Programmation en Python pour les sciences de la vie

2e édition

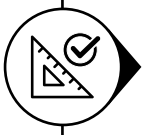
Patrick Fuchs
Maître de conférences à Université Paris Cité

Pierre Poulain
Maître de conférences à Université Paris Cité

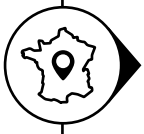
DUNOD

NOUS NOUS ENGAGEONS EN FAVEUR DE L'ENVIRONNEMENT :

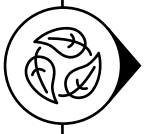
Nos livres sont imprimés sur des papiers certifiés pour réduire notre impact sur l'environnement.



Le format de nos ouvrages est pensé afin d'optimiser l'utilisation du papier.



Depuis plus de 30 ans, nous imprimons 70% de nos livres en France et 25% en Europe et nous mettons tout en œuvre pour augmenter cet engagement auprès des imprimeurs français.



Nous limitons l'utilisation du plastique sur nos ouvrages (film sur les couvertures et les livres).

11, rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-086411-9

86411 – (I) – OSB 80° – PAF/LUM – MGS

Dépôt légal : juin 2024

Achevé d'imprimer par la Nouvelle Imprimerie Laballery 58500 Clamecy
N° d'impression :

Table des matières

Avant-propos

Quelques mots sur l'origine de ce cours

La toute première version de ce cours de Python a été créée en 2003 par Patrick Fuchs, dans le cadre du master de biologie informatique de l'université Paris Diderot - Paris 7. En 2007, Pierre Poulain s'est associé à Patrick Fuchs pour continuer le développement du cours. Bien que destiné en priorité aux étudiants de l'université Paris Diderot, le cours a toujours été proposé sous licence libre et accessible par tout un chacun sur internet. Il est rapidement devenu une ressource importante dans le monde francophone. En 2017, le cours ayant atteint une taille conséquente, l'idée d'en faire un livre a germé. Une première version a ainsi été publiée en 2019 aux éditions Dunod. Cinq ans plus tard, le langage Python ayant beaucoup évolué, il est apparu intéressant d'en proposer une deuxième édition que vous tenez actuellement entre les mains. Nous espérons que cet ouvrage sera utile à un maximum de personnes, scientifiques ou non.

Contenu

Vous apprendrez dans ce livre les bases du langage Python et son utilisation dans les domaines de la biologie et de la bioinformatique. De nombreux exercices s'inspirent d'exemple tirés de ces domaines. Toutefois, l'ouvrage pourra servir également à toute personne ayant quelques notions scientifiques. Les chapitres 1 à 16 traitent des bases du langage. Les chapitres 17 à 23 abordent des notions plus avancées en Python qui peuvent être utiles aux biologistes et aux bioinformaticiens, ou plus généralement, à tout scientifique. Le chapitre 17 traite des expressions régulières, outil très puissant pour rechercher du texte dans des fichiers. Le chapitre 18 présente les *notebooks* Jupyter, véritables « cahiers de laboratoire » interactifs. Les chapitres 19 à 22 introduisent des modules incontournables en bioinformatique et en analyse de données (*Biopython*, *matplotlib*, *NumPy* et *pandas*). Le chapitre 23 introduit le concept très puissant de programmation orientée objet en Python.

L'annexe A présente quelques formats de données rencontrés en biologie et les outils pour les manipuler avec Python.

L'ensemble du cours décrit dans cet ouvrage est toujours accessible en ligne, à l'adresse suivante : `https://python.sdv.u-paris.fr/`.

Vous y trouverez du matériel supplémentaire. Le chapitre 24 aborde des notions plus avancées en programmation orientée objet. Le chapitre 25 introduit la programmation graphique avec le module *Tkinter*. Le chapitre 26 traite de notions complémentaires et vous donne quelques pistes si vous êtes confrontés à un programme écrit en Python 2. Le chapitre 27 vous propose des mini-projets pour développer vos compétences en programmation. Vous trouverez enfin dans l'annexe B des procédures pas à pas décrivant l'installation de Python et des modules scientifiques décrits dans cet ouvrage.

Cette page web est régulièrement mise à jour en fonction des retours de nos étudiants et plus généralement des utilisateurs. Ainsi, il se peut qu'elle diffère du présent ouvrage dans le futur.

Afin de promouvoir le partage des connaissances et le logiciel libre, nos droits d'auteurs provenant de la vente de cet ouvrage seront reversés à deux associations. Wikimédia France [1] qui s'occupe notamment de l'encyclopédie libre Wikipédia. NumFOCUS [2] qui soutient le développement de logiciels libres scientifiques et notamment l'écosystème scientifique autour de Python.

Nous espérons que vous aurez autant de plaisir à apprendre Python que nous en avons à l'enseigner, tous les ans, avec la même ardeur !

Nouveautés par rapport à la première édition

En 2024, au vu de l'importance de l'utilisation de Python en science des données, nous avons décidé de remanier les chapitres par rapport à la première édition. Nous avons donc scinder l'ancien chapitre *Quelques modules d'intérêt en bioinformatique* en autant de chapitres différents : chapitre 18 *Jupyter et ses notebooks*, chapitre 19 *Module Biopython*, chapitre 20 *Module NumPy*, chapitre 21 *Module Matplotlib* et chapitre 22 *Module Pandas*. Ainsi les notions introduites sur ces modules essentiels sont plus approfondis et représentent un bon point de départ pour aborder la gestion de données en Python. Les bases du langage sont bien sûr toujours présentes, mais nous avons également approfondi les notions de conteneurs (chapitre 14), et introduit les dictionnaires et les tuples plus en amont (chapitre 8). L'espace n'étant pas infini, l'introduction sur la programmation orientée objet a été raccourcie (chapitre 23). Toutefois, comme dit ci-dessus, la suite sur la programmation orientée objet (chapitre 24) ainsi que ce qui traite des fenêtres graphiques et du module Tkinter (chapitre 25) sont disponibles sur notre site en ligne.

Remerciements

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

Nous remercions tout particulièrement Sander Nabuurs pour la première version de ce cours remontant à 2003, Denis Mestivier pour les idées de certains exercices et Philip Guo pour son site*Python Tutor* [3].

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

1. https://www.wikimedia.fr/
2. https://numfocus.org/
3. http://pythontutor.com/

1 Introduction

1.1 Qu'est-ce que Python ?

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

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

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

Ce langage de programmation présente de nombreuses caractéristiques intéressantes :

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

1. `https://www.python.org/psf/`
2. Nous sommes d'accord, cette notion est très relative.

concernant cette installation. Tous les détails et la marche à suivre pas-à-pas sont donnés à l'adresse `https://python.sdv.u-paris.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.

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

1.2.2 Miniconda

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

Avec le gestionnaire de paquets *conda*, fourni avec Miniconda, vous pourrez installer des modules supplémentaires qui sont très utiles en bioinformatique (*NumPy*, *scipy*, *matplotlib*, *pandas*, *Biopython*), mais également Jupyter Lab qui vous permettra d'éditer des *notebooks* Jupyter. Vous trouverez en ligne [4] 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 ligne. Bien sûr, si vous préférez d'autres éditeurs comme *Visual Studio Code*, *Sublime Text*, *emacs*, *vim*, *geany*… utilisez-les !

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

1.3 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.

3. `https://conda.io/miniconda.html`
4. `https://python.sdv.u-paris.fr/livre-dunod`

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* Unix [5] fonctionnant sous Mac OS X et Linux ;
- du *shell* PowerShell [6] fonctionnant sous Windows.

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

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

```
$ python
```

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

1.5 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 :

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

pour Mac OS X :

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

ou pour Linux :

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

Les blocs

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

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

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

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

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

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

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

```
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.

En résumant, l'interpréteur fonctionne sur le modèle :

```
1 >>> instruction python
2 résultat
```

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

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

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

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

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

Il existe de nombreux autres langages interprétés comme Perl [7] ou R [8]. Le gros avantage de ce type de langage est qu'on peut immédiatement tester une commande à l'aide de l'interpréteur, ce qui est très utile pour débugger (c'est-à-dire trouver et corriger les éventuelles erreurs d'un programme). Gardez bien en mémoire cette propriété de Python qui pourra parfois vous faire gagner un temps précieux !

1.6 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 [9]) 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 :

```
$ python test.py
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 16 *Bonnes pratiques en programmation Python*.

Voici un exemple :

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

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

Remarque

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

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

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.

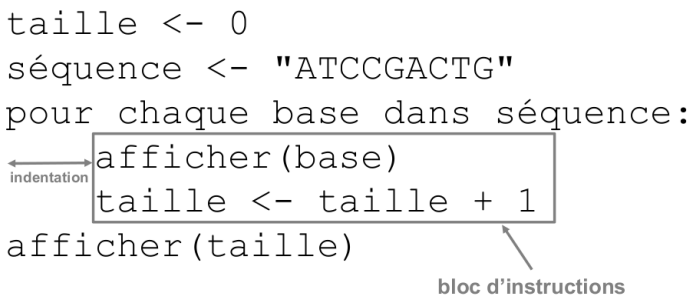


FIGURE 1.1 – Notion d'indentation et de bloc d'instructions.

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

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

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

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

1.9 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 Py-

thon d'une autre manière. Nous vous conseillons les ressources suivantes en langue française :

- Le livre *Apprendre à programmer avec Python 3* de Gérard Swinnen. Cet ouvrage est téléchargeable gratuitement sur le site de Gérard Swinnen [10]. Les éditions Eyrolles proposent également la version papier de cet ouvrage.
- Le livre *Apprendre à programmer en Python avec PyZo et Jupyter Notebook* de Bob Cordeau et Laurent Pointal, publié aux éditions Dunod. Une partie de cet ouvrage est téléchargeable gratuitement sur le site de Laurent Pointal [11].
- Le livre *Apprenez à programmer en Python* de Vincent Legoff [12] que vous trouverez sur le site *Openclassroms*.

Et pour terminer, une ressource incontournable en langue anglaise :

- Le site www.python.org [13]. Il contient énormément d'informations et de liens sur Python. La page d'index des modules [14] est particulièrement utile (et traduite en français).

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

2 Variables

2.1 Définition et création

Définition

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

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

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

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

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

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

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

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

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

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

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

Revenons sur le signe = ci-dessus.

Définition

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

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

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

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

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

Attention

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

2.2 Les types de variables

Définition

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

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

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

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

Remarque

Python reconnaît certains types de variables automatiquement (entier, *float*). Par contre, pour une chaîne de caractères, il faut l'entourer de guillemets (doubles, simples, voire trois guillemets successifs doubles ou simples) afin d'indiquer à Python le début et la fin de la chaîne de caractères.
Dans l'interpréteur, l'affichage direct du contenu d'une chaîne de caractères se fait avec des guillemets simples, quel que soit le type de guillemets utilisé pour définir la chaîne de caractères.
En Python, comme dans la plupart des langages de programmation, c'est le point qui est utilisé comme séparateur décimal. Ainsi, `3.14` est un nombre reconnu comme un *float* en Python alors que ce n'est pas le cas de `3,14`.

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

1. https://docs.python.org/fr/3.12/library/stdtypes.html
2. https://fr.wikipedia.org/wiki/Bool%C3%A9en

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

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

2.3 Nommage

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

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

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

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

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

2.4 Écriture scientifique

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

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

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

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

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

3. `https://thepopulationproject.org/`

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

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

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

2.5 Opérations

2.5.1 Opérations sur les types numériques

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

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

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

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

```
>>> 3 / 4
0.75
>>> 2.5 / 2
1.25
```

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

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

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

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

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

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

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

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

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

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

2.5.2 Opérations sur les chaînes de caractères

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

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

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

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

⚠ Attention

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

2.5.3 Opérations illicites

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

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

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

2.6 La fonction type()

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

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

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

⚠ Attention

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

2.7 Conversion de types

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

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

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

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

2.8 Note sur le vocabulaire et la syntaxe

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

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

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

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

2.9 Minimum et maximum

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

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

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

2.10 Exercices

Conseil

Pour ces exercices, utilisez l'interpréteur Python.

2.10.1 Nombres de Friedman

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

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

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

- $7+3^6$
- $(3+4)^3$
- 3^6-5
- $(1+2^8)\times 5$
- $(2+1^8)^7$

2.10.2 Prédire le résultat : opérations

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

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

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

2.10.3 Prédire le résultat : opérations et conversions de types

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

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