_2
4 | ...
5 | >>> ma_fonction("texte très long", "tigre",
6 | ...                   "singe", "souris")
7 | 'texte très longtigre'

```

Les parenthèses sont également très pratiques pour répartir sur plusieurs lignes une chaîne de caractères qui sera affichée sur une seule ligne :

```

1 | >>> print("ATGCGTACAGTATCGATAAC"
2 | ...     "ATGACTGCTACGATCGGATA"
3 | ...     "CGGGTAACGCCATGTACATT")
4 | ATGCGTACAGTATCGATAACATGACTGCTACGGATACGGTAACGCCATGTACATT

```

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 dans entre parenthèses, Python les concatène automatiquement.

On peut aussi utiliser les parenthèses pour évaluer un expression trop longue :

```

1 | >>> ma_variable = 3
2 | >>> if (ma_variable > 1 and ma_variable < 10
3 | ... and ma_variable % 2 == 1 and ma_variable % 3 == 0):
4 | ...     print("ma variable vaut {}".format(ma_variable))
5 | ...
6 | ma variable vaut 3

```

Les parenthèses sont aussi très utiles lorsqu'on a besoin d'enchaîner des méthodes les unes à la suite des autres. Un exemple se trouve dans le chapitre 17 *Quelques modules d'intérêt en bioinformatique*, dans la partie consacrée au module *pandas*.

Enfin, il est possible de créer des listes ou des dictionnaires sur plusieurs lignes, en sautant une ligne après une virgule :

```

1 | >>> ma_liste = [1, 2, 3,
2 | ...             4, 5, 6,
3 | ...             7, 8, 9]
4 | >>> mon_dico = {"clé1": 13,
5 | ...           "clé2": 42,
6 | ...           "clé3": -10}

```

15.1.6 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 c'est à utiliser avec parcimonie.

15.1.7 Commentaires

Les commentaires débutent toujours par le symbole `#` suivi d'un espace. Ils donnent des explications claires sur l'utilité du code et doivent être synchronisés avec le code, c'est-à-dire que si le code est modifié, les commentaires doivent l'être aussi (le cas échéant).

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 très fortement d'écrire les commentaires en anglais, sauf si vous êtes à 120% sûr 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 :

```
1 | x = x + 1    # My wonderful comment.
```

Remarque

Nous terminerons par une remarque qui concerne la syntaxe, mais qui n'est pas incluse dans la PEP 8. On nous pose souvent la question du type de guillemets à utiliser pour déclarer une chaîne de caractères. Simples ou doubles ?

```
1 | >>> var_1 = "Ma chaîne de caractères"
2 | >>> var_1
3 | 'Ma chaîne de caractères'
4 | >>> var_2 = 'Ma chaine de caractères'
5 | >>> var_2
6 | 'Ma chaîne de caractères'
7 | >>> var_1 == var_2
8 | True
```

Vous constatez dans l'exemple ci-dessus que pour Python, c'est exactement la même chose. Et à notre connaissance, il n'existe pas de recommandation officielle sur le sujet.

Nous vous conseillons cependant d'utiliser les guillemets doubles car ceux-ci sont, de notre point de vue, plus lisibles.

15.2 Les *docstrings* et la PEP 257

Les *docstrings*, que l'on pourrait traduire par « chaînes de documentation » en français, sont un élément essentiel de nos programmes Python comme on l'a vu au chapitre 14 *Création de modules*. À nouveau, les développeurs de Python ont émis des recommandations dans la PEP 8 et plus exhaustivement dans la PEP 257⁴ sur la manière de rédiger correctement les *docstrings*. En voici un résumé succinct.

De manière générale, écrivez des *docstrings* pour les modules, les fonctions, les classes et les méthodes. Lorsque l'explication est courte et compacte comme dans certaines fonctions ou méthodes simples, utilisez des *docstrings* d'une ligne :

```
1 | """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.

```
1 | """Docstring de plusieurs lignes, la première ligne est un résumé.
2 |
3 | Après avoir sauté une ligne, on décrit les détails de cette docstring.
4 | blablabla
5 | blablabla
6 | blublublu
7 | bliblibli
8 | On termine la docstring avec les triples guillemets sur la ligne suivante.
9 | """
```

Remarque

4. <https://www.python.org/dev/peps/pep-0257/>

La PEP 257 recommande d'écrire des *docstrings* avec des triples 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 14 *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.

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*⁵.
- La solution NumPy avec le *NumPy Style Python Docstrings*⁶. NumPy qui est un module complémentaire à Python, très utilisé en analyse de données et dont on parlera dans le chapitre 17 *Quelques modules d'intérêt en bioinformatique*.

On illustre ici la solution NumPy pour des raisons de goût personnel. Sentez-vous libre d'aller explorer la proposition de Google. Voici un exemple très simple :

```
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
13    Avec une description plus longue.
14    Sur plusieurs lignes.
15
16    Returns
17    -----
18    int
19        Le produit des deux nombres.
20
21    return nombre1 * nombre2
```

Lignes 6 et 7. La section *Parameters* précise les paramètres de la fonction. Les tirets sur la ligne 7 permettent de souligner le nom de la section et donc de 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 14. Même chose pour le second paramètre. La description du paramètre peut s'étaler sur plusieurs lignes.

Lignes 16 et 17. La section *Returns* indique ce qui est renvoyé par la fonction (le cas échéant).

Lignes 18 et 19. 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 !

5. https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html

6. https://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_numpy.html

15.3 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 des sites internet ou des outils dédiés.

Le site pep8online⁷, par exemple, est très simple d'utilisation. On copie / colle le code à évaluer puis on clique sur le bouton *Check code*.

Les outils pycodestyle, pydocstyle et pylint doivent par contre être installés sur votre machine. Avec la distribution Miniconda, cette étape d'installation se résume à une ligne de commande :

```
1 | $ 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 :

```
1 """Un script de multiplication.
2 """
3
4 import os
5
6 def Multiplie_nombres(nombre1, nombre2):
7     """Multiplication de deux nombres entiers
8     Cette fonction ne sert pas à grand chose.
9
10    Parameters
11    -----
12    nombre1 : int
13        Le premier nombre entier.
14    nombre2 : int
15        Le second nombre entier.
16
17    Avec une description plus longue.
18    Sur plusieurs lignes.
19
20    Returns
21    -----
22    int
23        Le produit des deux nombres.
24
25    """
26
27    return nombre1 *nombre2
28
29 if __name__ == "__main__":
30     print("2 x 3 = {}".format(Multiplie_nombres(2,3)))
31     print ("4 x 5 = {}".format(Multiplie_nombres(4, 5)))
```

Ce script est d'ailleurs parfaitement fonctionnel :

```
1 | $ python script_quality_ok.py
2 | 2 x 3 = 6
3 | 4 x 5 = 20
```

On va tout d'abord vérifier la conformité avec la PEP 8 avec l'outil pycodestyle :

```
1 | $ pycodestyle script_quality_not_ok.py
2 | script_quality_not_ok.py:6:1: E302 expected 2 blank lines, found 1
3 | script_quality_not_ok.py:6:30: E231 missing whitespace after ','
4 | script_quality_not_ok.py:6:38: E202 whitespace before ')'
5 | script_quality_not_ok.py:26:21: E225 missing whitespace around operator
6 | script_quality_not_ok.py:30:50: E231 missing whitespace after ','
7 | 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.

7. <http://pep8online.com/>

8. https://python.sdv.univ-paris-diderot.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 manque un espace après la virgule séparant les deux arguments lors de l'appel de la fonction `Multiplie_nombres()` à la ligne 30 (colonne 50) du script.

Ligne 7. Il y a un espace de trop entre `print` et `(` à la ligne 31 (colonne 10) du script.

Remarquez que curieusement, `pycodestyle` n'a pas détecté que le nom de la fonction `Multiplie_nombres()` ne respecte pas la convention de nommage.

Ensuite, l'outil `pycodestyle` va vérifier la conformité avec la PEP 257 et s'intéresser particulièrement aux *docstrings* :

```
1 $ pydocstyle script_quality_not_ok.py
2 script_quality_not_ok.py:1 at module level:
3     D200: One-line docstring should fit on one line with quotes (found 2)
4 script_quality_not_ok.py:7 in public function `Multiplie_nombres`:
5     D205: 1 blank line required between summary line and description (found 0)
6 script_quality_not_ok.py:7 in public function `Multiplie_nombres`:
7     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 :

```
1 $ pylint script_quality_not_ok.py
2 **** Module script_quality_not_ok
3 script_quality_not_ok.py:6:29: C0326: Exactly one space required after comma
4 def Multiplie_nombres(nombre1, nombre2):
5     ^ (bad-whitespace)
6 script_quality_not_ok.py:6:38: C0326: No space allowed before bracket
7 def Multiplie_nombres(nombre1, nombre2):
8     ^ (bad-whitespace)
9 script_quality_not_ok.py:30:49: C0326: Exactly one space required after comma
10    print("2 x 3 = {}".format(Multiplie_nombres(2,3)))
11    ^ (bad-whitespace)
12 script_quality_not_ok.py:31:10: C0326: No space allowed before bracket
13    print ("4 x 5 = {}".format(Multiplie_nombres(4, 5)))
14    ^ (bad-whitespace)
15 script_quality_not_ok.py:6:0: C0103: Function name "Multiplie_nombres"
16 doesn't conform to snake_case naming style (invalid-name)
17 script_quality_not_ok.py:4:0: W0611: Unused import os (unused-import)
18
19 -----
20 Your code has been rated at 0.00/10
```

Lignes 3 à 5. `pylint` indique qu'il manque un espace entre les paramètres de la fonction `Multiplie_nombres()` (ligne 6 et colonne 29 du script). La ligne du script qui pose problème est affichée, ce qui est pratique.

Lignes 6 à 8. `pylint` identifie un espace de trop après le second paramètre de la fonction `Multiplie_nombres()`.

Lignes 9 à 11. Il manque un espace entre le premier et le second argument de la fonction `Multiplie_nombres()`.

Ligne 12 à 14. Il y a un espace de trop entre `print` et `(`.

Lignes 15 et 16. Le nom de la fonction `Multiplie_nombres()` ne respecte pas la convention PEP 8. La fonction devrait s'appeler `multiplie_nombres()`.

Ligne 17. Le module `os` est chargé mais pas utilisé (ligne 4 du script).

Ligne 20. `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.

9. https://python.sdv.univ-paris-diderot.fr/data-files/script_quality_ok.py

15.4 Organisation du code

Il est fondamental de toujours structurer et organiser 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.feuilletée.fr")
13 __version__ = "1.0.0"
14 __copyright__ = "copyleft"
15 __date__ = "2130/01"
16 __version__ = "1.2.3"
17
18 import module_interne
19 import module_interne_2
20
21 import module_externe
22
23 UNE_CONSTANTE = valeur
24 UNE_AUTRE_CONSTANTE = une_autre_valeur
25
26
27 class UneSuperClasse():
28     """Résumé de la docstring décrivant la classe.
29
30     Description détaillée ligne 1
31     Description détaillée ligne 2
32     Description détaillée ligne 3
33     """
34
35     def __init__(self):
36         """Résumé de la docstring décrivant le constructeur.
37
38         Description détaillée ligne 1
39         Description détaillée ligne 2
40         Description détaillée ligne 3
41         """
42
43         [...]
44
45     def une_méthode_simple(self):
46         """Docstring d'une ligne décrivant la méthode."""
47
48         [...]
49
50     def une_méthode_complexe(self, arg1):
51         """Résumé de la docstring décrivant la méthode.
52
53         Description détaillée ligne 1
54         Description détaillée ligne 2
55         Description détaillée ligne 3
56         """
57
58         [...]
59         return un_truc
60
61
62     def une_fonction_complexe(arg1, arg2, arg3):
63         """Résumé de la docstring décrivant la fonction.
64
65         Description détaillée ligne 1
66         Description détaillée ligne 2
67         Description détaillée ligne 3
68         """
69
70         [...]
71         return une_chose
72
73
74     def une_fonction_simple(arg1, arg2):
75         """Docstring d'une ligne décrivant la fonction."""
76
77         [...]
78         return autre_chose
79
80
81 if __name__ == "__main__":
82     # ici débute le programme principal

```

78 | [. . .]

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 14 *Création de modules*).

Lignes 11 à 16. On définit ici un certain nombres 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. Bien sûr, ces métadonnées ne sont pas obligatoires, mais elles sont utiles lorsque le code est distribué à la communauté.

Lignes 18 à 21. Importation des modules. D'abord les modules internes à Python (fournis en standard), puis les modules externes (ceux qu'il faut installer en plus), un module par ligne.

Lignes 23 et 24. Définition des constantes. Le nom des constantes est en majuscule.

Ligne 27. Définition d'une classe. On a laissé deux lignes vides avant.

Lignes 28 à 33. *Docstring* décrivant la classe.

Lignes 34, 43 et 47. Avant chaque méthode de la classe, on laisse une ligne vide.

Lignes 59 à 73. Après les classes, on met les fonctions « classiques ». Avant chaque fonction, on laisse deux lignes vides.

Lignes 76 à 78. On écrit le programme principal. Le test ligne 76 n'est vrai que si le script est utilisé en tant que programme. Les lignes suivantes ne sont donc pas exécutées si le script est chargé comme un module.

15.5 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 des mots en français (ou en anglais) les consignes qui vous ont été données ou le cahier des charges qui vous a été communiqué. Dessinez ou construisez des schémas si cela vous aide.
- Découpez en fonctions chaque élément de votre programme. Vous pourrez ainsi tester chaque élément indépendamment du reste. Pensez à écrire les *docstrings* en même temps que vous écrivez vos fonctions.
- 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.
- Documentez-vous. L'algorithme dont vous avez besoin existe-t-il déjà dans un autre module ? Existe-t-il sous la forme de pseudo-code ? De quels outils mathématiques avez-vous besoin dans votre algorithme ?
- Si vous créez ou manipulez une entité cohérente avec des propriétés propres, essayez de construire une classe. Jetez, pour cela, un œil au chapitre 19 *Avoir la classe avec les objets*.
- 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 et illustrer l'organisation de cette structure de données sur un exemple simple.
- Testez toujours votre code sur un jeu de données **simple** pour pouvoir comprendre rapidement ce qui se passe. Par exemple, une séquence de 1000 bases est plus facile à gérer que le génome humain ! Cela vous permettra également de retrouver plus facilement une erreur lorsque votre programme ne fait pas ce que vous souhaitez.
- 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 claire ?
- Si vous distribuez votre code :
 - Rédigez une documentation claire.
 - Testez votre programme (jetez un œil aux tests unitaires).
 - Précisez une licence d'utilisation. Voir par exemple le site *Choose an open source license*¹⁰.

15.6 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` :

```
1 | >>> import this
2 | The Zen of Python, by Tim Peters
3 |
```

10. <https://choosealicense.com/>

```
4 | Beautiful is better than ugly.
5 | Explicit is better than implicit.
6 | Simple is better than complex.
7 | Complex is better than complicated.
8 | Flat is better than nested.
9 | Sparse is better than dense.
10| Readability counts.
11| Special cases aren't special enough to break the rules.
12| Although practicality beats purity.
13| Errors should never pass silently.
14| Unless explicitly silenced.
15| In the face of ambiguity, refuse the temptation to guess.
16| There should be one-- and preferably only one --obvious way to do it.
17| Although that way may not be obvious at first unless you're Dutch.
18| Now is better than never.
19| Although never is often better than *right* now.
20| If the implementation is hard to explain, it's a bad idea.
21| If the implementation is easy to explain, it may be a good idea.
22| Namespaces are one honking great idea -- let's do more of those!
23| >>>
```

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.

Pour aller plus loin

- L'article *Python Code Quality : Tools & Best Practices*¹¹ 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*¹² du site *OpenClassrooms* et *Structuring Python Programs*¹³ du site *Real Python* rappellent les règles d'écriture et les bonnes pratiques vues dans ce chapitre.

11. <https://realpython.com/python-code-quality/>

12. <https://openclassrooms.com/fr/courses/4425111-perfectionnez-vous-en-python/4464230-assimilez-les-bonnes-pratiques-de-la-pep-8>

13. <https://realpython.com/python-program-structure/>

Chapitre 16

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, spécialement lorsque vous souhaitez récupérer des informations dans de gros fichiers.

Cette action de recherche de données dans un fichier est appelée plus 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 comme 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 régulières¹ sur le site officiel de Python.

16.1 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*), motif 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 :

```
1 $ egrep '^DEF' herp_virus.gbk
2 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.

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 :

1. <https://docs.python.org/fr/3/library/re.html>

^ 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 à 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 ATTG.

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 10 *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.

Il est important de savoir par ailleurs que les regex sont « avides » (*greedy* en anglais) lorsqu'on utilise les métacaractères + et *. C'est-à-dire que la regex cherchera à « s'étendre » au maximum. Par exemple, si on utilise la regex `A+` pour faire une recherche dans la chaîne TTTAAAAAAAGC, tous les A de cette chaîne (8 en tout) seront concernés, bien que AA, AAA, etc. « fonctionnent » également avec cette regex.

16.2 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 :

- <https://regexone.com/> : tutoriel en ligne très bien fait.
- <https://regexpr.com/> : visualise tous les endroits où une *regex* est retrouvée dans un texte.
- <https://www.regular-expressions.info> : documentation exhaustive sur les *regex* (il y a même une section sur Python).
- <https://pythex.org/> : interface simple et efficace, dédiée à Python.

N'hésitez pas à tester ces sites avant de vous lancer dans les exercices ou dans l'écriture de vos propres *regex* !

16.3 Le module *re*

16.3.1 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, chaîne)`. Si *motif* est retrouvé dans *chaîne*, 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. Regardez cet exemple dans lequel on va rechercher le motif `tigre` dans la chaîne de caractères "`girafe tigre singe`" :

```

1 | >>> import re
2 | >>> animaux = "girafe tigre singe"
3 | >>> re.search("tigre", animaux)
4 | <_sre.SRE_Match object at 0x7fefdaefc2a0>
5 | >>> if re.search("tigre", animaux):
6 | ...     print("OK")
7 | ...
8 | 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`.

16.3.2 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).

```

1 | >>> animaux = "girafe tigre singe"
2 | >>> re.search("tigre", animaux)
3 | <_sre.SRE_Match object at 0x7fefdaefc718>
4 | >>> re.match('tigre', animaux)
5 | >>>
6 | >>> animaux = "tigre singe"
7 | >>> re.match("tigre", animaux)
8 | <_sre.SRE_Match object; span=(0, 5), match='tigre'>
9 | >>>

```

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.

```

1 | >>> animaux = "tigre "
2 | >>> re.fullmatch('tigre', animaux)
3 | >>> animaux = "tigre"
4 | >>> re.fullmatch('tigre', animaux)
5 | <_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 `$`, par exemple `^tigre$`.

16.3.3 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` :

```
1 | >>> regex = re.compile("^tigre")
2 | >>> regex
3 | <_sre.SRE_Pattern object at 0x7fefdaef0df0>
```

On peut alors utiliser directement cet objet avec la méthode `.search()` :

```
1 | >>> animaux = "girafe tigre singe"
2 | >>> regex.search(animaux)
3 | >>> animaux = "tigre singe"
4 | >>> regex.search(animaux)
5 | <_sre.SRE_Match object at 0x7fefdaef718>
6 | >>> animaux = "singe tigre"
7 | >>> regex.search(animaux)
```

16.3.4 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 :

```
1 | >>> 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()`.

```
1 | >>> resultat = regex.search("pi vaut 3.14")
2 | >>> resultat.group(0)
3 | '3.14'
4 | >>> resultat.group(1)
5 | '3'
6 | >>> resultat.group(2)
7 | '14'
8 | >>> resultat.start()
9 | 8
10 | >>> resultat.end()
11 | 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 :

```
1 | >>> resultat = regex.search("pi vaut 3.14 et e vaut 2.72")
2 | >>> resultat.group(0)
3 | '3.14'
```

16.3.5 La méthode `.findall()`

Pour récupérer chaque zone, vous pouvez utiliser la méthode `.findall()` qui renvoie une liste des éléments en correspondance.

```
1 | >>> regex = re.compile("[0-9]+\\.[0-9]+")
2 | >>> resultat = regex.findall("pi vaut 3.14 et e vaut 2.72")
3 | >>> resultat
4 | ['3.14', '2.72']
```

L'utilisation des groupes entre parenthèses est également possible et ceux-ci sont alors renvoyés sous la forme de tuples.

```

1 | >>> regex = re.compile("[0-9]+\.(0-9)+")
2 | >>> resultat = regex.findall("pi vaut 3.14 et e vaut 2.72")
3 | >>> resultat
4 | [('3', '14'), ('2', '72')]

```

16.3.6 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` :

```

1 | >>> regex = re.compile("[0-9]+\.[0-9]+")
2 | >>> regex.sub("quelque chose", "pi vaut 3.14 et e vaut 2.72")
3 | 'pi vaut quelque chose et e vaut quelque chose'
4 | >>> regex.sub("quelque chose", "pi vaut 3.14 et e vaut 2.72", count=1)
5 | '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.

```

1 | >>> regex = re.compile("[0-9]+\.(0-9)+")
2 | >>> phrase = "pi vaut 3.14 et e vaut 2.72"
3 | >>> regex.sub("approximativement \\\1", phrase)
4 | 'pi vaut approximativement 3 et e vaut vaut approximativement 2'
5 | >>> regex.sub("approximativement \\\1 (puis .\\\2)", phrase)
6 | '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 et à vos *regex* !

16.4 Exercices

Conseil : pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

16.4.1 Regex de base

Dans cet exercice, nous allons manipuler le fichier GenBank NC_001133.gbk² 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.

2. https://python.sdv.univ-paris-diderot.fr/data-files/NC_001133.gbk

Conseils :

- Vous utiliserez des *regex* pour trouver les lignes demandées.
- Vous trouverez des explications sur le format GenBank et des exemples de code dans l'annexe A *Quelques formats de données rencontrés en biologie*.

16.4.2 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
BbvAI	GAANNNNTTC

16.4.3 Nettoyeur d'espaces

Le fichier `cigale_fourmi.txt`³ contient le célèbre poème 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 espace dans le texte ci-dessus. Le nouveau texte « propre » sera enregistré dans un fichier `cigale_fourmi_propre.txt`.

16.4.4 Liste des protéines humaines

Téléchargez le fichier `human-proteome.fasta`⁴ qui contient le protéome humaine, 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*, de toutes les lignes de commentaires des séquences, le numéro d'accession de la protéine ;
- affiche le mot `protein`, suivi d'un numéro qui s'incrémentera, suivi du numéro d'accession.

Voici un exemple de sortie attendue :

```

1 | protein 00001 095139
2 | protein 00002 075438
3 | protein 00003 Q8N4C6
4 | [...]
5 | protein 20371 Q8IZJ1
6 | protein 20372 Q9UKP6
7 | protein 20373 Q96HZ7

```

Conseils :

- Vous trouverez des explications sur le format FASTA et des exemples de code dans l'annexe A *Quelques formats de données rencontrés en biologie*.
- La ligne de commentaire d'une séquence au format FASTA est de la forme
`>sp|095139|NDUB6_HUMAN NADH dehydrogenase [...]`
 Elle débute toujours pas le caractère `>`. Le numéro d'accession 095139 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'incrémentera débutera à 1 et sera affiché sur 5 caractères avec des 0 à sa gauche si nécessaires (formatage `{:05d}`).

3. https://python.sdv.univ-paris-diderot.fr/data-files/cigale_fourmi.txt

4. <https://python.sdv.univ-paris-diderot.fr/data-files/human-proteome.fasta>

16.4.5 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`⁵ 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

16.4.6 Nettoyeur de doublons (exercice +++)

Téléchargez le fichier `breves_doublons.txt`⁶ qui contient des mots répétés deux fois. Par exemple :

```
1| Le cinéma est devenu parlant , la radio radio finira en images .  
2| La sardine , c'est un petit petit poisson sans tête qui vit dans l'huile .  
3| [...]
```

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()`.

5. https://python.sdv.univ-paris-diderot.fr/data-files/fichier_a_dhtmliser.html

6. https://python.sdv.univ-paris-diderot.fr/data-files/breves_doublons.txt

Chapitre 17

Quelques modules d'intérêt en bioinformatique

Nous allons aborder dans ce chapitre quelques modules très importants en bioinformatique. Le premier *NumPy* permet notamment de manipuler des vecteurs et des matrices. Le module *Biopython* permet de travailler sur des données biologiques, comme des séquences (nucléiques et protéiques) ou des structures (fichiers PDB). Le module *matplotlib* permet de créer des graphiques depuis Python. Enfin, le module *pandas* est très performant pour l'analyse de données, et *scipy* étend les possibilités offertes par *NumPy*, notamment en proposant des algorithmes couramment utilisés en calcul scientifique.

Ces modules ne sont pas fournis avec la distribution Python de base (contrairement à tous les autres modules vus précédemment). Avec la distribution Miniconda que nous vous avons conseillé d'utiliser (consultez pour cela la documentation en ligne¹), vous pouvez rapidement les installer avec la commande :

```
| $ conda install -y numpy pandas matplotlib scipy biopython
```

Dans ce chapitre, nous vous montrerons quelques exemples d'utilisation de ces modules pour vous convaincre de leur pertinence.

17.1 Module *NumPy*

Le module *NumPy*² 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*.

On charge le module *NumPy* avec la commande :

```
| >>> import numpy
```

On peut également définir un nom raccourci pour *NumPy* :

```
| >>> import numpy as np
```

17.1.1 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 objet séquentiel (comme une liste ou un tuple) en un objet de type *array*. Voici un exemple simple de conversion d'une liste à une dimension en objet *array* :

```
1 | >>> import numpy as np
2 | >>> a = [1, 2, 3]
3 | >>> np.array(a)
4 | array([1, 2, 3])
5 | >>> b = np.array(a)
6 | >>> b
7 | array([1, 2, 3])
8 | >>> type(b)
9 | <type 'numpy.ndarray'>
```

Nous avons converti la liste [1, 2, 3] en *array*. Nous aurions obtenu le même résultat si nous avions converti le tuple (1,2,3) en *array*.

1. <https://python.sdv.univ-paris-diderot.fr/livre-dunod>

2. <http://numpy.scipy.org/>

Par ailleurs, lorsqu'on demande à Python d'afficher le contenu d'un objet *array*, 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 :

```
1 | >>> a = np.array([1, 2, "tigre"])
2 | >>> a
3 | array(['1', '2', 'tigre'], dtype='|<U21')
4 | >>> type(a)
5 | <class 'numpy.ndarray'>
```

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* :

```
1 | >>> b = np.array([1, 2, 3.5])
2 | >>> b
3 | array([1., 2., 3.5])
4 | >>> type(b)
5 | <class 'numpy.ndarray'>
```

Ici, la notation 1. indique qu'il s'agit du *float* 1.0000... et pas de l'entier 1.

Le module *NumPy* est très performant pour manipuler des valeurs numériques (entières ou *floats*). Nous vous recommandons de ne pas utiliser que des valeurs numériques avec *NumPy*.

Contrairement à la fonction `range()`, la fonction `arange()` permet de construire un *array* à une dimension de manière simple.

```
1 | >>> np.arange(10)
2 | 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 :

```
1 | >>> np.arange(10, 0, -1)
2 | 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* selon l'argument qu'on lui passe :

```
1 | >>> np.arange(10)
2 | array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
3 | >>> np.arange(10.0)
4 | 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 **élément par élément** sur ce type d'objet, ce qui est bien commode lorsqu'on analyse de grandes quantités de données. Regardez ces exemples :

```
1 | >>> v = np.arange(4)
2 | >>> v
3 | array([0, 1, 2, 3])
4 | >>> v + 1
5 | array([1, 2, 3, 4])
6 | >>> v + 0.1
7 | array([ 0.1,  1.1,  2.1,  3.1])
8 | >>> v * 2
9 | array([0, 2, 4, 6])
10 | >>> v * v
11 | array([0, 1, 4, 9])
```

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.

Notez également que, dans le dernier exemple de multiplication (ligne 10), l'*array* final correspond à la multiplication élément par élément des deux *array* initiaux.

17.1.2 array et dimensions

Il est aussi possible de construire des objets *array* à deux dimensions, il suffit de passer en argument une liste de listes à la fonction `array()` :

```

1 | >>> w = np.array([[1,2],[3,4],[5,6]])
2 | >>> w
3 | array([[1, 2],
4 |       [3, 4],
5 |       [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 :

```

1 | >>> x = np.array([[[1,2],[2,3]],[[4,5],[5,6]]])
2 | >>> x
3 | array([[[1, 2],
4 |       [2, 3]],
5 |
6 |       [[4, 5],
7 |       [5, 6]]])

```

La fonction `array()` peut créer des tableaux à n'importe quel nombre de dimensions. Toutefois ça devient vite compliqué lorsqu'on dépasse trois dimensions. Retenez qu'un objet *array* à une dimension peut être considéré comme un **vecteur** et un *array* à deux dimensions comme une **matrice**.

Voici quelques attributs intéressants pour décrire un objet *array* :

- `.ndim` renvoie le nombre de dimensions (par exemple, 1 pour un vecteur et 2 pour une matrice);
- `.shape` renvoie les dimensions sous forme d'un tuple;
- `.size` renvoie le nombre total d'éléments contenus dans l'*array*.

```

1 | >>> v = np.arange(4)
2 | >>> v
3 | array([0, 1, 2, 3])
4 | >>> v.ndim
5 | 1
6 | >>> v.shape
7 | (4,)
8 | >>> v.size
9 | 4
10 | >>> w = np.array([[1,2],[3,4],[5,6]])
11 | >>> w
12 | array([[1, 2],
13 |       [3, 4],
14 |       [5, 6]])
15 | >>> w.ndim
16 | 2
17 | >>> w.shape
18 | (3, 2)
19 | >>> w.size
20 | 6

```

Et la méthode `.reshape()` modifie les dimensions d'un *array* :

```

1 | >>> a = np.arange(0, 6)
2 | >>> a
3 | array([0, 1, 2, 3, 4, 5])
4 | >>> a.shape
5 | (6,)
6 | >>> b = a.reshape((2, 3))
7 | >>> b
8 | array([[0, 1, 2],
9 |       [3, 4, 5]])
10 | >>> b.shape
11 | (2, 3)

```

Notez que `a.reshape((2, 3))` n'est pas la même chose que `a.reshape((3, 2))` :

```

1 | >>> c = a.reshape((3, 2))
2 | >>> c
3 | array([[0, 1],
4 |       [2, 3],
5 |       [4, 5]])
6 | >>> c.shape
7 | (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 *array* 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.

```

1 | >>> a = np.arange(0, 6)
2 | >>> a
3 | array([0, 1, 2, 3, 4, 5])
4 | >>> a.shape
5 | (6,)
6 | >>> d = a.reshape((3, 4))
7 | Traceback (most recent call last):
8 | File "<stdin>", line 1, in <module>
9 | 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 ou coupe la liste initiale jusqu'à ce que le nouvel *array* soit rempli.

```

1 | >>> a = np.arange(0, 6)
2 | >>> a.shape
3 | (6,)
4 | >>> a.resize((3,3))
5 | >>> a.shape
6 | (3, 3)
7 | >>> a
8 | array([[0, 1, 2],
9 |        [3, 4, 5],
10|         [0, 0, 0]])
```



```

1 | >>> b = np.arange(0, 10)
2 | >>> b.shape
3 | (10,)
4 | >>> b.resize((2,3))
5 | >>> b.shape
6 | (2, 3)
7 | >>> b
8 | array([[0, 1, 2],
9 |        [3, 4, 5]])
```

Notez qu'il existe aussi la fonction `np.resize()` qui, dans le cas d'un nouvel *array* plus grand que l'*array* initial, va répéter l'*array* initial :

```

1 | >>> a = np.arange(0, 6)
2 | >>> a.shape
3 | (6,)
4 | >>> c = np.resize(a, (3, 5))
5 | >>> c.shape
6 | (3, 5)
7 | >>> c
8 | array([[0, 1, 2, 3, 4],
9 |        [5, 0, 1, 2, 3],
10|         [4, 5, 0, 1, 2]])
```

17.1.3 Indices

Pour récupérer un ou plusieurs élément(s) d'un objet *array*, vous pouvez utiliser les indices ou les tranches, de la même manière qu'avec les listes :

```

1 | >>> a = np.arange(10)
2 | >>> a
3 | array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
4 | >>> a[5:]
5 | array([5, 6, 7, 8, 9])
6 | >>> a[::2]
7 | array([0, 2, 4, 6, 8])
8 | >>> a[1]
9 | 1
```

Dans le cas d'un objet *array* à deux dimensions, vous pouvez récupérer une ligne complète (d'indice *m*), une colonne complète (d'indice *n*) ou bien un seul élément.

```

1 | >>> a = np.array([[1,2],[3,4]])
2 | >>> a
3 | array([[1, 2],
4 |        [3, 4]])
5 | >>> a[:,0]
6 | array([1, 3])
```

```

7 | >>> a[0,:]
8 | array([1, 2])
9 | >>> a[1,1]
10| 4

```

La syntaxe `a[m, :]` renvoie la ligne $m-1$, et `a[:, n]` renvoie la colonne $n-1$. Les tranches sont évidemment aussi utilisables sur un tableau à deux dimensions.

17.1.4 Construction automatique de matrices

Il peut être 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 construire des matrices à partir de rien. 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.

```

1 | >>> np.zeros((2,3))
2 | array([[0., 0., 0.],
3 |        [0., 0., 0.]])
4 | >>> np.ones((3,3))
5 | array([[1., 1., 1.],
6 |        [1., 1., 1.],
7 |        [1., 1., 1.]])

```

Par défaut, les fonctions `zeros()` et `ones()` génèrent des *floats*, mais vous pouvez demander des entiers en passant l'option `int` en second argument :

```

1 | >>> np.zeros((2,3), int)
2 | array([[0, 0, 0],
3 |        [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)
2 | array([[7, 7, 7],
3 |        [7, 7, 7]])
4 | >>> np.full((2,3), 7, float)
5 | array([[ 7.,  7.,  7.],
6 |        [ 7.,  7.,  7.]])

```

Nous construisons ainsi une matrice constituée de 2 lignes et 3 colonnes et qui ne contient que le chiffre 7, sous formes d'entiers (`int`) dans le premier cas et de *floats* dans le second.

17.1.5 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 d'un *array*. Par exemple, pour une matrice :

```

1 | >>> a = np.resize(np.arange(1,10), (3,3))
2 | >>> a
3 | array([[1, 2, 3],
4 |        [4, 5, 6],
5 |        [7, 8, 9]])
6 | >>> np.transpose(a)
7 | array([[1, 4, 7],
8 |        [2, 5, 8],
9 |        [3, 6, 9]])

```

La fonction `dot()` vous permet de réaliser une multiplication de matrices.

```

1 | >>> a = np.resize(np.arange(4), (2,2))
2 | >>> a
3 | array([[0, 1],
4 |        [2, 3]])
5 | >>> np.dot(a, a)
6 | array([[ 2,  3],
7 |        [ 6, 11]])
8 | >>> a * a
9 | array([[0, 1],
10|       [4, 9]])

```

Notez bien que `dot(a, a)` renvoie le **produit matriciel** entre deux matrices, alors que `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 `inv()` renvoie l'inverse d'une matrice carrée, `det()` son déterminant et `eig()` ses vecteurs et valeurs propres.

```

1 | >>> a = np.resize(np.arange(4), (2,2))
2 | >>> a
3 | array([[0, 1],
4 |        [2, 3]])
5 | >>> np.linalg.inv(a)
6 | array([[-1.5,  0.5],
7 |        [ 1.,  0. ]])
8 | >>> np.linalg.det(a)
9 | -2.0
10 | >>> np.linalg.eig(a)
11 | (array([-0.56155281,  3.56155281]), array([-0.87192821, -0.27032301],
12 | [ 0.48963374, -0.96276969]))
13 | >>> np.linalg.eig(a)[0]
14 | array([-0.56155281,  3.56155281])
15 | >>> np.linalg.eig(a)[1]
16 | array([-0.87192821, -0.27032301],
17 | [ 0.48963374, -0.96276969]))
```

Notez que la fonction `eig()` renvoie un tuple dont le premier élément correspond aux valeurs propres et le second aux vecteurs propres.

17.2 Module *Biopython*

Le module *Biopython*³ propose des fonctionnalités très utiles en bioinformatique. Le tutoriel⁴ est particulièrement bien fait, n'hésitez pas à le consulter.

17.2.1 Manipulation de données

Voici quelques exemples de manipulation de données avec *Biopython*.

Définition d'une séquence :

```

1 | >>> import Bio
2 | >>> from Bio.Seq import Seq
3 | >>> from Bio.Alphabet import IUPAC
4 | >>> ADN = Seq("ATATCGGCTATAGCATGCA", IUPAC.unambiguous_dna)
5 | >>> ADN
6 | Seq('ATATCGGCTATAGCATGCA', IUPACUnambiguousDNA())
```

Ligne 1. Le module *Biopython* s'appelle *Bio*.

Ligne 4. L'expression `IUPAC.unambiguous_dna` signifie que la séquence entrée est bien une séquence d'ADN.

Obtention de la séquence complémentaire et complémentaire inverse :

```

1 | >>> ADN.complement()
2 | Seq('TATAGCCGATATCGTACGT', IUPACUnambiguousDNA())
3 | >>> ADN.reverse_complement()
4 | Seq('TGCATGCTATAGCCGATAT', IUPACUnambiguousDNA())
```

Traduction en séquence protéique :

```

1 | >>> ADN.translate()
2 | Seq('ISAIAC', IUPACProtein())
```

Dans l'annexe A *Quelques formats de données rencontrés en biologie*, vous trouverez de nombreux exemples d'utilisation de *Biopython* pour manipuler des données aux formats FASTA, GenBank et PDB.

3. <http://biopython.org/>

4. <http://biopython.org/DIST/docs/tutorial/Tutorial.html>

[Biometals](#), 2012 Aug;25(4):677-86. doi: 10.1007/s10534-012-9520-3.

Known and potential roles of transferrin in iron biology.

[Bartnikas TB¹](#).

[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 17.1 – Aperçu de la publication *Known and potential roles of transferrin in iron biology* depuis le site PubMed.

17.2.2 Interrogation de la base de données PubMed

Le sous-module *Entrez* de *Biopython* permet d'utiliser les ressources du NCBI et notamment d'interroger le site PubMed⁵.

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 = "votreemail@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.

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()
2 | dict_keys(['Count', 'RetMax', 'RetStart', 'IdList', 'TranslationSet',
3 | '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"]
2 | ['30411489', '30409795', '30405884', '30405827', '30402883', '30401570',
3 | '30399508', '30397276', '30395963', '30394734', '30394728', '30394123',
4 | '30393423', '30392910', '30392664', '30391706', '30391651', '30391537',
5 | '30391296', '30390672']
6 | >>> len(res_esearch["IdList"])
7 | 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 33900 résultats.

En réalité, le nombre exact de publications est connu :

```
1 | >>> res_esearch["Count"]
2 | '33988'
```

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 22294463⁶ et dont un aperçu est sur la figure 17.1.

Nous allons pour cela utiliser la fonction *Entrez.esummary()*

```
1 | >>> req_esummary = Entrez.esummary(db="pubmed", id="22294463")
2 | >>> res_esummary = Entrez.read(req_esummary)
```

5. <https://www.ncbi.nlm.nih.gov/pubmed/>

6. <https://www.ncbi.nlm.nih.gov/pubmed/22294463>

La variable `res_esummary` n'est pas réellement une liste mais en a plusieurs propriétés. Cette pseudo-liste n'a qu'un seul élément, qui est lui-même un pseudo-dictionnaire dont voici les clés :

```

1 | >>> res_esummary[0].keys()
2 | dict_keys(['Item', 'Id', 'PubDate', 'EPubDate', 'Source', 'AuthorList',
3 | 'LastAuthor', 'Title', 'Volume', 'Issue', 'Pages', 'LangList',
4 | 'NlmUniqueID', 'ISSN', 'ESSN', 'PubTypeList', 'RecordStatus', 'PubStatus',
5 | 'ArticleIds', 'DOI', 'History', 'References', 'HasAbstract', 'PmcRefCount',
6 | '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 :

```

1 | >>> res_esummary[0]["Title"]
2 | 'Known and potential roles of transferrin in iron biology.'
3 | >>> res_esummary[0]["DOI"]
4 | '10.1007/s10534-012-9520-3'
5 | >>> res_esummary[0]["PubDate"]
6 | '2012 Aug'
7 | >>> res_esummary[0]["Source"]
8 | 'Biometals'

```

Enfin, pour récupérer le résumé de la publication précédente, nous allons utiliser la fonction `Entrez.efetch()` :

```

1 | >>> req_efetch = Entrez.efetch(db="pubmed", id="22294463", rettype="txt")
2 | >>> res_efetch = Entrez.read(req_efetch)

```

La variable `res_efetch` est un pseudo-dictionnaire qui contient une pseudo-liste, qui contient un pseudo-dictionnaire, qui contient... Oui, c'est compliqué ! Pour faire court, le résumé peut s'obtenir avec l'instruction :

```

1 | >>> res_efetch['PubmedArticle'][0]['MedlineCitation']['Article'] \
2 | ... ['Abstract']['AbstractText'][0]
3 | 'Transferrin is an abundant serum metal-binding protein best known
4 | for its role in iron delivery. The human disease congenital atransf
5 | errinia and animal models of this disease highlight the essential
6 | role of transferrin in erythropoiesis and iron metabolism. Patient
7 | s and mice deficient in transferrin exhibit anemia and a paradoxica
8 | l iron overload attributed to deficiency in hepcidin, a peptide hor
9 | mone synthesized largely by the liver that inhibits dietary iron ab
10 | sorption and macrophage iron efflux. Studies of inherited human dis
11 | ease and model organisms indicate that transferrin is an essential
12 | regulator of hepcidin expression. In this paper, we review current
13 | literature on transferrin deficiency and present our recent finding
14 | s, including potential overlaps between transferrin, iron and manga
15 | nese in the regulation of hepcidin expression.'

```

Ce qui est bien le résumé que nous obtenons sur la figure 17.1.

17.3 Module *matplotlib*

Le module *matplotlib*⁷ permet de générer des graphiques depuis Python. Il est l'outil complémentaire de *NumPy*, *scipy* ou *pandas* (qu'on verra juste après) lorsqu'on veut faire de l'analyse de données.

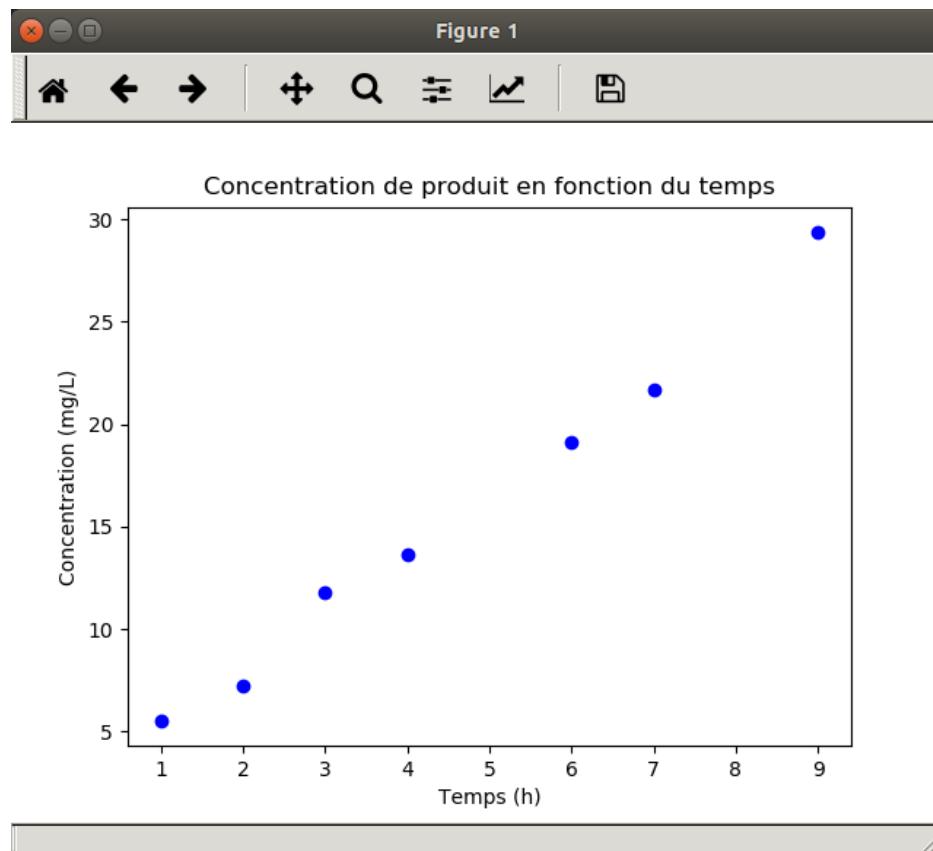
17.3.1 Représentation sous forme de points

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 heure).

Nous avons les résultats suivants :

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

7. <https://matplotlib.org/>

FIGURE 17.2 – Fenêtre interactive de *matplotlib*.

Nous allons maintenant représenter l'évolution de la concentration en fonction du temps :

```

1| import matplotlib.pyplot as plt
2|
3| temps = [1, 2, 3, 4, 6, 7, 9]
4| concentration = [5.5, 7.2, 11.8, 13.6, 19.1, 21.7, 29.4]
5| plt.scatter(temps, concentration, marker="o", color="blue")
6| plt.xlabel("Temps (h)")
7| plt.ylabel("Concentration (mg/L)")
8| plt.title("Concentration de produit en fonction du temps")
9| plt.show()

```

Vous devriez obtenir une fenêtre graphique **interactive** qui vous permet de manipuler le graphe (se déplacer, zoomer, enregistrer comme image, etc.) et qui ressemble à celle de la figure 17.2.

Revenons maintenant sur le code.

Ligne 1. Tout d'abord, on importe le sous-module `pyplot` du module `matplotlib` et on lui donne le nom court `plt` pour l'utiliser plus rapidement ensuite.

Lignes 3 et 4. On définit les variables `temps` et `concentration` comme des listes. Les deux listes doivent avoir la même longueur (7 éléments dans le cas présent).

Ligne 5. La fonction `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 6 et 7. Les fonctions `xlabel()` et `ylabel()` sont utilisées pour donner un nom aux axes.

Ligne 8. La fonction `title()` définit le titre du graphique.

Ligne 9. Enfin, la fonction `show()` affiche le graphique généré à l'écran.

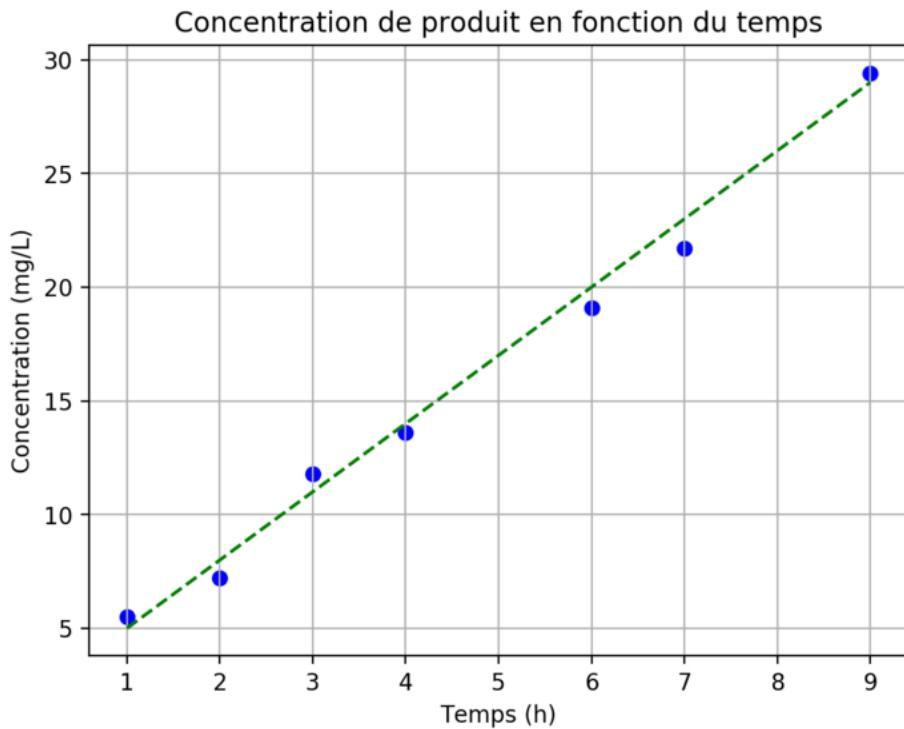


FIGURE 17.3 – Concentration du produit en fonction du temps.

17.3.2 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 $f(x) = 2 + 3 \times x$. 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 plt.scatter(temps, concentration, marker='o', color = 'blue')
7 plt.xlabel("Temps (h)")
8 plt.ylabel("Concentration (mg/L)")
9 plt.title("Concentration de produit en fonction du temps")
10 x = np.linspace( min(temps), max(temps), 50)
11 y = 2 + 3 * x
12 plt.plot(x, y, color='green', ls="--")
13 plt.grid()
14 plt.savefig('concentration_vs_temps.png', bbox_inches='tight', dpi=200)

```

Le résultat est représenté sur la figure 17.3.

Les étapes supplémentaires par rapport au graphique précédent (figure 17.2) sont :

Ligne 1. On charge le module *numpy* sous le nom *np*.

Ligne 10. 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 générerons une liste de 50 valeurs. La variable *x* ainsi créée est du type *array*.

Ligne 11. On construit ensuite la variable *y* à partir de la formule modélisant l'évolution de la concentration 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 12. La fonction *plot()* permet de construire une courbe à partir des coordonnées en abscisse et en ordonnées des points à représenter. On indique ensuite des arguments facultatifs comme le style de la ligne (*ls*) et sa couleur (*color*).

Ligne 13. La fonction *grid()* affiche une grille.

Ligne 14. Enfin, la fonction *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

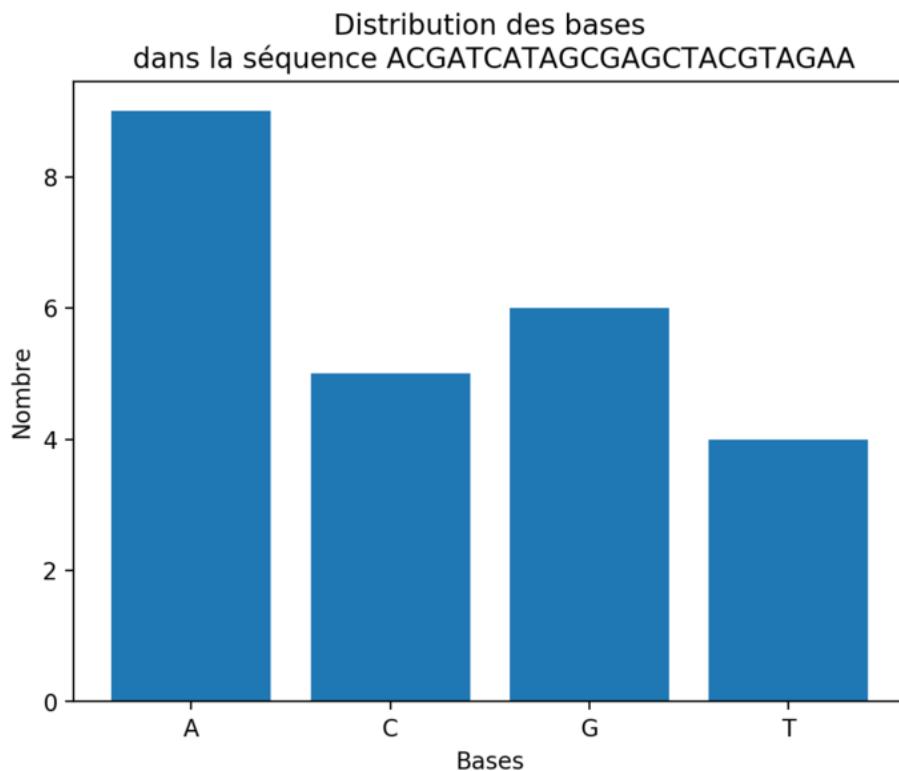


FIGURE 17.4 – Distribution des bases.

l'image (dpi).

17.3.3 Représentation sous forme de 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 plt.bar(x, distribution)
12 plt.xticks(x, bases)
13 plt.xlabel("Bases")
14 plt.ylabel("Nombre")
15 plt.title("Distribution des bases\n dans la séquence {}".format(sequence))
16 plt.savefig('distribution_bases.png', bbox_inches="tight", dpi=200)

```

On obtient alors le graphique de la figure 17.4.

Prenons le temps d'examiner les différentes étapes du script précédent :

Lignes 3 à 5. On définit les variables `sequence`, `bases` et `distribution` qu'on utilise ensuite.

Lignes 6 et 7. 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 9. On définit la position en abscisse des barres. Dans cet exemple, la variable `x` vaut `array([0, 1, 2, 3])`.

Ligne 10. La fonction `bar()` construit le diagramme en bâtons. Elle prend en argument la position des barres (`x`) et leurs hauteurs (`distribution`).

Ligne 11. La fonction `xticks()` redéfinit les étiquettes (c'est-à-dire le nom des bases) sur l'axe des abscisses.

Lignes 12 à 14. 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 15. 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. Le site du *matplotlib* fournit de nombreux exemples détaillés⁸, n'hésitez pas à le consulter.

17.4 Module *pandas*

Le module *pandas*⁹ a été conçu pour la manipulation et l'analyse de données. Il est particulièrement puissant pour manipuler des données structurées sous forme de tableau.

Pour charger *pandas* dans la mémoire de Python, on utilise la commande `import` habituelle :

```
1 | >>> import pandas
```

Pandas est souvent chargé avec un nom raccourci, comme pour *NumPy* et *matplotlib* :

```
1 | >>> import panda as pd
```

17.4.1 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
3 | a    10
4 | b    20
5 | c    30
6 | d    40
7 | dtype: int64
```

Avec *pandas*, chaque élément de la série de données possède une étiquette qui permet d'appeler les éléments. Ainsi, pour appeler le premier élément de la série, on peut utiliser son index, comme pour une liste (0 pour le premier élément) ou son étiquette (ici, "a") :

```
1 | >>> s[0]
2 | 10
3 | >>> s["a"]
4 | 10
```

Bien sûr, on peut extraire plusieurs éléments, par leurs indices ou leurs étiquettes :

```
1 | >>> s[[1,3]]
2 | b    20
3 | d    40
4 | dtype: int64
5 | >>> s[["b","d"]]
6 | b    20
7 | d    40
8 | dtype: int64
```

Les étiquettes permettent de modifier et d'ajouter des éléments :

```
1 | >>> s["c"] = 300
2 | >>> s["z"] = 50
3 | >>> s
4 | a    10
5 | b    20
6 | c    300
7 | d    40
8 | z    50
9 | dtype: int64
```

Enfin, on peut filtrer une partie de la *series* :

```
1 | >>> s[s>30]
2 | c    300
3 | d    40
4 | z    50
5 | dtype: int64
```

8. <https://matplotlib.org/gallery/index.html>

9. <https://pandas.pydata.org/>

et même combiner plusieurs critères de sélection :

```

1 >>> s[(s>20) & (s<100)]
2   d    40
3   z    50
4   dtype: int64

```

17.4.2 Dataframes

Un autre type d'objet particulièrement intéressant et introduit par *pandas* sont les *dataframes*, qui 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.

Voici comment créer un *dataframe* avec *pandas* à partir de données fournies comme liste de lignes :

```

1 >>> df = pd.DataFrame(columns=["a", "b", "c", "d"],
2 ...                 index=["chat", "singe", "souris"],
3 ...                 data=[np.arange(10, 14),
4 ...                       np.arange(20, 24),
5 ...                       np.arange(30, 34)])
6 >>> df
7          a    b    c    d
8 chat    10   11   12   13
9 singe   20   21   22   23
10 souris 30   31   32   33

```

Ligne 1. Le *dataframe* est créé avec la fonction `pd.DataFrame()` à laquelle on fournit plusieurs arguments. L'argument `columns` indique le nom des colonnes, sous forme d'une liste.

Ligne 2. L'argument `index` définit le nom des lignes, sous forme de liste.

Lignes 3-5. 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.from_dict(data)
6 >>> df.index = ["chat", "singe", "souris"]
7 >>> df
8          a    b    c    d
9 chat    10   11   12   13
10 singe   20   21   22   23
11 souris 30   31   32   33

```

Lignes 1-4. 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.from_dict()` 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`.

17.4.3 Quelques propriétés

Les dimensions d'un *dataframe* sont données par l'attribut `.shape` :

```

1 >>> df.shape
2 (3, 4)

```

Ici, le *dataframe* `df` a 3 lignes et 4 colonnes.

L'attribut `.columns` renvoie le nom des colonnes et permet aussi de renommer les colonnes d'un *dataframe* :

```

1 >>> df.columns
2 Index(['a', 'b', 'c', 'd'], dtype='object')
3 >>> df.columns = ["Paris", "Lyon", "Nantes", "Pau"]
4 >>> df
5          Paris    Lyon    Nantes    Pau
6 chat      10      11      12      13
7 singe     20      21      22      23
8 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)
2 |      Paris   Lyon   Nantes   Pau
3 | chat       10     11      12     13
4 | singe      20     21      22     23

```

17.4.4 Sélection

Les mécanismes de sélection fournis avec *pandas* sont très puissants. En voici un rapide aperçu :

Sélection de colonnes

On peut sélectionner une colonne par son étiquette :

```

1 | >>> df["Lyon"]
2 | chat      11
3 | singe     21
4 | souris    31

```

ou plusieurs colonnes en même temps :

```

1 | >>> df[["Lyon", "Pau"]]
2 |      Lyon   Pau
3 | chat     11    13
4 | singe    21    23
5 | souris   31    33

```

Pour la sélection de plusieurs colonnes, les étiquettes d'intérêt sont rassemblées dans une liste.

Sélection de lignes

Pour sélectionner une ligne, il faut utiliser l'instruction `.loc()` et l'étiquette de la ligne :

```

1 | >>> df.loc["singe"]
2 | Paris      20
3 | Lyon       21
4 | Nantes     22
5 | Pau        23
6 | Name: singe, dtype: int64

```

Ici aussi, on peut sélectionner plusieurs lignes :

```

1 | >>> df.loc[["singe", "chat"]]
2 |      Paris   Lyon   Nantes   Pau
3 | singe     20     21      22     23
4 | 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) :

```

1 | >>> df.iloc[1]
2 | Paris      20
3 | Lyon       21
4 | Nantes     22
5 | Pau        23
6 | Name: singe, dtype: int64

```

```

1 | >>> df.iloc[[1,0]]
2 |      Paris   Lyon   Nantes   Pau
3 | singe     20     21      22     23
4 | chat      10     11      12     13

```

On peut également utiliser les tranches (comme pour les listes) :

```

1 | >>> df.iloc[0:2]
2 |      Paris   Lyon   Nantes   Pau
3 | chat      10     11      12     13
4 | singe     20     21      22     23

```

Sélection sur les lignes et les colonnes

On peut bien sûr combiner les deux types de sélection (en ligne et en colonne) :

```

1 | >>> df.loc["souris", "Pau"]
2 | 33
3 | >>> df.loc[["singe", "souris"], ['Nantes', 'Lyon']]
4 |      Nantes   Lyon
5 | singe        22    21
6 | 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).

Sélection par condition

Remémorons-nous d'abord le contenu du *dataframe* `df` :

```

1 | >>> df
2 |      Paris   Lyon   Nantes   Pau
3 | chat      10     11     12     13
4 | singe     20     21     22     23
5 | souris    30     31     32     33

```

Sélectionnons maintenant toutes les lignes pour lesquelles les effectifs à Pau sont supérieurs à 15 :

```

1 | >>> df[ df["Pau"]>15 ]
2 |      Paris   Lyon   Nantes   Pau
3 | singe     20     21     22     23
4 | souris    30     31     32     33

```

De cette sélection, on ne souhaite garder que les valeurs pour Lyon :

```

1 | >>> df[ df["Pau"]>15 ]["Lyon"]
2 | singe      21
3 | souris    31
4 | Name: Lyon, dtype: int64

```

On peut aussi combiner plusieurs conditions avec `&` pour l'opérateur **et** :

```

1 | >>> df[ (df["Pau"]>15) & (df["Lyon"]>25) ]
2 |      Paris   Lyon   Nantes   Pau
3 | souris    30     31     32     33

```

et `|` pour l'opérateur **ou** :

```

1 | >>> df[ (df["Pau"]>15) | (df["Lyon"]>25) ]
2 |      Paris   Lyon   Nantes   Pau
3 | singe     20     21     22     23
4 | souris    30     31     32     33

```

17.4.5 Combinaison de *dataframes*

En biologie, on a souvent besoin de combiner deux tableaux de chiffres à partir d'une colonne commune.

Par exemple, si on considère les deux *dataframes* suivants :

```

1 | >>> data1 = {"Lyon": [10, 23, 17], "Paris": [3, 15, 20]}
2 | >>> df1 = pd.DataFrame.from_dict(data1)
3 | >>> df1.index = ["chat", "singe", "souris"]
4 | >>> df1
5 |      Lyon   Paris
6 | chat      10     3
7 | singe     23     15
8 | souris    17     20

```

et

```

1 | >>> data2 = {"Nantes": [3, 9, 14], "Strasbourg": [5, 10, 8]}
2 | >>> df2 = pd.DataFrame.from_dict(data2)
3 | >>> df2.index = ["chat", "souris", "lapin"]
4 | >>> df2
5 |      Nantes   Strasbourg
6 | chat        3        5
7 | souris     9        10
8 | lapin     14        8

```

On souhaite combiner ces deux *dataframes*, c'est-à-dire connaître pour les 4 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()`¹⁰ qui prend comme argument une liste de *dataframes* :

```

1 | >>> pd.concat([df1, df2])
2 |          Lyon   Nantes   Paris   Strasbourg
3 | chat      10.0      NaN     3.0      NaN
4 | singe     23.0      NaN    15.0      NaN
5 | souris    17.0      NaN    20.0      NaN
6 | chat      NaN      3.0      NaN     5.0
7 | souris   NaN      9.0      NaN    10.0
8 | lapin     NaN     14.0      NaN     8.0

```

Ici, `NaN` indique des valeurs manquantes. 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 :

```

1 | >>> pd.concat([df1, df2], axis=1)
2 |          Lyon   Paris   Nantes   Strasbourg
3 | chat      10.0     3.0     3.0     5.0
4 | lapin     NaN      NaN    14.0     8.0
5 | singe     23.0    15.0      NaN      NaN
6 | 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"` :

```

1 | >>> pd.concat([df1, df2], axis=1, join="inner")
2 |          Lyon   Paris   Nantes   Strasbourg
3 | chat      10       3       3       5
4 | 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`.

17.5 Un exemple plus complet

Pour illustrer les possibilités de *pandas*, voici un exemple plus complet.

Le fichier `transferrin_report.csv` que vous pouvez télécharger ici¹¹ contient une liste de structures de la transferrine¹². Cette protéine est responsable du transport du fer dans l'organisme.

Si vous n'êtes pas familier avec le format de fichier `.csv`, nous vous conseillons de consulter l'annexe A *Quelques formats de données rencontrés en biologie*.

Voyons maintenant comment explorer les données contenues dans ce fichier avec *pandas*.

17.5.1 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("transferrin_report.csv")
```

Le contenu est chargé sous la forme d'un *dataframe* dans la variable `df`.

Le fichier contient 41 lignes de données plus une ligne d'entête. Cette dernière est automatiquement utilisée par *pandas* pour nommer les différentes colonnes. Voici un aperçu des premières lignes :

```

1 | >>> df.head()
2 |          PDB ID      Source Deposit Date Length      MW
3 | 0      1A8E      Homo sapiens 1998-03-24    329  36408.4
4 | 1      1A8F      Homo sapiens 1998-03-25    329  36408.4
5 | 2      1AIV      Gallus gallus 1997-04-28    686  75929.0
6 | 3      1AOV Anas platyrhynchos 1996-12-11    686  75731.8
7 | 4      1B3E      Homo sapiens 1998-12-09    330  36505.5

```

Nous avons 5 colonnes de données :

— l'identifiant de la structure (PDB ID);

10. <https://pandas.pydata.org/pandas-docs/stable/merging.html>

11. https://python.sdv.univ-paris-diderot.fr/data-files/transferrin_report.csv

12. <https://fr.wikipedia.org/wiki/Transferrine>

- l'organisme d'où provient cette protéine (**Source**) ;
- la date à laquelle cette structure a été déposée dans la *Protein Data Bank* (**Deposit Date**) ;
- le nombre d'acides aminés qui constituent la protéine (**Length**) ;
- et la masse molaire de la protéine (**MW**).

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. La colonne PDB ID s'y prête très bien car cette colonne ne contient que des identifiants uniques :

```

1 | >>> df = pd.read_csv("transferrin_report.csv", index_col="PDB ID")
2 | >>> df.head()
3 |          Source Deposit Date   Length      MW
4 | PDB ID
5 | 1A8E      Homo sapiens 1998-03-24     329  36408.4
6 | 1A8F      Homo sapiens 1998-03-25     329  36408.4
7 | 1AIV     Gallus gallus 1997-04-28     686  75929.0
8 | 1AOV    Anas platyrhynchos 1996-12-11     686  75731.8
9 | 1B3E      Homo sapiens 1998-12-09     330  36505.5

```

Avant d'analyser un jeu de données, il est intéressant de l'explorer un peu. Par exemple, connaître ses dimensions :

```

1 | >>> df.shape
2 | (41, 4)

```

Notre jeu de données contient donc 41 lignes et 4 colonnes. En effet, la colonne PDB ID 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 :

```

1 | >>> df.dtypes
2 | Source        object
3 | Deposit Date    object
4 | Length       int64
5 | MW           float64
6 | dtype: object

```

Les colonnes Length et MW contiennent des valeurs numériques, respectivement des entiers (`int64`) et des *floats* (`float64`). Le type `object` est un type par défaut.

17.5.2 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 `Source`. Mais on sait par contre que la colonne `Deposit 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, *pandas* va se débrouiller pour trouver automatiquement le format de date utilisé. On peut alors explicitement demander à *pandas* de considérer la colonne `Deposit Date` comme une date :

```

1 | >>> df["Deposit Date"] = pd.to_datetime(df["Deposit Date"])

```

L'affichage des données n'est pas modifié :

```

1 | >>> df.head()
2 |          Source Deposit Date   Length      MW
3 | PDB ID
4 | 1A8E      Homo sapiens 1998-03-24     329  36408.4
5 | 1A8F      Homo sapiens 1998-03-25     329  36408.4
6 | 1AIV     Gallus gallus 1997-04-28     686  75929.0
7 | 1AOV    Anas platyrhynchos 1996-12-11     686  75731.8
8 | 1B3E      Homo sapiens 1998-12-09     330  36505.5

```

Mais le type de données de la colonne `Deposit Date` est maintenant une date (`datetime64[ns]`) :

```

1 | >>> df.dtypes
2 | Source        object
3 | Deposit Date  datetime64[ns]
4 | Length       int64
5 | MW           float64
6 | dtype: object

```

17.5.3 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()` :

```

1 | >>> df.describe()
2 |          Length           MW
3 | count    41.000000  41.000000
4 | mean     477.341463 52816.090244
5 | std      175.710217 19486.594012
6 | min      304.000000 33548.100000
7 | 25%     331.000000 36542.300000
8 | 50%     337.000000 37229.300000
9 | 75%     679.000000 75298.500000
10 | max     696.000000 77067.900000

```

On apprend ainsi que la masse moléculaire (colonne MW) a une valeur moyenne de 52816.090244 avec un écart-type de 19486.594012 et que la plus petite valeur est 33548.100000 et la plus grande 77067.900000. Pratique !

La colonne Source contient des chaînes de caractères, on peut rapidement déterminer le nombre de protéines pour chaque organisme :

```

1 | >>> df["Source"].value_counts()
2 | Homo sapiens      26
3 | Gallus gallus     10
4 | Anas platyrhynchos 2
5 | Oryctolagus cuniculus 2
6 | Sus scrofa        1
7 | Name: Source, dtype: int64

```

Ainsi, 26 protéines sont d'origine humaine (Homo sapiens) et 10 proviennent de la poule (Gallus gallus).

17.5.4 Statistiques par groupe

On peut aussi déterminer, pour chaque organisme, la taille et la masse moléculaire moyennes des transferrines :

```

1 | >>> df.groupby(["Source"]).mean()
2 |          Length           MW
3 | Source
4 | Anas platyrhynchos  686.000000  75731.800000
5 | Gallus gallus       509.300000  56324.080000
6 | Homo sapiens       439.615385  48663.392308
7 | Oryctolagus cuniculus 490.000000  54219.600000
8 | Sus scrofa          696.000000  77067.900000

```

La méthode .groupby() rassemble d'abord les données suivant la colonne Source puis la méthode .mean() calcule la moyenne pour chaque groupe.

Si on souhaite obtenir deux statistiques (par exemple la valeur minimale et maximale) en une seule fois, il convient alors d'utiliser la méthode .pivot_table() plus complexe mais aussi beaucoup plus puissante :

```

1 | >>> df.pivot_table(index="Source", values=["Length", "MW"], aggfunc=[min, max])
2 |          min           max
3 |          Length           MW
4 | Source
5 | Anas platyrhynchos   686  75731.8    686  75731.8
6 | Gallus gallus         328  36105.8    686  75957.1
7 | Homo sapiens         327  36214.2    691  76250.2
8 | Oryctolagus cuniculus 304  33548.1    676  74891.1
9 | Sus scrofa            696  77067.9    696  77067.9

```

L'argument index précise la colonne dont on agrège les données.

L'argument values indique sur quelles colonnes les statistiques sont calculées.

Enfin, aggfunc liste les statistiques calculées, ici la valeur minimale et maximale.

Notez que les valeurs renvoyées sont d'abord les valeurs minimales pour Length et MW puis les valeurs maximales pour Length et MW.

17.5.5 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 moléculaire va être élevée.

Pour vérifier cela graphiquement, on représente la masse moléculaire de la protéine en fonction de sa taille (c'est-à-dire du nombre d'acides aminés).

```

1 | >>> import matplotlib.pyplot as plt
2 | >>> plt.scatter(df["Length"], df["MW"])
3 | <matplotlib.collections.PathCollection object at 0x7f62c2501780>
4 | >>> plt.xlabel("Taille (nombre d'acides aminés)")
5 | Text(0.5, 0, "Taille (nombre d'acides aminés)")
6 | >>> plt.ylabel("Masse moléculaire (Dalton)")

```

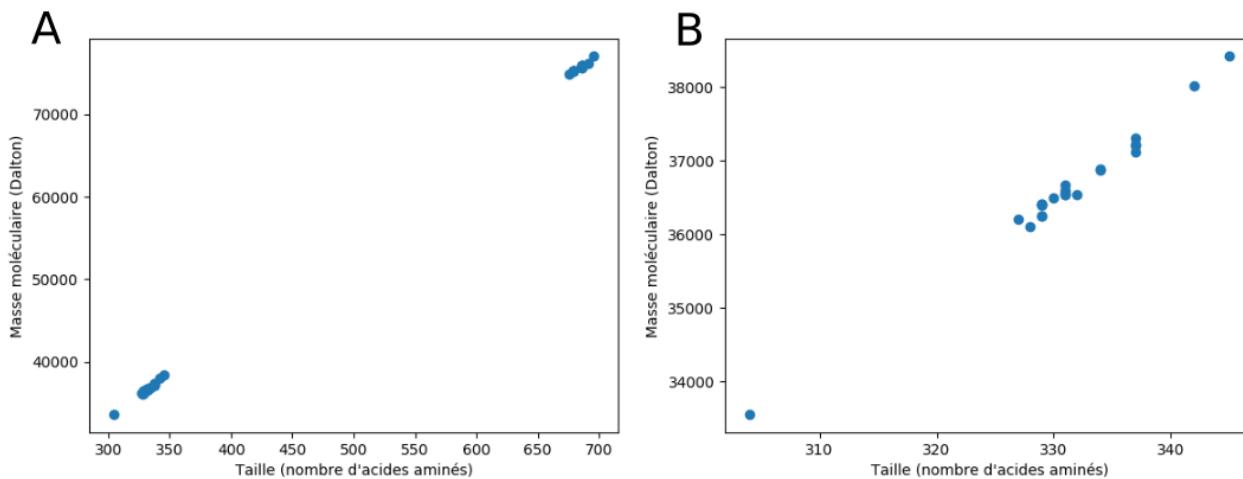


FIGURE 17.5 – (A) Masse moléculaire en fonction de la taille. (B) Zoom.

```
7 | Text(0, 0.5, 'Masse moléculaire (Dalton)')
8 | >>> plt.savefig("transferrine1.png")
```

On obtient un graphique similaire à celui de la figure 17.5 (A) avec deux groupes de points distincts (car certaines structures sont incomplètes).

On peut zoomer sur le groupe de points le plus à gauche en ne sélectionnant que les protéines constituées de moins de 400 résidus :

```
1 | >>> dfz = df[df["Length"]<400]
```

Puis en créant un deuxième graphique :

```
1 | >>> plt.clf()
2 | >>> plt.scatter(dfz["Length"], dfz["MW"])
3 | <matplotlib.collections.PathCollection object at 0x7f85bb4852e8>
4 | >>> plt.xlabel("Taille (nombre d'acides aminés)")
5 | Text(0.5, 0, "Taille (nombre d'acides aminés)")
6 | >>> plt.ylabel("Masse moléculaire (Dalton)")
7 | Text(0, 0.5, 'Masse moléculaire (Dalton)')
8 | >>> plt.savefig("transferrine2.png")
```

Ligne 1. L'instruction `plt.clf()` efface le graphe précédent mais conserve les noms des axes des abscisses et des ordonnées.

Le graphique 17.5 (B) obtenu met en évidence une relation linéaire entre le nombre de résidus d'une protéine et sa masse moléculaire.

En réalisant une régression linéaire, on détermine les paramètres de la droite qui passent le plus proche possible des points du graphique.

```
1 | >>> from scipy.stats import linregress
2 | >>> lr = linregress(dfz["Length"], dfz["MW"])
3 |
4 | LinregressResult(slope=116.18244897959184, intercept=-1871.6131972789153,
5 | rvalue=0.993825553885062, pvalue=1.664932379936294e-22,
6 | stderr=2.765423239336685)
```

Ce modèle linéaire nous indique qu'un résidu a une masse d'environ 116 Dalton, ce qui est cohérent. On peut également comparer ce modèle aux différentes protéines :

```
1 | >>> plt.clf()
2 | >>> plt.scatter(dfz["Length"], dfz["MW"])
3 | <matplotlib.collections.PathCollection object at 0x7f85b97bfef0>
4 | >>> plt.plot(dfz["Length"], dfz["Length"]*lr.slope+lr.intercept, ls=":")
5 | [<matplotlib.lines.Line2D object at 0x7f85b933e208>]
6 | >>> plt.xlabel("Taille (nombre d'acides aminés)")
7 | Text(0.5, 0, "Taille (nombre d'acides aminés)")
8 | >>> plt.ylabel("Masse moléculaire (Dalton)")
9 | Text(0, 0.5, 'Masse moléculaire (Dalton)')
10| >>> plt.savefig("transferrine3.png")
```

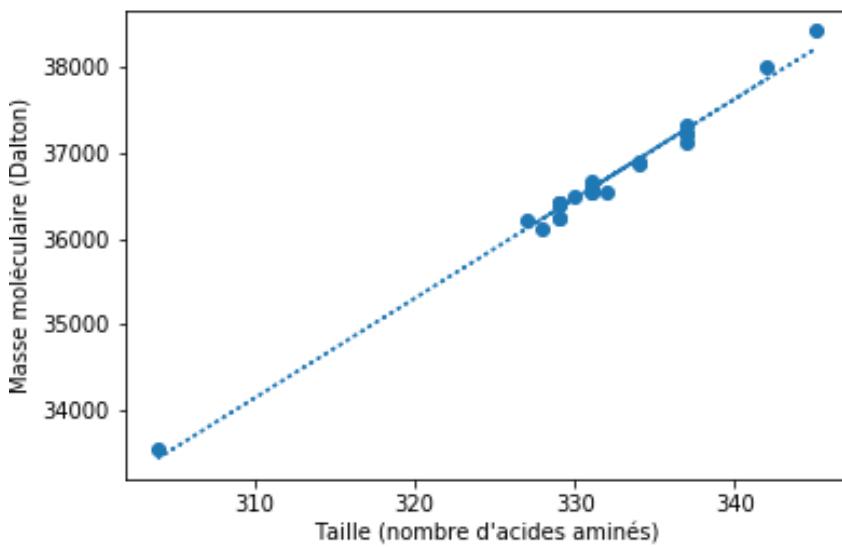


FIGURE 17.6 – Masse moléculaire en fonction de la taille (zoom) avec un modèle linéaire.

On obtient ainsi le graphique de la figure 17.6.

17.5.6 Analyse de données temporelles

Il peut être intéressant de savoir, pour chaque organisme, quand les premières et les dernières structures de transferrines ont été déposées dans la PDB.

La méthode `.pivot_table()` apporte un élément de réponse :

```

1 >>> df.pivot_table(index="Source", values=["Deposit Date"], aggfunc=[min, max])
2
3           min          max
4      Source Deposit Date Deposit Date
5 Anas platyrhynchos    1995-08-03    1996-12-11
6 Gallus gallus        1993-09-15    2005-09-28
7 Homo sapiens         1992-02-10    2018-03-22
8 Oryctolagus cuniculus 1990-08-16    2001-07-24
9 Sus scrofa            2001-07-03    2001-07-03

```

Chez l'homme (*Homo sapiens*), la première structure de transferrine a été déposée dans la PDB le 10 février 1992 et la dernière le 22 mars 2018.

Une autre question est de savoir combien de structures de transferrines ont été déposées en fonction du temps.

La méthode `.value_counts()` peut être utilisée mais elle ne renvoie que le nombre de structures déposées dans la PDB pour un jour donné. Par exemple, deux structures ont été déposées le 4 septembre 2000.

```

1 >>> df["Deposit Date"].value_counts().head()
2 1999-01-07    2
3 2000-09-04    2
4 2002-11-18    2
5 2003-03-10    1
6 2001-07-24    1
7 Name: Deposit Date, dtype: int64

```

Si on souhaite une réponse plus globale, par exemple, à l'échelle de l'année, la méthode `.resample()` calcule le nombre de structures déposées par année (en fournissant l'argument A) :

```

1 >>> df["Deposit Date"].value_counts().resample("A").count()
2 1990-12-31    1
3 1991-12-31    0
4 1992-12-31    1
5 1993-12-31    1

```

Les dates apparaissent maintenant comme le dernier jour de l'année mais désignent bien l'année complète. Dans cet exemple, une seule structure de transferrine a été déposée dans la PDB entre le 1er janvier 1990 et le 31 décembre 1990.

Pour connaître en quelle année le plus de structures ont été déposées dans la PDB, il faut trier les valeurs obtenus 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 dépôts), on utilisera également la méthode `.head()` :

```

1 | >>> (df["Deposit Date"].value_counts()
2 | ... .resample("A")
3 | ... .count()
4 | ... .sort_values(ascending=False)
5 | ... .head())
6 | 2001-12-31    5
7 | 2003-12-31    4
8 | 1998-12-31    3
9 | 1999-12-31    3
10 | 2002-12-31   3
11 | Name: Deposit Date, dtype: int64

```

En 2001, cinq structures de transferrine ont été déposées dans la PDB. La deuxième « meilleure » année est 2003 avec quatre structures.

Toutes ces méthodes, enchaînées les unes à la suite des autres, peuvent vous sembler complexes mais chacune d'elles correspond à une étape particulière du traitement des données. L'utilisation des parenthèses (ligne 1, juste avant `df["Deposit Date"]` et ligne 5, juste après `head()`) permet de répartir élegamment cette longue instruction sur plusieurs lignes.

Bien sûr, on aurait pu créer des variables intermédiaires pour chaque étape mais cela aurait été plus lourd :

```

1 | >>> date1 = df["Deposit Date"].value_counts()
2 | >>> date2 = date1.resample("A")
3 | >>> date3 = date2.count()
4 | >>> date4 = date3.sort_values(ascending=False)
5 | >>> date4.head()
6 | 2001-12-31    5
7 | 2003-12-31    4
8 | 1998-12-31    3
9 | 1999-12-31    3
10 | 2002-12-31   3
11 | Name: Deposit Date, dtype: int64

```

Pour aller plus loin

- Le livre de Nicolas Rougier *From Python to Numpy*¹³ est une excellente ressource pour explorer plus en détails les possibilités de NumPy.
- Les ouvrages *Python for Data Analysis* de Wes McKinney et *Pandas Cookbook* de Theodore Petrou sont d'excellentes références pour pandas.

17.6 Exercices

La barstar est un inhibiteur de ribonucléase. C'est une protéine relativement simple qui contient 89 acides aminés. Sa structure tridimensionnelle, obtenue par cristallographie aux rayons X, se trouve dans la *Protein Data Bank* (PDB) sous le code 1BTA.

17.6.1 Distance entre deux atomes carbones alpha consécutifs de la barstar

L'objectif de cet exercice est de calculer la distance entre carbones alpha consécutifs le long de la chaîne peptidique avec module *NumPy*.

Extraction des coordonnées atomiques

Téléchargez le fichier `1bta.pdb` qui correspond à la structure de la barstar¹⁴ sur le site de la PDB (lien direct vers le fichier¹⁵).

Voici le code pour extraire les coordonnées atomiques des carbones alpha de la barstar :

13. <https://www.labri.fr/perso/nrougier/from-python-to-numpy/>
 14. <http://www.rcsb.org/pdb/explore.do?structureId=1BTA>
 15. <https://files.rcsb.org/download/1BTA.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("{} {} {} ".format(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.

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.

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.

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 (-, +, $\ast\ast 2$) 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.

17.6.2 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 barstar publiés chaque année.

Vous utiliserez les modules `Biopython` et `matplotlib`.

Requête avec un mot-clé

Sur le site de PubMed¹⁶, 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 ?

Récupération des informations d'une publication

Récupérez les informations de la publication dont le PMID est 29701945¹⁷. 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.

16. <https://www.ncbi.nlm.nih.gov/pubmed/>

17. <https://www.ncbi.nlm.nih.gov/pubmed/29701945>

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. Combien de caractères contient-il ?

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 bien 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 dans votre boucle afficher un message qui indique où vous en êtes dans la récupération des articles.

À la fin 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 valeurs le nombre de publications associées à une année données.

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 années. 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`, vous allez pouvoir afficher la distribution des publications en fonction des années :

```
1| import matplotlib.pyplot as plt
2| plt.bar(x, y)
3| plt.show()
```

Vous pouvez également ajouter un peu de cosmétique et enregistrer le graphique sur votre disque dur :

```
1| import matplotlib.pyplot as plt
2|
3| plt.bar(x, y)
4|
5| # redéfinition des valeurs affichées sur l'axe des ordonnées
6| plt.yticks(list(range(0, max(y), 2)))
7|
8| # étiquetage des axes et du graphique
9| plt.xlabel("Années")
10| plt.ylabel("Nombre de publications")
11| plt.title("Distribution des publications qui mentionnent la barstar")
12|
13| # enregistrement sur le disque
14| plt.savefig('distribution_barstar_annee.png', bbox_inches='tight', dpi=200)
```

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 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 son nom), 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 (notées avec In []).

18.1 Installation

Avec la distribution Miniconda, les notebooks Jupyter s'installent avec la commande :

```
| $ conda install -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 Lancement de Jupyter et création d'un notebook

Pour lancer les notebooks Jupyter, utilisez la commande suivante depuis un *shell* :

```
| $ jupyter notebook
```

Une nouvelle page devrait s'ouvrir dans votre navigateur web et ressembler à la figure 18.2.

Cette interface liste les notebooks Jupyter existants (pour le moment aucun).

Pour créer un notebook, cliquez sur le bouton à droite *New* puis sélectionnez *Python 3*. Vous noterez au passage qu'il est également possible de créer un fichier texte, un répertoire ou bien encore de lancer un *shell* via un *Terminal* (voir 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 en cliquant sur *Untitled*, à droite du logo de Jupyter. Si le nom de votre notebook est *test* alors le fichier *test.ipynb* sera créé dans le répertoire depuis lequel vous avez lancé Jupyter.

Remarque

L'extension *.ipynb* est l'extension de fichier des notebooks Jupyter.

Vous pouvez entrer des instructions Python dans la première cellule. Par exemple :

```
| 1| a = 2
| 2| b = 3
| 3| print(a+b)
```

Exemple de notebook Jupyter

① Cette cellule contient du texte formaté en Markdown.
On peut ajouter du texte en **gras** ou bien en *italique*.

In [1]: # cette cellule contient du code Python
mais qui est exécuté
print("Hello Python !")
② Hello Python !

In [2]: # une autre cellule avec du code Python
mais qui ne renvoie rien
def ma_fonction(x):
 return x + 2
③

In [3]: # une autre cellule avec du code Python
mais qui renvoie quelque chose
même si la fonction print() n'est pas utilisée
ce comportement ressemble à celui de l'interpréteur Python
ma_fonction(3)
④

Out[3]: 5
⑤ Encore du texte
avec une équation :
$$\prod_{i=1}^n i = n!$$

FIGURE 18.1 – Exemple de notebook Jupyter. Les chiffres entourés désignent les différentes cellules.

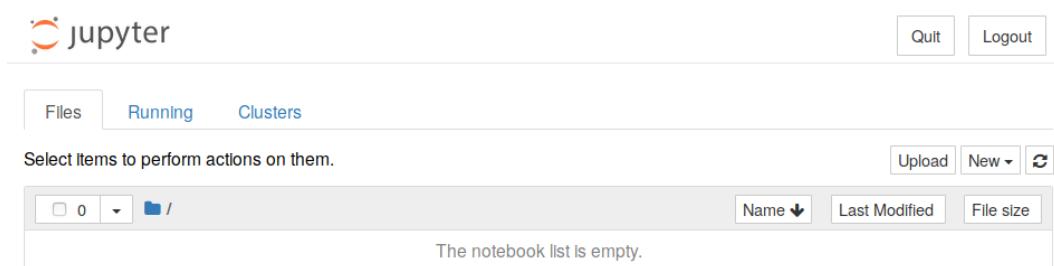


FIGURE 18.2 – Interface de Jupyter.

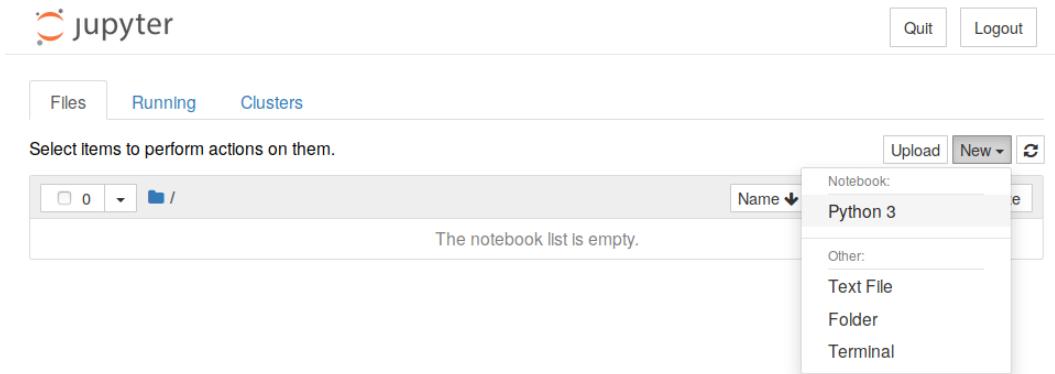


FIGURE 18.3 – Création d'un nouveau notebook.

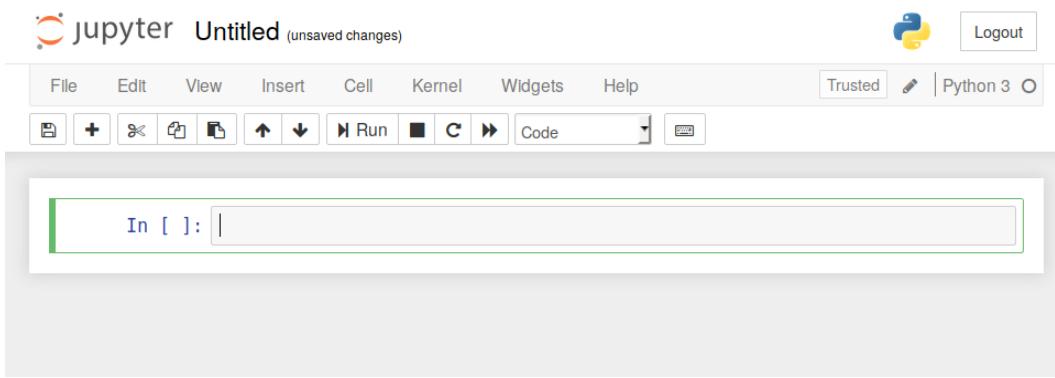


FIGURE 18.4 – Nouveau notebook.

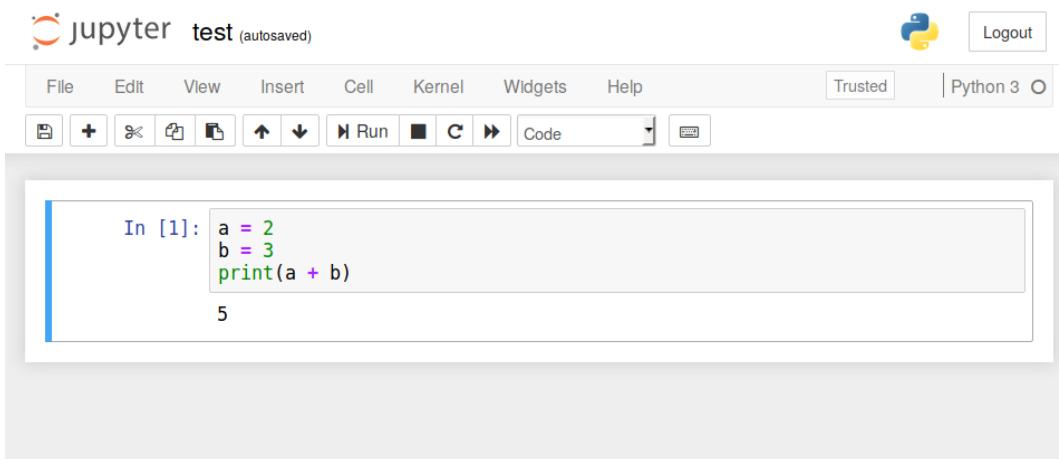


FIGURE 18.5 – Exécution d'une première cellule.

Pour exécuter le contenu de cette cellule, vous avez plusieurs possibilités :

- Cliquer sur le menu *Cell*, puis *Run Cells*.
- Cliquer sur le bouton *Run* (sous la barre de menu).
- Presser simultanément les touches *Ctrl + Entrée*.

Dans tous les cas, vous devriez obtenir quelque chose qui ressemble à l'image 18.5. La notation *In [1]* à gauche de la cellule indique qu'il s'agit de la première cellule exécutée.

Pour créer une nouvelle cellule, vous avez, ici encore, plusieurs possibilités :

- Cliquer sur l'icône *+* sous la barre de menu.
- Cliquer sur le menu *Insert*, puis *Insert Cell Below*.

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 passé de *In [1]* à *In [2]* car vous avez exécuté la première cellule *puis* créez 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. Vous remarquerez que pour les cellules 3 et 4, le résultat renvoyé par le code Python est précédé par *Out [3]* et *Out [4]*.

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 *In [5]* (voir figure 18.7).

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 d'une cellule à l'autre.

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 & Run All* et enfin de valider le message *Restart and Run All Cells*.

18.3 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 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 une notebook Jupyter (figure 18.8) et le rendu lorsque la cellule est

The screenshot shows a Jupyter Notebook interface with the title "jupyter test (unsaved changes)". The toolbar includes File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted, Python 3, and Logout. Below the toolbar are standard notebook controls: New, Open, Save, Run, Cell, Kernel, Help, and Code dropdown. The main area contains four code cells:

- In [1]:

```
a = 2
b = 3
print(a + b)
```

Output: 5
- In [2]:

```
def ma_fonction(x, y):
    return x + y
```
- In [3]:

```
ma_fonction(a, 10)
```

Output: 12
- In [4]:

```
ma_fonction("Bonjour", "Jupyter")
```

Output: 'BonjourJupyter'

FIGURE 18.6 – Notebook avec plusieurs cellules de code Python.

The screenshot shows a Jupyter Notebook interface with the title "jupyter test (autosaved)". The toolbar and controls are identical to Figure 18.6. The main area contains the same four code cells as Figure 18.6, but the fourth cell (In [4]) has a blue selection bar on its left, indicating it is currently selected or being edited.

FIGURE 18.7 – Notebook avec une cellule ré-exécutée.

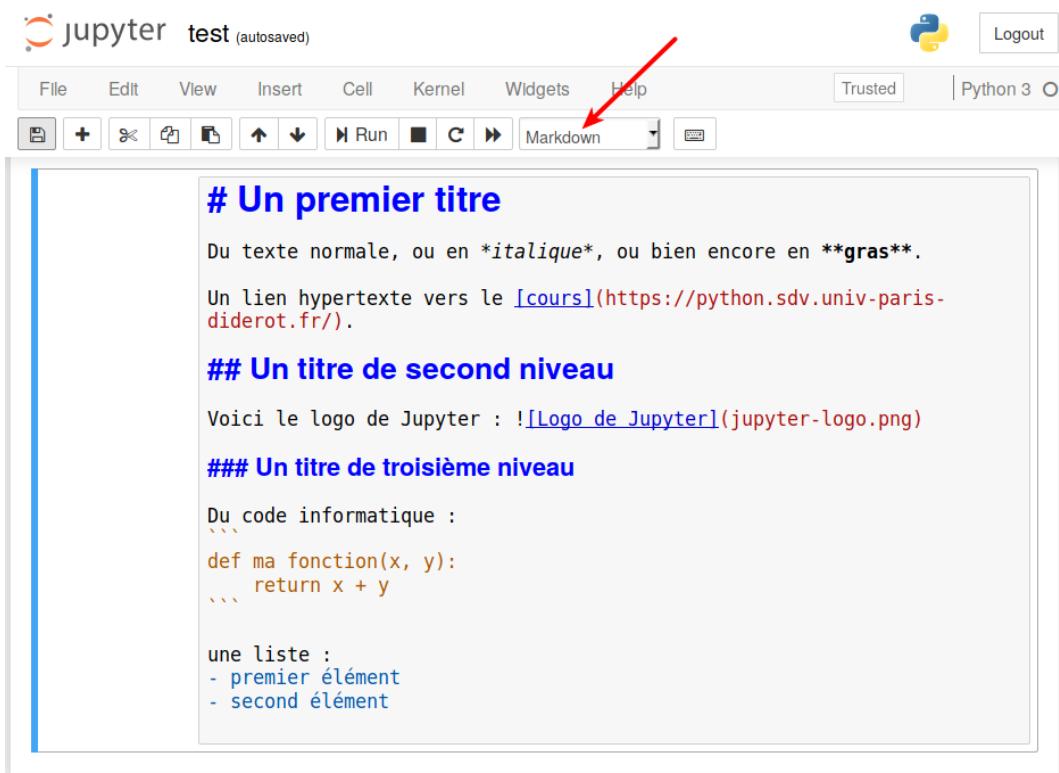


FIGURE 18.8 – Notebook avec une cellule au format Markdown.

exécutée (figure 18.9).

Notez qu'une cellule Markdown n'a pas le marqueur In [] à sa gauche.

Le format Markdown permet de rapidement et très simplement rédiger du texte structuré. Ce cours est par exemple complètement rédigé en Markdown ;-)

Nous vous conseillons d'explorer les possibilités du Markdown en consultant la page Wikipédia¹ ou directement la page de référence².

18.4 Des graphiques dans les notebooks

Un autre intérêt des notebooks Jupyter est de pouvoir y incorporer des graphiques réalisés avec la bibliothèque *matplotlib*.

Voici un exemple en reprenant un graphique présenté dans le chapitre 17 *Quelques modules d'intérêt en bioinformatique* (figure 18.10).

La différence notable est l'utilisation de la commande :

`%matplotlib inline`

qui n'est à lancer qu'une seule fois (en général dans la première cellule du notebook) et qui permet l'incorporation de figures dans un notebook Jupyter.

Remarque

Pour quitter l'interface des notebooks Jupyter, il faut, dans le premier onglet qui est apparu, cliquer sur le bouton *Quit* (figure 18.2).

Une méthode plus radicale est de revenir sur le *shell* depuis lequel les notebooks Jupyter ont été lancés puis de presser deux fois la combinaison de touches *Ctrl + C*.

1. <https://fr.wikipedia.org/wiki/Markdown>

2. <https://daringfireball.net/projects/markdown/syntax>

The screenshot shows a Jupyter Notebook interface. At the top, there's a toolbar with File, Edit, View, Insert, Cell, Kernel, Widgets, Help, Trusted (checkbox), Python 3 (radio button), and Logout. Below the toolbar is a menu bar with icons for file operations like Open, Save, New, and Run, along with a dropdown for Cell type (Cell, Text, Code, Markdown) and a Help icon.

The main content area contains a single Markdown cell. The cell displays the following text:

Un premier titre
Du texte normale, ou en *italique*, ou bien encore en **gras**.
Un lien hypertexte vers le [cours](#).

Un titre de second niveau
Voici le logo de Jupyter :



Un titre de troisième niveau
Du code informatique :

```
def ma fonction(x, y):  
    return x + y
```

une liste :

- premier élément
- second élément

FIGURE 18.9 – Notebook avec une cellule au format Markdown (après exécution).

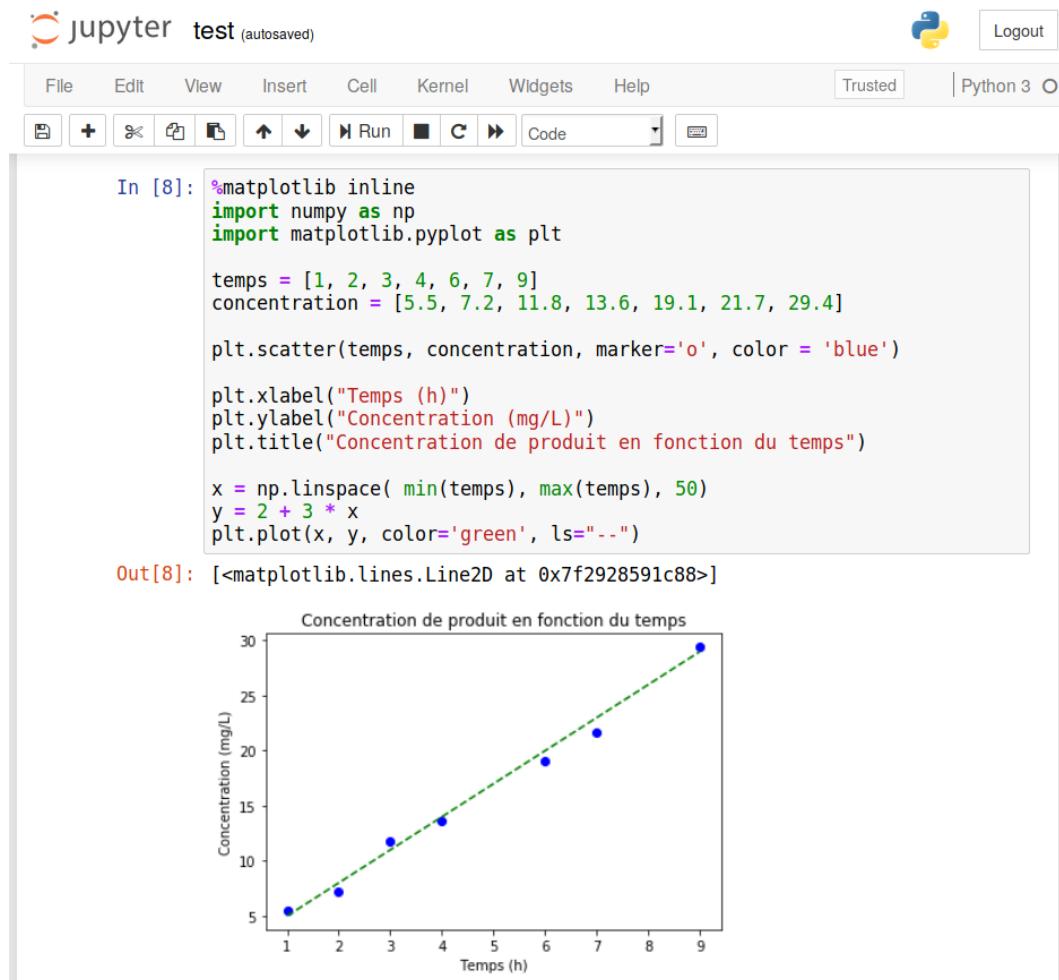


FIGURE 18.10 – Incorporation d'un graphique dans un notebook Jupyter.

In [9]:	%whos		
Variable	Type	Data/Info	
a	int	2	
b	int	3	
concentration	list	n=7	
ma_fonction	function	<function ma_fonction at 0x7f2950b46048>	
np	module	<module 'numpy' from '/ho...>kages(numpy/_init_.py'>	
plt	module	<module 'matplotlib.pyplot'>es/matplotlib	
/pyplot.py'>			
temps	list	n=7	
x	ndarray	50: 50 elems, type `float64`, 400 bytes	
y	ndarray	50: 50 elems, type `float64`, 400 bytes	

FIGURE 18.11 – Magic command %whos.

18.5 Les *magic commands*

La commande précédente (%matplotlib inline) est une *magic command*. Il en existe beaucoup, en voici deux :

— %whos liste tous les objets (variables, fonctions, modules...) utilisés dans le notebook (voir figure 18.11).

— %history liste toutes les commandes Python lancées dans un notebook (voir figure 18.12).

Enfin, avec les environnements Linux ou Mac OS X, il est possible de lancer une commande Unix depuis un notebook Jupyter. Il faut pour cela précéder la commande du symbole !. La figure 18.13 illustre cette possibilité avec la commande ls qui affiche le contenu d'un répertoire.

Remarque

Le lancement de n'importe quelle commande Unix depuis un notebook Jupyter (en précédant cette commande de !) est une fonctionnalité très puissante.

Pour vous en rendre compte, jetez un œil au notebook³ produit par les chercheurs Zichen Wang et Avi Ma'ayan qui reproduit l'analyse complète de données obtenues par séquençage haut débit. Ces analyses 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*⁴ » (F1000 Research, 2016).

18.6 JupyterLab

En 2018, le consortium Jupyter a lancé *JupyterLab* qui est un environnement complet d'analyse. Pour obtenir cette interface, lancez la commande suivante depuis un *shell* :

```
1| $ jupyter lab
```

Une nouvelle page devrait s'ouvrir dans votre navigateur web et vous devriez obtenir une interface similaire à la figure 18.14.

L'interface proposée par JupyterLab est très riche. On peut y organiser un notebook Jupyter « classique » avec une figure en encart, un *shell* (voir figure 18.15)... Les possibilités sont infinies !

3. <https://github.com/MaayanLab/Zika-RNAseq-Pipeline/blob/master/Zika.ipynb>

4. <https://f1000research.com/articles/5-1574/>

```
In [7]: %history
a = 2
b = 3
print(a + b)
def ma_fonction(x, y):
    return x + y
ma_fonction(a, 10)
ma_fonction("Bonjour", "Jupyter")
%matplotlib inline
import numpy as np
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]

plt.scatter(temps, concentration, marker='o', color = 'blue')

plt.xlabel("Temps (h)")
plt.ylabel("Concentration (mg/L)")
plt.title("Concentration de produit en fonction du temps")

x = np.linspace( min(temps), max(temps), 50)
y = 2 + 3 * x
plt.plot(x, y, color='green', ls="--")
%whos
%history
```

FIGURE 18.12 – Magic command %history.

```
In [12]: !ls
jupyter-exemple.ipynb  jupyter-logo.png  test.ipynb
```

FIGURE 18.13 – Lancement d'une commande Unix.

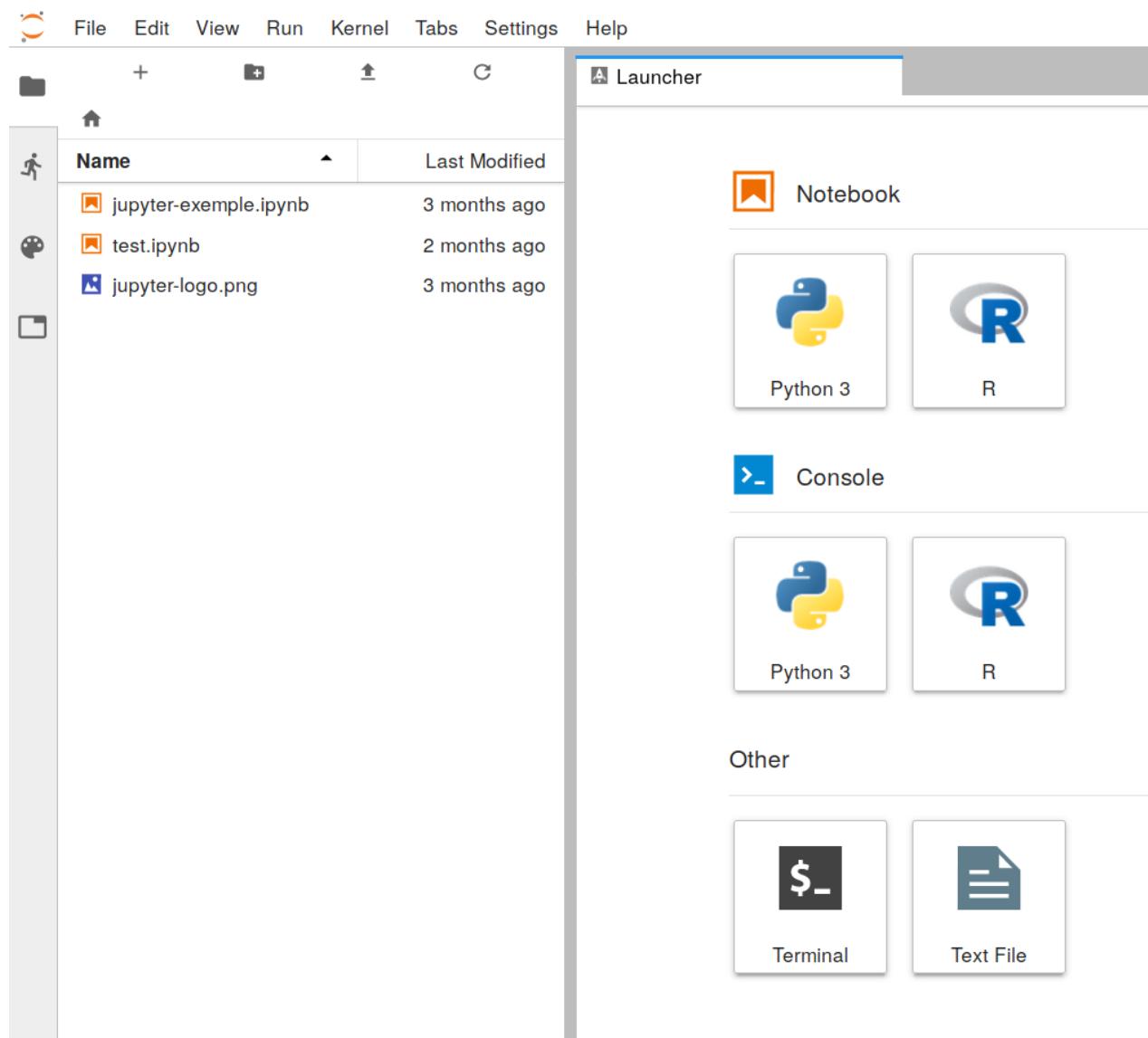


FIGURE 18.14 – Interface de JupyterLab.

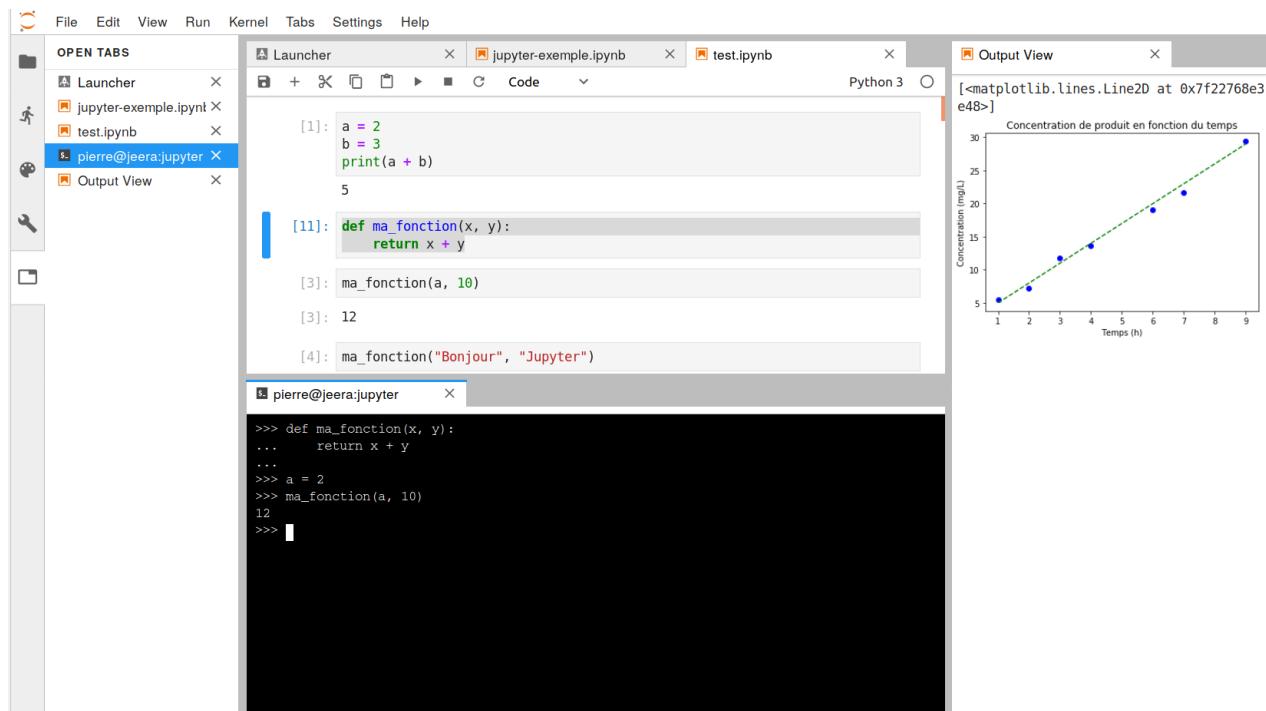


FIGURE 18.15 – JupyterLab comme environnement d’analyse.

Chapitre 19

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*. Pareil pour les *float* et *string*. Mais également 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 rédiger du code plus compact et mieux ré-utilisable. L'utilisation de classes évite l'utilisation de variables globales en créant ce qu'on appelle un *espace de noms* propre à chaque objet 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, nous avons déjà vu ces mots vous en souvenez-vous ?), 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 / algorithmes. 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 UML¹.

Dans ce chapitre nous allons vous présenter une introduction à la POO avec Python. Nous allons vous donner tous les éléments pour démarrer la construction de vos premières classes.

Après la lecture de ce chapitre, 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 ce chapitre avant d'aborder le chapitre 20 *Fenêtres graphiques et Tkinter*.

19.1 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.

1. [https://fr.wikipedia.org/wiki/UML_\(informatique\)](https://fr.wikipedia.org/wiki/UML_(informatique))

19.1.1 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 15 *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 | ...
5 | >>> Citron
6 | <class '__main__.Citron'>
7 | >>> type(Citron)
8 | <class 'type'>
9 | >>> citron1 = Citron()
10 | >>> citron1
11 | <__main__.Citron object at 0x7ff2193a20f0>
>>>
```

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 sont 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 interne `isinstance()` :

```

1 | >>> isinstance(citron1, Citron)
2 | True
```

19.1.2 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 caractère point qui lie les deux.

Lignes 4 à 6. La fonction `dir()` nous montre que l'attribut `.couleur` a bien été ajouté à l'objet.

Lignes 7. La notation `instance.attribut` donne accès à l'attribut de l'objet, de même que l'on utilisait par exemple `math.pi` avec les modules.

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 est spécifique à cette instance. 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 vous-même.

19.1.3 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 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| if __name__ == "__main__":
5|     citron1 = Citron()
6|     print(citron1.couleur)
7|     citron2 = Citron()
8|     print(citron2.couleur)
9|

```

L'exécution de ce code affichera :

```

1| jaune
2| jaune

```

Attention

Même si on peut retrouver un attribut de classe avec une syntaxe `instance.attribut`, un tel attribut ne peut pas être modifié avec une syntaxe `instance.attribut = nouvelle_valeur` (voir la rubrique *Différence entre les attributs de classe et d'instance*).

19.1.4 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 6 et 7. 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 recuper_saveur(self):
3         return "acide"
4
5
6 if __name__ == "__main__":
7     citron1 = Citron()
8     saveur_citron1 = citron1.recuper_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`.

19.1.5 Le constructeur

Lors de l'instanciation d'un objet à partir d'une classe, il peut être intéressant de lancer certaines instructions comme par exemple 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

2. <http://www.pythontutor.com>

Python 3.6

```

1 class Citron:
2     def __init__(self):
3         self.couleur = "jaune"
4
5
6 if __name__ == '__main__':
7     citron1 = Citron()
8     print(citron1.couleur)

```

[Edit this code](#)

line that has just executed
next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<< First < Back Step 2 of 7 Forward > Last >>

FIGURE 19.1 – Fonctionnement d'un constructeur (étape 1).

Python 3.6

```

1 class Citron:
2     def __init__(self):
3         self.couleur = "jaune"
4
5
6 if __name__ == '__main__':
7     citron1 = Citron()
8     print(citron1.couleur)

```

[Edit this code](#)

line that has just executed
next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<< First < Back Step 4 of 7 Forward > Last >>

FIGURE 19.2 – Fonctionnement d'un constructeur (étape 2).

utilisé dans les chapitres 9 et 12 sur les fonctions). N'hésitez pas à copier/coller ce petit 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 (voir figure 19.1). 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 (voir figure 19.2). 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 en cours (instance que nous appellerons `citron1` de retour 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.

Étape 3 (voir figure 19.3). 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

The screenshot shows the Python 3.6 code and the corresponding memory state in Python Tutor.

```

Python 3.6
1 class Citron:
2     def __init__(self):
3         self.couleur = "jaune"
4
5
6 if __name__ == '__main__':
7     citron1 = Citron()
8     print(citron1.couleur)

```

Frames:

- Global frame: Contains the `Citron` class definition.
- Citron instance: Contains the `__init__` method and its return value `None`.

Objects:

- `Citron class`: Contains `__init__` and a link to the `__init__` method in the `Citron` class.
- `Citron instance`: Contains the attribute `couleur` with the value `"jaune"`.

Print output: Shows the output `jaune`.

Execution status: Step 6 of 7. Breakpoints: Line 6 (next line to execute).

FIGURE 19.3 – Fonctionnement d'un constructeur (étape 3).

The screenshot shows the Python 3.6 code and the corresponding memory state in Python Tutor.

```

Python 3.6
1 class Citron:
2     def __init__(self):
3         self.couleur = "jaune"
4
5
6 if __name__ == '__main__':
7     citron1 = Citron()
8     print(citron1.couleur)

```

Frames:

- Global frame: Contains the `Citron` class definition.
- Citron instance: Contains the `__init__` method and its return value `None`.

Objects:

- `Citron class`: Contains `__init__` and a link to the `__init__` method in the `Citron` class.
- `Citron instance`: Contains the attribute `couleur` with the value `"jaune"`.

Print output: Shows the output `jaune`.

Execution status: Program terminated. Breakpoints: None.

FIGURE 19.4 – Fonctionnement d'un constructeur (étape 4).

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` (voir l'exemple de la figure 19.5). 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.

Étape 4 (voir figure 19.4). 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`.

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.

19.1.6 Passage d'argument(s) à linstanciation

Lors de linstanciation, 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 en veut (voir chapitre 9 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'}

```

19.1.7 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 lattribut couleur que l'on accroche à linstance avec le self.

Ligne 4. Nous créons cette fois-ci une variable var sans laccrocher au 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 lobjet ou instance créé(e). On va voir plus bas ce qu'elle contient exactement.

Attention

On peut appeler cette référence comme on veut, toutefois nous vous conseillons vivement de lappeler self car cest une convention générale 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 lattribut de classe .couleur ainsi que la variable var créée dans le constructeur .__init__().

L'exécution de ce code donnera :

```

1 $ python classe_exemple1.py
2 <__main__.Citron object at 0x7f4e5fb71438>
3 jaune
4 Traceback (most recent call last):
5   File "classe_exemple1.py", line 14, in <module>
6     citron1.affiche_attributs()
7   File "classe_exemple1.py", line 9, in affiche_attributs
8     print(var)
9   NameError: name 'var' is not defined

```

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 (cf. chapitres 9 et 12). 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. On voit que 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.

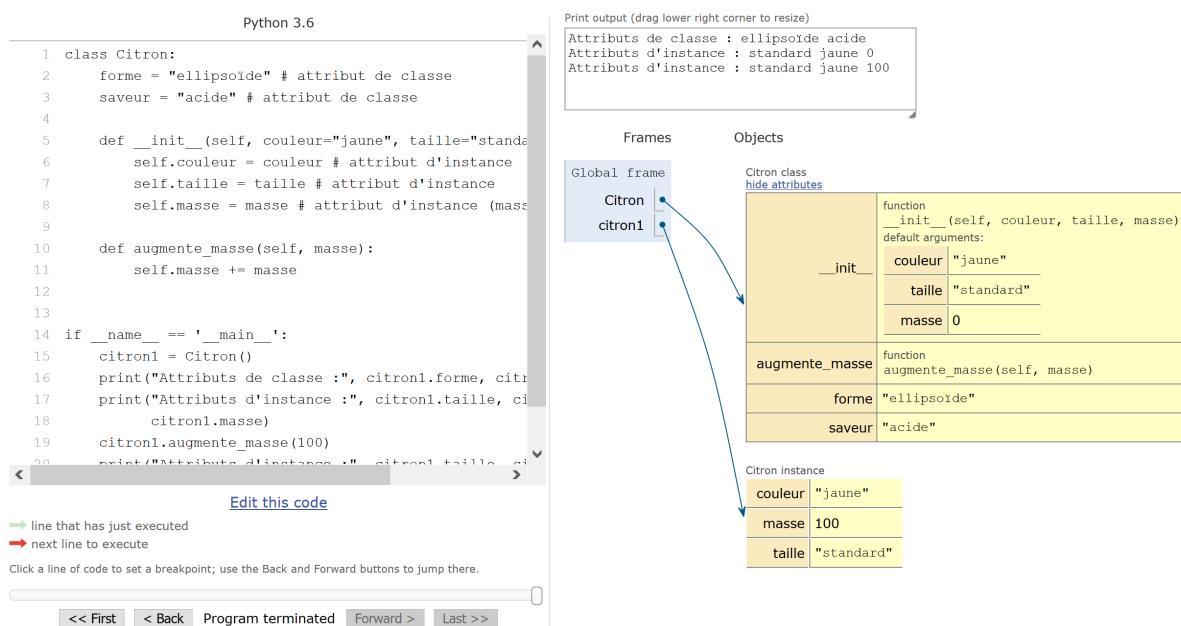
Ligne 11. On crée l'instance `citron1` en lui passant l'argument `"jaune pâle"`. La variable d'instance `couleur` prendra ainsi cette valeur au lieu de celle par défaut (`"jaune"`). À noter, l'instanciation affichera le message `Le citron c'est trop bon !` puisque la méthode `.affiche_attributs()` est 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*.

19.1.8 Différence entre les attributs de classe et d'instance

On a vu ci-dessus 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`.

FIGURE 19.5 – Illustration de la signification des attributs de classe et d'instance avec *Python Tutor*.

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 :

```

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 aumente_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.aumente_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 `.ajoute_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 19.5 montre l'état des variables après avoir exécuté ce code grâce au site *Python Tutor*³ dans .

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` 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`.

3. <http://www.pythontutor.com>

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("attribut de classe: {}, attribut d'instance: {}"
9             .format(self.saveur, self.couleur))
10
11 if __name__ == "__main__":
12     citron1 = Citron()
13     citron1.affiche_attributs()

```

Ce code va afficher la phrase :

```
1 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 if __name__ == "__main__":
5     citron1 = Citron()
6     print(citron1.saveur)
7     citron1.saveur = "sucrée"
8     print(citron1.saveur) # on regarde ici avec Python Tutor
9     del citron1.saveur
10    print(citron1.saveur) # on regarde ici avec Python Tutor
11    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 19.7 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 Tutor* nous montre que la notation `citron1.saveur` pointe maintenant vers l'espace `Citron instance` qui est vide ; ainsi, Python utilisera alors l'attribut de classe `.saveur` qui contient toujours la valeur `acide` (cf. figure 19.7).

La ligne 11 va tenter de détruire l'attribut de classe `.saveur`. Toutefois, Python interdit cela, ainsi l'erreur suivante sera générée :

```

1 Traceback (most recent call last):
2   File "./test.py", line 10, in <module>
3       del(citron1.saveur)
4 AttributeError: saveur

```

En fait, la seule manière de modifier un attribut de classe est d'utiliser une syntaxe
`NomClasse.attribut_de_classe = nouvelle_valeur`,

Python 3.6

```

1 class Citron:
2     saveur = "acide"
3
4 if __name__ == '__main__':
5     citron1 = Citron()
6     print(citron1.saveur)
7     citron1.saveur = "sucrée"
8     print(citron1.saveur) # on regarde ici avec Python tutor
9     del citron1.saveur
10    print(citron1.saveur) # on regarde ici avec Python tutor
11    del citron1.saveur

```

[Edit this code](#)

→ line that has just executed
→ next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<< First < Back Step 7 of 9 Forward > Last >>

Print output (drag lower right corner to resize)
acide
sucrée

Frames Objects

Global frame

Citron class
hide attributes

citron1

citron1

Citron instance

saveur "acide"

saveur "sucrée"

FIGURE 19.6 – Illustration avec *Python Tutor* de la non destruction d'un attribut de classe (étape 1).

Python 3.6

```

1 class Citron:
2     saveur = "acide"
3
4 if __name__ == '__main__':
5     citron1 = Citron()
6     print(citron1.saveur)
7     citron1.saveur = "sucrée"
8     print(citron1.saveur) # on regarde ici avec Python tutor
9     del citron1.saveur
10    print(citron1.saveur) # on regarde ici avec Python tutor
11    del citron1.saveur

```

[Edit this code](#)

→ line that has just executed
→ next line to execute

Click a line of code to set a breakpoint; use the Back and Forward buttons to jump there.

<< First < Back Step 9 of 9 Forward > Last >>

Print output (drag lower right corner to resize)
acide
sucrée
acide

Frames Objects

Global frame

Citron class
hide attributes

citron1

citron1

Citron instance

saveur "acide"

FIGURE 19.7 – Illustration avec *Python Tutor* de la non destruction d'un attribut de classe (étape 3).

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__()`.

19.2 Espace de noms

Nous faisons ici une petite digression sur le concept d'**espace de noms**, car il est important de bien le comprendre lorsqu'on étudie les classes. Nous avons déjà croisé ce concept à plusieurs reprises. D'abord dans le chapitre 12 *Plus sur les fonctions*, puis dans le chapitre 14 *Création de modules*, et maintenant dans ce chapitre. De quoi s'agit-il ?

Définition

Dans la documentation officielle⁴, 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 9 et 12 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 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.

19.2.1 Rappel sur la règle LGI

Comme vu dans le chapitre 9 *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*⁵ et des *Built-in Constants*⁶). Si aucun objet n'est trouvé, Python renvoie une erreur.

19.2.2 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
```

4. <https://docs.python.org/fr/3/tutorial/classes.html#python-scopes-and-namespaces>

5. [https://docs.python.org/fr/3/library/functions.html%20comme%20par%20exemple%20%60print\(\)%60](https://docs.python.org/fr/3/library/functions.html%20comme%20par%20exemple%20%60print()%60)

6. <https://docs.python.org/fr/3/library/constants.html>

```

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 :

```

1 $ python ./test_var_module.py
2 Dans prog principal i: 1000000
3 Dans prog principal j: 2
4 Dans module, i local: -27478524
5 Dans module, j global: -76
6 Dans prog principal i: 1000000
7 Dans prog principal j: 2

```

Lignes 2 et 3. On a bien les valeurs de `i` et `j` définies dans le programme principal de `test.py`.

Lignes 4 et 5. 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 6 et 7. 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`.

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 la méthode 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.

19.2.3 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 :

```

1 class Citron:
2     def __init__(self, saveur="acide", couleur="jaune"):
3         self.saveur = saveur
4         self.couleur = couleur
5         print("Dans __init__(), vous venez de créer un citron:",
6             self.affiche_attributs())
7
8     def affiche_attributs(self):
9         return "{}{}, {}".format(self.saveur, self.couleur)
10
11
12 if __name__ == "__main__":
13     saveur = "sucrée"
14     couleur = "orange"
15     print("Dans prog principal: {}, {}".format(saveur, couleur))
16     citron1 = Citron("très acide", "jaune foncé")
17     print("Dans citron1.affiche_attributs():", citron1.affiche_attributs())
18     print("Dans prog principal: {}, {}".format(saveur, couleur))

```

Lorsqu'on exécutera ce code, on obtiendra :

```

1| Dans prog principal: sucrée, orange
2| Dans __init__(), vous venez de créer un citron: jaune foncé, très acide
3| Dans citron1.affiche_attributs(): jaune foncé, très acide
4| Dans prog 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 :

```

1| class Citron:
2|     def __init__(self):
3|         self.couleur = "jaune"
4|         self.affiche_coucou()
5|         affiche_coucou()
6|
7|     def affiche_coucou(self):
8|         print("Coucou interne !")
9|
10| def affiche_coucou():
11|     print("Coucou externe")
12|
13|
14|
15| if __name__ == "__main__":
16|     citron1 = Citron()
17|     citron1.affiche_coucou()
18|     affiche_coucou()

```

Lorsqu'on va exécuter le code, on obtiendra :

```

1| Coucou interne !
2| Coucou externe
3| Coucou interne !
4| 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 9 et 12 sur les fonctions, sont à proscrire absolument. Tout cela concourt à rendre le code plus lisible.

Dans le chapitre 20 *Fenêtres graphiques et Tkinter*, vous verrez une démonstration de l'utilité de tout encapsuler dans une classe afin d'éviter les variables globales.

19.2.4 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**.

19.2.5 Pour aller plus loin

Il existe d'autres règles concernant les espaces 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 :

```

1| def fonction1():
2|     [...]
3|
4|     def fct_dans_fonction1():
5|         [...]

```

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 officielle⁷.

- D'autres subtilités concerteront la gestion des noms en cas de définition d'une nouvelle classe héritant d'une classe mère. Ces aspects sont présentés dans la rubrique *Héritage* de ce chapitre.

19.3 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.

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 :

```
1 | >>> sorted("citron")
2 | ['c', 'i', 'n', 'o', 'r', 't']
3 | >>> sorted([1, -67, 42, 0, 81])
4 | [-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 :

```
1 | >>> 2 + 2
2 | 4
3 | >>> "ti" + "ti"
4 | '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.

Comment Python permet-il ces prouesses que sont le polymorphisme et la redéfinition des opérateurs ? Et bien, il utilise des méthodes dites *magiques*.

Définition

Une méthode magique (*magic method*) est une méthode spéciale dont le nom est entouré de double *underscores*. Par exemple, la méthode `__init__()` est une méthode magique. Ces méthodes sont, la plupart du temps, destinées au fonctionnement interne de la classe. Beaucoup 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.

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 magique qui permettra cela est nommée `__str__()` : elle redéfinit le comportement d'une instance avec la fonction `print()`.

7. <https://docs.python.org/3/tutorial/classes.html#python-scopes-and-namespaces>

```

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 ("Votre citron est de couleur {} et de taille {}".format(self.couleur, self.taille))
14
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 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 magique `__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 magiques, 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 magiques 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 magiques font des choses assez impressionnantes. 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, nbcitrons, age):
3         self.nbcitrons, self.age = nbcitrons, age
4
5     def __call__(self, nbcitrons, age):
6         self.nbcitrons, self.age = nbcitrons, age
7
8     def __str__(self):
9         return "Ce citronnier a {} ans et {} citrons" \
10            .format(self.age, self.nbcitrons)
11
12
13 if __name__ == "__main__":
14     citronnier1 = Citronnier(10, 3)
15     print(citronnier1)
16     citronnier1(30, 4)
17     print(citronnier1)

```

À la ligne 15, on utilise une notation `instance(arg1, arg2)`, ce qui va automatiquement appeler la méthode magique `__call__()` qui mettra à jour les deux attributs d'instance `nbcitrons` et `age` (Lignes 5 et 6). Ce code affichera la sortie suivante :

```

1 | Ce citronnier a 3 ans et 10 citrons
2 | 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 magiques va bien sûr au-delà du présent ouvrage. Toutefois, si vous souhaitez aller plus loin, nous vous conseillons la page de Rafe Kettler⁸ qui est particulièrement complète et très bien faite.

19.4 Héritage

19.4.1 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 :

```

1 class Mere1:
2     # contenu de la classe mère 1
3
4
5 class Mere2:
6     # contenu de la classe mère 2
7
8
9 class Fille1(Mere1):
10    # contenu de la classe fille 1
11
12
13 class Fille2(Mere1, Mere2):
14    # 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`. Voyons maintenant un exemple concret :

```

1 class Mere:
2     def bonjour(self):
3         return "Vous avez le bonjour de la classe mère !"
4
5
6 class Fille(Mere):
7     def salut(self):
8         return "Un salut de la classe fille !"
9
10
11 if __name__ == "__main__":
12     fille = Fille()
13     print(fille.salut())
14     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 instantiation de la classe `Fille`, on utilise la méthode `.salut()`, puis la méthode `.bonjour()` héritée de la classe mère.

Ce code affiche la sortie suivante :

8. <https://rszalski.github.io/magicmethods>

```

1| Un salut de la classe fille !
2| 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. En reprenant l'exemple ci-dessus dans l'interpréteur :

```

1| >>> fille = Fille()
2| >>> isinstance(fille, Fille)
3| True
4| >>> isinstance(fille, Mere)
5| 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 magique (comme par exemple la méthode `__add__()` pour l'opérateur `+`).

Voyons un exemple :

```

1| class Mere:
2|     def bonjour(self):
3|         return "Vous avez le bonjour de la classe mère !"
4|
5|
6| class Fille2(Mere):
7|     def bonjour(self):
8|         return "Vous avez le bonjour de la classe fille !"
9|
10|
11| if __name__ == "__main__":
12|     fille = Fille2()
13|     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).

19.4.2 Ordre de résolution

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` :

```

1| >>> import tkinter as tk
2| >>> racine = tk.Tk()

```

En invoquant la commande `help(racine)`, l'interpréteur nous montre :

```

1| Help on class Tk in module tkinter:
2|

```

```

3| class Tk(Misc, Wm)
4| | Toplevel widget of Tk which represents mostly the main window
5| | of an application. It has an associated Tcl interpreter.
6|
7| | Method resolution order:
8| | Tk
9| | Misc
10| | Wm
11| | builtins.object
12| [...]

```

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 :

```

1| >>> class Citron:
2| ...     pass

```

Puis ensuite regarder l'aide sur l'une de ses instances :

```

1| Help on class Citron in module __main__:
2|
3| class Citron(builtins.object)
4| | Data descriptors defined here:
5|
6| |   __dict__
7| |       dictionary for instance variables (if defined)
8| [...]

```

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)` :

```

1| >>> import builtins
2| >>> dir(builtins)
3| ['ArithError', 'AssertionError', 'AttributeError', ...]
4| ['ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable', 'chr', ...]
5| ['list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', ...]
6| ['str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']

```

Au début, on y trouve les exceptions commençant par une lettre majuscule (cf. chapitre 21 *Remarques complémentaires* 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 :

```

1| >>> class Citron:
2| ...     pass
3| ...
4| >>> c = Citron()
5| >>> dir(c)
6| ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',

```

```

7 | '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
8 | '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
9 | '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
10| '__str__', '__subclasshook__', '__weakref__']
>>> o = object()
12| >>> dir(o)
13| ['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
14| '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
15| '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
16| '__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*).

19.4.3 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 souhaitons 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, ajouter des méthodes d'affichage commune à n'importe quel fruit, et ajouter toute 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 # en gramme
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 ("Instance {}\ntaille: {}, masse: {},\n"
15            "saveur: {}, forme: {}\\nconseil: {}\\n"
16            .format(type_fruit, self.taille, self.masse, self.saveur,
17                    self.forme, conseil))
17|
18|
19 class Citron(Fruit):
20    def __init__(self, taille=None, masse=None, saveur=None, forme=None):
21        print("(1) Je rentre dans le constructeur de Citron, et je vais"
22            "appeler\\n"
23            "le constructeur de la classe mère Fruit !")
24        Fruit.__init__(self, taille, masse, saveur, forme)
25        print("(3) J'ai fini dans le constructeur de Citron, "
26            "les attributs sont: \\nself.taille: {}, self.masse: {},\\n"
27            "self.saveur: {}, self.forme: {}\\n"
28            .format(self.taille, self.masse, self.saveur, self.forme))
29|
30    def __str__(self):
31        print("(1) Je rentre dans la méthode __str__() de la classe " \
32            "Citron")
33        print("Je vais lancer la méthode .affiche_conseil() héritée " \
34            "de la classe Fruit")
35        return self.affiche_conseil("Citron", "Bon en tarte :-p !")
36|
37|
38 if __name__ == "__main__":
39    # on crée un citron
40    citron1 = Citron(taille="petite", saveur="acide", forme="ellipsoïde",
41                      masse=50)
42    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 28. 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 25-28 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 30 à 34. On définit la méthode `__str__()` qui va modifier le comportement de notre classe avec `print()`. Celui-ci fait également appel à une méthode 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 37 à 40. 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 | (1) Je rentre dans le constructeur de Citron, et je vais appeler
2 | le constructeur de la classe mère Fruit !
3 | (2) Je suis dans le constructeur de la classe Fruit
4 | Je viens de créer self.taille, self.masse, self.saveur et self.forme
5 | (3) J'ai fini dans le constructeur de Citron, les attributs sont:
6 | self.taille: petite, self.masse: 50,
7 | self.saveur: acide, self.forme: ellipsoïde
8 |
9 | (1) Je rentre dans la méthode __str__() de la classe Citron
10 | Je vais lancer la méthode .affiche_conseil() héritée de la classe Fruit
11 | (2) Je suis dans la méthode .affiche_conseil() de la classe Fruit
12 |
13 Instance Citron
14 taille: petite, masse: 50,
15 saveur: acide, forme: ellipsoïde
16 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 chapitre 20) 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 utilisée, 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 ré-écrire les mêmes méthodes pour chaque fruit, simplement à les appeler. Par exemple :

```

1 class Kaki(Fruit):
2     def __init__(self, taille=None, masse=None, saveur=None, forme=None):
3         Fruit.__init__(self, taille, masse, saveur, forme)
4
5     def __str__(self):
6         return Fruit.affiche_conseil(self, "Kaki",
7                                       "Bon à manger cru, miam !")
8
9
10 class Orange(Fruit):
11     def __init__(self, taille=None, masse=None, saveur=None, forme=None):
12         Fruit.__init__(self, taille, masse, saveur, forme)
13
14     def __str__(self):
15         return Fruit.affiche_conseil(self, "Orange", "Trop bon en jus !")

```

Cet exemple illustre 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 20 *Fenêtres graphiques et Tkinter* 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.

19.4.4 Pour aller plus loin

À ce stade, nous pouvons émettre deux remarques :

- L'héritage et le polymorphisme 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 aller plus loin sur la POO, nous vous conseillons de lire des ressources supplémentaires. En langue française, vous trouverez les livres de Gérard Swinnen⁹, Bob Cordeau et Laurent Pointal¹⁰, et Vincent Legoff¹¹.

19.5 Accès et modifications des attributs depuis l'extérieur

19.5.1 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

```

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>

```

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 17 nous testons si la masse est négative, si tel est le cas nous générerons une erreur avec le mot-clé `raise` (cf. chapitre 21 *Remarques complémentaires*). 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 21 à 28. Après instantiation, on affiche la valeur des attributs avec les deux fonctions *getters*, puis on les modifie avec les *setters* et on les réaffiche à nouveau.

L'exécution de ce code donnera la sortie suivante :

```

1 jaune 0
2 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 26, 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 comprends 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 ajouter les lignes 9 à 12 ci-dessous) :

```

1 if __name__ == "__main__":
2     # définition de citron1
3     citron1 = Citron()
4     print(citron1.get_couleur(), citron1.get_masse())
5     # on change les attributs de citron1 avec les setters
6     citron1.set_couleur("jaune foncé")
7     citron1.set_masse(100)
8     print(citron1.get_couleur(), citron1.get_masse())
9     # on les recharge sans les setters
10    citron1.couleur = "pourpre profond"
11    citron1.masse = -15
12    print(citron1.get_couleur(), citron1.get_masse())

```

Cela donnera la sortie suivante :

```

1 jaune 0
2 jaune foncé 100
3 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 15 *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 15).

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` :

```
1 | # pythonique
2 | obj.x + obj.y + obj.z
3 |
4 | # non pythonique
5 | 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 *sucré 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.

19.5.2 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` :

```
1 | 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 fonctions que l'on a définies précédemment dans notre classe, mais on ne précise aucun argument ni parenthèse (voir le chapitre 20 *Fenêtres graphiques et Tkinter*). 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("Z'avez déjà vu une masse négative ?" \
14             "C'est nawak")
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("La masse de notre citron est {} g".format(citron.masse))
26     # on mange le citron
27     citron.masse = 25
28     print("La masse de notre citron est {} g".format(citron.masse))
29     print(citron.__dict__)

```

Pour une fois, nous allons commenter les lignes dans le désordre :

Ligne 16. 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 0x7fd3615aeeef8>`). 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 *Déférence 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.

Lignes 6 à 14. 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 (1) Dans le programme principal, je vais instancier un Citron
2 (2) J'arrive dans le __init__
3 Coucou je suis dans le set
4 (3) Je reviens dans le programme principal
5 Coucou je suis dans le get
6 La masse de notre citron est 100 g
7 Coucou je suis dans le set
8 Coucou je suis dans le get
9 La masse de notre citron est 25 g
10 {'_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).

Pour aller plus loin

Il existe une autre syntaxe considérée comme plus élégante pour mettre en place les objets *property*. Il s'agit des *décorateurs* `@property`, `@attribut.setter` et `@attribut.deleter`. Toutefois, la notion de décorateur va au-delà du présent ouvrage. Si vous souhaitez plus d'informations, vous pouvez consulter par exemple le site [programiz¹²](https://www.programiz.com/python-programming/property%3E) ou le livre de Vincent Legoff¹³.

19.6 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.

19.6.1 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 15 *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 pister 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).

19.6.2 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)* (cf. ci-dessous) !

Au lieu de ça, on parle en Python d'attributs publics et *non publics*.

12. <https://www.programiz.com/python-programming/property%3E>

13. <https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python>

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* :

```

1 | >>> citron = Citron()
2 | Coucou je suis dans le set
3 | >>> citron.masse
4 | Coucou je suis dans le get
5 | 0
6 | >>> citron.masse = -16
7 | Coucou je suis dans le set
8 | Traceback (most recent call last):
9 |   File "<stdin>", line 1, in <module>
10 |     File "<stdin>", line 10, in set_masse
11 |   ValueError: Z'avez déjà vu une masse négative ? C'est nawak
12 | >>> citron.masse = 16
13 | Coucou je suis dans le set
14 | >>> citron.masse
15 | Coucou je suis dans le get
16 | 16
17 | >>> citron._masse
18 | 16
19 | >>> citron._masse = -8364
20 | >>> citron.masse
21 | Coucou je suis dans le get
22 | -8364
23 | >>>
```

Nous n'avons pas encore croisé d'attribut dont le nom commence par deux caractères *underscores*. 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 :

```

1 | class Citron:
2 |     def __init__(self):
3 |         self.__mass = 100
4 |
5 |     def get_mass(self):
6 |         return self.__mass
7 |
8 |
9 | if __name__ == "__main__":
10 |     citron1 = Citron()
11 |     print(citron1.get_mass())
12 |     print(citron1.__mass)
```

Ce code va donner la sortie suivante :

```

1 | 100
2 | Traceback (most recent call last):
3 |   File "./pyscripts/mangling.py", line 11, in <module>
4 |     print(citron1.__mass)
5 | 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 :

```

1 | if __name__ == "__main__":
2 |     citron1 = Citron()
3 |     print(citron1.__dict__)
```

14. https://en.wikipedia.org/wiki/Name_mangling

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._NomClasseAttribut` 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 :

```

1 class Fruit:
2     def __init__(self):
3         self.__mass = 100
4
5
6 class Citron(Fruit):
7     def __init__(self):
8         Fruit.__init__(self)
9         self.__mass = 200
10
11    def print_masse(self):
12        print(self._Fruit__mass)
13        print(self.__mass)
14
15
16 if __name__ == "__main__":
17     citron1 = Citron()
18     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 ? Et bien selon le concepteur 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 !

19.6.3 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` :

```

1 class Citron:
2     """Voici la classe Citron.
3
4     Il s'agit d'une classe assez impressionnante qui crée des objets
5     citrons.
6     Par défaut une instance de Citron contient l'attribut de classe
7     saveur.
8     """
9
10    saveur = "acide"
11
12    def __init__(self, couleur="jaune", taille="standard"):
13        """Constructeur de la classe Citron.
14
15        Ce constructeur prend deux arguments par mot-clé
16        couleur et taille."""
17        self.couleur = couleur
18        self.taille = taille
19
20    def __str__(self):
21        """Redéfinit le comportement avec print()."""
22        return "saveur: {}, couleur: {}, taille: {}".format(
23            saveur, couleur, taille)
24
25    def affiche_coucou(self):
26        """Méthode inutile qui affiche coucou."""
27        print("Coucou !")
```

Si on fait `help(Citron)` dans l'interpréteur, on obtient :

```

1 Help on class Citron in module __main__:
2
3 class Citron(builtins.object)
4     Citron(couleur='jaune', taille='standard')
5
6     Voici la classe Citron.
7
8     Il s'agit d'une classe assez impressionnante qui crée des objets
9     citrons.
10    Par défaut une instance de Citron contient l'attribut de classe
11    saveur.
12
13    Methods defined here:
14
15        __init__(self, couleur='jaune', taille='standard')
16            Constructeur de la classe Citron.
17
18            Ce constructeur prend deux arguments par mot-clé
19            couleur et taille.
20
21        __str__(self)
22            Redéfinit le comportement avec print().
23
24        affiche_coucou(self)
25            Méthode inutile qui affiche coucou.
26
27    [...]
28
29    Data and other attributes defined here:
30
31        saveur = 'acide'
```

Python formate automatiquement l'aide comme il le fait avec les modules (voir chapitre 14 *Modules*). Comme nous l'avons dit dans le chapitre 15 *Bonnes pratiques en programmation Python*, n'oubliez pas que les *docstrings* sont destinées aux utilisateurs de votre classe. Elle 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 :

```

1 >>> citron1 = Citron()
2 >>> print(citron1.__doc__)
3 Voici la classe Citron.
4
5     Il s'agit d'une classe assez impressionnante qui crée des objets
6     citrons.
7     Par défaut une instance de Citron contient l'attribut de classe
8     saveur.
9
10 >>> print(citron1.affiche_coucou.__doc__)
11 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.).

19.6.4 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), utiliser plutôt les dictionnaires.
- 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 *Difference entre les attributs de classe et d'instance*, nous vous conseillons de ne jamais modifier les attributs de classe. Vous pouvez néanmois 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.

19.7 Exercices

Conseil : pour ces exercices, créez des scripts puis exécutez-les dans un *shell*.

19.7.1 Classe Rectangle

Téléchargez le script [rectangle.py]((<https://python.sdv.univ-paris-diderot.fr/data-files/rectangle.py>)) 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 `coul`;
- 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(ve) !) et qui affiche les attributs et la surface de ce nouveau rectangle.

19.7.2 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`).

19.7.3 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.

19.7.4 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()`. Redéfinissez le comportement avec `print()` (à l'aide de la méthode magique `__str__()`) de manière à afficher les coordonnées et la masse de l'atome.

19.7.5 Autres exercices +++

D'autres exercices plus approfondis sur les classes sont accessibles sur notre site internet ¹⁵. N'hésitez pas à aller le visiter.

15. <https://python.sdv.univ-paris-diderot.fr/livre-dunod>

Chapitre 20

Fenêtres graphiques et *Tkinter*

Conseil

Dans ce chapitre, nous allons utiliser des classes, nous vous conseillons de bien relire le chapitre 19 sur le sujet. Par ailleurs, nous vous conseillons de relire également la rubrique *Arguments positionnels et arguments par mot-clé* du chapitre 9 sur les fonctions.

20.1 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 -option1 blabla -option2 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 dévelopez 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 20.1.

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

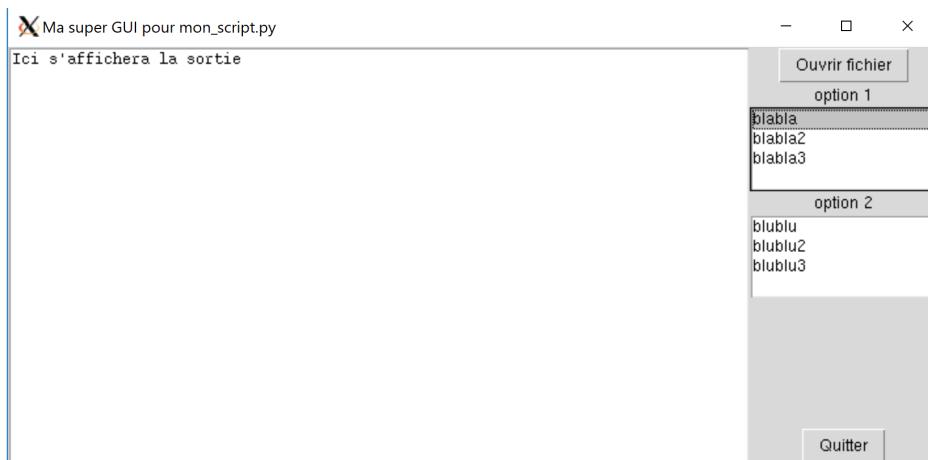


FIGURE 20.1 – Exemple de GUI.

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 de nombreux 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, *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.).

20.2 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 20.1, 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

Une fonction *callback* est une fonction passée en argument d'une autre fonction.

Un exemple de fonction *callback* est présenté dans la rubrique suivante.

20.3 Notion de fonction *callback*

Conseil : pour les débutants, vous pouvez passer cette rubrique.

Jusqu'à maintenant nous avons toujours appelé les fonctions ou les méthodes de cette manière :

```

1 var = fct(arg1, arg2)
2
3 obj.methode(arg)

```

1. <https://wiki.python.org/moin/TkInter>
2. <http://www.wxpython.org/>
3. <https://pyqt.readthedocs.io>
4. <https://pygobject.readthedocs.io/en/latest/>

FIGURE 20.2 – Exemple de fonction *callback* dans *Python Tutor*.

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 :

```

1 def fct_callback(arg):
2     print("J'aime bien les {}".format(arg))
3
4
5 def une_fct(ma_callback):
6     print("Je suis au début de une_fct(), et je vais exécuter la fonction callback :")
7     ma_callback("fraises")
8     print("Aye, une_fct() se termine.")
9
10
11 # Programme principal.
12 une_fct(fct_callback)

```

Si on exécute ce code, on obtient :

```

1 Je suis au début de une_fct() et je vais exécuter la fonction callback :
2 J'aime bien les fraises !
3 Aye, 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 parenthèse 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 12) ? C'était la même chose, une référence était envoyée plutôt qu'une copie. *Python Tutor*⁵ nous confirme cela (cf. figure 20.2).

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.

20.4 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

5. <http://pythontutor.com>

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*).

20.4.1 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 `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 strictement **obligatoire**. 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 exhaustive⁶. Dans les fonctions *Tkinter* qui construisent 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 9 *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 ici⁷.

Lignes 5 et 6. 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 20.3.

20.4.2 Le même exemple dans un script.

Tentons maintenant de faire la même chose dans un script `tk_exemple.py` :

6. <http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/label.html>

7. <http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/button.html>

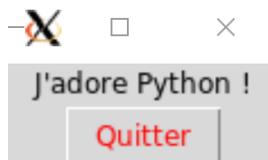


FIGURE 20.3 – Exemple basique de fenêtre Tkinter.

```

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 20.3). 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.

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.

20.5 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 PEP8⁸, cf. chapitre 15). 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 15).

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 10), on utilisera directement `self` plutôt que `racine` ou tout autre nom donné à l'instance. Comme vu dans le chapitre 19 *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 défini ci-dessus).

Lignes 11 et 12. On place les deux *widgets* dans la fenêtre avec la méthode `.pack()`.

Ligne 14. 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 14 *Création de modules*).

Ligne 15. On instancie notre application.

Ligne 16. 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 deviendra visible. On pourra noter que cette méthode `.title()` est héritée de la classe mère Tk.

Ligne 17. On lance le gestionnaire d'événements.

Au final, vous obtiendrez le même rendu que précédemment (cf. figure 20.3). 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. le code générant la figure 20.5).

20.6 Le widget canvas

20.6.1 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 :

8. <https://www.python.org/dev/peps/pep-0008/>

9. <http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/canvas.html>

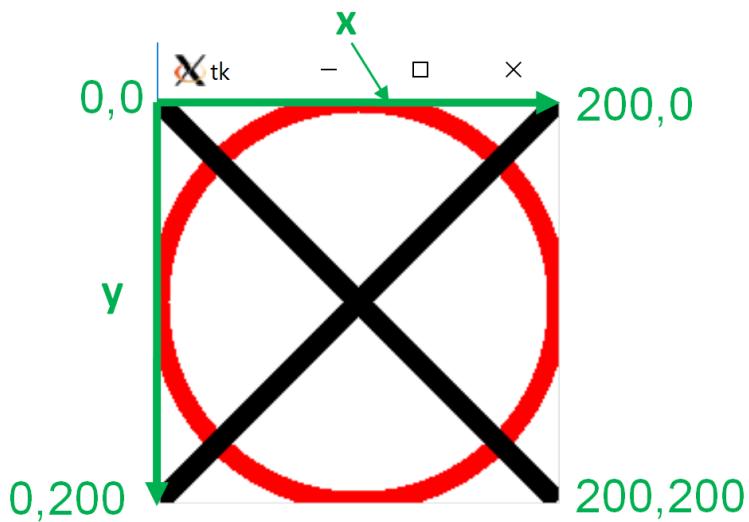


FIGURE 20.4 – Exemple 1 de *canvas* avec le système de coordonnées (le système de coordonnées est montré en vert et n'apparaît pas sur la vraie fenêtre *Tkinter*).

```

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 obligatoire, 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 le 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 obligatoires sont les coordonnées de l'ellipse (les deux points englobant l'ellipse, cf. ce lien¹⁰ 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 20.4 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.

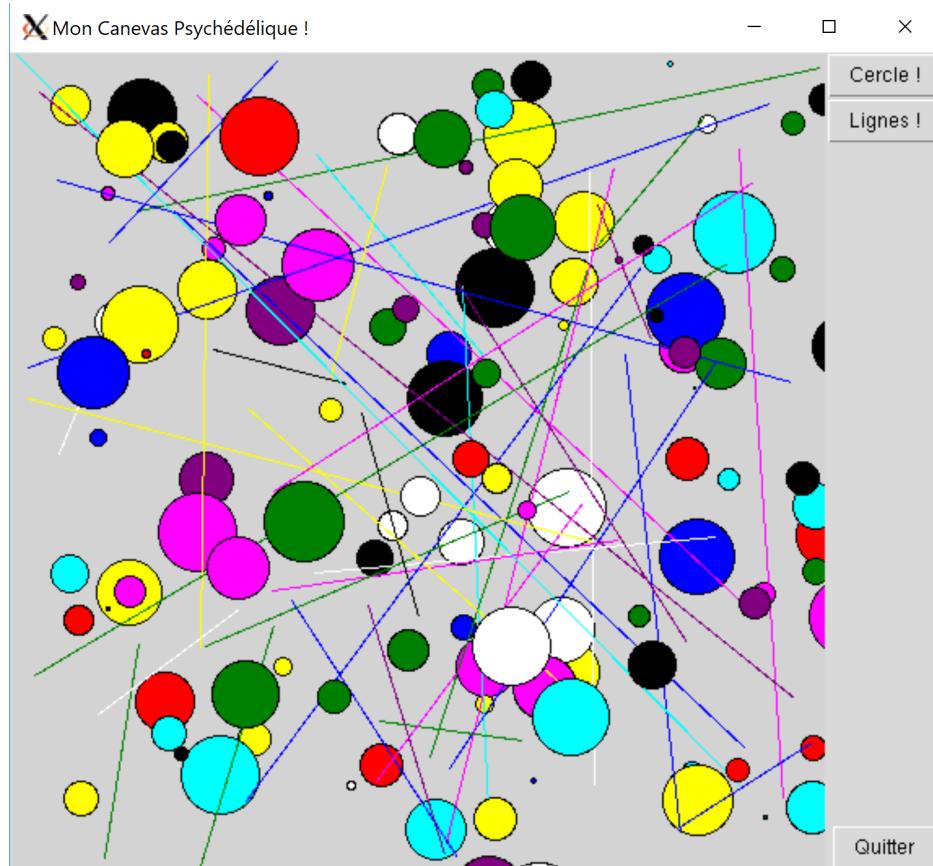
Attention

L'axe des *y* est inversé par rapport à ce que l'on représente en mathématique. Ainsi, si l'on souhaite présenter une fonction mathématique (ou tout autre objet dans un repère régi par un repère mathématique), il faudra faire un changement de repère.

20.6.2 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 20.5).

¹⁰ http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/create_oval.html

FIGURE 20.5 – Exemple 2 de *canvas*.

Le code suivant crée une telle application :

```

1 import tkinter as tk
2 import random as rd
3
4 class AppliCanevas(tk.Tk):
5     def __init__(self):
6         tk.Tk.__init__(self)
7         self.size = 500
8         self.creer_widgets()
9
10    def creer_widgets(self):
11        # création canevas
12        self.canv = tk.Canvas(self, bg="light gray", height=self.size,
13                             width=self.size)
14        self.canv.pack(side=tk.LEFT)
15        # boutons
16        self.bouton_cercles = tk.Button(self, text="Cercle !",
17                                         command=self.dessine_cercles)
18        self.bouton_cercles.pack(side=tk.TOP)
19        self.bouton_lignes = tk.Button(self, text="Lignes !",
20                                       command=self.dessine_lignes)
21        self.bouton_lignes.pack()
22        self.bouton_quitter = tk.Button(self, text="Quitter",
23                                        command=self.quit)
24        self.bouton_quitter.pack(side=tk.BOTTOM)
25
26    def rd_col(self):
27        return rd.choice(("black", "red", "green", "blue", "yellow", "magenta",
28                          "cyan", "white", "purple"))
29
30    def dessine_cercles(self):
31        for i in range(20):
32            x, y = [rd.randint(1, self.size) for j in range(2)]
33            diameter = rd.randint(1, 50)
34            self.canv.create_oval(x, y, x+diameter, y+diameter,
35                                  fill=self.rd_col())
36
37    def dessine_lignes(self):
38        for i in range(20):
39            x, y, x2, y2 = [rd.randint(1, self.size) for j in range(4)]
40            self.canv.create_line(x, y, x2, y2, fill=self.rd_col())
41
42
43 if __name__ == "__main__":
44     app = AppliCanevas()
45     app.title("Mon Canevas Psychédélique !")
46     app.mainloop()

```

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 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 instantiant 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. 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 grâce à *self.canv*.

20.6.3 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 20.6).

```

1 """Super appli baballe !!!
2
3 Usage: python tk_baballe.py
4 - clic gauche: faire grossir la baballe
5 - clic droit: faire rétrécir la baballe
6 - clic central: relance la baballe (depuis le point du clic)
7         dans une direction aléatoire
8 - touche Esc: quitte l'appli baballe
9 """
10
11 import tkinter as tk
12 import random as rd
13
14 class AppliBaballe(tk.Tk):
15     def __init__(self):
16         """Constructeur de l'application."""
17         tk.Tk.__init__(self)
18         # coord ini baballe
19         self.x, self.y = 200, 200
20         # rayon ini baballe
21         self.size = 50
22         # pas de déplacement
23         self.dx, self.dy = 20, 20
24         # création et packing du canvas
25         self.canv = tk.Canvas(self, bg='light gray', height=400, width=400)
26         self.canv.pack()
27         # création de la baballe
28         self.baballe = self.canv.create_oval(self.x, self.y,
29                                           self.x+self.size,
30                                           self.y+self.size,
31                                           width=2, fill="blue")
32
33         # binding des actions
34         self.canv.bind("<Button-1>", self.incr)
35         self.canv.bind("<Button-2>", self.boom)
36         self.canv.bind("<Button-3>", self.decr)
37         self.bind("<Escape>", self.stop)
38         # lancer la baballe
39         self.move()
40
41     def move(self):
42         """Déplace la baballe (appelée itérativement avec la méthode after)."""
43         # incr coord baballe
44         self.x += self.dx
45         self.y += self.dy
46         # vérifier que la baballe ne sort pas du canvas (choc élastique)
47         if self.x < 10:
48             self.dx = abs(self.dx)
49         if self.x > 400-self.size-10:
50             self.dx = -abs(self.dx)
51         if self.y < 10:
52             self.dy = abs(self.dy)
53         if self.y > 400-self.size-10:
54             self.dy = -abs(self.dy)
55         # mise à jour des coord
56         self.canv.coords(self.baballe, self.x, self.y, self.x+self.size,
57                           self.y+self.size)
58         # rappel de move toutes les 50ms
59         self.after(50, self.move)
60
61     def boom(self, mclick):
62         """Relance la baballe dans une direction aléatoire au point du clic."""
63         self.x = mclick.x
64         self.y = mclick.y
65         self.canv.create_text(self.x, self.y, text="Boom !", fill="red")
66         self.dx = rd.choice([-30, -20, -10, 10, 20, 30])
67         self.dy = rd.choice([-30, -20, -10, 10, 20, 30])
68
69     def incr(self, lclick):
70         """Augmente la taille de la baballe."""
71         self.size += 10
72         if self.size > 200:
73             self.size = 200
74
75     def decr(self, rclick):
76         """Diminue la taille de la baballe."""
77         self.size -= 10
78         if self.size < 10:
79             self.size = 10
80
81     def stop(self, esc):
82         """Quitte l'application."""
83         self.quit()

```

```

83
84
85 if __name__ == "__main__":
86     myapp = AppliBaballe()
87     myapp.title("Baballe !")
88     myapp.mainloop()

```

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 (*x₀*, *y₀*) dans la documentation officielle¹¹).

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=...*, cf. ci-dessus). 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 *windows*.

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 83).

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 12 (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 while True:
5     move()
6     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 60 et 61, 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*.

11. http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/create_oval.html

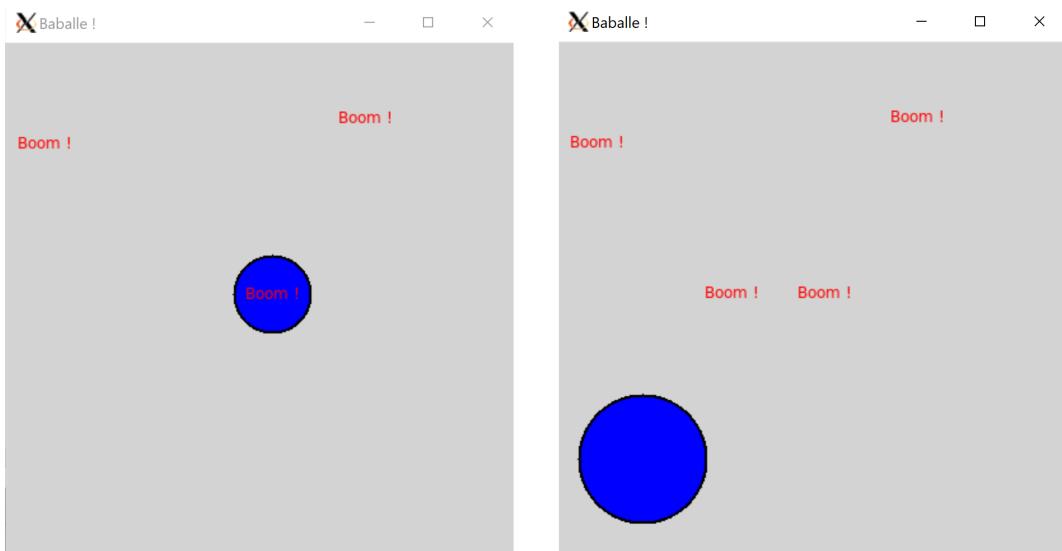


FIGURE 20.6 – Exemple de *canvas* animé à deux instants de l'exécution (panneau de gauche : au moment où on effectue un clic central ; panneau de droite : après avoir effectué plusieurs clics gauches).

Il existe de nombreux autres événements que l'on peut capturer et lier à des méthodes / fonctions *callback*. Vous trouverez une liste complète ici¹².

20.7 Pour aller plus loin

20.7.1 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 20.1).
- *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 20.1).
- *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 container* 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* : crée des fenêtres indépendantes.
- *PanedWindow* : container 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 Mexique¹³ (MNT). Par ailleurs, la page *Universal*

12. <http://effbot.org/tkinterbook/tkinter-events-and-bindings.htm>

13. <http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html>

*widget methods*¹⁴ 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 Python¹⁵, ainsi que la documentation très complète du site du MNT¹⁶.

20.7.2 Autres pistes à approfondir

Si vous souhaitez aller un peu plus loin en *Tkinter*, voici quelques notions / remarques qui pourraient vous être utiles. Pour les débutants, vous pouvez passer cette rubrique.

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 MNT¹⁷.

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()`¹⁸;
- `.grid()`^{19 20};
- `.place()`²¹.

14. <https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/universal.html>

15. <https://docs.python.org/3/library/tkinter.ttk.html>

16. <http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/ttk.html>

17. <https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/control-variables.html>

18. <http://effbot.org/tkinterbook/pack.htm>

19. <http://effbot.org/tkinterbook/grid.htm>

20. <https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/grid.html>

21. <http://effbot.org/tkinterbook/place.htm>

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 if __name__ == "__main__":
17    racine = tk.Tk()
18    racine.title("Ma Première App :-)")
19    app = Application(racine)
20    racine.mainloop()
21

```

Lignes 17 à 20. 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 poser 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` !

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.

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 :

```

1| class MonApplication(tk.Tk):
2|     def __init__(self, *args, **kwargs):
3|         tk.Tk.__init__(self, *args, **kwargs)
4|         # ici débute la construction de votre appli
5|     [...]
6|
7| # programme principal
8| if __name__ == "__main__":
9|     [...]
10|    app = MonApplication()
11|    [...]

```

Les arguments `*args` et `**kwargs` récupèrent facilement tous les arguments « positionnels » et « par mot-clé ». Pour plus de détails sur comment `*args` et `**kwargs` fonctionnent, reportez-vous au chapitre 21 *Remarques complémentaires*.

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 à passer au constructeur, par exemple :

```
1| 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 Application 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 (cf. chapitre 15 *Bonnes Pratiques*), 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.

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 :

```

1| >>> import tkinter as tk
2| >>> bouton = tk.Button(text="Quitter")
3| >>> 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` :

```

1| >>> racine = bouton.master
2| >>> racine
3| <tkinter.Tk object: ...
4| >>> racine.children
5| {'!button': <tkinter.Button object: ...>}
6| >>> 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*.

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 21 *Remarques complémentaires*. 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 [pythonprogramming²²](https://pythonprogramming.net/passing-functions-parameters-tkinter-using-lambda/) et le livre de Gérard Swinnen²³.

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 Swinnen²⁴ et le site [pythonprogramming²⁵](https://pythonprogramming.net/change-show-new-frame-tkinter/) sont des bonnes sources pour commencer et voir concrètement comment faire cela.

20.7.3 Bibliographie pour aller plus loin

Voici quelques ressources que vous pouvez utiliser pour continuer votre apprentissage de *Tkinter* :

1. En anglais :

- La Documentation officielle²⁶ de Python.
- Le manuel²⁷ de référence sur le site du MNT.
- Le site²⁸ de Fredrik Lundh est également très complet.
- Pour avoir un exemple²⁹ rapide de code pour chaque *widget*.
- Le livre³⁰ 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 site³¹ 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 site³² bien complet d'Étienne Florent.
- Le livre³³ de Gérard Swinnen qui montre de nombreux exemples d'applications tkinter (pdf en libre téléchargement).

20.8 Exercices

Conseil : dans tous les exercices qui suivent nous vous recommandons de concevoir une classe pour chaque exercice.

20.8.1 Application de base

Concevez une application qui affiche l'heure dans un *label* (par exemple 09:10:55) et qui possède un bouton 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*.

20.8.2 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 :

22. <https://pythonprogramming.net/passing-functions-parameters-tkinter-using-lambda/>
 23. <https://inforef.be/swi/python.htm>
 24. <https://inforef.be/swi/python.htm>
 25. <https://pythonprogramming.net/change-show-new-frame-tkinter/>
 26. <https://wiki.python.org/moin/TkInter>
 27. <https://infohost.nmt.edu/tcc/help/pubs/tkinter/web/index.html>
 28. <http://effbot.org/tkinterbook/>
 29. https://www.tutorialspoint.com/python/python_gui_programming.htm
 30. <https://github.com/Dvlv/Tkinter-By-Example>
 31. <https://pythonprogramming.net/tkinter-depth-tutorial-making-actual-program/>
 32. <http://tkinter.fdex.eu/index.html>
 33. <https://inforef.be/swi/python.htm>



FIGURE 20.7 – Compte à rebours.

```
self.label.configure(text=heure)
où heure est une chaîne de caractères représentant l'heure actuelle.
```

20.8.3 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 bouton « 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 l'interpréteur et quittera automatiquement le script. Une image du résultat attendu est montrée dans la figure 20.7.

20.8.4 Triangle de Sierpinski

Le triangle de Sierpinski³⁴ est une fractale classique. On se propose ici de la dessiner avec un algorithme tiré du jeu du chaos³⁵. Celui-ci se décompose en pseudo-code de la façon suivante :

```
1 définir les 3 sommets d'un triangle isocèle
2 point <- coordonnées (x, y) du centre du triangle
3 dessiner(point) # une pixelle de large
4 pour i de 0 à 25000:
5     sommet_tmp <- choisir un sommet du triangle au hasard
6     point <- calculer(coordonnées (x, y) du centre entre et sommet_tmp)
7     dessiner(point)
```

Le rendu final attendu est montré dans la figure 20.8. 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.

20.8.5 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 !

```
1 import tkinter as tk
2
3 class MaListbox(tk.Tk):
4     def __init__(self):
5         # Instanciation fenêtre Tk.
6         tk.Tk.__init__(self)
```

34. https://fr.wikipedia.org/wiki/Triangle_de_Sierpi%C5%84ski

35. https://fr.wikipedia.org/wiki/Jeu_du_chaos

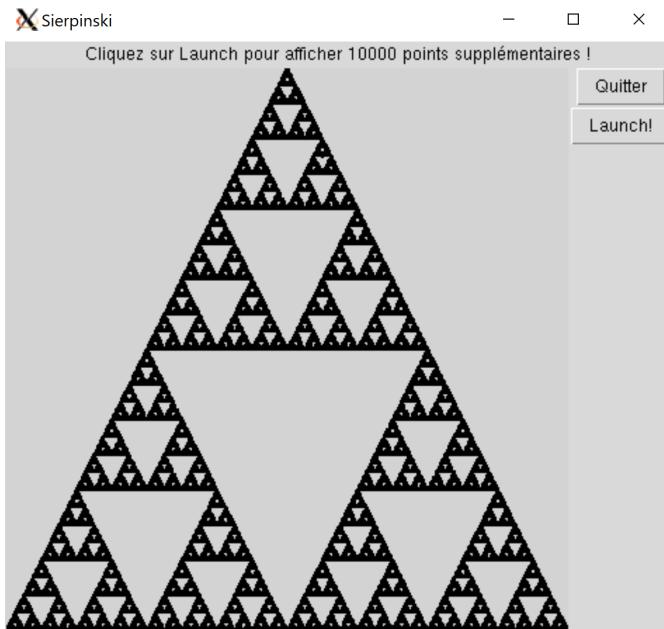


FIGURE 20.8 – Triangle de Sierpinski.

```

7  self.listBox = tk.Listbox(self, height=10, width=4)
8  self.listBox.pack()
9  # On ajoute des items à la listBox (entiers).
10 for i in range(1, 10+1):
11     # Utilisation de ma méthode .insert(index, element)
12     # On ajoute l'entier i (tk.END signifie en dernier).
13     self.listBox.insert(tk.END, i)
14 # Selectionne premier élément de listBox.
15 self.listBox.select_set(0)
16 # Lier une méthode quand clic sur listBox.
17 self.listBox.bind("<<ListboxSelect>>", self.clic_listbox)
18
19 def clic_listbox(self, event):
20     # Récup du widget à partir de l'objet event.
21     widget = event.widget
22     # Récup du choix sélectionné dans la listBox (tuple).
23     # (par exemple renvoie `(5,)` si on a cliqué sur `5`)
24     selection = widget.curselection()
25     # Récup du nombre sélectionné (déjà un entier).
26     choix_select = widget.get(selection[0])
27     # affichage
28     print("Le choix sélectionné est {}, son type est {}".format(choix_select, type(choix_select)))
29
30
31 if __name__ == "__main__":
32     app = MaListBox()
33     app.title("MaListBox")
34     app.mainloop()
35

```

20.8.6 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 22 *Mini projets*. 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 21

Remarques complémentaires

21.1 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 :

21.1.1 La fonction print()

La fonction `print()` en Python 2 s'utilise sans parenthèse. Par exemple :

```
1 | >>> print 12
2 | 12
3 | >>> print "girafe"
4 | girafe
```

Par contre en Python 3, si vous n'utilisez pas de parenthèse, Python vous renverra une erreur :

```
1 | >>> print 12
2 |   File "<stdin>", line 1
3 |     print 12
4 | ^
5 | SyntaxError: Missing parentheses in call to 'print'
```

21.1.2 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 :

```
1 | >>> 3 / 4
2 | 0.75
```

Il est également possible de réaliser une division entière avec l'opérateur `//` :

```
1 | >>> 3 // 4
2 | 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 :

```
1 | >>> 3 / 5
2 | 0
3 | >>> 4 / 3
4 | 1
```

Faites très attention à cet aspect si vous programmez encore en Python 2, c'est une source d'erreur récurrente.

21.1.3 La fonction range()

En Python 3, la fonction `range()` est un générateur, c'est-à-dire que cette fonction va itérer sur le nombre entier donner en argument. On ne peut pas l'utiliser seule :

```
1 | >>> range(3)
2 | range(0, 3)
```

Lorsqu'on l'utilise dans une boucle `for`, `range(3)` va produire successivement les nombres 0, puis 1 puis 2. Par exemple :

```
1 | >>> for i in range(3):
2 | ...     print(i)
3 |
4 | 0
5 | 1
6 | 2
```

En Python 2, la fonction `range()` renvoie une liste. Par exemple :

```
1 | >>> range(3)
2 | [0, 1, 2]
3 | >>> range(2, 6)
4 | [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 !

```
1 | >>> range(3)
2 | [0, 1, 2]
3 | >>> xrange(3)
4 | 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 suffit de l'associer avec la fonction `list()`. Par exemple :

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

21.1.4 Encodage et utf-8

En Python 3, vous pouvez utiliser des caractères accentués dans les commentaires ou dans les chaîne 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 :

```
1 | SyntaxError: Non-ASCII character '\xc2' in file xxx on line yyy, but no encoding
2 | 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 :

```
1 | # 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 importante¹) de votre script :

```
1 | #! /usr/bin/env python
2 | # coding: utf-8
```

Remarque

L'encodage utf-8 peut aussi être déclaré de cette manière :

```
1 | # -*- coding: utf-8 -*-
```

mais c'est un peu plus long à écrire.

1. <http://www.python.org/dev/peps/pep-0263/>

21.2 Liste de compréhension

Une manière originale et très puissante de générer des listes est la compréhension de listes. Pour plus de détails, consultez à ce sujet le site de Python² et celui de Wikipédia³.

Voici quelques exemples.

21.2.1 Nombres pairs compris entre 0 et 30

```
1 | >>> print([i for i in range() if i%2 == 0])
2 | [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30]
```

21.2.2 Jeu sur la casse des mots d'une phrase

```
1 | >>> message = "C'est sympa la BioInfo"
2 | >>> msg_lst = message.split()
3 | >>> print([[m.upper(), len(m)] for m in msg_lst])
4 | [["C'EST", 5], ['SYMPA', 5], ['LA', 2], ['BIOINFO', 7]]
```

21.2.3 Formatage d'une séquence avec 60 caractères par ligne

Exemple d'une séquence constituée de 150 alanines :

```
1 | # Exemple d'une séquence de 150 alanines.
2 | >>> seq = "A"*150
3 | >>> width = 60
4 | >>> seq_split = [seq[i:i+width] for i in range(0,len(seq),width)]
5 | >>> print("\n".join(seq_split))
6 | AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
7 | AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
8 | AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

21.2.4 Formatage FASTA d'une séquence (avec la ligne de commentaire)

Exemple d'une séquence constituée de 150 alanines :

```
1 | >>> com = "Séquence de 150 alanines"
2 | >>> seq = "A"*150
3 | >>> width = 60
4 | >>> seq_split = [seq[i:i+width] for i in range(0,len(seq),width)]
5 | >>> print(">" + com + "\n" + "\n".join(seq_split))
6 | > séquence de 150 alanines
7 | AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
8 | AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
9 | AAAAAAAAAAAAAAAAAAAAAAA
```

21.2.5 Sélection des carbones alpha dans un fichier pdb

Exemple avec la structure de la barstar⁴ :

```
1 | >>> with open("1bta.pdb", "r") as f_pdb:
2 | ...     CA_lines = [line for line in f_pdb if line.startswith("ATOM")
3 | ...                     and line[12:16].strip() == "CA"]
4 | ...
5 | >>> print(len(CA_lines))
6 | 89
```

21.3 Gestion des erreurs

La gestion des erreurs permet d'éviter 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.

2. <http://www.python.org/dev/peps/pep-0202/>
 3. http://fr.wikipedia.org/wiki/Comprehension_de_liste
 4. <http://www.rcsb.org/pdb/explore.do?structureId=1BTA>

```

1 >>> nb = int(input("Entrez un nombre entier : "))
2 Entrez un nombre entier : 23
3 >>> print(nb)
4 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 :

```

1 >>> nb = int(input("Entrez un nombre entier : "))
2 Entrez un nombre entier : ATCG
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5 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.

Le jeu d'instructions `try / except` permet de tester l'exécution d'une commande et d'intervenir en cas d'erreur.

```

1 >>> try:
2 ...     nb = int(input("Entrez un nombre entier : "))
3 ...     except:
4 ...         print("Vous n'avez pas entré un nombre entier !")
5 ...
6 Entrez un nombre entier : ATCG
7 Vous n'avez pas entré un nombre entier !

```

Dans cette exemple, l'erreur renvoyée 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.

```

1 >>> while True:
2 ...     try:
3 ...         nb = int(input("Entrez un nombre entier : "))
4 ...         print("Le nombre est", nb)
5 ...         break
6 ...     except:
7 ...         print("Vous n'avez pas entré un nombre entier !")
8 ...         print("Essayez encore")
9 ...
10 Entrez un nombre entier : ATCG
11 Vous n'avez pas entré un nombre entier !
12 Essayez encore
13 Entrez un nombre entier : toto
14 Vous n'avez pas entré un nombre entier !
15 Essayez encore
16 Entrez un nombre entier : 3.2
17 Vous n'avez pas entré un nombre entier !
18 Essayez encore
19 Entrez un nombre : 55
20 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 erreurs 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.

Vous pouvez par exemple vérifier qu'un fichier a bien été ouvert.

```

1 >>> nom = "toto.pdb"
2 >>> try:
3 ...     with open(nom, "r") as fichier:
4 ...         for ligne in fichier:
5 ...             print(ligne)
6 ...     except:
7 ...         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 :

```

1 >>> try:
2 ...     nb = int(input("Entrez un nombre entier : "))
3 ...     except ValueError:
4 ...         print("Vous n'avez pas entré un nombre entier !")
5 ...
6 Entrez un nombre entier : ATCG
7 Vous n'avez pas entré un nombre entier !

```

Ici, on intercepte une erreur de type `ValueError`, ce qui correspond bien à un problème de conversion avec `int()`. Il existe d'autres types d'erreurs comme `RuntimeError`, `TypeError`, `NameError`, `IOError`, etc.

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.

```

1 import urllib.request
2
3 def download_page(address):
4     error = ""
5     page = ""
6     try:
7         data = urllib.request.urlopen(address)
8         page = data.read()
9     except IOError as e:
10        if hasattr(e, 'reason'):
11            error = "Cannot reach web server: " + str(e.reason)
12        if hasattr(e, 'code'):
13            error = "Server failed {:.d}".format(e.code)
14    return page, error
15
16 data, error = download_page("https://files.rcsb.org/download/1BTA.pdb")
17
18 if error:
19     print("Erreur rencontrée : {}".format(error))
20 else:
21     with open("protéine.pdb", "w") as prot:
22         prot.write(data.decode('utf-8'))
23     print("Protéine enregistrée")

```

La variable `e` est une instance (un représentant) de l'erreur de type `IOError`. Certains de ces 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 et stockée dans la variable `data`, puis ensuite enregistrée sur le disque dur.

21.4 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 :

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 : `print('Hello World !')` il va alors contenir : “`#!/usr/bin/env python`
`print('Hello World !')`”
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 par 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) :

```

1 $ ./test.py
2 Hello World !

```

Définition

Le **shebang**⁵ 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.

5. <http://fr.wikipedia.org/wiki/Shebang>

21.5 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 9 Fonctions.

Dans le chapitre 9, 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 :

```

1 | >>> def fct(**kwargs):
2 | ...     print(kwargs)
3 | ...
4 | >>> fct()
5 | {}
6 | >>> fct(z=1, gogo="toto")
7 | {'gogo': 'toto', 'z': 1}
8 | >>> fct(z=1, gogo="toto", y=-67)
9 | {'y': -67, 'gogo': 'toto', 'z': 1}
10| >>> fct(1, 2)
11| Traceback (most recent call last):
12|   File "<stdin>", line 1, in <module>
13| 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 :

```

1 | >>> def fct(*args, **kwargs):
2 | ...     print(args)
3 | ...     print(kwargs)
4 | ...
5 | >>> fct()
6 | ()
7 | {}
8 | >>> fct(1, 2)
9 | (1, 2)
10| {}
11| >>> fct(z=1, y=2)
12| {}
13| {'y': 2, 'z': 1}
14| >>> fct(1, 2, 3, z=1, y=2)
15| (1, 2, 3)
16| {'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.

Enfin, il est possible de combiner des arguments positionnels avec `*args` et `**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. 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 `Tkinter` présenté dans le chapitre 20.

Enfin, il est possible d'utiliser ce mécanisme d'empaquetage / déempaquetage (*packing / unpacking*) dans l'autre sens :

```

1 | >>> def fct(a, b, c):
2 | ...     print(a,b,c)
3 | ...
4 | >>> t = (-5,6,7)
5 | >>>
6 | >>> fct(*t)
7 | -5 6 7

```

Avec la syntaxe `*t` on déempaquette le tuple à la volée lors de l'appel à la fonction. Cela est aussi possible avec un dictionnaire :

```

1 | >>> def fct(x, y, z):
2 | ...     print(x, y, z)
3 | ...
4 | >>> dico = {'x': -1, 'y': -2, 'z': -3}
5 | >>> fct(**dico)
6 | -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 déempaqueter les valeurs du dictionnaire (si vous utilisez une seule étoile, Python déempaquettera les clés !).

21.6 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).

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_Abstf` 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 $0,15 \text{ radian}^{-1}$, ce qui correspond bien à 2π (c'est plutôt bon signe de retrouver ce résultat).

21.7 Sauvegardez votre historique de commandes

Vous pouvez sauvegarder l'historique des commandes utilisées dans l'interpréteur Python avec le module `readline`.

```
1 | >>> print("hello")
2 | hello
3 | >>> a = 22
4 | >>> a = a + 11
5 | >>> print(a)
6 | 33
7 | >>> import readline
8 | >>> readline.write_history_file()
```

Quitez Python. L'historique de toutes vos commandes est dans votre répertoire personnel, dans le fichier `.history`. Relancez l'interpréteur Python.

```
1 | >>> import readline
2 | >>> 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 22

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.

22.1 Description des projets

22.1.1 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 3000 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 page³.

Conseil : vous trouverez des explications sur le format FASTA et des exemples de code dans l'annexe A *Quelques formats de données rencontrés en biologie*.

22.1.2 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 du boulanger *Saccharomyces cerevisiae*. Vous pouvez télécharger ce fichier :

- soit via le lien sur le site du cours `NC_001133.gbk`⁴ ;
- 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 rencontrés en biologie*.

Vous pouvez réaliser ce projet sans ou avec des expressions régulières (abordées dans le chapitre 15).

22.1.3 Simulation d'un pendule

On se propose de réaliser une simulation d'un pendule simple⁶ en Tkinter. 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. On néglige les effets de frottement

1. <https://python.sdv.univ-paris-diderot.fr/data-files/english-common-words.txt>

2. <https://python.sdv.univ-paris-diderot.fr/data-files/human-proteome.fasta>

3. https://www.uniprot.org/help/human_proteome

4. https://python.sdv.univ-paris-diderot.fr/data-files/NC_001133.gbk

5. https://www.ncbi.nlm.nih.gov/nuccore/NC_001133

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

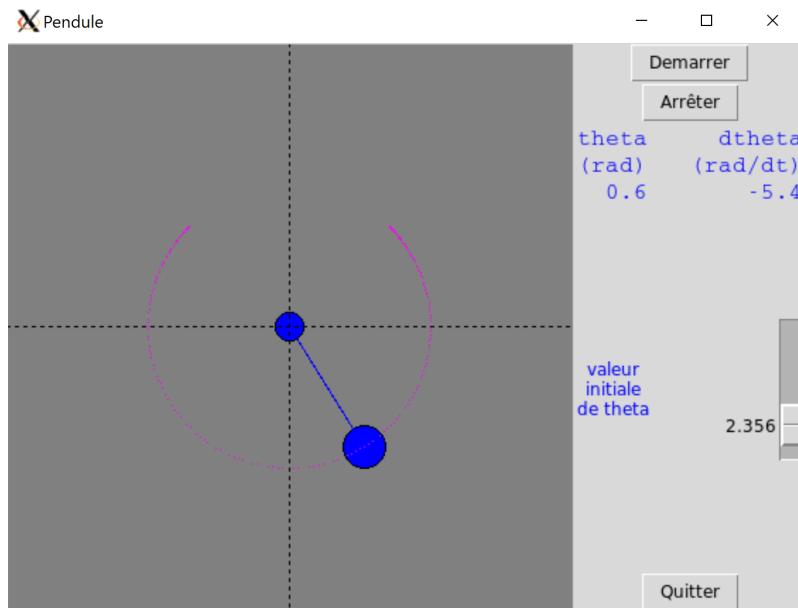


FIGURE 22.1 – Application pendule.

et on considère le champ gravitationnel comme uniforme. Le mouvement du pendule sera calculé en résolvant numériquement l'équation différentielle suivante :

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l} * \sin(\theta)$$

où θ représente l'angle entre la verticale et la tige du pendule, g la gravité, et l la longueur de la tige (note : pour la dérivation d'une telle équation vous pouvez consulter la page wikipedia⁷).

Pour trouver la valeur de θ en fonction du temps, on pourra utiliser la méthode d'Euler⁸ de résolution d'équation différentielle. La formule ci-dessus donne l'accélération angulaire $a_\theta = \frac{d^2\theta}{dt^2}$. À partir de celle-ci, la méthode d'Euler propose le calcul de la vitesse angulaire $v_\theta = a_\theta * \delta t$ (où δt représente le pas de temps entre deux étapes successives de la simulation). Enfin, cette vitesse v_θ donne l'angle θ lui-même : $\theta = v_\theta * \delta t$. 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.

Pour la visualisation, vous pourrez utiliser le *widget canvas* du module *Tkinter* (voir le chapitre 20 *Fenêtres graphiques et Tkinter*, rubrique *Un canvas animé dans une classe*). On cherche à obtenir un résultat comme montré dans la figure 22.4.

Nous vous conseillons de procéder d'abord par 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 20 *Fenêtres graphiques et Tkinter*.

22.2 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.

7. [https://en.wikipedia.org/wiki/Pendulum_\(mathematics\)#math_Eq._1](https://en.wikipedia.org/wiki/Pendulum_(mathematics)#math_Eq._1)

8. https://en.wikipedia.org/wiki/Euler_method

22.2.1 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.

Composition aminée

Dans un premier temps, composez 5 mots anglais avec les 20 acides aminés.

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.

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 page¹¹.

Voici les premières lignes de ce fichier ([...] indique une coupure que nous avons faite) :

```

1 | >sp|095139|NDUB6_HUMAN NADH dehydrogenase [ubiquinone] 1 beta [...]
2 | MTGYTPDEKLRLQQLRRELRRRWLKDQELSPREPVLPPQKMGPMEKFWNKFLENKSPWRKM
3 | VHGKVKKSIFVFTHVLPVWVIIHYYMKYHVSEKPYGIVEKKSRIFPGDTILETGEVIPPMM
4 | KEFPDQHH
5 | >sp|075438|NDUB1_HUMAN NADH dehydrogenase [ubiquinone] 1 beta [...]
6 | MVNLLQIVRDHWVHVLVPMGFVIGCYLDRKSDERLTAFRNKSMLFKRELQPSEEVTWK
7 | >sp|Q8N4C6|NIN_HUMAN Ninein OS=Homo sapiens OX=9606 GN=NIN PE=1 SV=4
8 | MDEVEQDGHEARLKELFDSFDTTGTGSLGQEELTDLCHMLSLEEVAPVLQQTLLQDNLLG
9 | RVHFDFQFKEALILILSRTLTSNEEHFQEPDCSLEAQPKYVRGGKRYGRRSLPEFQESVEEF
10 | PEVTVIEPLDEEARPSHIPAGDCSCEHWKTQRSEEEYAEQQLRFNPDDLNASQSGSSPPQ

```

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, 095139, 075438, 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 095139.

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 :

```

1 | ACCESS found in 1 sequences
2 | ACID found in 38 sequences
3 | ACT found in 805 sequences
4 | [...]

```

Cette étape prend quelques minutes. Soyez patient.

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 :

```
1 | => xxx found in yyy sequences
```

9. <https://python.sdv.univ-paris-diderot.fr/data-files/english-common-words.txt>

10. <https://python.sdv.univ-paris-diderot.fr/data-files/human-proteome.fasta>

11. https://www.uniprot.org/help/human_proteome

Quel est ce mot ?

Quel pourcentage des séquences du protéome contiennent ce mot ?

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.

22.2.2 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 rencontrés 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 indiquées dans la description du projet.

Dans cette rubrique, nous allons réaliser ce projet **sans expression régulière**.

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.

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.

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 gènes¹². 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

12. <https://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html#BaseSpanB>

```
1 |     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

```
1 |     gene          <2480..>2707
```

sera transformé en

```
1 | 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.

Extraction de la séquence nucléique du génome

La taille du génome est indiquée 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

```
1 | ORIGIN
```

et

```
1 | //
```

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 :

```
1 | is_dnaseq <- False
2 | Lire chaque ligne du fichier gbk
3 |   si la ligne contient "//"
4 |     is_dnaseq <- False
5 |   si is_dnaseq vaut True
6 |     accumuler la séquence
7 |   si la ligne contient "ORIGIN"
8 |     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.

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 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.

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 :

```
1 >commentaire
2 sequence sur une ligne de 80 caractères maxi
3 suite de la séquence .....
4 suite de la séquence .....
5 ...
```

Testez cette fonction avec :

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.

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 *os* dans le chapitre 8 *Modules*.

Testez votre script ainsi finalisé.

Bravo, si vous êtes arrivés jusqu'à cette étape.

22.2.3 genbank2fasta (avec expression régulière)

Ce projet consiste à écrire un convertisseur de fichier, du format GenBank au format FASTA. L'annexe A *Quelques formats de données rencontrés 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 indiquées dans la description du projet.

Dans cette rubrique, nous allons réaliser ce projet avec des expression régulières en utilisant le module `re`.

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.

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.

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.

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

```
| //
```

13. <https://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html#BaseSpanB>

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.

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 :

- On rappelle que construire la séquence complémentaire inverse d'une séquence d'ADN consiste à :

 - 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.

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 :

```
1 >commentaire
2 sequence sur une ligne de 80 caractères maxi
3 suite de la séquence .....
4 suite de la séquence .....
5
```

Testez cette fonction avec :

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.

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 `os` dans le chapitre 8 sur les modules.

Testez votre script ainsi finalisé.

22.2.4 Simulation d'un pendule

L'objectif de ce projet est de simuler un pendule simple¹⁴ en 2 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.

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 anglais¹⁵. 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 22.2 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 22.3 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 mis en mouvement 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 Newton¹⁶ 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

14. https://fr.wikipedia.org/wiki/Pendule_simple

15. [https://en.wikipedia.org/wiki/Pendulum_\(mathematics\)](https://en.wikipedia.org/wiki/Pendulum_(mathematics))

16. https://fr.wikipedia.org/wiki/Lois_du_mouvement_de_Newton

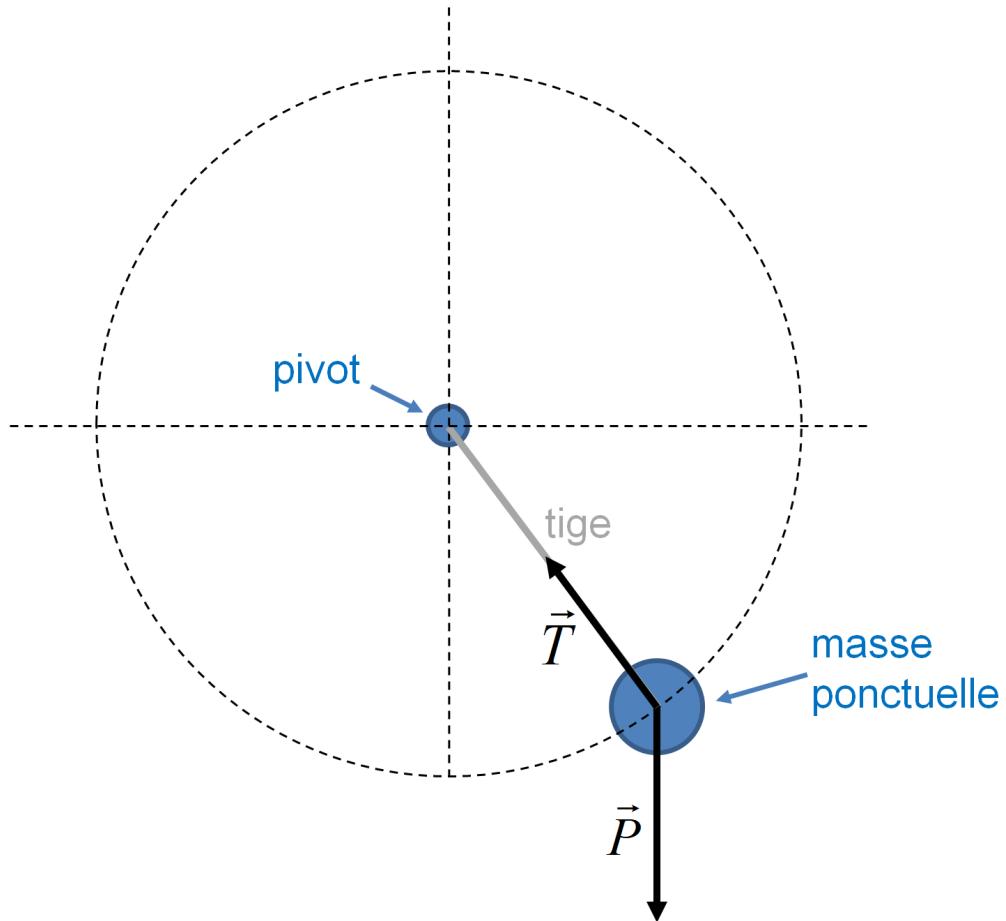


FIGURE 22.2 – Bilan des forces dans un pendule simple.

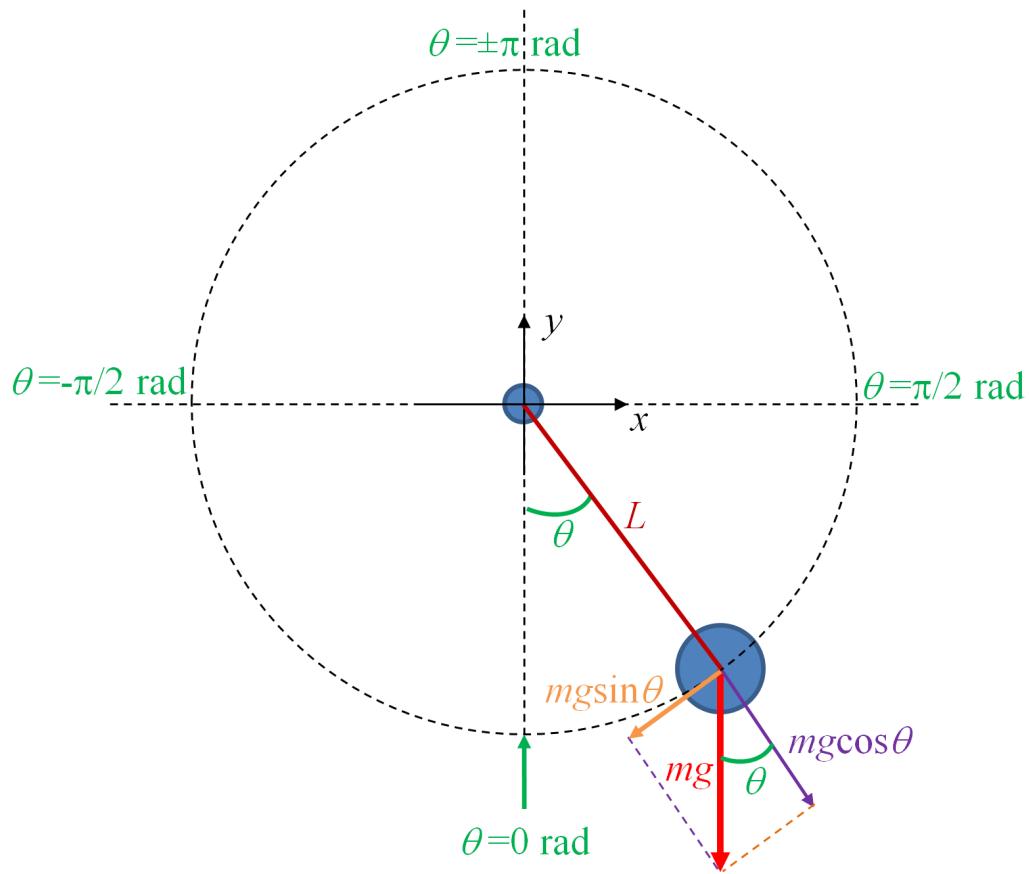


FIGURE 22.3 – Caractérisation géométrique d'un pendule simple.

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 $m g \cos \theta$ 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 $m g \sin \theta$ (cf. figure 22.3). Au final, on peut écrire l'expression suivante en raisonnant sur les valeurs scalaires :

$$F = ma = -m g \sin \theta$$

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 :

$$a = -g \sin \theta$$

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 :

$$a = \frac{d^2s}{dt^2} = L \frac{d^2\theta}{dt^2}$$

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 :

$$L \frac{d^2\theta}{dt^2} = -g \sin \theta$$

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.

Résolution de l'équation différentielle du pendule

Il existe de nombreuses méthodes numériques de résolution d'équations différentielles¹⁷. 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'Euler¹⁸. Celle-ci est relativement intuitive à comprendre.

Commençons d'abord par calculer l'accélération angulaire a_θ au temps t en utilisant l'équation différentielle précédemment établie :

$$a_\theta(t) = \frac{d^2\theta}{dt^2}(t) = -\frac{g}{L} \sin \theta(t)$$

17. https://en.wikipedia.org/wiki/Numerical_methods_for_ordinary_differential_equations

18. https://en.wikipedia.org/wiki/Semi-implicit_Euler_method

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 :

```

1 L <- 1           # longueur tige en m
2 g <- 9.8         # accélération gravitationnelle en m/s^2
3 t <- 0           # temps initial en s
4 dt <- 0.05       # pas de temps en s
5 # conditions initiales
6 theta <- pi / 4 # angle initial en rad
7 dtheta <- 0       # vitesse angulaire initiale en rad/s
8
9 afficher_position_pendule(t, theta) # afficher position de départ

```

L'initialisation des valeurs de θ et $d\theta$ 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.

A 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. A la fin de cette rubrique, nous proposons une solution en langage algorithmique. Essayez dans un premier temps de le faire vous-même. A 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 # plot
11 plt.figure(figsize=(8,8))
12 mini = min(theta) * 1.2
13 maxi = max(theta) * 1.2
14 plt.xlim(0, maxi)
15 plt.ylim(mini, maxi)
16 plt.xlabel('t (s)')
17 plt.ylabel('theta (deg)')
18 plt.plot(t, theta)
19 plt.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 :

```

1 tant qu'on n'arrête pas le pendule:
2     # acc angulaire au tps t (en rad/s^2)
3     d2theta <- -(g/L) * sin(theta)
4     # v angulaire mise à jour de t -> t + dt
5     dtheta <- dtheta + d2theta * dt
6     # theta mis à jour de t -> t + dt
7     theta <- theta + dtheta * dt
8     # t mis à jour
9     t <- t + dt
10    # mettre à jour l'affichage
11    afficher_position_pendule(t, theta)

```

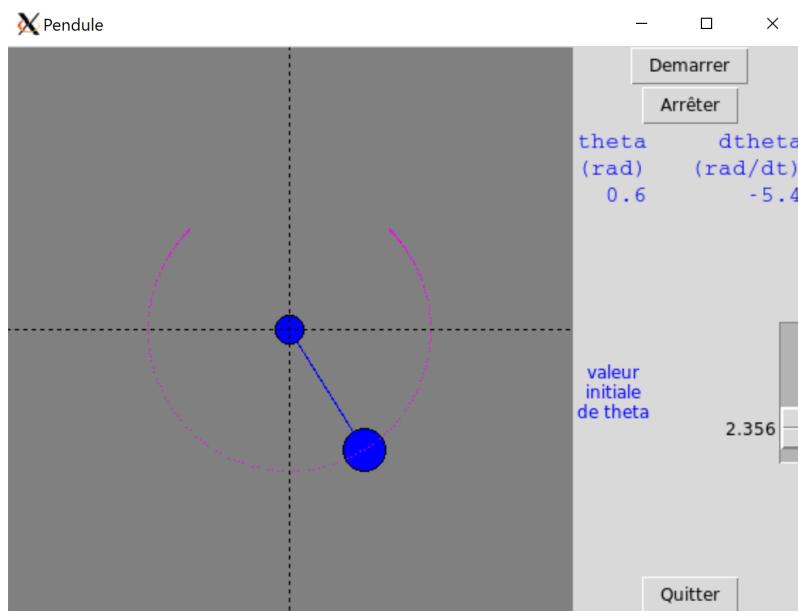


FIGURE 22.4 – Application pendule.

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 20 sur *tkinter* avant de vous lancer dans cette partie.

Comme expliqué largement dans le chapitre 20, 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 (cf. 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         # instantiation 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        # ici vous pouvez construire l'application graphique
13        self.canv = tk.Canvas(self, bg='gray', height=400, width=400)
14        # création d'un bouton demarrer, arreter, quitter
15        # penser à placer les widgets avec .pack()
16        [...]
```

La figure 22.4 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 violettes « 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 : 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).

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 cette dernière **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 22.3, 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 (cf. exercice sur la spirale dans le chapitre 7). 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 $(0, -1)$;
 - quand $\theta = +\pi/2 = 90$ deg, on a $(1, 0)$;
 - quand $\theta = -\pi/2 = -90$ deg, on a $(-1, 0)$;
 - quand $\theta = \pm\pi = \pm180$ deg, on a $(0, 1)$.

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
```

Méthode convertissant θ en coordonnées (x, y) Il nous faut maintenant convertir (x, y) en coordonnées (x_c, y_c) dans le *canvas*. 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 $1 \text{ m} \rightarrow 100 \text{ pixels}$;
- l'axe des ordonnées dans le *canvas* est **inversé** par rapport au repère mathématique.

Conseil

Il est essentiel d'écrire une méthode qui convertira θ en (x, y) puis (x_c, y_c) dans votre classe. Vous pouvez l'appeler par exemple `.theta2xc_yc()`.

Si vous n'avez pas trouvé, voici la solution :

```
1| self.conv_factor = 100
2| self.x_c = self.x * self.conv_factor + 200
3| self.y_c = -self.y * self.conv_factor + 200
```

TODO :

- mouvement du pendule : mth .start(), .stop(), .move() (se référer à la baballe)

Ressource complémentaire

Si vous souhaitez aller plus loin sur les différentes méthodes numériques de résolution d'équation différentielle associées au pendule, nous vous conseillons le site de James Sethna¹⁹ de l'université de Cornell.

19. <http://pages.physics.cornell.edu/~sethna/StatMech/ComputerExercises/Pendulum/Pendulum.html>

Annexe A

Quelques formats de données rencontrés 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 :

```
1 | >en-tête
2 | séquence avec un nombre maximum de caractères par ligne
3 | séquence avec un nombre maximum de caractères par ligne
4 | séquence avec un nombre maximum de caractères par ligne
5 | séquence avec un nombre maximum de caractères par ligne
6 | 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 la sous-unité β de l'hémoglobine humaine¹, extraite de la base de données UniProt, est :

```
1 | >sp|P68871|HBB_HUMAN Hemoglobin subunit beta OS=Homo sapiens OX=9606 GN=HBB PE=1 SV=2
2 | MVHLTPEEKSAVTALWGKVNDEVGGEALGRLLVVYPWTQRFESFGDLSTPDAMGNPK
3 | VKAHGKKVLGAFSDGLAHLDNLKGTFATLSELHCDKLHVDPENFRLGNVLVCVLAHHFG
4 | KEFTPVQAAQKVVAGVANALAHKYH
```

La première ligne contient la description de la séquence (*Hemoglobin subunit beta*), le type de base de données (ici *sp* qui signifie Swiss-Prot), son identifiant (*P68871*) et son nom (*HBB_HUMAN*) dans cette base de données, ainsi que d'autres informations (*S=Homo sapiens OX=9606 GN=HBB PE=1 SV=2*).

Les lignes suivantes contiennent la séquence sur des lignes ne dépassant pas, ici, 60 caractères. La séquence de la sous-unité β de l'hémoglobine humaine est composée de 147 acides aminés, soit deux lignes de 60 caractères et une troisième de 27 caractères.

Définition

UniProt² 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

1. <https://www.uniprot.org/uniprot/P68871>

2. <https://www.uniprot.org/>

automatiquement).

Voici maintenant la séquence nucléique (ARN), au format FASTA, de l'insuline humaine³, extraite de la base de données GenBank⁴ :

```

1 | >BT006808.1 Homo sapiens insulin mRNA, complete cds
2 | ATGCCCTGTGGATGCCCTCTGCCCTGCTGGCCTGCTGGCCCTCTGGGACCTGACCCAGCCGAG
3 | CCTTGTAACCAACACCTGTGCGCTCACACCTGGTGAAGCTCTACCTAGTGTGCGGGAAACGAGG
4 | CTTCTCTACACACCAGGAGGACCTGCAGGTGGAGCTGGAGCTGGCGGG
5 | GCCCTGTGCAGGAGCCCTGGAGGGTCCCTGCAGAACGCTGGCATGTGGAAC
6 | AATGCTGTACCAAGCATCTGCCTCTACCACTGGAGAACTACTGCAACTAG

```

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 chez humaine (*Homo sapiens*), féline (*Felis catus*) et bovine (*Bos taurus*) :

```

1 | >sp|P01308|INS_HUMAN Insulin OS=Homo sapiens OX=9606 GN=INS PE=1 SV=1
2 | MAlWMRLPLALLALWGPDPAAAFVNQHLCGSHLVEALYLVCGERGFFYTPKTRREAED
3 | LQVGQVELGGPGAGSLQPLALEGSQKRGIVEQCCTTSICSLYQLENYC
4 | >sp|P06306|INS_FELCA Insulin OS=Felis catus OX=9685 GN=INS PE=1 SV=2
5 | MAPWTRLPLALLSLWIPAPTRAFFVNQHLCGSHLVEALYLVCGERGFFYTPKARREAED
6 | LQGKDAELGEAPGAGGLQPSALEAPLQKRGIVEQCASVCSLYQLEHYCN
7 | >sp|P01317|INS_BOVIN Insulin OS=Bos taurus OX=9913 GN=INS PE=1 SV=2
8 | MAlWTRLRPLALLALWPPPAPARAFVNQHLCGSHLVEALYLVCGERGFFYTPKARREVEG
9 | PQVGALELAGGGPAGGLEGPQKRGIVEQCASVCSLYQLENYC

```

Ces séquences proviennent de la base de données UniProt et sont téléchargeables en suivant ce lien⁵.

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 :

```

1 prot_dict = {}
2 with open("insulin.fasta", "r") as fasta_file:
3     prot_id = ""
4     for line in fasta_file:
5         if line.startswith(">"):
6             prot_id = line[1:].split()[0]
7             prot_dict[prot_id] = ""
8         else:
9             prot_dict[prot_id] += line.strip()
10    for id in prot_dict:
11        print(id)
12        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 :

```

1 | sp|P06306|INS_FELCA
2 | MAPWTRLPLALLSLWIPAPTRAFFVNQHL
3 | sp|P01317|INS_BOVIN
4 | MAlWTRLRPLALLALWPPPAPARAFVNQHL
5 | sp|P01308|INS_HUMAN
6 | MAlWMRLPLALLALWGPDPAAAFVNQHL

```

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* :

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/uniprot/?sort\(score&desc=&compress=no&query=id:P01308%20OR%20id:P01317%20OR%20id:P06306&format=fasta](https://www.uniprot.org/uniprot/?sort(score&desc=&compress=no&query=id:P01308%20OR%20id:P01317%20OR%20id:P06306&format=fasta)

```

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.

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 porte le plus souvent l'extension .gbk.

Le format GenBank est décrit de manière très complète sur le site du NCBI⁶. En voici néanmoins les principaux éléments avec l'exemple du gène qui code pour la trypsine⁷ chez l'Homme.

A.2.1 L'en-tête

```

1| LOCUS      HUMTRPSGNA          800 bp      mRNA      linear    PRI 14-JAN-1995
2| DEFINITION Human pancreatic trypsin 1 (TRY1) mRNA, complete cds.
3| ACCESSION  M22612
4| VERSION    M22612.1
5| KEYWORDS   trypsinogen.
6| SOURCE     Homo sapiens (human)
7| ORGANISM   Homo sapiens
8|           Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
9|           Mammalia; Eutheria; Euarchontoglires; Primates; Haplorrhini;
10|          Catarrhini; Hominidae; Homo.
11| [...]

```

Ligne 1 (LOCUS) : le nom du locus (*HUMTRPSGNA*), la taille du gène (800 paires de base), 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).

A.2.2 Les features

```

1| [...]
2| FEATURES          Location/Qualifiers
3|     source          1..800
4|                   /organism="Homo sapiens"
5|                   /mol_type="mRNA"
6|                   /db_xref="taxon:9606"
7|                   /map="7q32-qter"
8|                   /tissue_type="pancreas"
9|     gene            1..800
10|                /gene="TRY1"
11|                7..750
12|                /gene="TRY1"
13|                /codon_start=1
14|                /product="trypsinogen"
15|                /protein_id="AAA61231.1"
16|                /db_xref="GDB:G00-119-620"
17|                /translation="MNPLLILTFVAAALAAPFDDDKIVGGYNEENSVPYQVSLNSG
18| YHFCGGSLINEQWVVSAGHCYKSRIQVRLGEHNIEVLEGNEQFINAAKIRHPQYDRK
19| TLNNNDIMLIKLSRAVINARVSTISLPTAPPATGTCCLISGWGNTASSGADYPDELQC
20| LDAPVLSQAKCEASYPGKITSNMFCVGFLEGGKDSCQDGGPVVCNGQLQGVVSWGD
21| GCAQKNKPGVYTAKVYNYVWKWIKNTIAANS"
22|     sig_peptide    7..51
23|                /gene="TRY1"
24|                /note="G00-119-620"
25| [...]

```

6. <https://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html>

7. <https://www.ncbi.nlm.nih.gov/nuccore/M22612.1>

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 complément(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.

Ligne 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

```

1 [...]
2 ORIGIN
3      1 accaccatga atccacttctt gatccttacc tttgtggcag ctgctttgc tgcccccttt
4      61 gatgatgatg acaagatgtt tggggctac aactgtgagg agaattctgt ccccttaccag
5      121 gtgtccctga attctggta ccacttctgt ggtggctccc tcataacaga acagtgggtg
6      181 gatatcagcag gccacttctt caagtcccgcc atccaggatgaa gactgggaga gcacacatc
7      241 aggttcctgg aggggaatga gcaggatc aatgcaggcca agatcatccg ccaccccaa
8      301 taccacaggaa agactctgaa caatgacatc atgttaatca agtcttcctc acgtgcagta
9      361 atcaacgccc gcgtgtccac catctctgtt cccacccgccc ctccagccac tggcacgaag
10     421 tgctctatctt ctggctgggg caacatcgcc agtcttgcgc ccgactacc agacgagctg
11     481 cagtgcctgg atgcttcctgt gctgagggccat gctaagtgtt aagcttccat ccctgaaag
12     541 attaccagca acatgttctg tggggcttc ttggggggag gcaaggatc atgtcagggt
13     601 gattctgttg gccctgttgtt ctgcaatggaa cagcttcaag gagttgttcc ctgggtgtat
14     661 ggctgtgccc agaagaacaa gcctggatgc tacaccaagg ttacaacta cgtgaaatgg
15     721 attaaaatgttccatagctgc caatagctaa agcccccaatgttccat tctcttccat
16     781 aataaaatgttccat
17 //
```

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* :

```

1 from Bio import SeqIO
2 with open("M22612.gbk", "r") as gbk_file:
3     record = SeqIO.read(gbk_file, "genbank")
4     print(record.id)
5     print(record.description)
6     print(record.seq[:60])
```

Pour la séquence lue dans le fichier GenBank, on affiche son identifiant, sa description et les 60 premiers résidus :

```

1 M22612.1
2 Human pancreatic trypsin 1 (TRY1) mRNA, complete cds.
3 ACCACCATGAATCCACTCCTGATCCTTACCTTGTCGGCAGCTGCTTGCTGCCCTT
```

Il est également possible de lire un fichier GenBank sans le module *Biopython*. Une activité dédiée est proposée dans le chapitre 22 *Mini-projets*.

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 éponyme⁹. Les principales 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 (et aussi par une machine). À l'inverse les coordonnées sont surtout utilisables par une 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 bovine¹⁰.

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>

A.3.1 En-tête

Pour la trypsine bovine, l'en-tête compte 510 lignes. En voici quelques unes :

```

1 HEADER      HYDROLASE (SERINE PROTEINASE)          26-OCT-81    2PTN
2 TITLE       ON THE DISORDERED ACTIVATION DOMAIN IN TRYPSINOGEN.
3 TITLE       2 CHEMICAL LABELLING AND LOW-TEMPERATURE CRYSTALLOGRAPHY
4 COMPND     MOL_ID: 1;
5 COMPND     2 MOLECULE: TRYPSIN;
6 COMPND     3 CHAIN: A;
7 [...]
8 SOURCE      2 ORGANISM_SCIENTIFIC: BOS TAURUS;
9 [...]
10 EXPDTA    X-RAY DIFFRACTION
11 [...]
12 REMARK    2 RESOLUTION.    1.55 ANGSTROMS.
13 [...]
14 DBREF      2PTN A 16 245 UNP P00760 TRY1_BOVIN 21 243
15 SEQRES     1 A 223 ILE VAL GLY GLY TYR THR CYS GLY ALA ASN THR VAL PRO
16 SEQRES     2 A 223 TYR GLN VAL SER LEU ASN SER GLY TYR HIS PHE CYS GLY
17 SEQRES     3 A 223 GLY SER LEU ILE ASN SER GLN TRP VAL VAL SER ALA ALA
18 SEQRES     4 A 223 HIS CYS TYR LYS SER GLY ILE GLN VAL ARG LEU GLY GLU
19 [...]
20 HELIX      1 H1 SER A 164 ILE A 176 1SNGL ALPHA TURN, REST IRREG. 13
21 HELIX      2 H2 LYS A 230 VAL A 235 5CONTIGUOUS WITH H3 6
22 HELIX      3 H3 SER A 236 ASN A 245 1CONTIGUOUS WITH H2 10
23 SHEET      1 A 7 TYR A 20 THR A 21 0
24 SHEET      2 A 7 LYS A 156 PRO A 161 -1 N CYS A 157 0 TYR A 20
25 [...]
26 SSBOND     1 CYS A 22 CYS A 157 1555 1555 2.04
27 SSBOND     2 CYS A 42 CYS A 58 1555 1555 2.02
28 [...]

```

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 Angstroems.

Ligne 14. DBREF indique les liens éventuels vers d'autres banques de données. Ici, l'identifiant correspondant à cette protéine dans UniProt (*UNP*) est P00760¹¹.

Ligne 15-18. SEQRES donnent à la séquence de la protéine. Les résidus sont représentés par leur code à 3 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 bonds 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 1700 lignes. En voici quelques unes correspondantes au résidu leucine 99 :

```

1 [...]
2 ATOM      601 N LEU A 99 10.007 19.687 17.536 1.00 12.25 N
3 ATOM      602 CA LEU A 99 9.599 18.429 18.188 1.00 12.25 C
4 ATOM      603 C LEU A 99 10.565 17.281 17.914 1.00 12.25 C
5 ATOM      604 O LEU A 99 10.256 16.101 18.215 1.00 12.25 O
6 ATOM      605 CB LEU A 99 8.149 18.040 17.853 1.00 12.25 C
7 ATOM      606 CG LEU A 99 7.125 19.029 18.438 1.00 18.18 C
8 ATOM      607 CD1 LEU A 99 5.695 18.554 18.168 1.00 18.18 C
9 ATOM      608 CD2 LEU A 99 7.323 19.236 19.952 1.00 18.18 C
10 [...]

```

11. <https://www.uniprot.org/uniprot/P00760>

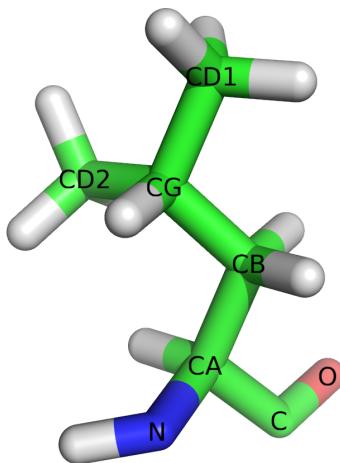


FIGURE A.1 – Structure tridimensionnelle d'un résidu leucine. Les noms des atomes sont indiqués en noir.

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 coordonnées x de l'atome.
- 19.687 : la coordonnées y de l'atome.
- 17.536 : la coordonnées 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. 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 PDB¹².

Les résidus sont ensuite décrits les uns après les autres, atome par atome. Voici par exemple les premiers résidus de la trypsin bovine :

1	[...]											
2	ATOM	1	N	ILE	A	16	-8.155	9.648	20.365	1.00	10.68	N
3	ATOM	2	CA	ILE	A	16	-8.150	8.766	19.179	1.00	10.68	C
4	ATOM	3	C	ILE	A	16	-9.405	9.018	18.348	1.00	10.68	C
5	ATOM	4	O	ILE	A	16	-10.533	8.888	18.870	1.00	10.68	O
6	ATOM	5	CB	ILE	A	16	-8.091	7.261	19.602	1.00	10.68	C
7	ATOM	6	CG1	ILE	A	16	-6.898	6.882	20.508	1.00	7.42	C
8	ATOM	7	CG2	ILE	A	16	-8.178	6.281	18.408	1.00	7.42	C
9	ATOM	8	CD1	ILE	A	16	-5.555	6.893	19.773	1.00	7.42	C
10	ATOM	9	N	VAL	A	17	-9.224	9.305	17.090	1.00	9.63	N
11	ATOM	10	CA	VAL	A	17	-10.351	9.448	16.157	1.00	9.63	C
12	ATOM	11	C	VAL	A	17	-10.500	8.184	15.315	1.00	9.63	C
13	ATOM	12	O	VAL	A	17	-9.496	7.688	14.748	1.00	9.63	O
14	ATOM	13	CB	VAL	A	17	-10.123	10.665	15.222	1.00	9.63	C
15	ATOM	14	CG1	VAL	A	17	-11.319	10.915	14.278	1.00	11.95	C
16	ATOM	15	CG2	VAL	A	17	-9.737	11.970	15.970	1.00	11.95	C
17	[...]											

12. <http://www.wwpdb.org/documentation/file-format-content/format33/sect9.html>

Vous remarquez 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 (code PDB 5X9X¹³) :

```

1 [...]
2 ATOM    762  HB1  ALA A  44      37.162   -2.955    2.220   1.00   0.00      H
3 ATOM    763  HB2  ALA A  44      38.306   -2.353    3.417   1.00   0.00      H
4 ATOM    764  HB3  ALA A  44      38.243   -1.621    1.814   1.00   0.00      H
5 TER     765      ALA A  44
6 ATOM    766  N   GLY B  95      -18.564    3.009   13.772   1.00   0.00      N
7 ATOM    767  CA  GLY B  95      -19.166    3.646   12.621   1.00   0.00      C
8 ATOM    768  C   GLY B  95      -20.207    2.755   11.976   1.00   0.00      C
9 [...]
```

La première chaîne est notée A et la seconde B. La séparation entre les deux est marquée par la ligne TER 765
ALA A 44.

Dans un fichier PDB, chaque structure porte un nom de chaîne différent.

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 :

```

1 [...]
2 MODEL      1
3 ATOM      1  N   MET A   1      -27.283   -9.772    5.388   1.00   0.00      N
4 ATOM      2  CA  MET A   1      -28.233   -8.680    5.682   1.00   0.00      C
5 [...]
6 ATOM    1499  HG2  GLU B  139      36.113   -5.242    2.536   1.00   0.00      H
7 ATOM    1500  HG3  GLU B  139      37.475   -4.132    2.428   1.00   0.00      H
8 TER     1501      GLU B  139
9 ENDMDL
10 MODEL      2
11 ATOM      1  N   MET A   1      -29.736   -10.759   4.394   1.00   0.00      N
12 ATOM      2  CA  MET A   1      -28.372   -10.225   4.603   1.00   0.00      C
13 [...]
14 ATOM    1499  HG2  GLU B  139      36.113   -5.242    2.536   1.00   0.00      H
15 ATOM    1500  HG3  GLU B  139      37.475   -4.132    2.428   1.00   0.00      H
16 TER     1501      GLU B  139
17 ENDMDL
18 MODEL      2
19 ATOM      1  N   MET A   1      -29.736   -10.759   4.394   1.00   0.00      N
20 ATOM      2  CA  MET A   1      -28.372   -10.225   4.603   1.00   0.00      C
21 [...]
```

Chaque structure est encadrée par les lignes

```

1 MODEL      n
2
3 et
4 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.

Chargement de 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 2e ligne par `parser = PDBParser(PERMISSIVE=1)`. Soyez néanmoins très prudent quant aux résultats obtenus.

13. <http://www.rcsb.org/structure/5X9X>

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 :

```
1| model = structure[0]
2| chain = model["A"]
3| res1 = chain[16]
4| res2 = chain[17]
5| print(res1.resname, res1["N"].coord)
6| print(res2.resname, res2["CA"].coord)
```

ce qui produit :

```
1| ILE [ -8.15499973  9.64799976 20.36499977]
2| VAL [-10.35099983  9.44799995 16.15699959]
```

L'objet `res1["N"].coord` est un *array* de *NumPy* (voir le chapitre 17 *Quelques modules d'intérêt en bioinformatique*). On peut alors obtenir simplement les coordonnées x, y et z d'un atome :

```
1| print(res1["N"].coord[0], res1["N"].coord[1], res1["N"].coord[2])
```

ce qui produit :

```
1| -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) :

```
1| res_start = 16
2| model = structure[0]
3| chain = model["A"]
4| for i in range(10):
5|     idx = res_start + i
6|     print(chain[idx].resname, idx, chain[idx]["CA"].coord)
```

avec pour résultat :

```
1| ILE 16 [ -8.14999962  8.76599979 19.17900085]
2| VAL 17 [-10.35099983  9.44799995 16.15699959]
3| GLY 18 [-12.02099991  6.63000011 14.25899982]
4| GLY 19 [-10.90200043  3.89899993 16.68400002]
5| TYR 20 [-12.65100002  1.44200003 19.01600075]
6| THR 21 [-13.01799965  0.93800002 22.76000023]
7| CYS 22 [-10.02000046 -1.16299999 23.76000023]
8| GLY 23 [-11.68299961 -2.86500001 26.7140007 ]
9| ALA 24 [-10.64799976 -2.62700009 30.36100006]
10| 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 :

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]	
3	atom name	1	{:.1s}	13–16 [12:16]	
4	alternate location indicator	4	{:^4s}	13–16 [16:17]	
5	residue name	3	{:.3s}	18–20 [17:20]	
6	chain identifier	1	{:.1s}	21–22 [22:26]	
7	residue sequence number	4	{:.4d}	23–26 [22:26]	
8	code for insertion of residues	1	{:.1s}	27 [26:27]	
9	orthogonal coordinates for X in Angstrom	3	{:.3f}	31–38 [30:38]	
10	orthogonal coordinates for Y in Angstrom	8	{:.8f}	39–46 [38:46]	
11	orthogonal coordinates for Z in Angstrom	8	{:.8f}	47–54 [46:54]	
12	occupancy	6	{:.2f}	55–60 [54:60]	
13	temperature factor	6	{:.2f}	61–66 [60:66]	
14	element symbol	10	{:>2s}	77–78 [76:78]	
15	charge on the atom	2	{:<2s}	79–80 [78:80]	

```
Python formatted string (old): %s %s %s %s  
Python formatted string (new): %s - %s %s %s
```



```

1 with open("2PTN.pdb", "r") as pdb_file:
2     res_count = 0
3     for line in pdb_file:
4         if line.startswith("ATOM"):
5             atom_name = line[12:16].strip()
6             res_name = line[17:20].strip()
7             res_num = int(line[22:26])
8             if atom_name == "CA":
9                 res_count += 1
10                x = float(line[30:38])
11                y = float(line[38:46])
12                z = float(line[46:54])
13                print(res_name, res_num, x, y, z)
14            if res_count >= 10:
15                break

```

ce qui donne :

```

1 ILE 16 -8.15 8.766 19.179
2 VAL 17 -10.351 9.448 16.157
3 GLY 18 -12.021 6.63 14.259
4 GLY 19 -10.902 3.899 16.684
5 TYR 20 -12.651 1.442 19.016
6 THR 21 -13.018 0.938 22.76
7 CYS 22 -10.02 -1.163 23.76
8 GLY 23 -11.683 -2.865 26.714
9 ALA 24 -10.648 -2.627 30.361
10 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 sont des formats de fichiers très largement utilisés en informatique. Ils sont tout autant très utilisés en biologie. En voici quelques exemples.

A.4.1 XML

Le format XML est un format de fichier 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 balise 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 :

```

1 <protein>
2 <element>élément 1</element>
3 <element>élément 2</element>
4 <element>élément 3</element>
5 </protein>

```

Dans cet exemple, nous avons trois balises element qui sont contenues dans une balise protein.

Voici un autre exemple avec l'enzyme trypsine¹⁵ humaine (code P07477¹⁶) telle qu'on peut la trouver décrite dans la base de données UniProt :

14. https://fr.wikipedia.org/wiki/Extensible_Markup_Language

15. <https://www.uniprot.org/uniprot/P07477>

16. <https://www.uniprot.org/uniprot/P07477.xml>

```

1 | <?xml version='1.0' encoding='UTF-8'?>
2 | <uniprot xmlns="http://uniprot.org/uniprot" xmlns:xsi=[...]>
3 | <entry dataset="Swiss-Prot" created="1988-04-01" modified="2018-09-12" [...]>
4 | <accession>P07477</accession>
5 | <accession>A1A509</accession>
6 | <accession>A6NJ71</accession>
7 | [...]
8 | <gene>
9 |   <name type="primary">PRSS1</name>
10 |  <name type="synonym">TRP1</name>
11 |  <name type="synonym">TRY1</name>
12 |  <name type="synonym">TRYP1</name>
13 | </gene>
14 | [...]
15 | <sequence length="247" mass="26558" checksum="DD49A487B8062813" [...]>
16 | MNPLILTFVAAALAAPFDDDKIVGGYNEEENSPVYQVSLNSGYHFCGGSLINEQWVVS
17 | AGHCYKSRIEHNIEVLEGNEQFINAAKIIIRHPQYDRKTLNNNDIMLIKLSRAVIN
18 | ARVSTISLPLTAPPATGKCLISGWGNTASSGADYPDELQCLDAPVLSQAKCEASYPGKIT
19 | SNMFCVGFLEGGKDSCQGDGGPVVCNGQLQGVVSWGDCAQKNKPGVYTKVYNVVKIK
20 | NTIAANS
21 | </sequence>
22 | </entry>
23 | [...]
24 | </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'agit d'une balise ouvrante avec plusieurs attributs (`dataset="Swiss-Prot"`, `created="1988-04-01"`...).

Les lignes 4-6 précisent les numéros d'acquisition dans la base de données UniProt qui font référence à cette même protéine.

Les lignes 8-13 listent les quatre gènes correspondants à cette protéine. Le premier gène porte l'attribut `type="primary"` est 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 15-21 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("gene : {} ({})".format(gene.text, gene.get("type")))
13|
14| sequence = root.xpath("/uniprot/entry/sequence")[0]
15| print("sequence: {}".format(sequence.text.strip()))
16| print("length: {}".format(sequence.get("length")))

```

Ligne 1. On utilise le sous-module `etree` du module `lxml` pour lire le fichier 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 11. 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")`.

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 :

```

1 | gene : PRSS1 (primary)
2 | gene : TRP1 (synonym)
3 | gene : TRY1 (synonym)

```

```

4 | gene : TRYP1 (synonym)
5 | sequence: MNPLLILTFVAAALAAPFDDDKIVGGYNCEENSPYQVSLNSGYHFCGGSLINEQWVVS
6 | AGHCYKSRIQVRLGEHNIEVLEGNEQFINAAKIRHPYDRKTLNNNDIMLIKLSSRAVIN
7 | ARVSTISLPTAPPATGKCLISGWGNTASSGADYPDELQCLDAPVLSQAKCEASYPGKIT
8 | SNMFCVGFLEGGKDSCQGDSGGPVVCNGQLQGVVSWGDGCAQKNKPGVYTKVVNYVKWIK
9 | NTIAANS
10| length: 247

```

A.4.2 CSV et TSV

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 tableau.

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
1B3E	Homo sapiens	1998-12-09	330	36505.50
1D3K	Homo sapiens	1999-09-29	329	36407.40
1D4N	Homo sapiens	1999-10-04	329	36399.40
1DOT	Anas platyrhynchos	1995-08-03	686	75731.80
[...]	[...]	[...]	[...]	[...]

Voici maintenant l'équivalent en CSV¹⁷ :

```

1 | PDB ID,Source,Deposit Date,Length,MW
2 | 1A8E,Homo sapiens,1998-03-24,329,36408.40
3 | 1A8F,Homo sapiens,1998-03-25,329,36408.40
4 | 1AIV,Gallus gallus,1997-04-28,686,75929.00
5 | 1AOV,Anas platyrhynchos,1996-12-11,686,75731.80
6 | 1B3E,Homo sapiens,1998-12-09,330,36505.50
7 | 1D3K,Homo sapiens,1999-09-29,329,36407.40
8 | 1D4N,Homo sapiens,1999-10-04,329,36399.40
9 | 1DOT,Anas platyrhynchos,1995-08-03,686,75731.80
10| [...]

```

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 TSV¹⁸ est :

```

1 | PDB ID    Source   Deposit Date   Length   MW
2 | 1A8E      Homo sapiens 1998-03-24 329 36408.40
3 | 1A8F      Homo sapiens 1998-03-25 329 36408.40
4 | 1AIV      Gallus gallus 1997-04-28 686 75929.00
5 | 1AOV      Anas platyrhynchos 1996-12-11 686 75731.80
6 | 1B3E      Homo sapiens 1998-12-09 330 36505.50
7 | 1D3K      Homo sapiens 1999-09-29 329 36407.40
8 | 1D4N      Homo sapiens 1999-10-04 329 36399.40
9 | 1DOT      Anas platyrhynchos 1995-08-03 686 75731.80
10| [...]

```

Sur chaque ligne, les différentes valeurs sont séparées par une tabulation.

Attention

17. https://python.sdv.univ-paris-diderot.fr/data-files/transferrin_report.csv
18. https://python.sdv.univ-paris-diderot.fr/data-files/transferrin_report.tsv

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.

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 module¹⁹.

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 fichiers CSV.

Ligne 6. Affichage des champs correspondants à *PDB ID*, *Deposit Date*, *Length*.

Le résultat obtenu est :

```
1 1A8E 1998-03-24 329
2 1A8F 1998-03-25 329
3 1AIV 1997-04-28 686
4 1AOV 1996-12-11 686
5 1B3E 1998-12-09 330
6 1D3K 1999-09-29 329
7 [...]
```

Il suffit de modifier légèrement le script précédent pour lire un fichier TSV :

```
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.

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 :

19. <https://docs.python.org/fr/3.7/library/csv.html>

```

1 | Name ,Quantity
2 | girafe ,5
3 | tigre ,3
4 | 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 :

```

1 | Name      Quantity
2 | girafe    5
3 | tigre     3
4 | 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 17 *Quelques modules d'intérêt en bioinformatique* 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.

Annexe B

Installation de Python

Python est déjà présent sous Linux ou Mac OS X et s’installe très facilement sous Windows. Toutefois, on décrit 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** : Anaconda¹ 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. Miniconda² 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** : pip³ 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, on recommande 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 très efficacement.

Par ailleurs, on vous recommande 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, on part 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 (7 et 10), Mac OS C (Mac OS High Sierra version 10.13.6) et Linux (Ubuntu 16.04, Ubuntu 18.04).

B.2 Installation de Python avec Miniconda

Nous vous conseillons l’installation de la distribution Miniconda⁴ 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 :

1 | \$

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 *64-bit (bash installer)* correspondant à Linux et Python 3.7. Bien sur, si votre machine est en 32-bit (ce qui est maintenant assez rare), vous cliquerez sur le lien *32-bit (bash installer)*.

1. <https://www.anaconda.com/>
2. <https://conda.io/miniconda.html>
3. <https://pip.pypa.io/en/stable/>
4. <https://conda.io/miniconda.html>

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 :

```
1 | $ bash Miniconda3-latest-Linux-x86_64.sh
```

Dans un premier temps, validez la lecture de la licence d'utilisation :

```
1 | Welcome to Miniconda3 4.5.11
2 |
3 | In order to continue the installation process, please review the license
4 | agreement.
5 | Please, press ENTER to continue
6 | >>>
```

En appuyant sur la touche *Espace* faites défiler la licence d'utilisation puis tapez `yes` puis appuyez sur la touche *Entrée* pour la valider :

```
1 | Do you accept the license terms? [yes|no]
2 | [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* :

```
1 | Miniconda3 will now be installed into this location:
2 | /home/pierre/miniconda3
3 |
4 | - Press ENTER to confirm the location
5 | - Press CTRL-C to abort the installation
6 | - Or specify a different location below
7 |
8 | [/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 de modifier le fichier de configuration de votre *shell* Bash pour que *conda* soit pris en compte (c'est-à-dire accessible à chaque fois que vous ouvrez un *shell*). Nous vous conseillons d'accepter en tapant `yes` puis en appuyant sur la touche *Entrée*.

```
1 | Do you wish the installer to initialize Miniconda3
2 | in your /home/pierre/.bashrc ? [yes|no]
3 | [no] >>> yes
```

L'installation de Miniconda est terminée. L'espace utilisé par Miniconda sur votre disque dur est d'environ 300 Mo.

Test de l'interpréteur Python

Ouvrez un nouveau *shell*. À partir de maintenant, lorsque vous taperez la commande `python`, c'est le Python 3 de Miniconda qui sera lancé :

```
1 | $ python
2 | Python 3.7.1 (default, Dec 14 2018, 19:28:38)
3 | [GCC 7.3.0] :: Anaconda, Inc. on linux
4 | Type "help", "copyright", "credits" or "license" for more information.
5 | >>>
```

Quittez Python en tapant la commande `exit()` puis appuyant sur la touche *Entrée*.

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 :

```
1 | $ conda
2 | usage: conda [-h] [-V] command ...
3 |
4 | conda is a tool for managing and deploying applications, environments and packages.
5 |
6 | Options:
7 |
8 | positional arguments:
9 |   command          Remove unused packages and caches.
10 |   clean
11 |   [...]
```

Si c'est bien le cas, bravo, *conda* est bien installé et vous pouvez passer à la suite (rendez-vous à la rubrique [Installation des modules supplémentaires](#)) !

Désinstallation de Miniconda

Si vous souhaitez supprimer Miniconda, rien de plus simple, il suffit de suivre ces deux étapes :

1. Supprimer le répertoire de Miniconda. Par exemple pour l'utilisateur pierre : \$ rm -rf /home/pierre/miniconda3
2. Restaurer votre fichier de configuration du *shell Bash* en utilisant la copie de sauvegarde qu'a créée Miniconda lors de l'installation : \$ mv .bashrc-miniconda3.bak .bashrc

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 *64-bit (bash installer)* correspondant à Mac OS X et Python 3.7. Sous Mac, seule la version 64-bit est disponible.

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 7 et 10

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 Python⁵) n'est installée sur votre ordinateur. Si tel est le cas, nous recommandons vivement de la désinstaller pour éviter des conflits de version.

Par ailleurs, la procédure détaillée ci-dessous rendra la version de Miniconda **prioritaire** sur toute autre version de Python (raison pour laquelle nous vous demandons de cocher une case non recommandée par l'installateur, cf. Figure B.6). Si vous désinstallez toute version de Python existante, tout se passera sans problème.

Dans un navigateur internet, ouvrez la page du site Miniconda <https://conda.io/miniconda.html> puis cliquez sur le lien *64-bit (exe installer)* correspondant à Windows et Python 3.7. Bien sûr, si votre machine est en 32-bit (ce qui est maintenant assez rare), vous cliquerez sur le lien *32-bit (exe installer)*.

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 :

Cliquez sur *Next*, puis vous arrivez sur :

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 3.7* cochée et cochez la case *Add Anaconda to my PATH environment variable*. En cochant cette dernière option, le texte s'est mis en rouge car ce n'est pas une option recommandée. Nous vous recommandons de choisir tout de même cette option car elle permet d'avoir accès à conda (et donc Python) dans le *shell* Windows nommé PowerShell (*shell* beaucoup plus puissant que l'ancien *shell* nommé cmd qui devient progressivement obsolète).

Cliquez ensuite sur *Install*, l'installation se lance et durera quelques minutes :

À la fin, vous obtiendrez cette fenêtre :

Cliquez sur *Next*, vous arriverez sur la dernière fenêtre :

⁵. <https://www.python.org/downloads/>



FIGURE B.1 – Installation Miniconda étape 1.

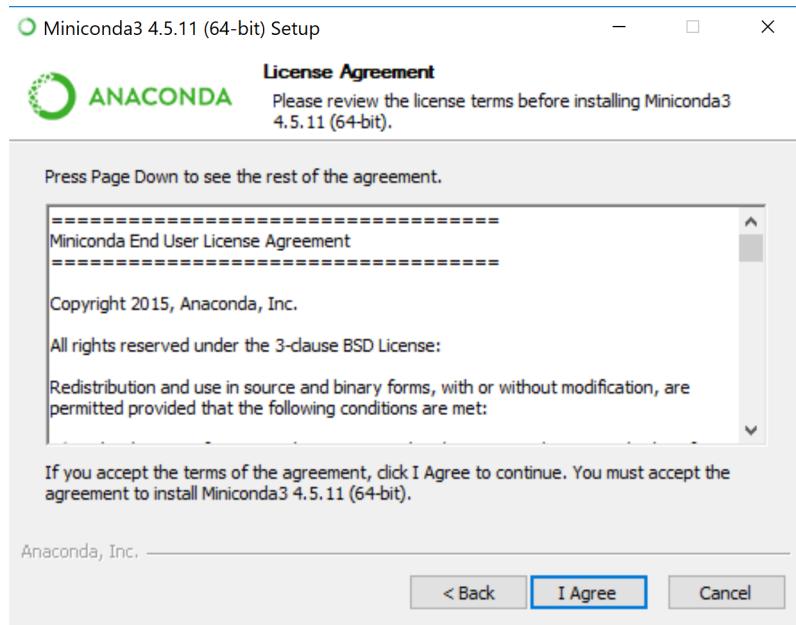


FIGURE B.2 – Installation Miniconda étape 2.

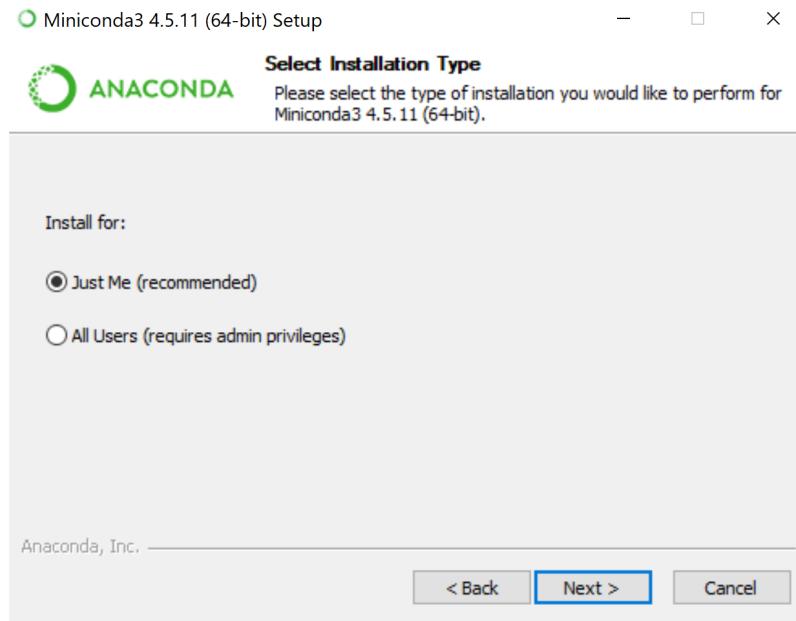


FIGURE B.3 – Installation Miniconda étape 3.

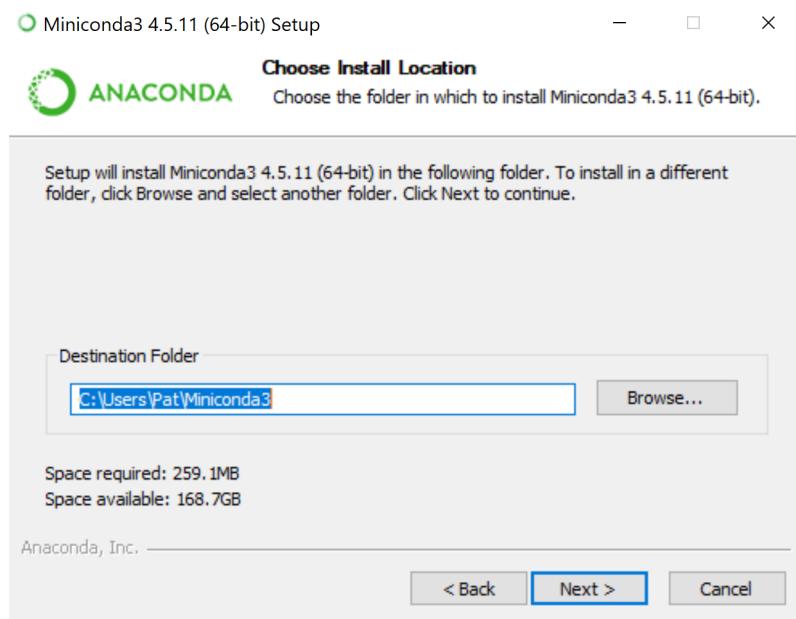


FIGURE B.4 – Installation Miniconda étape 4.

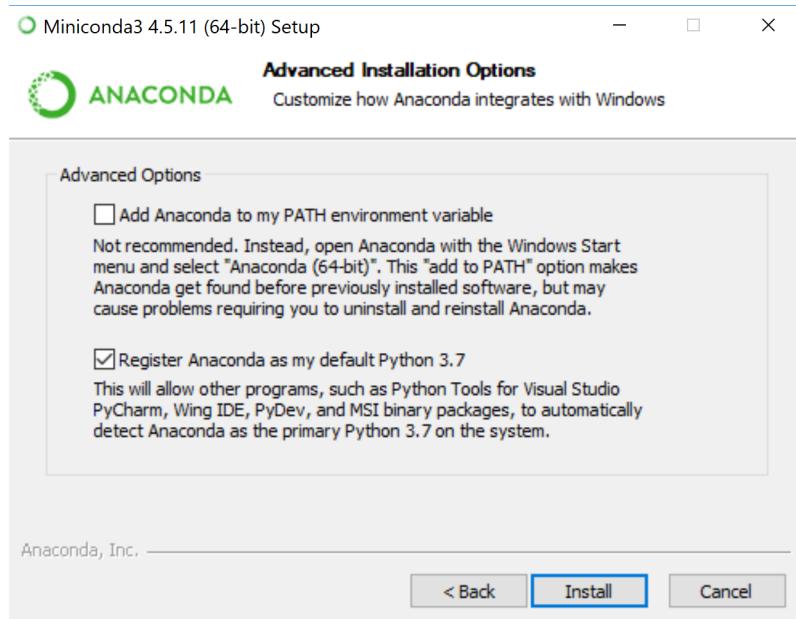


FIGURE B.5 – Installation Miniconda étape 5.

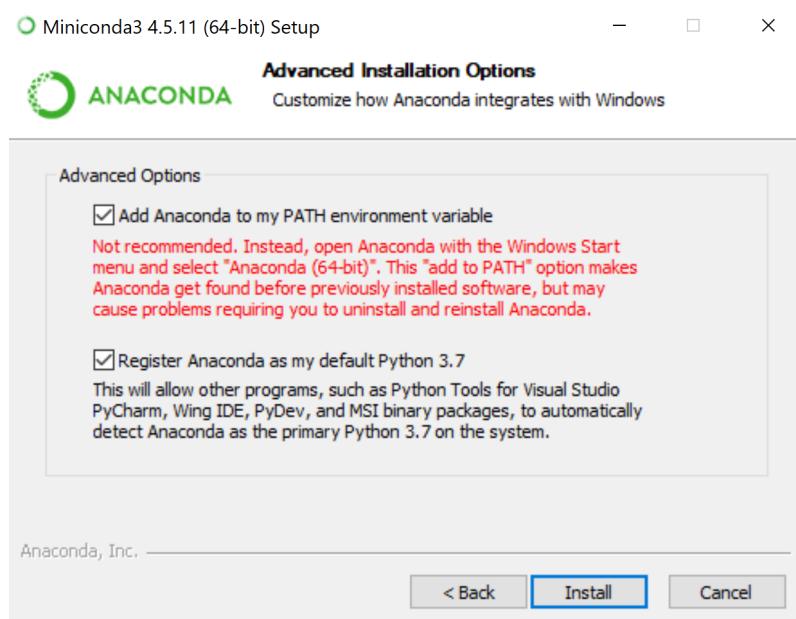


FIGURE B.6 – Installation Miniconda étape 5bis

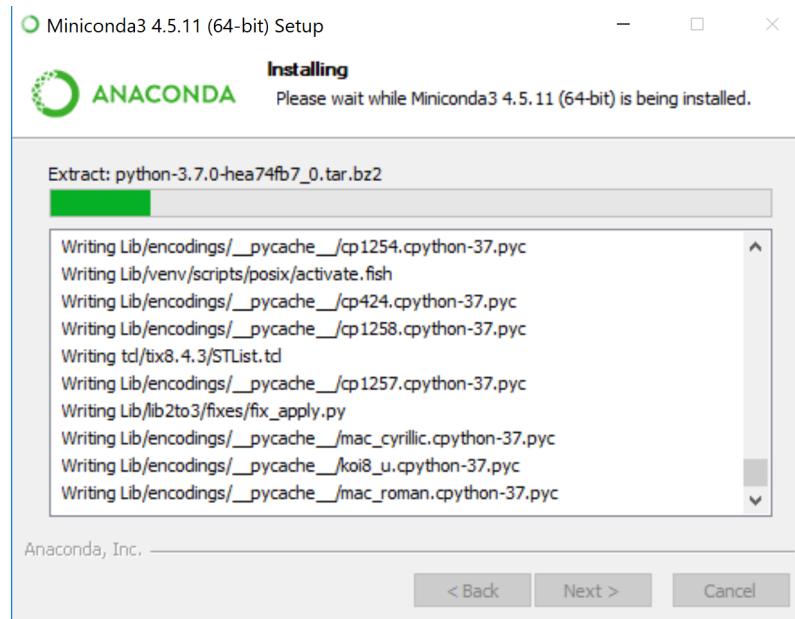


FIGURE B.7 – Installation Miniconda étape 6.

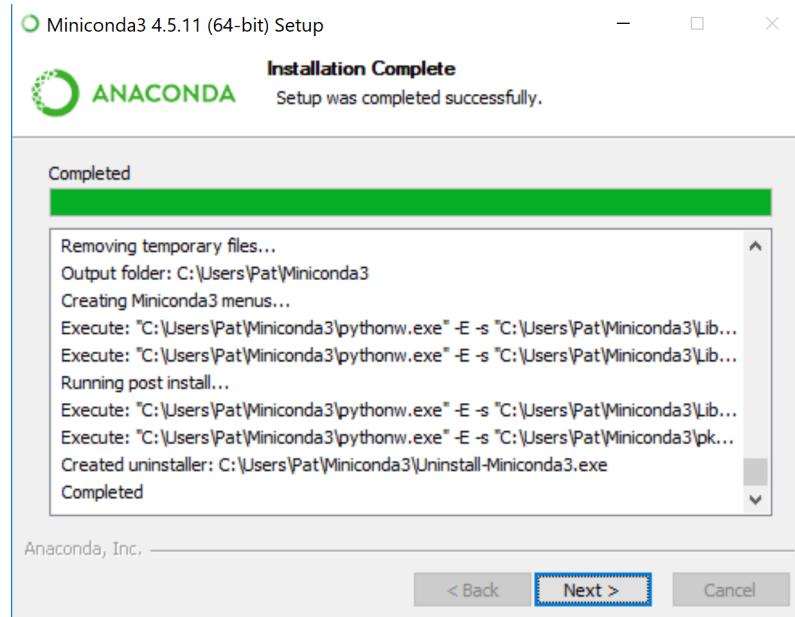


FIGURE B.8 – Installation Miniconda étape 7.

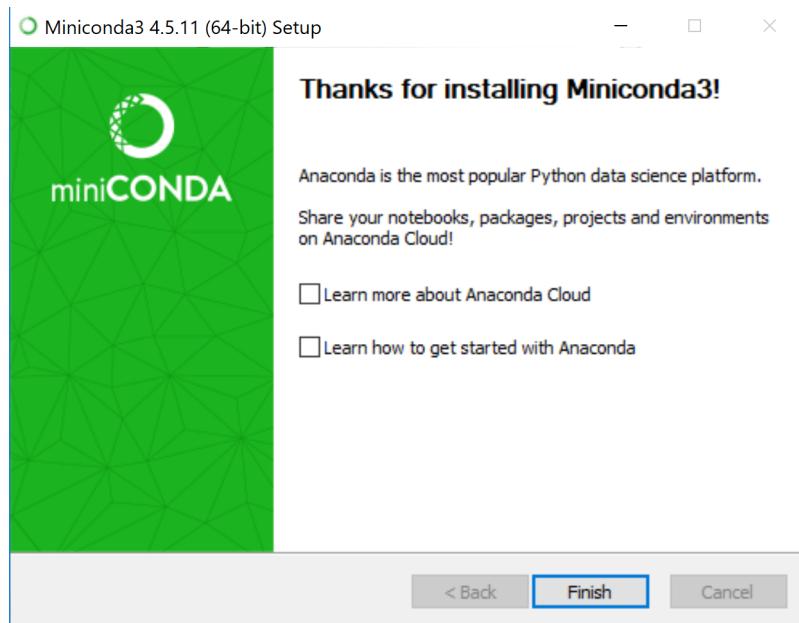


FIGURE B.9 – Installation Miniconda étape 8.

Décochez les cases *Learn more about Anaconda Cloud* et *Learn how to get started with Anaconda* et cliquez sur *Finish*. Miniconda est maintenant installé.

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 apparaître 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). 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 l'affichage suivant :

```
1 PS C:\Users\Pat> python
2 Python 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)] :: Anaconda, Inc. on win32
3 Type "help", "copyright", "credits" or "license" for more information.
4 >>>
```

Cela signifie que vous êtes bien dans l'interpréteur Python. À partir de là vous pouvez taper exit() puis appuyer sur la touche Entrée pour sortir de l'interpréteur Python.

Test du gestionnaire de paquets conda

Une fois revenu dans le *shell*, tapez la commande conda, vous devriez obtenir :

```
1 PS C:\Users\Pat> conda
2 usage: conda-script.py [-h] [-V] command ...
3
4 conda is a tool for managing and deploying applications, environments and packages.
5
6 Options:
7
8 positional arguments:
9   command
10    clean      Remove unused packages and caches.
11    config     Modify configuration values in .condarc. This is modeled
12                  after the git config command. Writes to the user .condarc
13                  file (C:\Users\Pat\.condarc) by default.
14    create     Create a new conda environment from a list of specified
15                  packages.
16    help       Displays a list of available conda commands and their help
17                  strings.
18    info       Display information about current conda install.
```

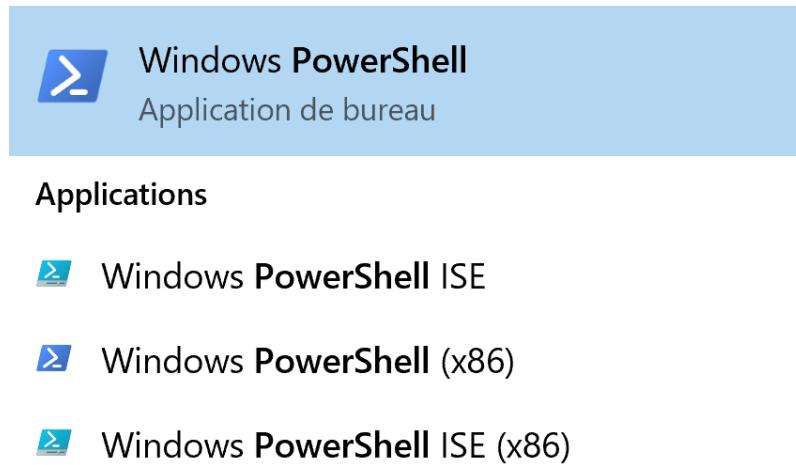


FIGURE B.10 – Menu pour lancer un PowerShell.

A screenshot of a Windows PowerShell window titled "Windows PowerShell". The window shows the following text:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Tous droits réservés.

PS C:\Users\Pat> python
Python 3.7.1 (default, Dec 10 2018, 22:54:23) [MSC v.1915 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

The window has a dark blue background and a light blue title bar.

FIGURE B.11 – Lancement de l'interpréteur Python dans un PowerShell.

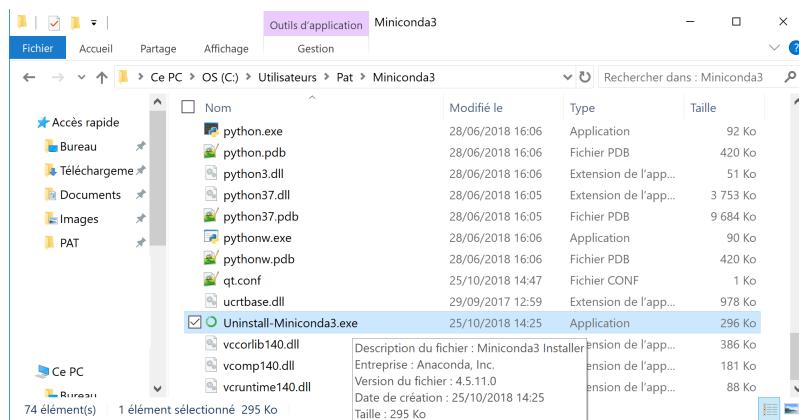


FIGURE B.12 – Désinstallation de Miniconda (étape 1).

```

19      init          Initialize conda for shell interaction. [Experimental]
20  install       Installs a list of packages into a specified conda
21          environment.
22  list          List linked packages in a conda environment.
23  package       Low-level conda package utility. (EXPERIMENTAL)
24  remove        Remove a list of packages from a specified conda environment.
25  uninstall     Alias for conda remove.
26  run           Run an executable in a conda environment. [Experimental]
27  search        Search for packages and display associated information. The
28          input is a MatchSpec, a query language for conda packages.
29          See examples below.
30  update        Updates conda packages to the latest compatible version.
31  upgrade       Alias for conda update.
32
33 optional arguments:
34 -h, --help      Show this help message and exit.
35 -V, --version   Show the conda version number and exit.
36
37 conda commands available from other packages:
38   env
39 PS C:\Users\Pat>

```

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](#)) !

Désinstallation de Miniconda

Si vous souhaitez désinstaller Miniconda, rien de plus simple. Dans un explorateur, dirigez-vous dans le répertoire où vous avez installé Miniconda (dans notre exemple il s'agit de C:\Users\ votre_nom_utilisateur\Miniconda3). Attention, si votre Windows est installé en français, il se peut qu'il faille cliquer sur C:\ puis sur Utilisateurs plutôt que Users comme montré ici :

Cliquez ensuite sur le fichier *Uninstall-Miniconda3.exe*. Vous aurez alors l'écran suivant :

Cliquez sur *Next*, puis à l'écran suivant cliquez sur *Uninstall* :

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.

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* :

```
1 | $ conda install numpy pandas matplotlib scipy biopython jupyterlab
```



FIGURE B.13 – Désinstallation de Miniconda (étape 2).

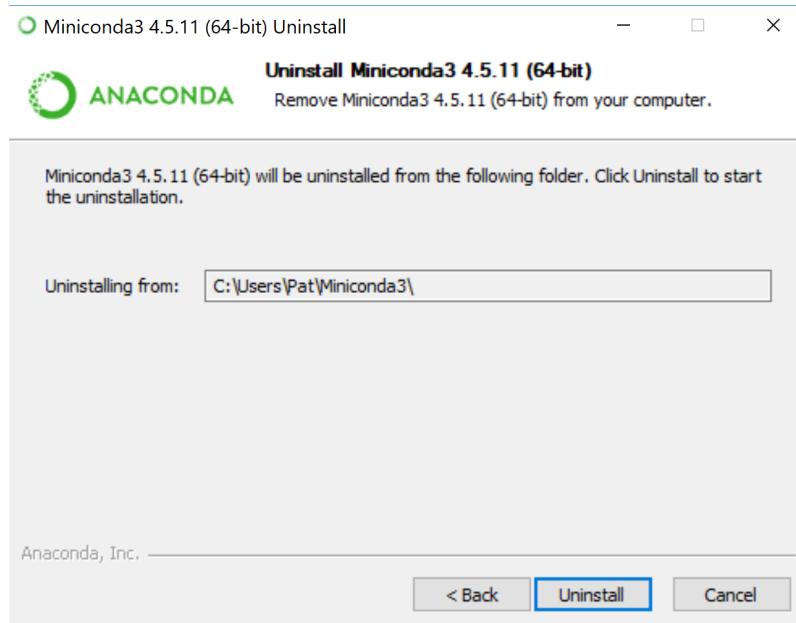


FIGURE B.14 – Désinstallation de Miniconda (étape 3).

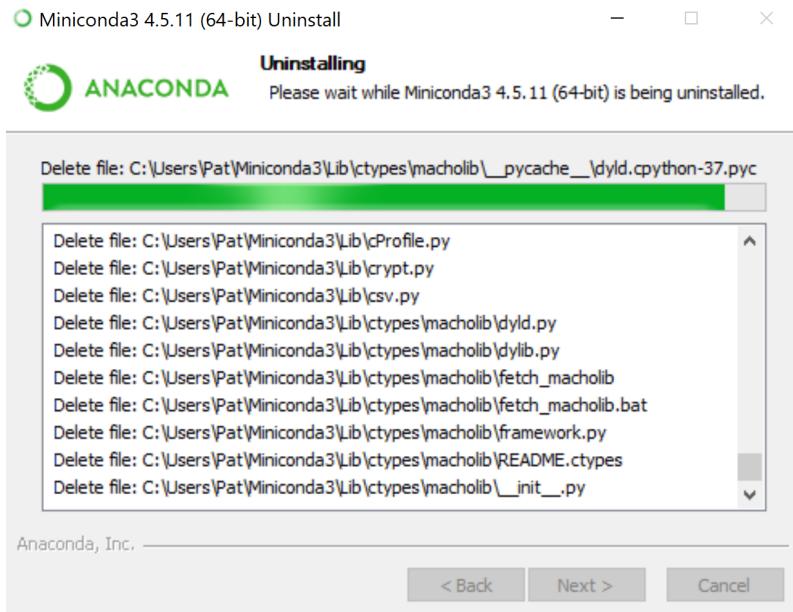


FIGURE B.15 – Désinstallation de Miniconda (étape 4).

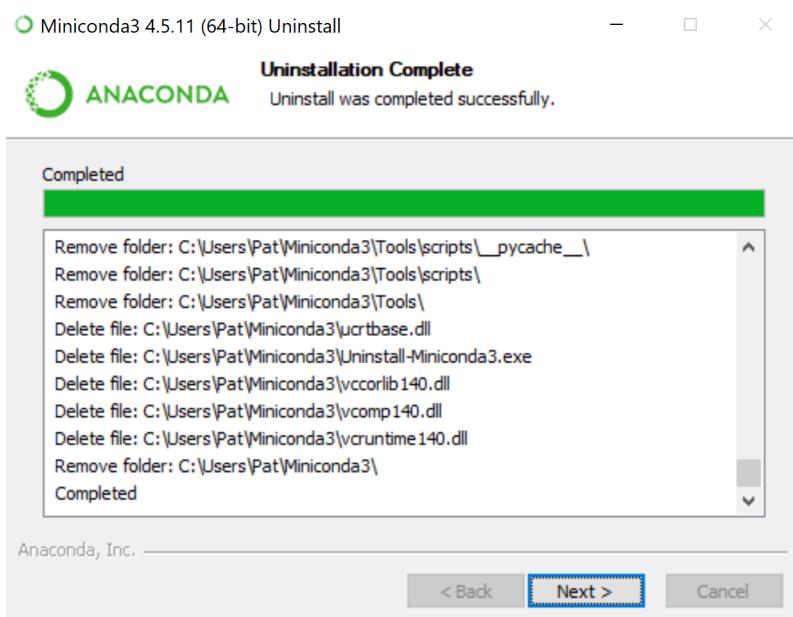


FIGURE B.16 – Désinstallation de Miniconda (étape 5).

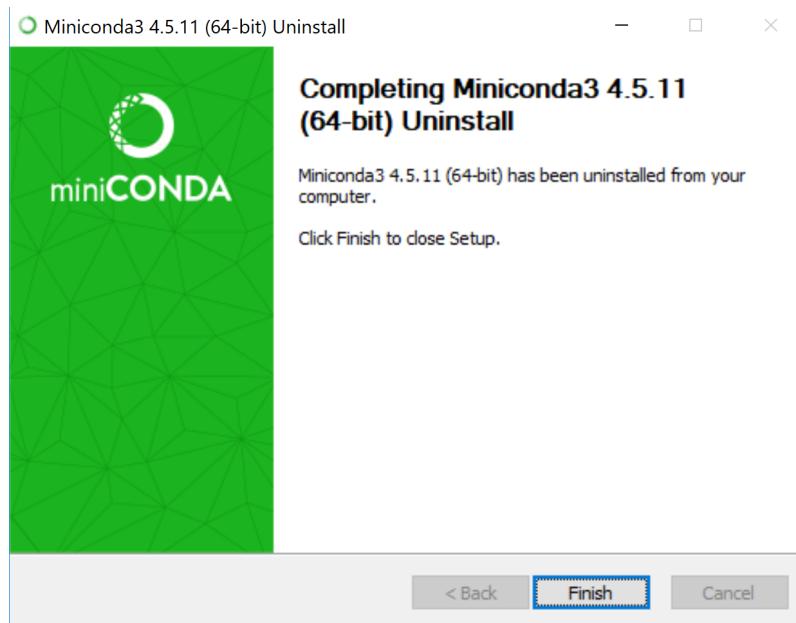


FIGURE B.17 – Désinstallation de Miniconda (étape 6).



FIGURE B.18 – Installation de packages avec conda (étape 1).

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 internet 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 (copies d’écran prise sous Windows avec le PowerShell) :

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 packages (cela prendra quelques minutes) :

Une fois que tout cela est terminé, vous récupérez la main dans le *shell* :

B.3.2 Test des modules supplémentaires

Pour tester la bonne installation des modules, lancez l’interpréteur Python :

```
1 | $ python
```

Puis tapez les lignes suivantes :

```
1 | import numpy
2 | import scipy
3 | import Bio
4 | import matplotlib
```

```

Sélection Windows PowerShell

pyqt:           5.9.2-py37h6538335_2
python-dateutil: 2.7.5-py37_0
pytz:           2018.9-py37_0
pywinpty:        0.5.5-py37_1000
pyzmq:          17.1.2-py37ha925a31_2
qt:              5.9.7-vc14h73c81de_0
scipy:          1.2.1-py37h29ff71c_0
send2trash:      1.5.0-py37_0
sip:             4.19.8-py37h6538335_0
terminado:       0.8.1-py37_1
testpath:        0.4.2-py37_0
tornado:         5.1.1-py37hfa6e2cd_0
traitlets:       4.3.2-py37_0
wcwidth:         0.1.7-py37_0
webencodings:   0.5.1-py37_1
winpty:          0.4.3-4
zeromq:          4.3.1-h33f27b4_3
zlib:            1.2.11-h62dc97_3

The following packages will be UPDATED:

ca-certificates: 2018.03.07-0      --> 2019.1.23-0
conda:           4.5.12-py37_0      --> 4.6.4-py37_0

Proceed ([y]/n)?

```

FIGURE B.19 – Installation de packages avec conda (étape 2).

```

Windows PowerShell

winpty:          0.4.3-4
zeromq:          4.3.1-h33f27b4_3
zlib:            1.2.11-h62dc97_3

The following packages will be UPDATED:

ca-certificates: 2018.03.07-0      --> 2019.1.23-0
conda:           4.5.12-py37_0      --> 4.6.4-py37_0

Proceed ([y]/n)?

Downloading and Extracting Packages
m2w64-gcc-libs-core- | 213 KB | #####| 100%
zlib-1.2.11          | 128 KB | #####| 100%
pyzmq-17.1.2         | 415 KB | #####| 100%
winpty-0.4.3         | 1.1 MB | #####| 100%
matplotlib-3.0.2    | 6.5 MB | #####| 100%
terminado-0.8.1     | 21 KB | #####| 100%
wcwidth-0.1.7        | 23 KB | #####| 100%
m2w64-gmp-6.1.0     | 689 KB | #####| 100%
ipython_genutils-0.2 | 39 KB | #####| 100%
pandas-0.24.1        | 9.6 MB | #####| 87%

```

FIGURE B.20 – Installation de packages avec conda (étape 3).

```

Windows PowerShell

pygments-2.3.1      | 1.3 MB | #####| 100%
backcall-0.1.0      | 19 KB | #####| 100%
blas-1.0            | 6 KB | #####| 100%
numpy-base-1.15.4   | 3.9 MB | #####| 100%
tornado-5.1.1       | 665 KB | #####| 100%
markupsafe-1.1.0    | 29 KB | #####| 100%
pickleshare-0.7.5   | 13 KB | #####| 100%
freetype-2.9.1       | 470 KB | #####| 100%
jupyterlab-0.35.3   | 10.5 MB | #####| 100%
m2w64-gcc-libs-5.3.0 | 518 KB | #####| 100%
libpng-1.6.36        | 550 KB | #####| 100%
send2trash-1.5.0     | 16 KB | #####| 100%
colorama-0.4.1       | 24 KB | #####| 100%
pyparsing-2.3.1      | 105 KB | #####| 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: | DEBUG menuinst_win32:_init_(196): Menu: name: 'Anaconda${PY_VER} ${PLATFOR
M}', prefix: 'C:\Users\Pat\Miniconda3', env_name: 'None', mode: 'user', used_mode: 'user'
DEBUG menuinst_win32:create(320): Shortcut cmd is C:\Users\Pat\Miniconda3\python.exe, args are ['C:\U
SERS\Pat\Miniconda3\cwp.py', 'C:\Users\Pat\Miniconda3', 'C:\Users\Pat\Miniconda3\python.e
xe', 'C:\Users\Pat\Miniconda3\Scripts\jupyter-notebook-script.py', '"%USERPROFILE%"']
done
PS C:\Users\Pat>
```

FIGURE B.21 – Installation de packages avec conda (étape 4).

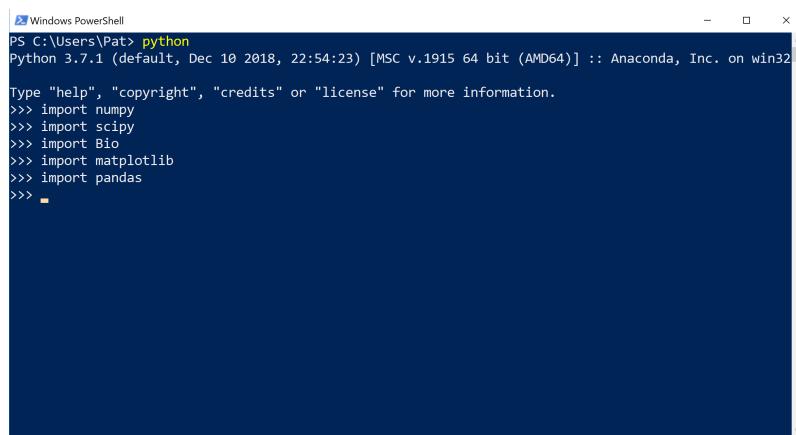


FIGURE B.22 – Test installation de modules Python avec Miniconda.

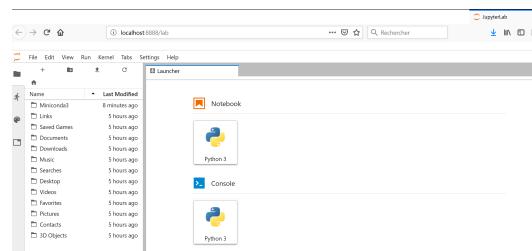


FIGURE B.23 – Test de Jupyter : ouverture dans un navigateur.

```
5| import pandas
```

Vous devriez obtenir la sortie suivante (ici sous Windows) :

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* :

```
1| $ jupyter lab
```

Cette commande devrait ouvrir votre navigateur internet par défaut et lancer Jupyter :

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 :

```
1| PS C:\Users\Pat> jupyter lab
2| [I 18:38:17.644 LabApp] JupyterLab extension loaded from C:\Users\Pat\Miniconda3\lib\site-packages\jupyterlab
3| [I 18:38:17.644 LabApp] JupyterLab application directory is C:\Users\Pat\Miniconda3\share\jupyter\lab
4| [...]
5| [I 18:38:28.904 LabApp] Shutting down on /api/shutdown request.
6| [I 18:38:28.919 LabApp] Shutting down 0 kernels
7| 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 : Pour les débutants, vous pouvez sauter cette rubrique.

FIGURE B.24 – Éditeur de texte *gedit*.

Comme indiqué au début de ce chapitre, pip⁶ 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 présent que dans les dépôts de *conda*. Toutefois, pour les modules classiques (comme *NumPy*, *scipy*, etc), tout est gérable avec *conda*.

Sauf cas exceptionnel, on vous conseille l'utilisation de *conda* pour gérer l'installation de modules supplémentaires.

Si vous souhaitez installer un package qui n'est pas présent sur un dépôt *conda* avec *pip*, la syntaxe est très simple :

```
1 | $ pip install nom_du_package
```

B.4 Choisir un bon éditeur de texte

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

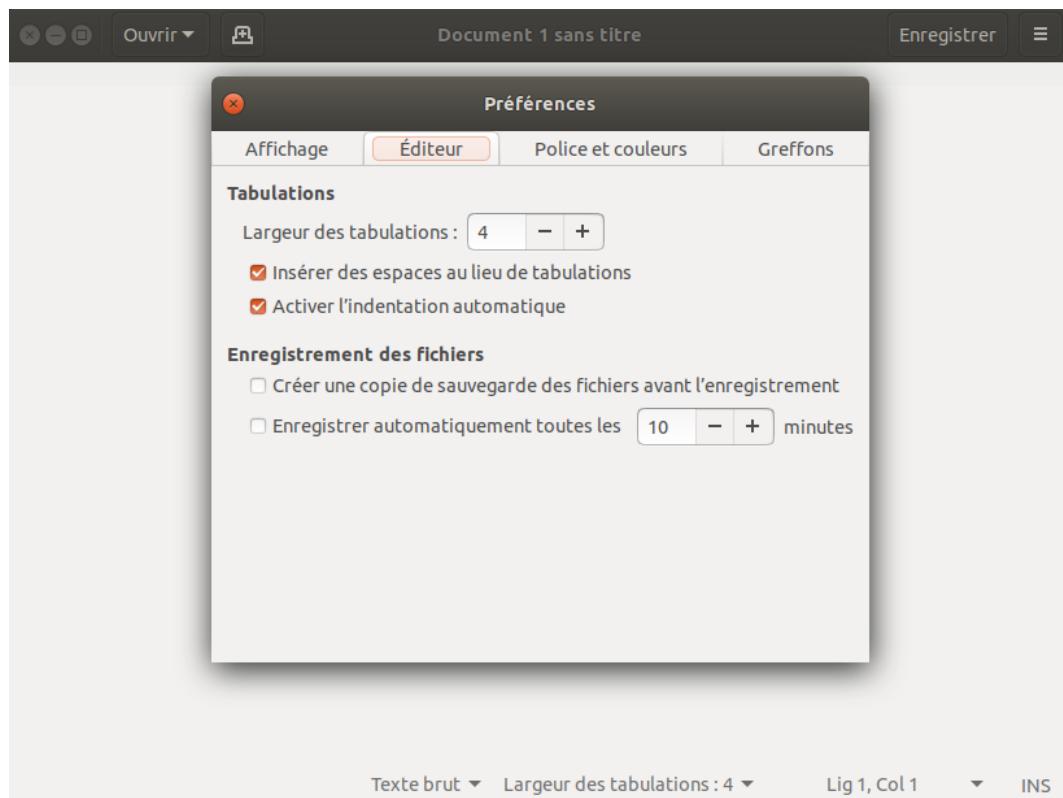
```
1 | $ sudo apt install -y gedit
```

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 de la figure B.24.

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

6. <https://pip.pypa.io/en/stable/>

FIGURE B.25 – Configuration de *gedit*.

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* (comme sur la figure B.25).

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.

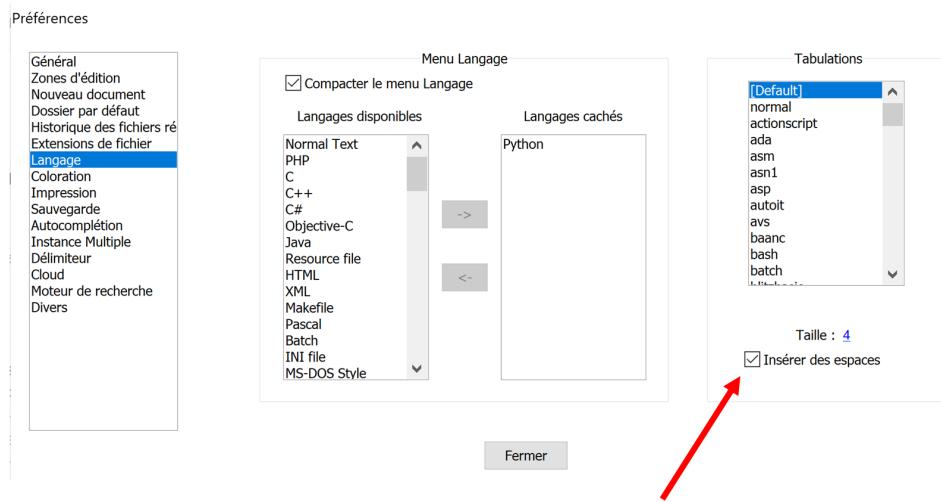
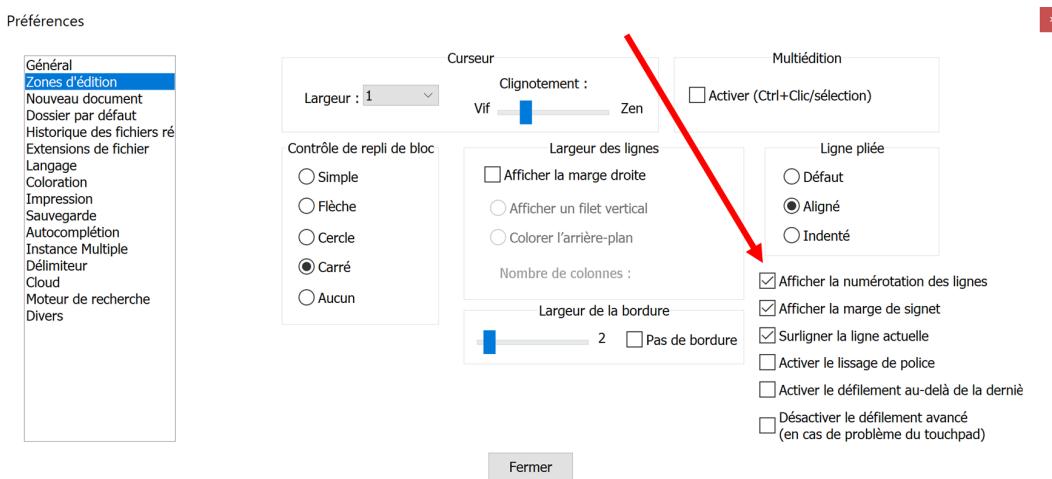
B.4.2 Installation et réglage de Notepad++ sous Windows

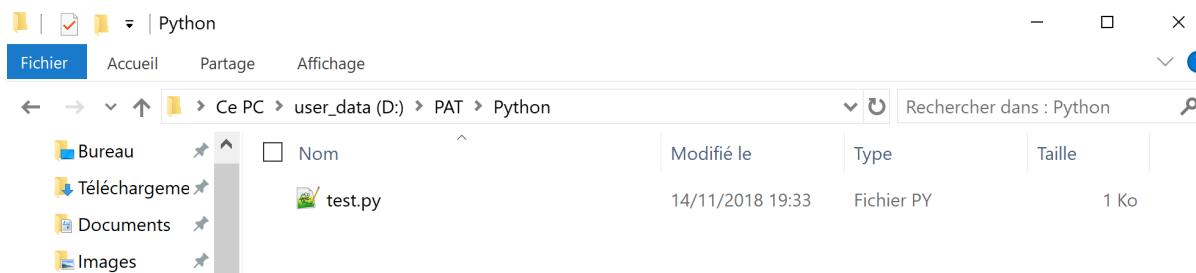
Sous Windows, nous vous recommandons l'excellent éditeur Notepad++⁷. Une fois cet éditeur installé, il est important de le régler correctement. En suivant le menu Paramètres, Préférences, vous arriverez sur un panneau vous permettant de configurer Notepad++.

En premier on va configurer l'appui sur la touche *Tab* afin qu'il corresponde à une indentation de 4 espaces, comme recommandée par la PEP 8 (chapitre 15 *Bonnes pratiques en programmation Python*). Dans la liste sur la gauche, cliquez sur Langage, puis à droite dans le carré Tabulations cochez la case *Insérer des espaces* en réglant sur 4 espaces comme indiqué dans la Figure B.26.

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 dans Figure B.27.

7. <https://notepad-plus-plus.org/download>

FIGURE B.26 – Configuration de *Notepad++* : indentation avec des espaces.FIGURE B.27 – Configuration de *Notepad++* : numéro de ligne.

FIGURE B.28 – Lancement d'un *powershell* depuis un répertoire donné (étape 1).FIGURE B.29 – Lancement d'un *powershell* depuis un répertoire donné (étape 2).

B.4.3 Installation et réglage de TextWrangler/BBedit sous Mac

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 ...

Sous Windows, il existe une astuce très pratique. Lorsqu'on utilise l'explorateur Windows et que l'on est dans un répertoire donné, par exemple :

Il suffit de taper `powershell` dans la barre qui indique le chemin :

puis on appuie sur entrée et le PowerShell se lance en étant directement dans le bon répertoire :

Dans ce PowerShell, 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`.

A 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 !")

```

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*.

```

Windows PowerShell
Copyright (C) Microsoft Corporation. Tous droits réservés.

PS D:\PAT\Python> ls

Répertoire : D:\PAT\Python

Mode          LastWriteTime    Length Name
----          -----          --  --
-a---  14/11/2018      19:33        249 test.py

PS D:\PAT\Python> python test.py
C'est fini !
PS D:\PAT\Python>

```

FIGURE B.30 – Lancement d'un *powershell* depuis un répertoire donné (étape 3).

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.it⁸ ;
- Tutorials Point⁹ ;
- et bien sur l'incontournable Python Tutor¹⁰.

Des applications mobiles vous permettent aussi de « pythonner » avec votre smartphone :

- Pydroid 3¹¹ pour Android ;
- Pythonista 3¹² 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.

8. <https://repl.it/languages/python3>

9. https://www.tutorialspoint.com/execute_python3_online.php

10. <http://pythontutor.com/visualize.html#mode=edit>

11. <https://play.google.com/store/apps/details?id=ru.iiec.pydroid3>

12. <https://itunes.apple.com/us/app/pythonista-3/id1085978097>