PYTHON de A à Y

krystof

6 octobre 2023

Table des matières

Première partie

Avant de commencer

Chapitre 1

Apprendre à programmer

Python a l'avantage d'être un langage polyvalent, permettant le développement d'applications de bureau ou pour mobiles, comme le développement Web. Il bénéficie d'une documentation fournie, et de nombreux tutoriels et cours figurent sur le net, dans à peu près toutes les langues. La communauté *Python* est importante et active, ainsi trouver de l'aide en ligne sera aisé. De plus c'est un des langages les plus faciles à apprendre. Au regard de tels avantages, nous pouvons dire que le langage *Python* en fait un très bon langage pour qui veut débuter dans la programmation.

Quand on débute dans l'apprentissage de la programmation il est souvent difficile de se centrer sur les concepts fondamentaux sans s'éparpiller. Un tel apprentissage nécessite donc de la rigueur en évitant de vouloir apprendre trop de choses en même temps. Il est nécessaire de rester centré sur un même concept, un même langage. Par la suite, quand les concepts de base seront maîtrisés avec un langage, il sera plus facile d'apprendre un autre langage, voire de s'orienter vers des concepts plus avancés.

Apprendre à coder implique de s'investir, et donc de coder souvent. Un peu chaque jour et vous mesurerez mieux votre progression. Il faut donc, en même temps que l'on apprend de nouveaux concepts, pratiquer, soit à l'aide d'exercices, soit par le développement de mini projets. Cela va donc nécessiter du temps, jusqu'à plusieurs mois d'apprentissage.

Une fois à l'aise avec l'ensemble des concepts de base, il faudra selon ses appétences se centrer sur un *framework* afin de réaliser des projets concrets et avancés. Et là encore, du temps sera nécessaire. L'apprenti développeur pourra se centrer sur :

Le *framework Django* : utilisé pour le développement *WEB*.

Le *framework Kivy* : pour le développement d'applications mobiles et de bureau.

La *data science* .

Chapitre 2

Installons Python et nos outils de développement

En plus de l'installation de *Python* lui-même, nous allons nous doter d'environnements de développement tels que *Pycharm*[1] (spécialement conçu pour le langage *Python*) et *Visual Studio Code*[2]. Nous verrons aussi comment interagir avec l'interpréteur `Python` et comment exécuter des scripts via le terminal. Nous présenterons rapidement quelques *IDE* autres que `Pycharm` et `VSC`, et notamment l'`IDLE Python` fournit avec `Python`. Et nous proposerons quelques outils en ligne, pratiques quand nous voulons coder avec une machine qui ne possède pas d'environnement `Python`.

2.1 *Pycharm*

Installation sous `GNU/Linux`

Télécharger l'archive de la version `Community` de `Pycharm`. Décompresser l'archive et se rendre dans le répertoire idoine puis dans le répertoire `bin`. La commande `./pycharm.sh` exécutée depuis ce même répertoire lancera `Pycharm` (il peut être intéressant de créer un raccourci à partir de cette commande).

Du code en français

Après l'installation il sera nécessaire de désactiver l'option `typo` qui ne reconnaît que l'anglais, et qui signale des erreurs lors de l'emploi de mots dans une autre langue. Même si l'emploi de l'anglais est fortement conseillé quand on code, il faut reconnaître que l'emploi de sa langue d'usage pour les non anglophones aide à la compréhension quand on apprend la programmation (pour l'emploi des noms de variables, de fonctions[3], etc.). Pour ce faire nous allons passer par les étapes suivantes :

```
File > Settings > Editor > Inspections
```

Saisir ensuite `typo` dans la case de recherche, puis décocher la case correspondante à la ligne `Typo`.

Raccourcis	Résultat
`[Shift] + [tab]`	Après avoir sélectionné une partie de code, presser ces touches ôte la tabulation s'il y en a une.
`[Ctrl] + [Z]`	A l'inverse de la commande précédente en remettant la tabulation.

TABLE 2.1 – Des raccourcis d'édition

Les raccourcis *Pycharm* :

2.2 *Visual Studio Code*

Page de téléchargement : `https://code.visualstudio.com/download`. Télécharger le fichier `.deb`, puis lancer la commande :

```
# dpkg -i fichier_telecharge.deb
```

`Visual Studio Code` est alors fonctionnel. Il y a une mise à jour du dépot `apt` incluant la clé de signature qui se fait automatiquement, permettant une mise à jour de l'IDE quand une nouvelle version est publiée sur le site de *Microsoft*.

Il peut être intéressant d'y ajouter le package qui permet de franciser *VSCode* : *French language pack*.

Si le message *Linter Pylint Not Install* apparaît, choisir justement de l'installer.

2.3 L'interpréteur `Python`

Si `Python` a été correctement installé, vous devriez automatiquement disposer d'un interpréteur `Python` qui permet d'utiliser du code de manière interactive.

Pour s'en persuader ouvrons un terminal et exécutons la commande toute simple : `$ python`. L'interpréteur devrait s'ouvrir sur ces lignes :

```
Python 3.10.8 (main, Nov  4 2022, 09:21:25) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Les trois chevrons (`>>>`) nous indiquent que nous sommes alors prêts à saisir du code `Python`. Essayons une toute première ligne de code avec la commande (fonction) `print()` qui nous sert à l'affichage :

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

Après avoir taper `[Entrée]` la ligne suivante affichera le message `Hello World!`, sans les chevrons. Il s'agit en fait de notre sortie console consécutive à l'exécution du code.

Et si nous avions malencontreusement mal saisi notre instruction, oubliant de les guillemets de fin par exemple :

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

1. https ://www.jetbrains.com/pycharm/
2. https ://code.visualstudio.com/
3. Notions qui seront abordés par la suite

Nous aurions alors droit à une petit message d'avertissement de la part de `Python` :

```
File "<stdin>", line 1
  print("Hello World!)
      ^
SyntaxError: unterminated string literal (detected at line 1)
```

Et pour sortir de l'interpréteur, rien de plus simple avec l'instruction `exit()`. Et nous voilà à nouveau dans notre terminal.

Entrer des commandes dans l'interpréteur `Python` de manière interactive est idéal pour tester rapidement et explorer des caractéristiques ou des fonctionnalités. Mais pour des besoins un peu plus élaborer, et surtout pour sauvegarder notre code, nous aurons vire recours aux scripts.

2.4 Les scripts `Python` exécutés depuis le terminal

Un script Python est un ensemble de code réutilisable, autrement dit il s'agit d'une séquence d'instructions `Python` contenue dans un fichier. Les scripts `Python` se présentent sous la forme de texte brut, ce qui vous permet de les modifier avec n'importe quel éditeur de texte (`vim` par exemple).

Passons à la pratique :

1. Créer un fichier portant l'extension `.py` (`hello.py` par exemple).
2. Y écrire du code `Python`, comme par exemple, la simple ligne `python("Hello World!")`).
3. Sauvegarder le fichier ainsi modifier.
4. L'exécuter depuis l'invite de commande avec la commande `python` (`$ python hello.py`, pour suivre notre exemple).

Ce qui nous donnera :

```
$ python hello.py
Hello World!
```

2.5 Interagir avec `Python` via un *IDE*

Un environnement de développement intégré (*IDE*) est une application qui combine plus ou moins toutes les fonctionnalités vues précédemment. Un bon *IDE* doit fournir des fonctionnalités intéressantes telles que :

- La coloration syntaxique du code pour en faciliter la lecture.
- Une aide contextuelle provenant de la documentation connexe ou interne (documentation des fonctions par exemple).
- L'auto-complétion du code qui offre un gain de temps et une aide à la rédaction non négligeables pour le développeur.
- Un bon *IDE* doit aussi proposer un débogueur.

L'*IDLE* `Python`

La plupart des installations `Python` contiennent un *IDE* rudimentaire appelé *IDLE*, pour « *Integrated Development and Learning Environment* » (que l'on peut traduire par « environnement intégré de développement et d'apprentissage »).

Installation sous *GNU/Linux* :

```
# aptitude install idle3 idle-python3.[num_version]
```

Lancement :

```
$ idle
```

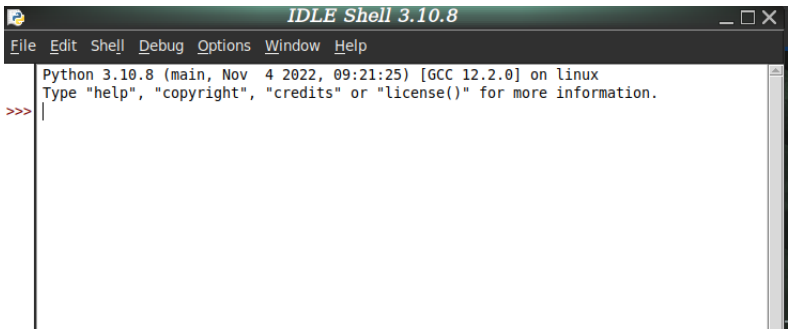

FIGURE 2.1 – Lancement de l'*IDLE* `Python`

A la suite du prompt figuré par les trois chevrons, nous pouvons alors saisir du code `Python` de façon interactive. L'interface *IDLE* ajoute l'avantage d'afficher les différents éléments syntaxiques dans des couleurs distinctes pour rendre les choses plus lisibles. Une aide contextuelle est aussi fournie.

Une autre fonctionnalité intéressante de l'*IDLE* est le rappel des instructions. Les touches `[Alt] + [p]` (revenir en arrière) et `[Alt] + [n]` (avancer) permettent de naviguer dans l'historique de saisie des instructions `Python`.

L'*IDLE* `Python` permet de créer des scripts `Python` et de travailler directement avec. Pour cela :

```
File  --> New file
```

Une fois votre script enregistré vous pouvez l'exécuter via :

```
Run --> Run Module
```

La sortie apparaît alors dans la fenêtre shell dans l'interpréteur.

Pour mieux connaître l'ensemble des fonctionnalités de l'*IDLE* se reporter à la documentation officielle : `https://docs.python.org/fr/3/library/idle.html`

`Thonny` un *IDE* pour `Python`

`Thonny` est un *IDE* `Python` gratuit développé et maintenu par l'Institut d'informatique de l'Université de Tartu, en Estonie. Il s'adresse spécifiquement aux débutants en `Python`, de sorte que l'interface est simple et épurée et qu'il est facile de la comprendre et de s'y habituer rapidement.

Site officiel : `https://thonny.org/`

2.6 Les outils en ligne

Pour s'exercer à coder depuis n'importe quel ordinateur, où *Python* n'est peut-être pas installé, il est possible de faire appel à des outils en ligne tels que la console interactive du site officiel[4], *Python Cloud IDE*[5], *replit.com*[6] ou *trinket*[7] qui nous permettent d'écrire du code depuis un navigateur Internet. Si cela peut s'avérer pratique, on ne pourra toutefois pas utiliser des outils avancés.

2.7 `Jupyter Notebook`

`Jupyter Notebook` est une application *web* qui permet de stocker des lignes de code `Python`, le résultat de l'exécution du code (notamment des graphiques, tableaux, etc.), ainsi que du texte formaté. Elle offre donc la possibilité d'écrire des petits bouts de code exécutables (insérés dans ce que l'on appelle des *cellules*), de les documenter et d'afficher la résultat de l'exécution. Tout cela est ensuite stocké dans un document partageable. C'est une application fréquemment utilisée par ceux qui travaillent dans l'analyse de données.

Installation :

```
# aptitude install jupyter-notebook
```

`Jupyter Notebook` s'ouvre dans un *navigateur*. Pour créer un *notebook* :

1. Cliquer sur `Nouveau`.
2. Puis sur `Python3` (Un nouvel onglet est alors ouvert)
3. Saisir du code `Python` dans une *cellule*, puis lancer `Exécuter`.

Il existe quatre types de *cellules* :

`Code` : *cellule* qui permet d'insérer du code `Python`.

`Heading` : type devenu obsolète dans la mesure car avec le type `markdown` il est beaucoup plus aisé d'insérer des titres.

`Text Brut` **(pour** `NbConvert` : permet de contrôler le formatage du document lors d'une conversion du *notebook* en un autre format.

`Markdown` : permet la documentation du *notebook* à l'aide de *balises* `HTML` ou de la syntaxe `markdown`[8].

4. `https://www.python.org/shell/`
5. `http://pythonfiddle.com/`
6. `www.replit.com`
7. `https://trinket.io/`
8. Pour la syntaxe `markdown` voir : `https://stacklima.com/cellule-markdown-dans-le-bloc-notes-jupyter/`

Deuxième partie

Premiers pas en *Python* : notions de base

Chapitre 3

Comment cela fonctionne-t-il ?

3.1 Des concepts de base en programmation

Nous allons progressivement voir et détailler les notions de *variables*, *fonctions*, *boucles*, *conditions* et *algorithmie*. Il s'agit des notions essentielles de base, communes à tous les langages de programmation.

Nous pouvons établir une similitude avec la recette de cuisine puisque le développeur, en codant, va finalement écrire une recette. L'ordinateur, à l'instar du cuisinier qui va réaliser la recette de cuisine, sera ensuite à exécuter ce code. Si ce code est bien écrit, alors l'ordinateur va l'exécuter sans difficulté, et le rendu de cette exécution sera celui voulu par le développeur. Au contraire, si le code comporte des erreurs, on dit alors qu'il contient des *bugs* et le rôle du développeur va consister à *déboguer* en analysant le code source, c'est-à-dire à en comprendre les erreurs. Il peut, pour être aidé dans sa tâche, faire appel à un *debugger*, logiciel destiné à investiguer le code en vue de retrouver les erreurs.

3.2 Les variables

Une variable est un nom (une étiquette) que l'on associe à une objet. On va alors dire que l'on affecte une valeur à une variable.

Exemple :

```
beurre = 50 (Pour des nombres)
ingredient1 = "lait" (Pour des caractères)
variable = valeur (syntaxe générale)
```

3.3 Les fonctions

En programmation, une fonction est un bloc de code autonome qui encapsule une tâche spécifique ou un groupe de tâches connexes. Les fonctions vont donc nous permettre de représenter un certain nombre d'actions, et vont aussi nous permettre de mieux structurer et organiser le code.

Les fonctions sont capables de retourner (mot clé `return`) un résultat (une valeur par exemple) que l'on peut ensuite affecter à une variable.

3.4 Les conditions

Les conditions, les boucles (cf. infra), mais aussi les fonctions ou les gestions d'erreurs, forment ce que l'on nomme en programmation (notamment utilisées en programmation impérative) les *structures de contrôle*. Les conditions permettent de réaliser des sauts conditionnels en proposant un branchement si une condition est vérifiée. Si la condition n'est pas vérifiée, l'exécution se poursuit séquentiellement. La condition est parfois appelée « condition de rupture » puisqu'elle implique en général une rupture dans le flot d'exécution lorsque la condition est vérifiée.

En `Python` les conditions fournissent le résultat de l'évaluation d'une expression. Les jeux de *structures de contrôle* que l'on rencontre dans le langage `Python` sont basés sur des blocs d'instruction balisés par l'indentation, forme syntaxique qui n'utilise pas de mots clés ou de symboles de ponctuation, comme nous pouvons le rencontrer avec d'autres langages de programmation.

```
Si...
    alors...
Sinon...
```

Exemple avec une recette de cuisine :

```
Si j'ai de la crème
    Alors j'ajoute de la crème
Sinon
    Je n'ajoute pas de crème
```

Mots clés en programmation pour exprimer les conditions : `if` (« *si* ») et `else` (« *sinon* »).

Ainsi, il est possible de réaliser divers tests qui viennent conditionner l'exécution du programme :

Le test « *si* » :

```
if condition :
    instruction
```

Test « *si sinon* » :

```
if condition :
    instruction 1
else:
    instruction 2
```

Test « *sinon si* » :

```
if condition :
    instruction 1
elif autre condition:
    instruction 2
instruction 3
```

Test « *selon* » : en `Python` cela est possible à partir de la version `10`, introduit sous le nom `match case` (cf. le chapitre dédiée aux structures conditionnelles).

3.5 Les boucles

Une boucle est une structure de contrôle destinée à exécuter une portion de code plusieurs fois de suite.

```
Faire n fois:
    mon action
```

En programmation on rencontre des boucles `for` et des boucles `while`.

3.6 Algorithmie (ou « façon de faire »)

3.7 Les objets et la programmation orientée objet (*POO)*

Chapitre 4

Un premier projet à l'aide d'un *IDE*

Tout projet *Python* possède la même structure commune de base :

```
- Répertoire (nom du projet)
      |
      |
      ---- Fichier principal (main.py)
```

Plus le projet est important, plus il contiendra de sous répertoires et fichiers (nous verrons cela en temps voulu).

4.1 Avec *Pycharm*

Au lancement de *Pycharm* on sélectionne `New Project`.

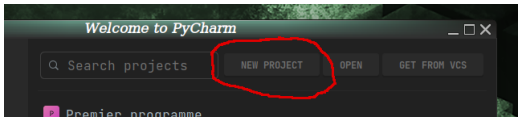

FIGURE 4.1 – Nouveau projet avec *Pycharm*

Puis dans la fenêtre qui s'ouvre on va spécifier le répertoire du projet (`Location`) et l'interpréteur *Python* que l'on va utiliser (`Base interpreter`). Ensuite, cliquer sur `Create`. Pour le reste on ne s'en occupe pas pour le moment.

Pour créer le premier fichier (notre `main.py`), faire un clic droit sur le nom du projet qui se situe dans le panneau de gauche (panneau *project*). Sélectionner :

```
New > Python File
```

Et nommer ce fichier `main`.

Pour illustrer l'exécution du code avec *Pycharm* nous allons tout d'abord insérer la ligne suivante dans le fichier `main.py` [1] :

```
print("Bonjour")
```

La première fois que l'on lance un programme avec *Pycharm* il faut faire un clic droit sur le panneau de l'éditeur, sélectionner dans le menu déroulant qui s'affiche `Run "main"`. Cette

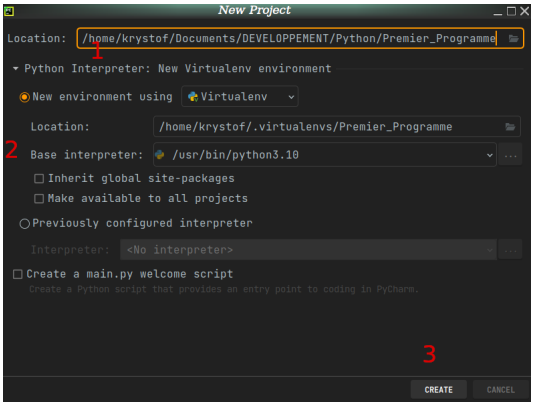

FIGURE 4.2 – Création du projet avec *Pycharm*

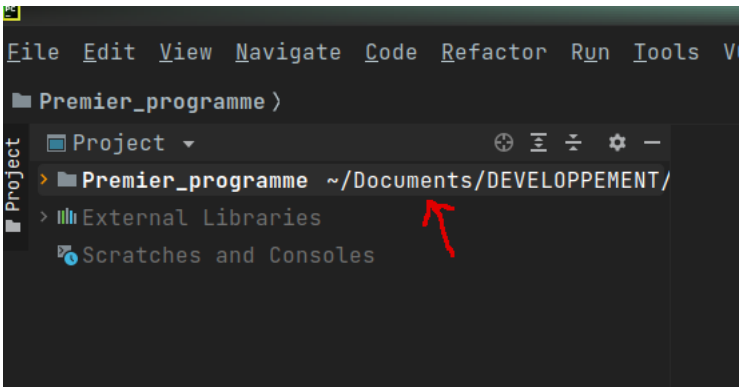

FIGURE 4.3 – Premier fichier d'un projet avec *Pycharm*

première fois permet à *Pycharm* de créer une configuration que nous pourrons alors utiliser pour les prochaines exécutions du code, à l'aide du bouton `Run` en haut à gauche de l'*IDE*.

S'ouvre alors le panneau d'exécution du code (panneau *Run*) en bas de l'interface *Pycharm* dans lequel on voit le résultat de notre programme précédé d'une ligne du type :

```
~/.virtualenvs/Premier_programme/bin/python  ~/Premier_
programme/main.py
```

`~/.virtualenvs/Premier_programme/bin/python` représente l'interpréteur auquel nous faisons appel et `~/Premier_programme/main.py` le fichier du code qui sera exécuté. medskip

4.2 Avec *Visual Studio Code*

Avec *VSC* nous ne sommes pas obligés d'en passer par la création d'un projet. Si le projet ne doit contenir qu'un unique fichier, il peut être en effet plus simple et plus léger d'en passer par cet *IDE*. Pour cela :

```
Fichier > Nouveau fichier > Enregistrer sous
```

On doit alors choisir le répertoire et donner un nom au fichier qui doit porter l'extension `.py`.

1. `print()` est une fonction et `"Bonjour"` son paramètre.

FIGURE 4.4 – Lancement du programme avec la touche `Run`.

4.3 L'exécution du code

Dans un programme la lecture des lignes de code est séquentielle, un ordre est donc suivi.

Maintenant que nous avons notre premier fichier, nous allons pouvoir entrer dans le coeur du langage *Python* en commençant par aborder les concepts de base.

Chapitre 5

Les variables

5.1 Une rapide introduction

Affectation d'une valeur à une variable

Insérons la ligne suivante dans notre fichier `main.py` :

```
nom_de_la_personne = "Krystof"
```

A son exécution, rien ne se passe, et pourtant nous venons de créer une nouvelle variable (`nom_de_la_personne`) de type *chaîne de caractères* et qui prend comme valeur la chaîne `Krystof`. Nous venons donc de stocker la valeur dans la variable. Et dorénavant nous pourrons faire référence à cette variable dès que nous en aurons besoin dans notre code. Une variable doit toujours être définie avant d'être utilisée.

Affichons la valeur de notre variable en ajoutant la ligne suivante :

```
print(nom_de_la_personne)
```

Nous pouvons ajouter du texte à cela par une concaténation :

```
print("Bonjour " + nom_de_la_personne)
```

En *Python*, le symbole `+` utilisé avec des chaînes de caractères permet donc la concaténation.

Il est aussi possible d'afficher de manières successive (avec un espace) les paramètres de la fonction *print()* à l'aide d'une virgule, mais cela n'est pas de la concaténation :

```
print("Bonjour", nom_de_la_personne)
```

Notre code est maintenant composé des deux lignes suivantes :

```
1  nom_de_la_personne = "Krystof"
2  print("Bonjour " + nom_de_la_personne)
```

Le résultat une fois ce code exécuté sera :

```
Bonjour Krystof
```

L'évolution d'une variable

Une variable peut se voir réaffecter une valeur selon le principe suivant :

`n = "Krystof"` : Création de la variable.

`n = "Paul"` : Réaffectation de la variable. On lui donne simplement une autre valeur.

5.2 Demander des données à l'utilisateur

Nous allons réaliser cela à l'aide de la fonction `input()`. Cette fonction a pour but de retourner une valeur que nous pouvons stocker dans une variable :

```
reponse = input("Quel est votre nom ? ")
```

reponse = variable

input() = fonction

"Quel est votre nom ? " = paramètre

Revenons à notre code qui devient celui-ci :

```
1  nom_de_la_personne = input("Quel est votre nom ? ")
2  print("Bonjour " + nom_de_la_personne)
```

A la question posée, on saisit un nom :

```
Quel est votre nom ? Krystof [Entrée]
Bonjour Krystof
```

5.3 Autres types de variables

```
age = 30
```

Nous venons d'affecter la valeur `30` à la variable `age` qui par conséquent devient une variable de type numérique, à la différence de si nous avions codé `age = "30"` où du coup la variable `age` aurait été de type chaîne de caractères.

Il existe une fonction `type()` qui permet d'indiquer le type d'une variable.

```
1  nom = "Krystof"
2  age = 30
3  print(type(nom))
4  print(type(age))
```

Ce qui donne en sortie :

```
<class 'str'>
<class 'int'>
```

Nous reviendrons plus tard sur la notions de *classe*. Ce qu'il faut retenir ici c'est que la première variable est identifiée comme `str` (abréviation de *string* qui veut dire « chaîne de caractères »), et la seconde est identifiée comme `int` (abréviation de *integer* qui veut dire « nombre entier »).

Comme autres valeurs numériques, nous avons les nombres décimaux (de type `float`), et les booléens dont les valeurs ne peuvent être que `True` (pour « vrai », de valeur `1`) et `False` (pour « Faux », de valeur `0`).

On ne peut concaténer une chaîne de caractères avec une valeur numérique. Nous aurions alors une erreur de type `TypeError`. Pour réaliser une telle opération il va nous falloir convertir notre valeur numérique en valeur de type chaîne de caractères. Nous allons utiliser la fonction `str()` :

```
1    age = 30
2    print("Notre texte..." + str(age))
```

La conversion peut aussi se réaliser dans l'autre sens avec la fonction `int()`.

```
1    age = "30"
2    ageX2 = int(age) * 2
3    print(ageX2)
```

Listing 5.1 – Conversion avec la fonction `int`

Dans le cas présent cela nous affiche `60`[1]. Par contre si la variable age ne contenait pas une valeur numérique (Si elle avait contenu la valeur `« trente »` par exemple) nous aurions eu alors une erreur de type `ValueError`[2].

En *Python*, les variables peuvent changer de type, ce qui n'est pas possible avec d'autres langages de programmation (Exemple : langage *C*). Mais une telle manipulation est déconseillée. On va alors plutôt faire appel à une autre variable. Il est donc préférable de faire :

```
age_int = int(age)
```

... plutôt que :

```
age = age(int)
```

L'incrémentation d'une variable numérique

```
1    nbre = 0
2    nbre = nbre + 1  # on ajoute 1 à la valeur de la variable
3    nbre += 1  # Autre méthode d'incrémentation
```

Listing 5.2 – Principe de l'incrémentation

1. Nous n'avons pas encore vu les calculs en *Python*, mais comme vous pouvez le voir sur l'exemple le symbole `*` correspond au signe de multiplication.
2. Nous reviendrons plus loin sur les erreurs et les types d'erreurs

La variable de type `float`

Il s'agit d'une valeur numérique décimale.

```
1 var = 1.75
2 print(type(var))
```

Ce qui donne en sortie :

```
<class 'float'>
```

5.4 Les constantes

En *Python* les variables constantes n'existent pas, mais par convention on va utiliser un nom de variable en lettres majuscules pour indiquer que nous souhaitons utiliser cette variable comme constante, c'est-à-dire que l'on en souhaite pas que sa valeur soit modifiée.

```
MA_CONSTANTE = 100  # Par exemple
```

Chapitre 6

Commenter et documenter son code

6.1 L'intérêt de commenter et documenter son code

Un code est bien plus souvent lu qu'il n'est écrit, et un code bien documenté est un code plus aisément lisible, à la fois pour les autres mais aussi pour soi-même (notamment quand nous devons revenir sur notre code plusieurs semaines après). Daniele Procida [1] dit : « *Peu importe la qualité de votre logiciel, car si la documentation n'est pas assez bonne, les gens ne l'utiliseront pas.* ». Il en est de même pour le code, surtout si nous attendons que d'autres viennent contribuer au projet.

Avant d'aller plus avant dans l'apprentissage du langage, apprentissage qui vous permettra de développer des projets de plus en plus fournis en code, nous allons profiter de ce chapitre pour voir comment documenter ce code vous serez amenés à lire et relire, et relire encore. S'il est bien documenter votre relecture en sera souvent bien plus efficace.

6.2 Le commentaire

Les commentaires font partie intégrante de tout programme. Il s'agit de lignes de code qui ne seront pas interprétées, et donc ne seront pas exécutées. Ils sont destinés aux développeurs, car ils décrivent des parties du code, lorsque cela s'avère nécessaire, afin d'en faciliter la compréhension des programmeurs, y compris pour vous-même.

Exemple d'une ligne de code que l'on ne veut pas interpréter, en mettant le symbole `#` devant :

```
# print("Bonjour")
```

On va donc, à l'aide de ce caractère (`#`) pouvoir documenter son code, en expliquant en langage commun ce que fait le programme ou telle et telle ligne de code. Il s'agit en fait de notes insérées dans le code. Les commentaires doivent être brefs, ne dépassant pas quelques phrases.

```
1    # Voici un programme de bienvenue
2    # Ici on apprécie les résultats
```

Listing 6.1 – Exemple de commentaire

1. Directeur de l'ingénierie chez Canonical, responsable de la documentation.

Les commentaires doivent être courts et précis. Selon la PEP 8[2], les commentaires doivent avoir une longueur maximale de 72 caractères. Si la longueur d'un commentaire est supérieure à la limite de caractères, il convient d'utiliser plusieurs lignes pour le commentaire. Deux méthodes sont possibles :

1 - L'emploi d'une succession de dièses (#) :

```
def commentaire_multilignes() :
    # Voici un très bon exemple
    # de la façon dont vous pouvez répartir des commentaires
    # sur plusieurs lignes en Python
```

2 - L'utilisation des chaînes de caractères avec l'emploi de guillemets triples :

```
    """Mon commentaire
    sur plusieurs lignes
    à la fois."""
```

Listing 6.2 – Commentaire sur plusieurs lignes

Bien que cela vous donne la fonctionnalité multi-lignes, ce n'est pas techniquement un commentaire. C'est une chaîne de caractères qui n'est assignée à aucune variable, et qui n'est donc pas appelée ou référencée par votre programme. Cependant, faites attention à l'endroit où vous placez ces « *commentaires* » multi-lignes. Selon l'endroit où ils se trouvent dans votre programme, ils peuvent se transformer en *docstrings*, qui sont des éléments de documentation associés à une fonction ou à une méthode[3].

Commenter avec un *IDE*

Avec *Pycharm* :

Sélectionner la partie du code à commenter, puis :

`Code > Comment with Line Comment`

Même opération pour décommenter (ou `[Ctrl] + [/]` qui est le raccourci clavier).

Avec *VSC* :

Une fois la partie de code sélectionnée, pour commenter et décommenter :

`Edition > Afficher/Masquer le commentaire de ligne`

Raccourci clavier : `[Ctrl] + [Maj] + [:]`

Pour commenter/décommenter sur plusieurs lignes (guillemets triples) :

`Edition > Afficher/Masquer le commentaire de bloc`

Raccourci clavier : `[Ctrl] + [Maj] + [A]`

2. https://peps.python.org/pep-0008/#maximum-line-length
3. Voir infra, « *Documenter son code à l'aide des docstrings* » page 36

6.3 Des usages et des règles pour de bons commentaires

Savoir comment écrire des commentaires en Python peut faciliter la vie de tous les développeurs, y compris la vôtre ! Ils peuvent aider les autres développeurs à comprendre ce que fait votre code, et vous aider à vous familiariser avec votre propre code.

Tout d'abord il faut savoir qu'en phase de développement le commentaire permet de rédiger du pseudo-code, comme par exemple décrire une portion de code avant de passer au codage proprement dit de la fonctionnalité ou des instructions que l'on souhaite implémenter.

De même, plutôt que d'effacer une portion de code que l'on souhaite modifier, il est possible de la conserver en la commentant. Ainsi si le nouveau code que l'on développe ne fonctionne pas aussi bien que le premier, il sera bien plus facile de réimplémenter celui-ci.

Bien commenter

- Avoir des commentaires aussi proches que possible du code décrit ;
- user d'un formatage simple (éviter les tableaux ou les figures `ASCII`) ;
- ne pas inclure d'informations redondantes ;
- concevoir un code qui se commente lui-même [4].

Des commentaires inutiles

- Eviter les redondances du type :

```
1     return a  # Retourne a
```

- Des commentaires qui peuvent s'éviter si le noms donnés aux variables sont suffisamment explicites

```
1     v = 4  # Nombre de vies
2     # Vaut mieux coder:
3     nbre_de_vies = 4
```

- Les commentaires doivent rarement être plus longs que le code qu'ils soutiennent.

6.4 Commenter *vs* documenter

En général, commenter consiste à décrire votre code, afin d'aider le travail des mainteneurs et des développeurs du code. Associés à un code bien écrit, les commentaires aident le lecteur à mieux comprendre votre code, son objectif et sa conception. Le code vous décrit la façon, les commentaires vos indiquent le but poursuivi.

Documenter le code, c'est décrire son utilisation et ses fonctionnalités à vos utilisateurs. Bien que cela puisse être utile dans le processus de développement, le principal public visé est constitué par les utilisateurs.

4. Voir à ce sujet les annotations de type page 55.

6.5 Documenter son code à l'aide des *docstrings*

Outre les commentaires il est possible de documenter son code à l'aide de *docstrings*. Cette documentation sera abordée tout au long du chapitre *Documenter son code et son projet* page 181, mais avant cela nous devrons d'abord aborder diverses notions essentielles de programmation en `Python`.

Sources pour la rédaction de ce chapitre

Writing Comments in Python (Guide) - trad : « *Écrire des commentaires en Python (Guide)* » - Par Jaya Zhané - 5 novembre 2018 - `https://realpython.com/python-comments-guide/`

Chapitre 7

Erreurs et gestion des exceptions

7.1 Les exceptions : tester son code

Dans cette partie on va utiliser des blocs de codes. En langage *Python* un bloc de codes s'initie par le double points (:). Puis concernant les lignes de code qui font partie de ce bloc on les insère après une tabulation. En *Python* la tabulation fait partie de la syntaxe du langage. Dans d'autres langages comme le *C* ce sont les accolades qui délimitent les blocs de codes.

```
1 try:  # Indique de tester le code qui suit
2     age = int(age)
3 except:  # Si ça ne fonctionne pas...
4     print("ERREUR: vous devez saisir un nombre")
5 else:  # Si la ligne sous le try fonctionne...
6     print("Vous avez " + age + " ans.")
```

A partir de la ligne avec `except` nous introduisons une condition à notre code.

Syntaxe de `try`...`except`...`else` :

```
1 try:
2     ...code testé...
3 except:
4     ...code exécuté si le test échoue...
5 else:
6     ...code exécuté si le test est réussi...
```

`except` seul ne va lever une exception que de façon générique. L'exception peut être levée en raison d'une erreur de syntaxe alors que nous avons prévu notre exception pour lever une erreur de type. Il est cependant possible de spécifier le type d'erreur et d'indiquer dans le bloc de codes de gestion d'exception, et même d'y lever plusieurs exceptions en fonction du type d'erreur. Toutefois, il est déconseillé de mettre toutes les lignes de codes dans un même bloc `try`, mais plusieurs blocs de `try` successifs sont envisageables.

Ainsi pour lever une exception d'erreur de type, notre `except` peut s'écrire de la façon suivante :

```
except ValueError:
```

A noter qu'un `except` seul est surligné dans *Pycharm*.

7.2 Afficher le message d'erreur

Il est possible de capturer l'exception et d'afficher le message d'erreur idoine à l'aide du mot clé `as`

```
try:
    # Bloc de code à tester
except TypeException as msg_exception:
    print(f"Voici l'erreur {msg_exception}")
```

7.3 Lever soi-même des exceptions

Par exemple, cela peut nous servir à invalider des saisies utilisateurs, alors que cette saisie peut être considérée comme correcte vis-à-vis du code. On utlise pour cela le mot clé `raise`.

Chapitre 8

La boucle `while`

Cette boucle veut dire *tant que...*. C'est est une structure de contrôle permettant d'exécuter un ensemble d'instructions de façon répétée sur la base d'une condition booléenne. Exemple simple d'introduction à la logique des boucles `while` :

```
1    n = 0  # 0 = valeur d'initialisation souvent prévue par défaut
2    while n < 10:  # On peut utiliser les signes: > < >= <= ==
3        print("n = " + str(n))
4        n = n + 1  # Incrémentation. Sortie de boucle quand n = 10
```

Listing 8.1 – Premier exemple avec la boucle *while*

Avec la boucle `while` on introduit une condition qui nous permet de faire évoluer des variables. Une boucle `while` est constituée d'un bloc de code source et d'une condition. À l'exécution, la condition est d'abord évaluée, et si elle est vraie, le bloc de code source est évalué. L'exécution du code est ensuite répétée jusqu'à ce que la condition soit fausse.

Autre exemple avec une condition du type *Tant que n'est pas égal à...* :

```
1    mdp = ""  # Initialisation de la variable pour tester une
2              # première fois la boucle while
3    while not mdp == "123":
4        mdp = input("Quel est le mode passe ? ")
5    print("Mot de passe correct !")
```

Listing 8.2 – Deuxième exemple avec la boucle *while*

Autre exemple :

```
1    age = 0  # Valeur d'initialisation de la variable que nous allons
2             # utiliser pour 'boucler'
3    while age == 0:
4        age_str = input ("Quel age avez-vous ? ")
5        try:
6            age = int(age_str)  # Si cela ne fonctionne pas, la
7                                # valeur de la variable 'age' reste
8                                # à 0, et donc on demeure dans la
9                                # boucle
10       except ValueError:
11           print("ERREUR: saisir un nombre.")
```

```
12      # Une fois sorti de la boucle:
13      print("Vous avez " + str(age) + " ans.")
```

Listing 8.3 – Troisième exemple avec la boucle *while*

Chapitre 9

Le debogage

Une des activités centrales d'un développeur est le *débogage* qui va consister à tester et rechercher les erreurs dans son code. Les *IDE* `Pycharm` et `Visual Studio Code`, dotés d'un débogueur, vont nous permettre d'investiguer les erreurs dans notre code. L'intérêt du *débogage* est aussi de pouvoir consulter l'état de nos *variables* tout en observant comment le code est exécuté. Apprendre le débogage dès le début de son apprentissage du langage `Python` offre ainsi des avantages, notamment dans le suivi pas à pas de l'exécution de notre code. Le débutant, pour visualiser le contenu de ses objets mutables (*variables*, *listes*, etc.) sera tenté de parsemer son code de diverses *fonctions* d'affichages (les fameux `print()`). Utiliser le *débogage* fait donc parti des bonnes habitudes à acquérir rapidement. C'est pourquoi il est ici proposé avant d'aller plus en avant dans l'apprentissage du code.

9.1 *Pycharm*

Activer le mode débogage

Entre le numéro de ligne de code et la ligne de code il y a une colonne. Cliquer au niveau d'une ligne de code : un point rouge apparaît. Il s'agit d'un point d'arrêt (*break point*).

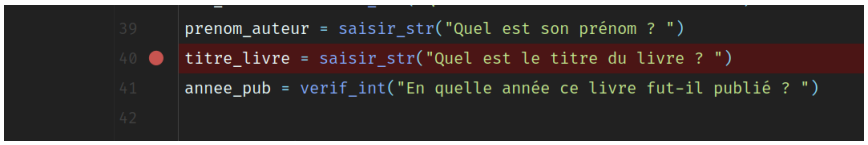

FIGURE 9.1 – *Pycharm* - Activer le débogage

Lancer ensuite le mode débogage en cliquant sur l'icône en forme d'insecte en haut à droite.

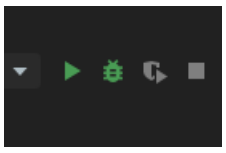

FIGURE 9.2 – *Pycharm* - Lancer le débogage

Quand le programme atteint le point d'arrêt, son exécution s'arrête, et nous pouvons alors observer ce qu'il s'est passé.

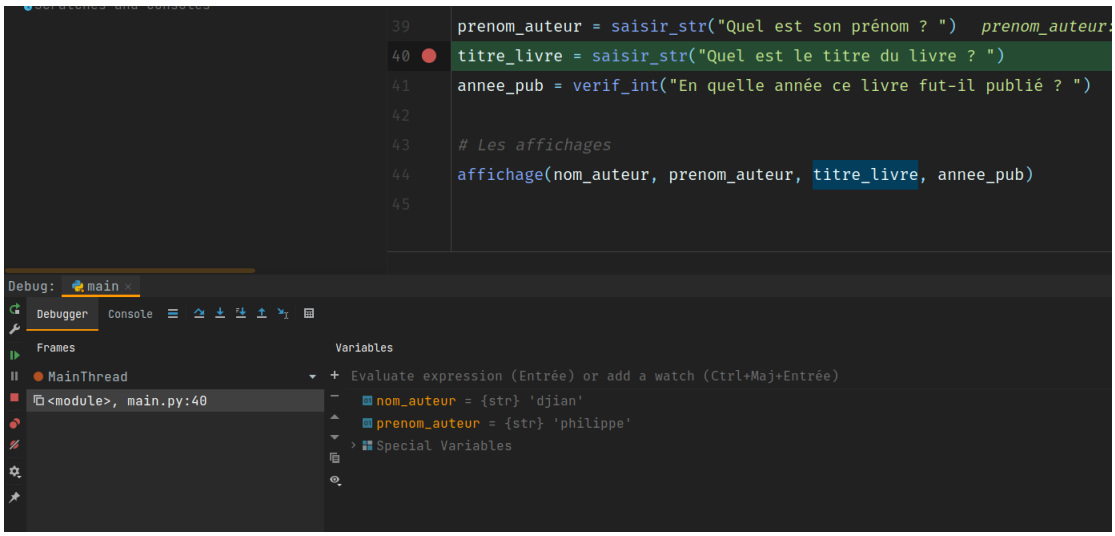

FIGURE 9.3 – *Pycharm* - Arrivée au point d'arrêt

Poursuivre l'exécution du code en mode débogage

Dans le panneau *debug*, cliquer sur le panneau *Step Over* (ce qui veut dire *Étape suivante*.

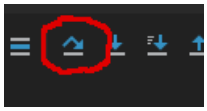

FIGURE 9.4 – *Pycharm* - Le bouton *Step Over*

En avançant ainsi dans le code, il est possible que nous soyons bloqué face à une instruction de type `input()`. Il faut alors revenir sur le panneau *Console* et saisir ce qui nous est demandé par le code.

9.2 *Visual Studio Code*

Activer le mode débogage

Le point d'arrêt se fixe par simple clic juste à gauche du numéro de ligne choisi.

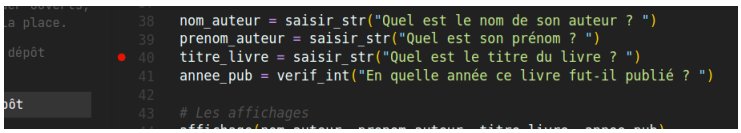

FIGURE 9.5 – *VSC* - Activer le débogage

Lancer le débogage :

`Exécuter > Démarrer le débogage`

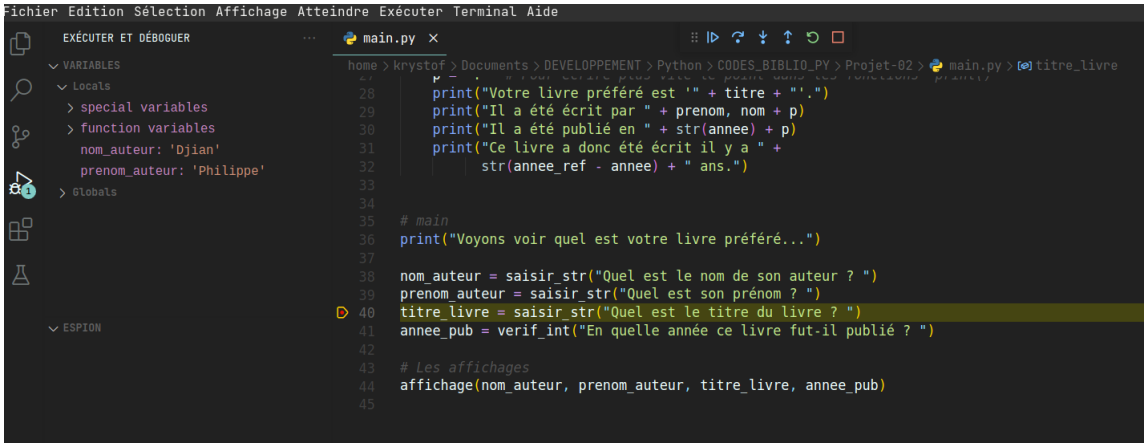

FIGURE 9.6 – *VSC* - Arrivée au point d'arrêt

Lors du premier lancement s'ouvre une fenêtre (*Select a debug configuration*) : sélectionner *Python File*. Puis l'exécution du programme stoppe au niveau du point d'arrêt.

Poursuivre l'exécution du code en mode débogage

Grâce au bouton *Pas à pas principal*.

FIGURE 9.7 – *VSC* - Le bouton *Pas à pas principal*

Chapitre 10

Les fonctions

10.1 Quel est l'intérêt des fonctions ?

Les *fonctions* sont fondamentales en programmation. Elles sont des éléments clés qui vont nous permettre de structurer notre code et d'en améliorer la qualité, en évitant d'avoir recourt à des répétitions de code [1]. Ce sont des objets dits *callables* (« *appelables* » pourrait-on traduire) qui sont associés à des actions. Pratiquement tous les langages de programmation utilisés aujourd'hui prennent en charge des fonctions définies par l'utilisateur, même si elles ne sont pas toujours appelées fonctions. Dans d'autres langages, elles peuvent être désignées par l'un des termes suivants : *sous-routines* (*subroutines*), *procédures* (*procedures*), *méthodes* (*methods*), *sous-programmes* (*subprograms*).

A l'aide des fonctions, notamment dans le développement de projets conséquents, nous saurons proposer un code aisément maintenable [2]. Les fonctions vont donc nous offrir la possibilité d'avoir :

Un code réutilisable : vous disposez d'une portion de code que vous avez besoin de reproduire à divers endroits de votre projet, et bien plutôt que de venir *copier-coller* ce code à tous ces endroits, vous pouvez simplement *encapsuler* ce code dans une fonction, et uniquement depuis les endroits où vous en avez besoin vous pouvez alors faire appel à ce morceau de code. Ce qui le rend d'ailleurs plus maintenable car si vous souhaitez le modifier il n'y a plus que le code de la fonction à modifier, et ce une seule fois au lieu de toutes les portions que vous auriez pu *copier-coller* par-ci par-là dans le code du projet [3].

Un code modulaire : les fonctions permettent de décomposer des processus complexes en étapes plus petites. Au lieu de regrouper tout le code de façon linéaire et continue, nous pouvons le diviser en fonctions distinctes, chacune d'entre elles se concentrant sur une tâche spécifique. Le fait de diviser une tâche importante en petites sous-tâches permet de la gérer plus facilement. Au fur et à mesure que les programmes se compliquent, il devient de plus en plus avantageux de modulariser les tâches plus ou moins complexes qu'il est censé accomplir.

Un code avec des *espaces de noms* bien distincts : un espace de noms est une région d'un programme dans laquelle les identifiants ont une signification. Lorsqu'une

1. En pratique un projet (ou programme) va contenir une fonction principale (`def main()`) qui va, comme son nom l'indique contenir le code principal, et c'est à partir de cette fonction principale que seront appelées la plupart des autres fonctions.

2. A noter que si l'on sur-structure un code avec des fonctions qui n'ont pas lieu d'être, il est très aisé de le destructurer.

3. « *Don't repeat yourself* » (aussi désigné par l'acronyme *DRY* - trad. « *Ne vous répétez pas* ») est une philosophie en programmation informatique consistant à éviter la redondance de code au sein d'une application afin d'en faciliter sa maintenance, les tests, le débogage et les évolutions de cette dernière.

fonction `Python` est définie, un nouvel espace de noms est créé pour cette fonction, un espace distinct de tous les autres espaces de noms existants. Le résultat pratique est que des variables peuvent être définies et utilisées dans une fonction même si elles portent le même nom que des variables définies dans d'autres fonctions ou dans le programme principal. Dans ces cas, il n'y aura pas de confusion ou d'interférence car elles sont conservées dans des espaces de noms distincts. Cela signifie que lorsque vous écrivez du code à l'intérieur d'une fonction, vous pouvez utiliser des noms de variables et des identificateurs sans vous soucier de savoir s'ils sont déjà utilisés ailleurs en dehors de la fonction. Cela permet de minimiser considérablement les erreurs dans le code.

10.2 Comment ça fonctionne ?

Première étape : la définition (ou implémentation) d'une fonction

A l'instar des variables, définir une fonction c'est lui donner un nom (une étiquette). Sa définition est finalement une affectation cachée. Une fonction est une boîte de codes (on parle d'*encapsulation* du code) à laquelle nous passons des *paramètres* lors de son *appel*.

```
def nom_fonction(paramètres):
    ...code de la fonction...
    ...ou bloc d'instructions...
```

Une fonction ne se définit qu'une seule fois, mais nous pouvons l'appeler (cf. infra *L'appel de fonction*) autant de fois que nous le souhaitons. Généralement on place les définitions de fonctions en début de code [4]. Au lancement du programme, elle ne sera pas exécutée. Seuls les noms des fonctions seront lus. Il faut la considérer comme du code séparé. Une fonction n'est exécutée que lors de son appel.

Seconde étape : l'appel de fonction

On appelle une fonction (cet appel implique que la fonction existe déjà) en fournissant son nom suivi de parenthèses. L'appel sans les parenthèses ne fera que l'afficher, au même titre que n'importe quel objet.

```
1 def afficher():
2     print("Mon texte affiché")
3
4
5 afficher()  # Appel
6 print(afficher)  # Affichage
```

Listing 10.1 – Utiliser une fonction

Ce petit bout de code produira en sortie :

```
Mon texte affiché
<function afficher at 0x7f820b269d80>
```

4. Deux fonctions sont séparées par deux lignes vides

Une variable se définit dans un contexte. Une variable définit dans le contexte d'une fonction ne sera pas accessible en dehors de la fonction, car elle n'existe que pour la fonction et non pour le programme principal. Pour permettre à une variable d'exister en dehors de la fonction, et d'être utilisée par le programme principal, on va retourner cette variable à l'aide du mot-clé `return`[5]. Du contenu de la fonction, le programme principal n'y a pas accès, il n'aura accès qu'à ce qui est retourné. Mais il faut savoir qu'une fonction ne retourne pas nécessairement quelque chose de façon explicite (par défaut la valeur de retour est `None`).

```
1  def dder_age():  # Notre fonction
2      age_int = 0
3      while age_int == 0:
4          try:
5              age_int = int(input("Quel âge avez-vous ? "))
6          except ValueError:
7              print("Erreur: veuillez saisir un nombre.")
8      return age_int
9
10 # Code principal
11 print("Bonjour")
12 age = dder_age()
13 print(f"Vous avez {age} ans.")
```

Listing 10.2 – Un premier exemple de fonction

Les fonctions intégrées à *Python* grâce à la bibliothèque standard

Il existe, à l'instar de la fonction `print()`, un certain nombre de fonctions contenues dans la bibliothèque standard de `Python`. On parle alors de *fonctions natives*. Il y a des fonctions qui exécutent du code (Exemple de la fonction `print()`), et des fonctions qui retournent des valeurs (Exemple de la fonction `input()`[6]), valeur que l'on récupère dans une variable.

Voyons deux exemples de fonctions natives : `len()` et `type()`.`len()` est une fonction qui nous permet d'obtenir la longueur d'une chaîne de caractères. On lui passe une chaîne de caractères en paramètre et elle retourne un objet de type `int`. `type()` est aussi une fonction intégrée issue de la bibliothèque standard.

```
1 a = "Python"
2 b = len(a)
3 print(b)
4 print(type(a))
5 print(type(b))
```

Listing 10.3 – Les fonctions `len()` et `type()`

Sortie dans la console :

```
6
<class 'str'>
<class 'int'>
```

5. Mot-clé qui ne s'utilise qu'à l'intérieur d'une fonction
6. La fonction `input()` a en plus la caractéristique d'être bloquante, et seul le fait de presser la touche `[Entrée]` nous fait sortir de la fonction et permet l'exécution de la suite du code.

Il en existe encore plein d'autres fonctions issues de la bibliothèque standard avec chacune leurs spécificités, que ce soit au niveau des entrées et des sorties.

10.3 La définition d'une fonction

Créer une fonction signifie que l'on « définit » une fonction, ou qu'on l'« implémente ». La syntaxe habituelle pour définir une *fonction* `Python` est la suivante :

```
def nom_fonction(arguments):
    ...bloc d'instructions...
```

Décortiquons cela :

`def` : mot-clé pour définir une fonction.

`nom_fonction` : identificateur de la fonction. Les conditions de nommage d'une fonction (de l'identificateur) sont les mêmes que pour les variables.

(arguments) : ils sont optionnels, et séparés par des virgules quand il y en a plusieurs. Ils correspondent aux paramètres qui sont passés à la fonction. Vous pouvez définir une fonction qui ne prend aucun *argument*, mais les parenthèses sont toujours nécessaires.

Les deux points (`:`) : ponctuation qui indique la fin de l'*en-tête* (nom et arguments) de la fonction, et que nous allons débuter un *bloc d'instructions* (de code).

L'*indentation* : tout bloc d'instructions respecte, en `Python`, les règles de l'indentation. Ce bloc d'instructions correspond au *corps de la fonction* qui sera exécuté à chaque appel de la fonction.

Une fonction doit être définie avant son appel, sur le même principe que pour les variables. Dans un codage « propre » on les retrouve au début du code, avant le lancement de la première ligne d'instructions.

Dans l'idéal, une fonction doit avoir une limite se situant autour d'une vingtaine de lignes de code. Au-delà, elle risque d'être longue et certainement mériterait-elle un découpage faisant appel à d'autres fonctions. Une fonction doit, en pratique, ne faire qu'une seule et unique action, mais la faire proprement et de façon à être aisément réutilisable.

10.4 L'appel de la *fonction*

La syntaxe pour appeler une *fonction* est la suivante :

```
nom_fonction(paramètres)
```

L'appel de la fonction permet l'exécution de la fonction, c'est-à-dire l'exécution du bloc d'instructions qu'elle contient. Ce n'est qu'à la suite de son appel qu'une fonction sera exécutée, sinon elle demeure ignorée. Sa définition permet au programme qu'elle soit connue et identifiée, mais son contenu demeure opaque tant qu'elle n'est pas appelée.

S'il n'y a aucun paramètre passé à la fonction son appel doit tout de même comporter des parenthèses (à l'instar de sa définition).

Quand il n'y a plus d'instructions à exécuter au sein de la fonction, elle retourne ce qui est explicitement attendu (une valeur que l'on va affecter à une variable, par exemple) par le mot-clé `return`, ou bien elle retourne un objet implicite (pas besoin du mot-clé `return`) qui est de type `None`.

```
def nom_poete():
    return "Charles BAUDELAIRE"


print("Qui a écrit 'Les Fleurs du Mal' ?")
nom = nom_poete()  # Appel de la fonction et valeur de retour
                   # de la fonction affectée à une variable.
print(f"C'est {nom}")  # Ligne exécutée après l'exécution de
                       # la fonction.
```

Listing 10.4 – Exemple du fonctionnement d'une fonction

La sortie en console sera alors :

```
Qui a écrit 'Les Fleurs du Mal' ?
C'est Charles BAUDELAIRE
```

10.5 Variable globale et paramètres

L'intérêt des fonctions résident aussi dans la passage de paramètres (de données) qui vont interagir sur le comportement de la fonction. Ces paramètres vont s'utiliser tels des variables nécessaires à l'exécution du bloc d'instructions.

Variable locale *vs* variable globale

Une variable à l'intérieur d'une fonction est une *variable locale*, locale à la fonction.

Avant la définition d'une fonction il est possible de définir une variable au sein du programme principal. Il s'agit alors d'une *variable globale*. Pour que la fonction prenne en considération une telle variable, on va alors au sein de la fonction utiliser l'instruction `global nom_variable`. Le code de la fonction peut être, par exemple, une simple incrémentation de la valeur de cette variable. Alors un simple appel de la fonction viendra incrémenter la valeur de la variable, et il n'y aura pas nécessité de recourir à l'instruction `return nom_variable` puisque étant globale elle sera directement modifiée au sein du programme principal. Mais une telle manipulation est à éviter car la fonction est alors dépendante des variables définies à l'extérieur de la fonction. Et justement, l'intérêt d'une fonction réside dans son indépendance et sa modularité. Elle doit pouvoir s'utiliser par ailleurs.

```
variable = 0


def fction():
    global variable
    variable += 10


print(variable)  # Avant le premier appel de fonction
fction()  # Premier appel de fonction
print(variable)
fction()  # Second appel de fonction
```

```
13 print(variable)
```

Listing 10.5 – Fonction et variable globale

Sortie en console :

```
0
10
20
```

A noter qu'il est possible pour une fonction d'utiliser une variable définie globalement, et sans avoir à recourir au mot-clé `global`, mais sachez que cette variable ne sera accessible qu'en lecture, et non en écriture, donc non modifiable au niveau global comme nous l'avons vu précédemment.

Les paramètres de fonctions - Les variables locales

Pour permettre à une fonction d'utiliser une variable définie ailleurs (à l'extérieur de la fonction), on va lui donner cette variable sous la forme d'un argument que l'on va passer en paramètre lors de l'appel. Ce paramètre est la valeur ou la variable que l'on souhaite passer à notre fonction. Dans certains textes consacrés à la programmation, les arguments donnés dans la définition de la fonction sont appelés *paramètres formels*, et les paramètres passés dans l'appel de la fonction sont appelés *paramètres réels* :

```
1 def double(a):
2   return a * 2
3
4 print(double(7))
```

Listing 10.6 – Exemple de fonction avec paramètre

Cela nous affichera la valeur `14` qui est retournée par notre fonction `double()`. A noter qu'ici la fonction `print()` a pour paramètre l'appel de la fonction `double()`. Nous venons de voir la façon la plus simple de transmettre des arguments à une fonction `Python` par l'utilisation d'un *argument positionnel* (également appelé *argument requis*, « *required argument* » en anglais). Dans la définition de la fonction, il est possible de spécifier une liste de paramètres que l'on sépare alors par des virgules :

```
1  def affiche_texte(nom, prenom, titre):
2      return f"{prenom.capitalize()} {nom.upper()} a écrit " \
3             f"'{titre.title()}'"
4
5
6  auteur_nom = "hemingway"
7  auteur_prenom = "ernest"
8  titre_livre = "pour qui sonne le glas"
9  print(affiche_texte(auteur_nom, auteur_prenom,
10                     titre_livre))
```

Listing 10.7 – Fonction avec plusieurs arguments positionnels

La sortie du code sera la suivante :

```
Ernest HEMINGWAY a écrit 'Pour Qui Sonne Le Glas'
```

Lorsque la fonction est appelée, la liste d'arguments spécifiée doit correspondre aux paramètres précisés lors de la définition de la fonction. Bien que les arguments positionnels constituent le moyen le plus simple de transmettre des données à une fonction, ils offrent également le moins de flexibilité. En effet, l'ordre des arguments dans l'appel doit correspondre à l'ordre des paramètres dans la définition. Avec les arguments positionnels, les arguments de l'appel et les paramètres de la définition doivent concorder non seulement dans l'ordre mais aussi en nombre. C'est également la raison pour laquelle les arguments positionnels sont également appelés arguments requis. Vous ne pouvez en omettre aucun lors de l'appel de la fonction.

L'utilisation des paramètres va finalement rendre nos fonctions beaucoup plus modulaires. L'intérêt d'une fonction réside justement dans son aspect générique, nous offrant alors la possibilité de faire appel selon divers cas de figure. Et il faut bien garder à l'esprit que les paramètres se comportent comme des variables définies localement (uniquement accessibles dans le corps de la fonction).

Les noms données aux paramètres et aux arguments sont différents, mais rien ne vous empêche d'utiliser les mêmes, car les valeurs associées seront exploités uniquement par la fonction, le programme principal n'y ayant pas accès. Cela n'a donc aucune importance.

Cependant l'ordre des paramètres et arguments a son importance, car sinon la fonction ne recevrait pas les bons arguments. De même pour le nombre : une fonction devant recevoir deux arguments ne peut en recevoir trois. Si tel était le cas, l'exécution du programme stopperait et une exception de type `TypeError` serait levée.

Paramètre optionnel

Un paramètre peut être rendu optionnel en lui donnant une valeur par défaut. Et il permet d'éviter de *lever une exception* si l'argument n'est pas passé à la fonction. Il est toujours placé après les paramètres qui ne sont pas optionnels.

```
def ma_fonction(var1, var2, var3=0):  # var3 = paramètre otpionnel
```

Et pour appeler la fonction :

```
ma_fonction(var1, var2)
```

En donnant une valeur par défaut au paramètre `var3`, on rend l'argument optionnel pour l'appel de la fonction.

Argument et paramètre par *mot-clé*

Il est aussi possible, lors de l'appel de fonction de donner le nom du paramètre auquel on affecte une valeur :

```
ma_fonction(var1, var2=4)
```

Une affectation explicite (par mot-clé) est plus intuitive qu'une affectation positionnelle [7]. L'utilisation d'arguments de type mot-clé lève la restriction sur l'ordre des arguments. Chaque argument mot-clé désigne explicitement un paramètre spécifique par son nom. Vous pouvez donc les spécifier dans n'importe quel ordre et `Python` saura toujours quel argument va avec quel paramètre. Ainsi, les arguments de type mot-clé permettent une certaine flexibilité dans l'ordre dans lequel les arguments de la fonction sont spécifiés, mais le nombre d'arguments reste rigide, car comme pour les arguments positionnels, le nombre d'arguments et de paramètres doit toujours correspondre. Par ailleurs, la référence à un mot-clé qui ne correspond à aucun des paramètres déclarés lève une exception de type `TypeError`.

Il est à noter que vous pouvez appeler une fonction en utilisant à la fois des arguments positionnels et des mots-clés. Une fois que vous avez spécifié un argument de type mot-clé, il ne peut y avoir aucun argument positionnel à sa droite.

10.6 Les retours de fonction

Le mot-clé `return` sert à deux choses :

1. Sortir de la fonction. Avec des structures conditionnelles cela peut avoir son utilité.
2. Potentiellement à renvoyer une valeur (non obligatoire), c'est-à-dire l'objet fabriqué dans la fonction.

Cependant, faire appel au mot-clé `return` n'est pas non plus obligatoire.

L'emploi du `return` sera par contre très utile avec l'utilisation de structures conditionnelles [8] au sein d'une fonction :

```
1 def fction():
2     if condition:
3         # corps de la condition
4         return valeur  # un objet
5     elif condition:
6         # corps de la condition
7         return valeur  # un objet
```

Listing 10.8 – Retours de fonction et structures conditionnelles

Retourner un objet (une valeur, une variable, etc.) :

```
return ...ce qui est retourné...
```

Un `return` sans valeur de retrour retourne `None`. Mais cela n'est pas une bonne pratique en programmation, car il faut savoir que ce n'est pas accepté par tous les langages de programmation. Donc, même avec la langage `Python` il est souhaitable de se référer aux pratiques générales de programmation.

Le mot-clé `return` ne s'utilise que dans des fonctions, et non dans le programme principal. Pour sortir d'un programme principal on va utiliser la fonction `exit(0)`.

Retourner plusieurs valeurs est possible à condition de les séparer par des virgules (attention toutefois à l'ordre) :

7. Dans *The Zen of Python* il est d'ailleurs dit : « *Explicit is better than implicit* ».
8. Les *structures conditionnelles* seront plus clairement abordées infra (cf. page 59).

```
def fction():
    return 1, 2


var1, var2 = fction()
print(var1 + var2)
```

Listing 10.9 – Fonction retournant plusieurs objets

Ce qui affiche la valeur `3` en sortie de console.

Voyons un autre exemple avec des récupérations différentes de valeurs :

```
def ma_fonction():
    return 1, 2, 3


print(ma_fonction())  # Affichage d'un tuple.
a, b, c = ma_fonction()
print(f"{a} - {b} - {c}")
a, *le_reste = ma_fonction()  # * -> Récupérer les valeurs
                              # restantes.
print(f"{a} - {le_reste}")  # le_reste = liste.
```

Listing 10.10 – Récupération de plusieurs valeurs de retour

Ce qui donne en sortie :

```
(1, 2, 3)
1 - 2 - 3
1 - [2, 3]
```

A noter que `return None` retourne un objet vide.

10.7 L'imbrication de fonctions

Il est possible d'imbriquer les fonctions entre elles, autrement dit de lancer l'appel d'une fonction depuis une autre fonction, mais il faut veiller à conserver un certain niveau de hiérarchie. C'est-à-dire qu'en haut de notre code nous définirons d'abord les fonctions les plus indépendantes puis nous définirons celles qui dépendent d'autres fonctions (en principe les premières).

```
def affichage(nbre):
    return f"Le résultat est {nbre}"


def calcul(a, b):
    produit = a * b
    return affichage(produit)
    # ou bien : return affichage(a*b)

```

```
10
11 print(calcul(5, 10))
12 print(calcul(10, 12))
```

Listing 10.11 – Exemple de fonctions imbriquées

En sortie console nous obtenons :

```
Le résultat est 50
Le résultat est 120
```

10.8 Documenter une fonction

La première documentation d'une fonction est son nom : avoir un nom explicite permet d'identifier rapidement la ou les actions qui lui correspondent. Mais dans l'idéal une fonction doit posséder une *docstring* (documentation) qui vient détailler ce que le nom ne peut exprimer. Cette *docstring* peut contenir l'objectif de la fonction, les *arguments* qu'elle prend, des informations sur les valeurs de retour ou toute autre information que vous jugez utile [9]. Et l'instruction `help(nom_fonction)` permettra de visualiser cette documentation interne à la fonction.

Exemple d'une fonction avec un nom explicite quant à son action, et documentée avec de la *docstring* :

```
1  def entre_min_max(nombre, mini, maxi):
2      """Pour vérifier si l'uitlisateur a bien saisi la valeur
3      attendue, comprise entre une valeur minimale et une valeur
4      maximale.
5      'nombre' : nombre saisi par l'utilisateur, de type int.
6      'mini' : valeur minimale, de type int.
7      'maxi' : valeur maximae, de type int.
8      La fonction retourne une valeur booléenne."""
9      if nombre < mini or nombre > maxi:
10         print(f"ERREUR : Veuillez saisir une valeur comprise "
11               f"entre {mini} et {maxi}.")
12         return False
13     return True
14
15
16 valeur_min, valeur_max = 0, 100
17 while True:
18     nbre = int(input(f"Veuillez saisir un nombre compris entre "
19                      f"{valeur_min} et {valeur_max} : "))
20     if entre_min_max(nbre, valeur_min,
21                      valeur_max):
22         print(f"{nbre} au carré = {nbre**2}")
23         break
```

Listing 10.12 – Exemple d'une fonction avec une *docstring*

9. Nous approfondirons la question de la documentation au travers du chapitre intitulé *Documenter son code et son projet* page 181.

10.9 Indiquer les types de paramètres et de retours

Depuis `Python 3.0`, lors de la définition de fonction il est possible de spécifier les types des paramètres et les types des retours, afin de documenter cette fonction, c'est ce que l'on nomme les *annotations de type*. Les annotations permettent d'attacher des métadonnées aux paramètres et à la valeur de retour d'une fonction.

Pour ajouter une annotation à un paramètre de fonction `Python`, insérez deux points (`:`) suivis d'une expression après le nom du paramètre, lors de la définition de la fonction. Pour ajouter une annotation à la valeur de retour, ajoutez les caractères « `->` ». Exemple :

```
def fction(param: int) -> bool:
```

Ici, par exemple, on indique que le paramètre de la fonction est de type `int` et que le `return` sera de type *booléen*. Cela ne corrige en aucun cas les erreurs. Il faut davantage voir cela comme une forme de commentaire utile dans la recherche d'erreurs potentielles de type. A noter que l'éditeur `Pycharm` signale cette potentielle erreur grâce aux avertissements que l'on retrouve en haut à droite.

FIGURE 10.1 – *Pycharm* - Les avertissements d'erreurs

Exemple :

```
1  def affichage(nb: int) -> str:
2      return f"Le résultat est {nb}"
3
4
5  def calcul(a: int, b: int) -> str:
6      produit = a * b
7      return affichage(produit)
8
9
10 print(calcul(5, 5))
```

Listing 10.13 – Les annotations de type

Il faut savoir qu'avec le langage `Python` indiquer le type de paramètres n'est pas obligatoire, contrairement à d'autres langages de programmation. Et par conséquent les annotations de type viennent apporter un éclairage quant aux valeurs des paramètres pour la personne amenée à lire le code.

Les annotations de type de la fonction `calcul()` présentée ci-dessus peuvent être affichées comme suit :

```
>>> def calcul(a: int, b: int) -> str:
...     produit = a * b
...     return affichage(produit)
...
>>> calcul.__annotations__
{'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'str'>}
```

L'interprète `Python` crée un *dictionnaire*[10] à partir des annotations et les affecte à un *attribut*[11] spécial de la fonction appelé `__annotations__`. Les *clés*[12] des paramètres sont les noms des paramètres. La clé de la valeur de retour est la chaîne de caractères `return`. Finalement, les annotations de type ne sont rien d'autre que des dictionnaires de métadonnées.

La fonction `Annoted` du module `typing` :

Depuis la version `3.9` de `Python`[13] les annotations de type se sont vues améliorées, par l'ajout d'un contenu informationnel plus libre. Pour cela nous allons faire appel à la fonction `Annoted` du module `typing` :

```
1 from typing import Annoted
```

Listing 10.14 – Import de la fonction `Annoted`

Voici alors la syntaxe que nous utiliserons dans la définition de la fonction :

```
def nom_fonction(var: Annoted[type, "information libre"])
```

Le `PEP 593` vient introduire un mécanisme afin d'étendre les annotations de type convenues précédemment avec le `PEP 484`[14], en y ajoutant des métadonnées variées, spécifiques au contexte. S'agissant des fonctions, cela vient en complément du `PEP 3107`[15] relatif aux annotations des fonctions.

Il est tout à fait possible d'utiliser des annotations imbriquées :

```
1 def ma_fonction(var: Annoted[list[int]]):
```

Listing 10.15 – Les annotations imbriquées

Il est aussi possible de déterminer des constantes de type.

Illustrons tout cela par un petit script permettant de calculer l'IMC d'un individu :

```
1  from typing import Annotated
2
3  # Constantes de type
4  ANNOTE_POIDS = Annotated[float, "Kg"]
5
6
7  def imc(poids: ANNOTE_POIDS, taille: Annotated[float, "mètre"]) \
8          -> float:
9      """Calcul de l'imc."""
10     return poids / (taille * taille)
11
12
```

10. Nous aborderons la notion de *dictionnaire* avec les *collections* (Cf. page 74).
11. La notion d'*attribut* sera abordée dans la partie consacrée à la *programmation objet* (Cf. page 109).
12. Les *clés* seront vues avec la notion de *dictionnaire*.
13. Cf. PEP 593 – Flexible function and variable annotations - `https://peps.python.org/pep-0593/`
14. Cf. PEP 484 – Type Hints - `https://peps.python.org/pep-0484/`
15. Cf. PEP 3107 – Function Annotations - `https://peps.python.org/pep-3107/`

```
13 print(imc(67.5, 1.75))
14
15 # Pour afficher les informations sur les annotations :
16 print(imc.__annotations__)
17 print(ANNOTE_POIDS)
```

Listing 10.16 – Exemple d'annotations de type avec la fonction `Annoted`

Le code peut rapidement s'alourdir, mais en même temps cela offre la possibilité de documenter son code, et la méthode `.__annotations__` vient nous offrir une lisibilité complète concernant les paramètres et retour d'une fonction. Voyons la sortie console du script précédent :

```
22.040816326530614
{'poids': typing.Annotated[float, 'Kg'], 'taille': typing.Annotated[float,
  'mètre'], 'return': <class 'float'>}
typing.Annotated[float, 'Kg']
```

10.10 Le débogage avec une fonction

Plaçons un point d'arrêt à la ligne correspondant à l'appel de fonction, puis on exécute le code en mode débogage. Arrivé au point d'arrêt on clique sur `Step into` pour entrer dans la fonction.

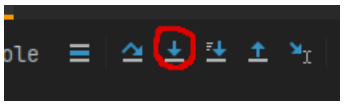

FIGURE 10.2 – *Pycharm* - Le bouton `Step into`

On poursuit ensuite l'exécution ligne par ligne avec le bouton `Step Over`. Pour sortir le débogueur de la fonction afin de revenir au programme principal et poursuivre avec `Step Over` depuis le programme principal, on clique sur `Step Out`.

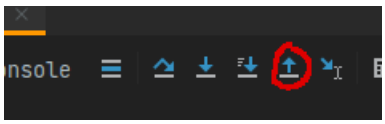

FIGURE 10.3 – *Pycharm* - Le bouton `Step Out`

Sur *Visual Studio Code* on emploie les boutons `Pas à pas détaillé` et `Pas à pas sortant`.

FIGURE 10.4 – *VSC* - Le bouton `Pas à pas détaillé`

FIGURE 10.5 – *VSC* - Le bouton `Pas à pas sortant`

Sources utilisées pour la rédaction de ce chapitre

- *Comprendre les fonctions* ***Python*** - Thierry CHAPPUIS - *WePynaire* du 3 novembre 2022, disponible à l'adresse `https://edu.placepython.fr/`
- *Defining Your Own Python Function* (trad. *Définir votre propre fonction* ***Python*** - John STURTZ - 9 mars 2020 - `https://realpython.com/defining-your-own-python-function/`

Chapitre 11

Les structures conditionnelles

11.1 Condition et variable de type « booléen »

Voici le principe :

```
Si condition:                    if condition:
    ...code...        ==>            ...code...
Sinon:                           else:
    ...code...                       ...code...
```

```
1  age = 8
2  if age >= 18:
3      print("Vous êtes majeur")
4  else:
5      print("Vous êtes mineur")
```

Listing 11.1 – Un premier exemple de structure conditionnelle

Ce code affiche comme résultat :

```
Vous êtes mineur
```

Les conditions retournent les valeurs `True` (*Vrai*) et `False` (*Faux*). Ce sont des valeurs dites *booléennes* (*boolean* en anglais).

Voyons cela de façon plus explicite :

```
1  age = 8
2  condition = age >= 18  # La condition est affectée à une variable
3  if condition:  # Revient à dire: si condition == True
4      print("Vous êtes majeur")
5  else:  # Si condition == False
6      print("Vous êtes mineur")
7  print(condition)
8  print(type(condition))
```

Listing 11.2 – Les valeurs *booléennes*

Ce code nous retourne :

```
Vous êtes mineur
False
<class 'bool'>
```

Le type de la variable `condition` est donc bien de type booléen.

11.2 Les opérateurs de comparaison

Nous avons donc utilisé l'opérateur de comparaison `>=`, mais il existe en *Python* huit opérateurs de comparaison.

Opérateur	Signification
`<`	Strictement inférieur
`<=`	Inférieur ou égal
`>`	Strictement supérieur
`>=`	Supérieur ou égal
`==`	Égal
`!=`	Différent
`is`	Identité d'objet
`is not`	Contraire de l'identité d'objet

TABLE 11.1 – Tableau des opérateurs de comparaison

Les opérateurs `is` et `is not` peuvent s'appliquer à deux objets quelconques. Ils ne retournent jamais d'exception.

11.3 La condition `elif`

`elif` signifie *sinon si...*, ce qui est la contraction de *else if*. `elif` rend la condition exclusive, c'est-à-dire que si la condition est vraie les autres ne seront pas testées.

```
1  if age == 17:
2      print("Vous êtes presque majeur")
3  elif age == 18:
4      print("Tout juste majeur")
5  elif age > 18:
6      print("Vous êtes majeur")
7  else:  # Exécuté si aucune des conditions d'avant n'est vrai
8      print("Vous êtes mineur")
```

Listing 11.3 – Sinon...si

Il peut y avoir autant de conditions `elif` que l'on en souhaite, cela fonctionne toujours comme un seul bloc de codes. L'ordre des conditions a son importance, ainsi que les formes conditionnelles, afin d'être sûr que toutes les conditions puissent être testées.

Attention à ne pas commettre ce type d'erreur, qui ne plantera pas nécessaire l'exécution du code, mais qui amènerait à des incohérences dans son exécution :

```
if condition:
    ...code...
elif condition:
    ...code...
if condition:
    ...code...
else:
    ...code...
```

Cela fonctionne comme deux blocs de codes distincts : `if...elif` et `if...else`.

11.4 and/or (*et*/*ou*)

```
1  elif age >= 1 and age <= 2:
2      print("Vous êtes encore un bébé")
```

Une telle condition `elif` peut être modifiée de la façon suivante, dans la mesure où c'est la même variable qui est utilisée (ce qui est d'ailleurs recommandé) :

```
1  elif 1 <= age <= 2:
```

C'est ce que l'on appelle l'enchaînement de conditions. Ainsi :

```
x < y <= z
```

C'est équivalent à :

```
x < y and y <= z
```

`y` n'est évalué qu'une seule fois, et `z` ne sera pas évalué si `x < y` est faux.

Nous aurions pu aussi utiliser `or` (pour *ou*) :

```
1  elif age == 1 or age == 2:
```

Utiliser `and` est intéressant quand dans notre condition nous voulons faire appel à deux variables différentes :

```
1  age = 20
2  taille = 2
3  if age >= 18 and taille >= 2:
4      print("Vous êtes majeur et grand")
```

Listing 11.4 – `and` avec deux variables différentes

11.5 La correspondance structurelle

La correspondance structurelle (ou *Structural pattern matching* en anglais) est l'une des fonctionnalités les plus importantes introduites dans la version `3.10` de `Python`[1]. Le `switch case` est une fonctionnalité déjà existante dans la majorité des langages de programmation, avec la version `3.10` elle fait donc son apparition dans le langage `Python`, sous le nom de `match case`. Désormais, avec `Python` nous pouvons réaliser de puissantes correspondances structurelles.

Illustrons cela par un exemple simple à comprendre :

```
while True:
    print("Choisir un nombre entre 1 et 5...(0 pour quitter)")
    choix = input("Votre choix : ")
    match choix:
        case "1" | "3" | "5":  # '|' pour 'or'
            print(f"{choix} est un chiffre impair")
        case "2" | "4":
            print(f"{choix} est un chiffre pair")
        case "0":
            break
        case _:  # Tous les autres cas
            print("Vous n'avez pas saisi le bon chiffre")
```

La correspondance structurelle de type `match case` offre un code beaucoup plus concis que l'utilisation d'une structure conditionnelle avec des `if...else...`. Et avec un métacaractère (*wild card* en anglais) comme l'underscore nous allons prendre en considération tous les autres cas de figure. Une déclaration de correspondance ne peut avoir qu'un seul et unique bloc `case` irréfutable, et ils doit être systématiquement positionné en dernier.

Avec le `match case` nous allons aussi pouvoir tester la structure des données[2].

```
# Divers dictionnaires avec des structures de données différentes :
auteur1 = {"nom": "DJIAN", "titre": "Sotos"}
auteur2 = {"nom": "KEROUAC", "livre": ("Sur la route", 1957)}
auteur3 = {"nom": "HEMINGWAY"}
auteur4 = {"titre": "Paris est une fête"}
auteur5 = {"nom": "DJIAN", "livre": ("Bleu comme l'enfer", 1981)}
auteur6 = {"nom": "BAUDELAIRE", "livre": ("Fleurs du Mal", 1857)}

auteurs = (auteur1, auteur2, auteur3, auteur4, auteur5, auteur6)

for auteur in auteurs:
    match auteur:
        case {"nom": nom, "livre": (titre, date_pub)} \
              if date_pub > 1900:
            print(f"{nom} a écrit {titre}, publié en {date_pub}.")
        case {"nom": nom, "livre": (titre, date_pub)}:
            print(f"{nom} a écrit {titre}, publié avant le "
```

1. Cf. PEP 634 – Structural Pattern Matching : Specification - `https://peps.python.org/pep-0634/`
2. Nous allons ici utiliser des dictionnaires, notion que nous aborderons plus loin avec les collections (page 69), mais j'estime qu'ici l'exemple est aisément compréhensible.

```
18                      f"XXe siècle")
19          case {"nom": nom, "titre": titre}:
20              print(f"{nom} a écrit {titre}.")
21          case _:
22              print(f"Aucune information exploitable pour {auteur}")
```

Il faut toutefois être vigilant à l'ordre des `case` et insérer les cas les plus particuliers au début.

Les mots clés `match` et `case` ne sont pas des mots réservés. Cela implique qu'ils ne sont reconnus comme tels que lorsqu'ils font partie d'un bloc `match`/`case`, en ayant bien les deux points à l'endroit prévu.

Chapitre 12

La boucle `for`

Elle permet de boucle un certain nombre de fois ou sur un certain nombre d'éléments (nous verrons cela avec les séquences).

```
for i in range(0, 5):
```

`i` représente une variable. *i* est utilisé par convention pour signifier *index*. Une telle ligne de code pourrait se traduire ainsi : pour `i` de 0 à 5 (`range` pour *ensemble*) on va boucler. Exemple de code fonctionnel :

```
1  for i in range(0, 5):
2      print(i)
```

Listing 12.1 – Un premier exemple avec une boucle `for`

Cela donne en sortie :

```
0
1
2
3
4
```

L'ensemble `range(0, 5)` est inclusif pour la première valeur (`0`) et exclusif pour la seconde (`5`). En expression mathématique cela donnerait l'ensemble `[0,5[`. En atteignant `5` (quand `i` vaut `5`) on va sortir de la boucle sans en exécuter le code qui suit.

Imaginons un code qui vienne demander le nom de plusieurs personnes, où chacun d'entre elles viendrait alors saisir son nom :

```
1  nbre_personnes = 3
2  for i in range(0, nbre_personnes):
3      nom = input("Quel est votre nom ? ")
4      print("Bonjour " + nom)
```

Listing 12.2 – Demander trois fois le nom

En sortie nous pourrions avoir :

```
Quel est votre nom ? Paul
Bonjour Paul
Quel est votre nom ? Mattéo
Bonjour Mattéo
Quel est votre nom ? Loreleï
Bonjour Loreleï
```

Idée de codage : imaginons demander à l'utilisateur non pas son livre préféré, mais ses trois livres préférés...

Chapitre 13

Les affichages

13.1 Les chaînes formatées

Au lieu de concaténer plusieurs chaînes de caractères on va faire appel au formatage qui permet de n'avoir qu'une seule et même chaîne de caractères. Ainsi, au lieu de coder :

```
print("Vous vous appelez " + nom + ", vous avez " + str(age) + " ans.")
```

Nous pouvons coder :

```
print(f"Vous vous appelez {nom}, vous avez {age} ans.")
```

Un autre formatage possible avec l'utilisation de `%s`. Les paramètres à afficher ne sont plus les noms des variables, mais on leur substitut `%s`, et les noms de variables sont insérés à la fin de la chaîne dans l'ordre voulu. Il s'agit de l'ancienne méthode de formatage de chaînes en *Python*, qui est toujours valable.

```
print("Vous vous appelez %s, vous avez %s ans." % (nom, age))
```

S'il y a plusieurs paramètres à insérer, ils doivent être placés entre parenthèses juste après le symbole `%` et séparés par une virgule. A noter qu'avec les deux dernières méthodes une valeur de type numérique est automatiquement convertie en une valeur de type *string*.

Il n'y a pas selon moi de méthode mieux qu'une autre, chacun préférant la sienne. Pour ma part je trouve la deuxième méthode plus lisible. Ce sera donc celle-ci que j'utiliserai par la suite.

13.2 `print()` sur plusieurs lignes

```
1  print("""Voici
2  votre texte
3  sur plusieurs
4  lignes""")
```

Ce qui donne en sortie :

```
Voici
votre texte
sur plusieurs
lignes
```

13.3 Ajout de paramètres à la fonction `print()`

```
1  nom = "Krystof"
2  depart = 64
3  print("Je m'appelle", nom, ". Je réside dasn le", depart)
```

Ce qui donne en sortie :

```
Je m'appelle Krystof . Je réside dans le 64
```

Chaque paramètre est suivi d'une espace, et les paramètres qui ne sont pas de type *string* sont considérés comme tel automatiquement.

13.4 `lower()` et `upper()`

```
print("Krystof".lower())  # Affiche 'krystof' (minuscules)
print("Krystof".upper())  # Affiche 'KRYSTOF' (majuscules)
```

Chapitre 14

Les collections d'objets

On va aborder ici des notions importantes de la programmation avec les collections qui nous permettent de contenir et gérer plusieurs éléments et ainsi optimiser notre code dans la manipulation de ces derniers (objets, variables). A cet effet *Python* dispose des listes et *tuples*, qui sont des objets fréquemment utilisés, et des dictionnaires et des sets.

Ici nous aborderons les notions essentielles liées à ces objets, et nous approfondirons par la suite dans un chapitre dédiée en deuxième partie.

14.1 Les *tuples*

Présentation

Syntaxe d'un *tuple* (on utilise des parenthèses) :

```
mon_tuple = (var1, var2, var3)
```

Un *tuple* n'est accessible, une fois créé, qu'en lecture seule, on ne peut donc le modifier. Il est dit *immutable* (terme de programmation). Une liste (cf. section suivante), par contre, est elle *mutable*.

Illustrons le fonctionnement d'un *tuple* :

```
auteurs = ("Djian", "Kerouac", "Hemingway")
print(auteurs)
```

Ce qui a pour effet d'afficher :

```
('Djian', 'Kerouac', 'Hemingway')
```

La notion d'indice

Il est cependant possible de n'afficher qu'un seul élément à l'aide de ce que l'on nomme des indices :

```
print(auteurs[0])
print(auteurs[2])
print(auteurs[3])
```

Respectivement, nous obtenons les sorties suivantes :

```
Djian
Hemingway
```

Mais pour le troisième `print()`, nous levons une exception de type `IndexError` avec la précision : `tuple index out of range`, que l'on peur traduire par *au-delà de l'index*. Notons que l'indice `0` correspond au premier élément, et l'indice `2` au troisième. En fait les indices débutent à `0`, d'où le décalage. Dans notre exemple il y a trois éléments dans le *tuple* `auteurs`. Ce que nous permet de vérifier l'instruction suivante :

```
print(len(auteurs))
```

La sortie pour cette instruction est bien 3.

Grâce aux indices nous allons ainsi pouvoir boucler sur notre objet de type *tuple* :

```
for i in range(0, len(auteurs)):
    print(auteurs[i]
```

Chaque élément de notre *tuple* sera alors affiché :

```
Djian
Kerouac
Hemingway
```

Une singularité du langage *Python* réside dans la possibilité d'utiliser des indices inversés (indices négatifs). `print(auteurs[-1])` affiche en sortie : `Hemingway`. Avec un indice `-1` nous avons accès au dernier élément, avec `-2` à l'avant dernier élément, et ainsi de suite.

Il est cependant possible de boucler sans l'aide des indices :

```
for i in auteurs:
    print(i)
```

Nous venons ainsi d'itérer sur chaque élément du *tuple*.

Similitudes avec ce que nous connaissons déjà

Une chaîne de caractères se comporte comme un *tuple*. Nous pouvons alors accéder aux divers éléments de la chaîne de caractères en bouclant sur cette dernière, avec ou sans indice.

Avec la fonction `range()` nous obtenons des objets de type collection que nous pouvons apparenter à des *tuples* :

```
a = range(0, 5)
print(a[-1])
```

Ce qui affiche `4` en sortie. En fait c'est comme si nous avions le *tuple* suivant : `(0, 1, 2, 3, 4)`, objet sur lequel nous pouvons itérer et avec lequel nous pouvons utiliser des indices.

14.2 Les listes

Présentation

La syntaxe d'une liste est quasi similaire à celle d'un *tuple*, sauf que l'on utilise des crochets au lieu de parenthèses.

```
ma_liste = [var1, var2, var3]
```

Avec les listes nous pouvons réaliser ce que nous venons de voir avec les *tuples*, mais également des manipulations supplémentaires. La différence entre ces deux types d'objets conteneurs est que les *tuples* utilisent moins de mémoire que les listes, et que ces dernières peuvent par contre être modifiées.

Modifier une liste

Les listes sont des conteneurs, et les valeurs contenues dans ce conteneur sont modifiables. Voyons cela en ajoutant un élément à une liste :

```
auteurs = ["Djian", "Kerouac", "Hemingway"]
new_auteur = "Ravalec"
auteurs.append(new_auteur)
print(auteurs)
```

Et notre liste ainsi modifiée s'affiche ainsi :

```
['Djian', 'Kerouac', 'Hemingway', 'Ravalec']
```

Supprimons maintenant l'élément situé à l'index `1` :

```
del auteurs[1]
print(auteurs)
```

Notre nouvelle liste affichée est alors la suivante :

```
['Djian', 'Hemingway', 'Ravalec']
```

Il est, bien évidemment envisageable de boucler sur les listes, à l'instar des *tuples*.

Petit exemple d'algorithme

```
1   """LISTES - ALGO : Valeur la plus petite
2   Détécter la valeur la plus petite et l'afficher en sortie,
3   à l'aide des indices.
4   Ici : afficher un nom à cette valeur minimale. """
5
6
7   noms_chauffeurs_km = [("Patrick", 1.5), ("Paul", 2.2),
8                         ("Marc", 0.4), ("Jean", 0.9),
9                         ("Pierre", 7.1), ("Marie", 1.1),
10                        ("Maxime", 0.6)]
11  # Première distance comme valeur de référence:
12  distance_min_et_nom = noms_chauffeurs_km[0]
```

```
13
14      for nom_et_distance in noms_chauffeurs_km:
15          if nom_et_distance[1] < distance_min_et_nom[1]:
16              distance_min_et_nom = nom_et_distance
17
18      print(f"Distance minimale:{distance_min_et_nom[1]} km. "
19            f"- Nom du chauffeur: {distance_min_et_nom[0]}.")
```

Listes et fonction

Voyons un exemple :

```
1   def modifier_valeur(a):
2       a[0] = 10
3
4
5   test = [1, 2, 3, 4]
6   print(test)
7   modifier_valeur(test)
8   print(test)
```

La sortie de ce code est :

```
[1, 2, 3, 4]
[10, 2, 3, 4]
```

14.3 *tuples* et fonction

A l'aide des *tuples* nous pouvons coder des fonctions qui retournent plusieurs valeurs, justement en retournant un *tuple*. Pour cela on va utiliser l'instruction `return` et des virgules.

```
return val1, val2, val3
```

Exemple :

```
1   def mon_tuple():
2       return "Djian", "Kerouac", "Hemingway"
3
4
5   # Affecter chaque valeur à une variable
6   auteur1, auteur2, auteur3 = mon_tuple()
7   # Ou créer un tuple:
8   mes_auteurs = mon_tuple()
```

Passer un *tuple* à une fonction en séparant les valeurs :

```
1   def afficher(nom1, nom2, nom3):
2       print(nom1)
```

```
3           print(nom2)
4           print(nom3)
5
6
7       mes_auteurs = ("Djian", "Kerouac", "Hemingway")
8       afficher(*mes_auteurs)  # Avec l'astérisque (*) on dit que l'on
9                               # unpack le tuple
10      print(mes_auteurs)
11      print(*mes_auteurs)  # Comme si on passait trois paramètres à
    la
12                           # fonction print()
```

Ce code nous donne la sortie suivante :

```
Djian
Kerouac
Hemingway
('Djian', 'Kerouac', 'Hemingway')
Djian Kerouac Hemingway
```

14.4 Les *slices*

Les *slices* sont des utilisations étendues des indices que l'on applique aux listes et aux *tuples*. Syntaxe d'un *slice* :

```
[start:stop:step]
ou
[début:fin(exclue):pas]
```

Voici quelques exemples :
[:] -> l'ensemble de la liste ;
[: :2] -> un élément sur deux (car avance de deux en deux dans la liste) ;
[: :-1] -> suit la liste dans l'ordre inversé ;
[: :-2] -> Idem mais avec un élément sur deux.

Les *slices* s'appliquent aussi sur les chaînes de caractères.

```
nom = "krystof"
print(nom[1:3])  # Affiche 'ry'
print(nom[::-1]  # Affiche 'fotsyrk'
```

14.5 La fonction `sort()`

```
ma_liste.sort()
```

Le tri de la liste s'opère directement sur la liste elle-même en suivant l'ordre alphabétique de la table `ASCII`, autrement dit de `A` à `Z`, puis de `a` à `z`.

Il est possible de modifier le comportement de la fonction `sort()`, et ce de deux façons :

```
ma_liste.sort(reverse=True)  # Opère un tri inverse
ma_liste.sort(key=mon_tri_personnalise)  # mon_tri_personnalise est une
                                         # fonction dans laquelle on code
                                         # le tri souhaité
```

14.6 Les dictionnaires

Le dictionnaire permet de structurer des données avec des champs, et par conséquent d'améliorer l'efficacité des certains algorithmes. Syntaxe d'un dictionnaire :

```
mon_dico = {"cle":val1, "cle2":val2, "cle3":val3}
```

L'utilisation des dictionnaires

```
1 pers = {'nom': 'jack', 'age': 25, 'taille': 1.60}
2 print(pers)
```

Cela donne en sortie :

```
{'nom': 'jack', 'age': 25, 'taille': 1.60}
```

Nous pouvons sélectionner uniquement certains champs :

```
1 print(pers['nom'])
```

Le résultat est :

```
jack
```

Un dictionnaire est mutable, nous pouvons donc lui réattribuer des valeurs.

```
1 pers['nom'] = 'Ernest'
```

Ou bien, depuis la version `3.9` de `Python`, à l'aide du « *pipe* » (`'|'`) :

```
1 pers|['nom', 'Ernest']
```

Nous pouvons ajouter de nouvelles clés dynamiquement :

```
1 pers['poste'] = "Développeur"
```

Les dictionnaires peuvent contenir tout type de données : `str`, `int`, `float`, *tuples*, listes, etc.

Boucler sur un dictionnaire :

```
for i in pers:
    print(i)
```

Cela permet de récupérer l'ensemble des clés du dictionnaire :

```
nom
age
taille
poste
```

Pour afficher les valeurs, le code est :

```
for i in pers:
    print(pers[i])
```

Ainsi, pour récupérer l'intégralité du dictionnaire :

```
for i in pers:
    print(f"clef: {i} - valeur: {pers[i]}")
```

Fusion de deux dictionnaires

Il existe la méthode du déballage, grâce à l'emploi des astérisques ('`*`'), mais il faut avouer que cela n'est pas très intuitif.

```
dict1 = {"Jean": (20, "Développeur"), "Pierre": (30, "Ingénieur")}
dict2 = {"Èmilie": (30, "Professeur"), "Marc": (25, "Chercheur")}

repertoire_complet = {**dict1, **dict2}
print(repertoire_complet)
```

Ce qui nous donne en sortie :

```
{'Jean': (20, 'Développeur'), 'Pierre': (30, 'Ingénieur'), 'Émilie': (30,
'Professeur'), 'Marc': (35, 'Chercheur')}
```

Il est possible d'utiliser la méthode du déballage de la façon suivante :

```
repertoire_complet = dict(dict1, **dict2)
```

A noter toutefois, que cette dernière méthode ne fonctionne que si `dict2` n'utilise que des clés de type `str`.

Pour un même résultat, mais avec l'emploi de la méthode `update()`, nous avons :

```
dict1.update(dict2)
```

Avec cette méthode, on ajoute `dict1` à `dict2`, ce qui finalement va écraser le `dict1` initial. De plus, avec des dictionnaires volumineux une telle opération va nécessité beaucoup de capacité mémoire.

Depuis la version `3.9` de `Python`[1]. nous pouvons recourir à l'utilisation de l'opérateur *pipe* ('`|`'), qui permet la concaténation.

```
1 dict_complet = dict1 | dict2
```

Ou bien encore :

```
1 dict1 |= dict2
```

Il s'agit d'une concaténation équivalente à `dict1 = dict1 + dict2`. C'est une affectation augmentée qui fonctionne « *in-place* », et qui se trouve être beaucoup plus efficiente au niveau mémoire comparativement à la méthode `dict1.update(dict2)`.

Avec toutes ces diverses méthodes, il faut savoir que les éventuels conflits de clés sont résolus, puisque seule la clé la plus à droite, autrement dit la dernière clé vue, sera prise en considération.

En terme de performance, si l'on souhaite fusionner un grand nombre de dictionnaires, il est préférable d'utiliser une boucle explicite et une union dite « *in-place* » :

```
1 new_dico = {}
2 for d in ensemble_dicos:
3     new_dico |= d
```

Liste *vs* dictionnaire

Avec une liste, pour rechercher un élément contenu dans la liste il va nous falloir itérer sur cette dernière. Avec le dictionnaire, grâce à la clé, nous avons un accès direct. Le dictionnaire va ainsi permettre une exécution plus rapide du programme, notamment si la masse de données est importante.

Si la clé n'existe pas

Quand la clé sollicitée n'existe par, *Python* lève une exception de type `KeyError`. Pour gérer ce cas de figure la méthode `.get()` retourne la valeur `None`.

```
1 habite = pers.get("ville")
2 if not habite:  # Equivalent à if habite == None:
3     ...code...
```

1. Cf. PEP 584 – Add Union Operators To dict - `https://peps.python.org/pep-0584/`

Troisième partie

Devenons plus intime avec *Python* : notions intermédiaires

Chapitre 15

Notions approfondies concernant les chaînes de caractères

15.1 Préfixes et suffixes d'une chaîne de caractères

La méthode `strip()`

Par défaut la fonction `strip()` supprime les espaces situés en fin et en début de chaîne.

```
1 p = "      ma chaîne     "
2 print(p.strip())
```

Ce qui nous donne en sortie :

```
ma chaîne
```

Avec un paramètre nous pouvons préciser ce qui doit être supprimer dans la chaîne :

```
1 p = "ma chaîne"
2 print(p.strip("ma "))
```

'`ma`' sera supprimé, et seul '`chaîne`' sera affiché. Mais cette méthode a ses limites.

```
1 p = "ceci est ma chaîne"
2 print(p.srip("chaîne"))
```

Cela aura pour effet d'afficher uniquement '`est ma`'. Cela s'explique par le fait que la fonction `strip()` va regarder au début de la chaîne et se centre sur '`ceci`'. '`c`' est dans '`chaîne`', alors on supprime '`c`'. Cela va fonctionner à l'identique pour les lettres '`e`', '`i`' et le second '`c`'. Puis la fonction `strip()` se centrera ensuite sur le mot '`chaîne`' afin de la supprimer.

Les fonctions `removeprefix()` et `removesuffix()`

La version `3.9` de `python` introduit deux nouvelles fonctions [1].

1. Cf. PEP 616 – String methods to remove prefixes and suffixes - `https://peps.python.org/pep-0616/`

```
p = "ceci est ma chaine"
# Pour supprimer uniquement 'chaine':
print(p.removesuffix("chaine"))
# Pour supprimer uniquement 'ceci':
print(p.removeprefix("ceci"))
# Pour supprimer 'ceci' et 'chaine':
print(p.removeprefix("ceci").removesuffix("chaine"))
```

Chapitre 16

Notions approfondies concernant les fonctions

16.1 Nombre variable de paramètres

Dans certains cas, lorsque vous définissez une fonction, vous ne savez pas à l'avance combien d'arguments vous souhaitez lui faire prendre. Mais, comme vous l'avez déjà vu, lorsque des arguments positionnels sont utilisés, le nombre d'arguments transmis doit correspondre au nombre de paramètres déclarés. Pour résoudre ce problème, nous faisons alors appel à l'empaquetage des arguments (« *Argument packing* »). Lorsqu'un nom de paramètre dans une définition de fonction `Python` est précédé d'un astérisque (`*`), cela indique que les tous les arguments correspondants dans l'appel de fonction sont empaquetés dans un *tuple* auquel la fonction peut faire référence par le nom du paramètre donné. Voici un exemple :

```
1  def somme(*nbres):
2      # '*' pour signifier le nombre inconnu de paramètres
3      resultat = 0
4      for n in nbres:
5          resultat += n
6      return resultat
7
8
9  print(somme(5, 2, 4, 7, 8))  # Affiche 26
10 print(somme(5, 2, 4))  # Affiche 11
11 print(somme())  # Affiche 0
```

Listing 16.1 – Nombre de paramètres non connu

Une opération analogue est disponible de l'autre côté de l'équation dans un appel de fonction `Python`. Lorsqu'un argument dans un appel de fonction est précédé d'un astérisque (`*`), cela indique que l'argument est un *tuple* qui doit être décomposé et transmis à la fonction en tant que valeurs séparées :

```
1  def somme(a, b, c):
2      resultat = a + b + c
3      return resultat
4
5
6  tuple_de_nbres = (5, 6, 9)
```

```
print(somme(*tuple_de_nbres))
```

Listing 16.2 – Décomposition de *tuple*

L'opérateur astérisque (*) peut être appliqué à tout itérable dans un appel de fonction `Python`, comme par exemple une liste ou un ensemble.

Vous pouvez même utiliser l'empaquetage et le dépaquetage de *tuple* en même temps :

```
def somme(*args):
    # '*' pour signifier le nombre inconnu de paramètres
    return sum(args)


a = (5, 6, 9)
b = (31, 54, 53, 5)
c = (3, 5, 86, 4, 4, 646, 42)
print(somme(*a))  # Affiche 20
print(somme(*b))  # Affiche 143
print(somme(*c))  # Affiche 790
```

Listing 16.3 – Empaquetage et dépaquetage

Peu importe le nom donné au paramètre, ce qui importe c'est la présence de l'astérisque (*). Cependant, par convention (par usage) c'est le mot `args` qui sera donné au paramètre, ainsi la fonction sera définie de la façon suivante :

```
def nom_fonction(*args)
```

`Python` dispose d'un opérateur similaire, le double astérisque (**), qui peut être utilisé avec les paramètres et les arguments des fonctions pour spécifier l'empaquetage et le dépaquetage des dictionnaires. Faire précéder un paramètre d'un double astérisque (**) dans une définition de fonction indique que les arguments correspondants, qui sont censés être des paires `clé=valeur`, doivent être empaquetés dans un dictionnaire :

```
def moyenne(**nbres):
    # '**' pour signifier des paramètres avec clé associée
    somme = 0
    for n in nbres.values():
        # Pour récupérer les valeurs associées à une clé
        somme += n
    return round(somme / len(nbres), 2)


print(f"Moyenne des notes : "
      f"{moyenne(maths=15, geo=11, anglais=18)}")
```

Listing 16.4 – Paramètres avec clé associée

Ce code donne en sortie :

```
Moyenne des notes : 14.67
```

Dans cet exemple, les arguments `maths=15, geo=11, anglais=18` sont empaquetés dans un dictionnaire que la fonction peut référencer par le nom `kwargs`. Encore une fois, n'importe quel nom peut être utilisé, mais le nom particulier `kwargs` (qui est l'abréviation de « *keyword args* ») est presque standard. Vous n'êtes pas obligé de l'utiliser, mais si vous le faites, toute personne connaissant les conventions de codage `Python` saura immédiatement ce que vous voulez dire.

Le dépaquetage d'un dictionnaire d'arguments est analogue au dépaquetage d'un *tuple* d'arguments. Lorsque le double astérisque (`**`) précède un argument dans un appel de fonction `Python`, il indique que l'argument est un dictionnaire qui doit être décomposé, et que les éléments résultants sont passés à la fonction en tant qu'arguments avec mots-clés :

```
notes = {"maths": 15, "geo": 11, "anglais": 18}
print(f"Moyenne des notes : {moyenne(**notes)}")
```

Listing 16.5 – Dépaquetage d'un dictionnaire

Les éléments du dictionnaire `notes` sont décompressés et passés à la fonction `moyenne()` comme arguments avec mots-clés. Ainsi, `moyenne(**notes)` est équivalent à `moyenne(maths=15, geo=11, anglais=18)`.

Les trois paramètres positionnels (standard, `*args` et `**kwargs`) peuvent être utilisés ensemble dans une définition de fonction `Python`. Si tel est le cas, ils doivent être spécifiés dans cet ordre :

```
def nom_fonction(standard, *args, **kwargs):
```

Voici un exemple :

```
def creer_liste(num_liste, *elements, **options):
    # '*' pour grouper les paramètres (capture les arguments
    # surnuméraires).
    # '**' pour grouper les arguments par mots clés.
    ma_liste = []
    print("-" * 10)
    print(f"Liste {num_liste} : ")
    print(f"elements = {type(elements)}")  # Affiche le tuple.
    print(f"options = {type(options)}")  # Le dictionnaire.
    for element in elements:
        if options.get("upper"):
            element = element.upper()
        elif options.get("capitale"):
            element = element.capitalize()
        ma_liste.append(element)
    print(f"Les paramètres optionels = {options}")
    return ma_liste  # Avec retour d'objet


def affichage_liste(*listes):
    num_liste = 1
    for liste in listes:
        print(f"Liste {num_liste}: {liste}")
        num_liste += 1
```

```
25
26
27 # Appels de fonction :
28 liste_mots = creer_liste(1, 'livre', 'bibliothèque', 'marque_page',
29                           option1=100, option2=200)
30 liste_noms_auteurs = creer_liste(2, "kerouac", "djian", "hemingway",
31                                   upper=True)
32 liste_nom_modifiee = creer_liste(3, "kerouac", "djian", "hemingway",
33                                   upper=False)
34 liste_prenoms_auteurs = creer_liste(4, "jack", "philippe", "ernest",
35                                      capitale=True)
36 # Affichages des listes à l'aide d'une autre fonction :
37 print("-" * 10)
38 affichage_liste(liste_mots, liste_noms_auteurs,
39                 liste_nom_modifiee, liste_prenoms_auteurs)
```

Listing 16.6 – Paramètres mixées

Ce qui donne en sortie :

```
----------
Liste 1 :
elements = <class 'tuple'>
options = <class 'dict'>
Les paramètres optionels = {'option1': 100, 'option2': 200}
----------
Liste 2 :
elements = <class 'tuple'>
options = <class 'dict'>
Les paramètres optionels = {'upper': True}
----------
Liste 3 :
elements = <class 'tuple'>
options = <class 'dict'>
Les paramètres optionels = {'upper': False}
----------
Liste 4 :
elements = <class 'tuple'>
options = <class 'dict'>
Les paramètres optionels = {'capitale': True}
----------
Liste 1: ['livre', 'bibliothèque', 'marque_page']
Liste 2: ['KEROUAC', 'DJIAN', 'HEMINGWAY']
Liste 3: ['kerouac', 'djian', 'hemingway']
Liste 4: ['Jack', 'Philippe', 'Ernest']
```

Cela offre à peu près toute la flexibilité dont vous pourriez avoir besoin dans une *interface de fonction*.

La version `3.5` de `Python` a introduit le *dépaquetage multiple* dans un appel de fonction, comme indiqué dans le `PEP 448`[1]. Une telle amélioration permet notamment de réaliser

1. `PEP 448` – *Additional Unpacking Generalizations* (trad. « *Généralités supplémentaires sur le déballage* ») - `https://peps.python.org/pep-0448/`

plusieurs dépaquetages dans un seul appel de fonction :

```
def ma_fonction(**kwargs):
    for k, v in kwargs.items():
        print(k, '->', v)


d1 = {'a': 1, 'b': 2}
d2 = {'x': 3, 'y': 4}
ma_fonction(**d1, **d2)
```

Listing 16.7 – Dépaquetage multiple

Ce qui nous donne en sortie :

```
a -> 1
b -> 2
x -> 3
y -> 4
```

16.2 Les fonctions récursives

Les fonctions récursives sont des fonctions qui peuvent être rappelées par elles-mêmes, mais il faut demeurer vigilant au fait qu'elles aient toujours une condition de sortie, au risque de se retrouver avec une boucle infinie.

```
  def f(nbre):
    print(nbre)
    f(nbre)


  f(2)  # Affichage du nombre 2 à l'infini
```

Listing 16.8 – Exemple de boucle sans fin

Voyons maintenant une vraie fonction récursive :

```
  def f(nbre, limite):
    if nbre > limite:
        return
    print(nbre)
    f(nbre*nbre, limite)


  f(2, 1000)
```

Listing 16.9 – Fonction récursive

Ce qui donne en sortie :

```
2
4
16
256
65536
```

16.3 Différence entre `break` et `return`

`break` permet de sortir d'une boucle et `return` permet de sortir d'une fonction. `break` peut s'utiliser en dehors d'une fonction tandis que ce n'est pas le cas pour `return`.

16.4 Les fonctions *callback* et *lambda*

Principe des fonctions *callback*

```
a = ma_fonction
```

Sans les parenthèses, la fonction ne sera pas exécutée. La fonction est simplement affectée à une variable qui devient alors une référence à la fonction. Du coup, `a()` est équivalent à `ma_fonction()`. Le fait de passer une fonction à une variable est ce que l'on appelle le *callback*.

Exemple à tester :

```
1  def afficher_table(n, operateur_str, operation_cbk):
2      # cbk = abréviation de callback
3      for i in range(1, 10):
4          print(f"{i} {operateur_str} {n} = "
5                f"{operation_cbk(i, n)}")
6
7
8  def mult_cbk(a, b):
9      return a*b
10
11
12 def add_cbk(a, b):
13     return a+b
14
15
16 afficher_table(2, "x", mult_cbk)
17 print("-" * 10)
18 afficher_table(2, "+", add_cbk)
```

Listing 16.10 – Le *callback*

Les fonctions *lambda*

Ce sont des fonctions qui n'ont pas de nom. Il s'agit de fonctions courtes qui ne contiennent que du code que l'on insère directement sans faire appel à la fonction (d'ailleurs impossible puisque la fonction n'a pas de nom).

Ainsi la fonction `mult_cbk()` vue précédemment pourrait s'écrire :

```
lambda a,b : a*b
```

L'appel de la fonction `afficher_table` peut donc au final s'écrire :

```
afficher_table(2, "x", lambda a,b : a*b)
```

Il n'y a alors plus besoin de la fonction `mult_cbk()`. Pareil avec la fonction `add_cbk()` qui devient `lambda a,b : a+b`.

Au final cela permet d'avoir du code très compact.

16.5 Les *décorateurs*

Un *décorateur* va venir décorer une fonction. C'est en fait une fonction qui prend en paramètres une autre fonction. Ils vont nous permettre d'exécuter du code avant et après celui de la fonction qu'ils décorent.

Illustrons cela par un script commenté :

```
1  from functools import wraps  # wraps nous permet de conserver
2  # nom et documentation de la fonction décorée.
3
4
5  def mon_decorateur(une_fonction):
6      @wraps(une_fonction)  # Au lieu des lignes 22 et 24
7      def mon_decorateur_interne(*args, **kwargs):
8          # *args pour tous les paramètres positionnés
9          # et **kwargs pour les paramètres nommés.
10         # Autrement nous pouvons juste utiliser les paramètres
11         # de la fonction qui doit être décorée.
12         # 'args' et 'kwargs' sont des conventions Python de
13         # nommage.
14         # Il est conseillé de mettre les deux, car on ne sait
15         # pas si nous aurons des arguments positionnels ou
16         # nommés.
17         print("Code exécuté avant la fonction...")
18         # Stockage de la variable décorée dans une variable :
19         # Pour un retour direct nous aurions codé :
20         # -> return une_fonction(*args, **kwargs)
21         valeur = une_fonction(*args, **kwargs)
22         print("Code exécuté après la fonction...")
23         return valeur
24     # Pour garder le nom de la fonction d'origine :
25     #mon_decorateur_interne.__name__ = une_fonction.__name__
26     # Pour garder la documentation de la fonction d'origine :
27     #mon_decorateur_interne.__doc__ = une_fonction.__doc__
28     return mon_decorateur_interne
29
30
31 @mon_decorateur  # C'est ainsi que l'on décor une fonction
32 # ==> Ceci est équivalent à :
```

```python
# ma_fonction = mon_decorateur(une_fonction=ma_fonction)
def ma_fonction(a):
    """Documentation de ma fonction :
    Ici, se contente d'afficher un message."""
    print(f"Dans ma fonction{a} : ")
    print(f"a = {a}")


@mon_decorateur
def ma_fonction2(a, b):
    print(f"Dans ma fonction{b} :")
    print(f"a = {a} - b = {b}")


@mon_decorateur
def ma_fonction3(a, b, c):
    print(f"Dans ma fonction{c} :")
    print(f"a = {a} - b = {b} - c = {c}")


# Une fonction est affectée au décorateur et est retournée
# via la variable :
# ma_fonction = mon_decorateur(une_fonction=ma_fonction)
ma_fonction(1)
print(f"print(ma_fonction) : {ma_fonction}")
print(f"ma_fonction.__name__ : {ma_fonction.__name__}")
print(f"ma_fonction.__doc__ : {ma_fonction.__doc__}")
print("-"*10)
ma_fonction2(1, 2)
print("-"*10)
ma_fonction3(1, 2, 3)
print("-"*10)
# Pour démontrer que le décorateur est bien une fonction :
print(f"La fonction 'mon_decorateur' : {mon_decorateur}")
```

Listing 16.11 – Fonction avec décorateur

Sources utilisées pour la rédaction de ce chapitre

- *Comprendre les fonctions* ***Python***, par Thierry Chappuis - *WePynaire* du 3 novembre 2022, disponible à l'adresse `https://edu.placepython.fr/`
- *Defining Your Own Python Function* (trad. « *Définir votre propre fonction* ***Python*** », par John Sturtz, 9 mars 2020 - `https://realpython.com/defining-your-own-python-function/`

Chapitre 17

Notions approfondies concernant les collections

17.1 append, extend, += et insert

Créons quelques listes :

```
auteurs = ["Djian", "Kerouac", "Marsé"]
poetes = ["Baudelaire", "Verlaine"]
classiques = ["Hugo", "Stendhal"]
philosophes = ["Morin", "Rousseau"]
auteurs2 = ["Ravalec", "McMurtry"]
```

La méthode `.append()` ne permet d'insérer qu'un seul item :

```
auteurs.append("Hemingway")
```

Pour ajouter tous les éléments un à un d'autre liste, nous allons utiliser la boucle `for` :

```
for e in poetes:
    auteurs.append(e)
```

Ajouter tous les éléments d'une liste à une autre (deux méthodes) :

```
auteurs.extend(classiques)
#    ou
auteurs += classiques
```

Insérer une nouvelle liste dans une liste à l'index souhaité. La nouvelle liste sera un élément de la liste :

```
auteurs.insert(2, auteurs2)  # Insertion à l'index 2
```

Comme avec les chaînes de caractères il est possible d'utiliser la concaténation avec des listes :

```
liste_complete_auteurs = auteurs + philosophes
```

Affichons maintenant tout cela :

```
print(f"Ma liste d'auteurs {auteurs}")
print(f"\nMa liste d'auteurs complète {liste_complete_auteurs}")
```

En sortie :

```
Ma liste d'auteurs ['Djian', 'Kerouac', ['Ravalec', 'McMurtry'], 'Marsé',
'Hemingway', 'Baudelaire', 'Verlaine', 'Hugo', 'Stendhal']

Ma liste d'auteurs complète ['Djian', 'Kerouac', ['Ravalec', 'McMurtry'], 'Marsé',
'Hemingway', 'Baudelaire', 'Verlaine', 'Hugo', 'Stendhal', 'Morin', 'Rousseau']
```

17.2 Copier une liste *vs* pointer sur une liste

`var = noms[:]` : Va permettre une copie de la liste et l'affecter à la variable `var`. Une modification de la liste contenue dans `var` n'aura aucune incidence sur celle contenue dans `noms`, car en mémoire il y a deux listes distinctes.

`var = nom` : Ici la variable `var` copie le nom de la variable `nom`. Toute modification de la liste par la variable `var` viendra modifier la liste affectée à la variable `nom`, car en fait `var` et `nom` pointent sur la même collection, il n'y a qu'une seule liste en mémoire.

17.3 Les tris : `sort()` *vs* `sorted()`

Trier une liste alphabétiquement :

```
liste.sort()
```

L'ordre des éléments est directement modifié au niveau de la liste. Aucune nouvelle liste n'est donc créée. On dit que l'opération est réalisée « en place ».

Pour créer une nouvelle liste on va utiliser la fonction `sorted()` :

```
liste_triee = sorted(liste)
```

Trier une liste dans l'ordre inverse de l'ordre alphabétique :

```
liste.sort(reverse=True)
liste_triee = sorted(liste, reverse=True)
```

Il est possible de réaliser des tris personnalisés à l'aide d'une fonction, et notamment une fonction *lambda*. L'option `key`, qui appelle une fonction, va nous permettre cela. Voyons l'exemple d'une fonction qui opérerait un tri sur la longueur de chaque élément, en les triant du plus petit au plus grand :

```
liste.sort(key=lambda x:len(x))
```

`x` représente l'entrée de la fonction et `len(x)` la sortie. Tenant compte de la longueur de chaque élément de la liste mais en les classant du plus grand au plus petit, le code est :

```
liste.sort(key=lambda x:len(x), reverse=True)
```

Nous pouvons aussi définir une fonction est l'appeler avec l'option `key`. `key=nom_fonction` : attention il n'y a pas les parenthèses d'appel.

Le paramètre `key` s'utilise aussi avec la fonction `sorted()`.

17.4 Opérations sur les éléments d'une collection avec les fonctions `min()`, `max()`, `sum()` et l'opérateur `in`

`min(collection)` : donne la valeur minimale ;
`max(collection)` : donne la valeur maximale

Appliqué à une chaîne de caractères, `min()` va retourner le caractère situé le plus bas dans l'ordre alphabétique, et `max` le plus haut. Cependant ce n'est pas une pratique recommandée car cela ne donne pas nécessairement un code lisible.

L'opérateur `in` permet de vérifier la présence d'un élément dans une collection. Est-ce que l'élément est contenu dans la liste :

```
if element in liste:
```

`sum(collection)` : calcule la somme de la collection, uniquement si les éléments de la collection sont des données de valeur numérique.

17.5 *Swapper* deux éléments d'une liste

Il s'agit ici d'échanger deux éléments d'une liste de façon atomique (c'est-à-dire en une seule ligne) :

```
liste[0], liste[4] = liste[4], liste[0]
```

17.6 `join` et `split`

`join` : signifie « rejoindre » et permet de coller ensemble les éléments de la collection ;
`split` : signifie « séparer ».

Avec les fonctions `join()` et `split()` on va utiliser des séparateurs.

```
mots = ["chat", "chien", "poule"]
mots_joints = "-".join(mots)
print(mots_joints)
```

Ce qui donne en sortie :

```
chat-chien-poule
```

A l'inverse :

```
new_liste = mots_joints.split("-")
print(new_liste)
```

La sortie est alors une nouvelle liste :

```
['chat', 'chien', 'poule']
```

17.7 index(), find() et count()

Pour obtenir l'index (indice) d'un élément dans une liste : `liste.index(element)`. Si l'élément recherché ne figure pas dans la liste, *Python* lève une exception de type `ValueError`. Si la liste contient deux éléments de même valeur, c'est l'index de la première occurrence qui est donné.

Il est toutefois possible de débuter la recherche de l'occurrence voulue à compter d'un index que l'on définit. Ainsi pour rechercher une occurrence à compter de l'index `3` (donc en ne tenant pas compte des éléments situés avant cet index) : `liste.index(element, 3)`.

`liste.count(element)` : donne le nombre d'occurrences (`element`) de même valeur.

Illustrons cela par le code suivant :

```
poetes = ["Baudelaire", "Verlaine", "Cros", "Verlaine",
          "Hugo", "Ronsard", "Verlaine"]

element_cherche = "Verlaine"
nb_occurences = poetes.count(element_cherche)
print(f"Nombre d'occurences pour '{element_cherche}': "
      f"{nb_occurences}")

if nb_occurences > 0:
    index_occurence = 0
    for i in range(nb_occurences):
        index_occurence = poetes.index(element_cherche,
                                       index_occurence)
        print(f"'{element_cherche}' trouvé à l'index "
              f"{index_occurence}")
        index_occurence += 1
else:
    print(f" '{element_cherche}' n'est pas dans la liste")
```

La sortie de ce code est :

```
Nombre d'occurences pour 'Verlaine': 3
'Verlaine' trouvé à l'index 1
'Verlaine' trouvé à l'index 3
'Verlaine' trouvé à l'index 6
```

`find()` ne fonctionne pas avec les collections mais peut s'utiliser avec des chaînes de caractères.

```
a = "Baudelaire-Verlaine-Hugo"
i = a.find("Verlaine")
print(i)
```

En sortie nous aurons le début de l'index pour « `Verlaine` » :

```
11
```

Si l'occurrence n'est pas trouvée, c'est la valeur `-1` qui est retournée.

17.8 Les compréhensions de listes

Les compréhensions de listes permettent de créer des listes avec une syntaxe autre que celle jusque là utilisée. Prenons l'exemple d'une liste qui devra contenir la liste du nombre de caractères de chaque éléments de type `str` contenus dans une autre liste. Nous connaissons la méthode `append()` que nous utilisons avec une boucle `for` :

```
auteurs = ["Djian", "Kerouac", "Baudelaire"]
lg_noms_auteurs = []
for nom in auteurs:
    lg_noms_auteurs.append(len(nom))
print(lg_noms_auteurs)
```

En sortie nous avons bien une liste contenant le nombre de caractères pour chaque nom :

```
[5, 7, 10]
```

Avec les compréhensions de listes nous pourrions coder cela de façon bien plus concise :

```
lg_noms_auteurs = [len(nom) for nom in auteurs]
```

Voyons de plus près l'instruction `len(nom) for nom in auteurs` :
`len(nom)` : correspond à ce que nous souhaitons obtenir.
`for` : on boucle.
`nom` : chaque élément.
`in` : dans.
`auteurs` : la liste auteurs.

Ce que nous pourrions littéralement traduire par : *obtenir la longueur de l'élément* ***nom*** *pour chaque élément* ***nom*** *contenu dans la liste* ***auteurs***.

Et nous pouvons aller encore plus loin en ajoutant des conditions afin de voir, par exemple, si nous incluons ou non certains éléments de la liste :

```
lg_noms_auteurs = [len(nom) for nom in auteurs if len(nom)<10]
print(lg_noms_auteurs]
```

Et avec notre condition la liste ainsi créée ne contiendra pas les nombres supérieurs ou égaux à 10 :

```
[5, 7]
```

Il y a une autre méthode usant d'une condition, qui elle va consister à déterminer la valeur de l'élément à inclure dans la liste :

```
1 lg_noms_auteurs = [len(nom) if len(nom)<10 else 0 for nom in auteurs]
2 print(lg_noms_auteurs)
```

Avec l'emploi d'une telle structure conditionnelle la nouvelle sortie est :

```
[5, 7, 0]
```

Voyons encore quelques exemples pour bien comprendre

Créons une liste contenant des nombres :

```
1 a = [i for i in range(5)]
```

Une liste ou nous excluons les nombres impairs :

```
1 a = [i for i in range(5) if (i//2)*2 == i]
```

Une liste qui contient l'élément booléen `True` pour l'élément correspondant pair, et l'élément booléen `False` si l'élément correspondant est impair :

```
1 a = [True if (i//2)*2 == 1 else False for i in range(5)]
2 print(a)
```

Et nous avons bien en sortie la liste suivante :

```
[True, False, True, False, True]
```

Construire une liste de *tuples* :

```
1 a = [(i, True if (i//2)*2 == 1 else False for i in range(5)]
2 print(a)
```

Ce qui donne en sortie :

```
[(0, True), (1, False), (2, True), (3, False), (4, True)]
```

Reconnaissons que dans ce dernier exemple la compréhension de listes s'avère bien plus parlante que si nous avions codé cela avec des `if`.

17.9 les fonctions `any()` et `all()`

`any` : que l'on peut traduire par `n'importe quel`.
`all` : traduisible par *tous*.

Les fonctions `any()` et **`all()`** s'utilisent avec des collections qui contiennent des booléns.

```
1 a = [True, False, False, True]
2 print(any(a))
```

La sortie de ce code est : `True`. En effet, à partir du moment où la fonction `any()` trouve un seul élément `True` dans la collection, alors elle retourne `True`. Et si tous les éléments sont `False`, alors elle retourne `False`.

Par contre avec la fonction `all()` il est nécessaire que tous les éléments soient `True` pour retourner `True`, dans le cas contraire c'est `False` qui est retourné.

Dans notre exemple ci-dessus, `print(all(a)` retourne `False`.

Illustrons l'utilité de tout ceci en imaginant chercher si une liste d'éléments de type `str` contient le caractère « `d` ».

```
1 auteurs = ["Baudelaire", "Djian", "Kerouac"]
2 noms_avec_d = [True if "d" in nom.lower() else False
3                for nom in auteurs]
4 print(noms_avec_d)
5 print(any(noms_avec_d)
```

Et nous avons en sortie :

```
[True, True, False]
True
```

Je vous laisse maintenant imaginer sans difficulté la sortie du code suivant :

```
1 noms_avec_d = any([True if "d" in nom.lower() else False
2                    for nom in auteus])
3 if nom_avec_d:
4     print("Trouvé !")
5 else:
6     print("Non trouvé !")
```

17.10 Tester si une chaîne contient des chiffres avec les fonctions `any()` et `isdigit()`

Pour tester si une chaîne de caractères contient des chiffres on va utiliser la fonction `isdigit()`, ainsi avec la chaîne `"toto"` la fonction retourne `False` et avec la chaîne `"2601"` elle retourne `True`.

```
1 print("Kerouac".isdigit())
2 print("123".isdigit())
```

Ce qui donne en sortie respectivement **False** et `True`. Par contre :

```
print("toto2601".isdigit())
```

retourne `False` car la chaîne ne contient pas que des chiffres. Pour vérifier si une chaîne de caractères contient des chiffres au milieu d'éléments qui ne sont pas des chiffres nous allons boucler sur la chaîne :

```
for c in "toto":
    if c.isdigit():
        print(True)
    print(False)
```

Cela retourne :

```
False
False
False
False
```

Et à l'aide d'une compréhension de listes :

```
print(any([e.isdigit() for e in "toto"]))
print(any([e.isdigit() for e in "toto123"]))
```

Nous avons comme retour :

```
False
True
```

17.11 La fonction `zip()`

Cette fonction permet de regrouper des listes.

```
auteurs = ["Kerouac", "Djian", "Hemingway"]
titres = ["Sur la route", "Sotos", "Mort dans l'après-midi"]
print(zip(auteurs, titres))
```

Le retour sera quelque chose du type `<zip object at 0x7F879cd49e80>`. En fait, on ne peut parcourir un objet de type `zip`. Pour en visualiser le contenu il est nécessaire de le transformer en liste.

```
print(list(zip(auteurs, titres)))
```

Nous obtenons en retour une liste de *tuples* :

```
[('Kerouac', 'Sur la route'), ('Djian', 'Sotos'),
 ('Hemingway', "Mort dans l'après-midi")]
```

Nous pouvons extraire chacun des éléments à l'aide d'une boucle `for` :

```
for (x,y) in zip(auteurs, titres):
    print(f"{x} a écrit '{y}'")
```

Ce qui nous donne en sortie :

```
Kerouac a écrit 'Sur la route'
Djian a écrit 'Sotos'
Hemingway a écrit 'Mort dans l'après-midi'
```

Voyons maintenant l'opération inverse :

```
auteurs_titres = zip(auteurs, titres)
dezippe = zip(*auteurs_titres)
print(list(dezippe))
```

Ce qui nous donne une liste avec deux *tuples* :

```
[('Kerouac', 'Djian', 'Hemingway'),
('Sur la route', 'Sotos', "Mort dans l'après-midi")]
```

Ou bien :

```
auteurs_titres = zip(auteurs, titres)
auteurs, titres = zip(*auteurs_titres)
print(auteurs)
print(titres)
```

Ce qui nous donne bien deux *tuples* :

```
('Kerouac', 'Djian', 'Hemingway')
('Sur la route', 'Sotos', "Mort dans l'après-midi")
```

17.12 Le *set*

Le *set* est une collection particulière.

```
auteurs = ["Djian", "Kerouac", "Baudelaire", "Djian"]
set_auteurs = set(auteurs)
print(set_auteurs)
```

Nous obtenons en retour :

```
{'Djian', 'Baudelaire', 'Kerouac'}
```

On notera :

1. une collection présentée avec des accolades ;

2. une collection dans un ordre modifié ;
3. l'absence d'éléments en doublon.

Un *set* va garantir que l'ensemble des éléments est unique. C'est un peu le principe des dictionnaires où l'on retrouve la présence de clés uniques. Si nous pouvons énumérer les éléments (valeurs) d'un *set*, on ne peut cependant y accéder à l'aide d'un index. Une tentative d'accès par indexation lève une exception de type `TypeError` et il est bien précisé que `'set' object is not subscriptable`.

Pour obtenir une liste sans doublon :

```
1 auteurs_sans_doublon = list(set(auteurs))
```

Notre liste sans doublon :

```
['Baudelaire', 'Djian', 'Kerouac']
```

Chapitre 18

La modularité

18.1 Un module pour regrouper des variables et des fonctions

La programmation modulaire consiste à décomposer une tâche de programmation volumineuse et difficile à gérer en sous-tâches ou modules distincts, plus petits et plus faciles à gérer. Les modules individuels peuvent ensuite être assemblés comme des blocs de construction pour créer une application plus importante.

La modularisation du code dans une grande application présente plusieurs avantages :

Simplicité : Plutôt que de se concentrer sur l'ensemble du problème, un module se concentre généralement sur une partie relativement petite du problème. Si vous travaillez sur un seul module, vous aurez un domaine de problème plus petit à gérer. Le développement est ainsi plus facile et moins sujet aux erreurs.

Maintenabilité : Si les modules sont écrits de manière à minimiser l'interdépendance, il est moins probable que les modifications apportées à un seul module aient un impact sur d'autres parties du programme (Vous pouvez même être en mesure d'apporter des modifications à un module sans avoir aucune connaissance de l'application en dehors de ce module).

La réutilisation : La fonctionnalité définie dans un seul module peut être facilement réutilisée par d'autres parties de l'application. Cela élimine le besoin de dupliquer le code.

Définition de la portée : Les modules définissent généralement un espace de noms distinct, ce qui permet d'éviter les collisions entre les identifiants dans différentes zones d'un programme.

Les fonctions, les modules et les paquets sont des constructions de `Python` qui favorisent la modularisation du code.

Dans un projet, les modules sont des fichiers sources qui viennent en complément du fichier principal (`main.py`) que l'on lance pour exécuter le programme. Ces modules doivent aussi porter l'extension `.py`.

Contenu classique d'un module :

```
1 """Docstring du module"""
2
3 def fonction1(arg1, arg1):
4     """Docstring de la fonction"""
5     ... Code de la fonction ...
6
```

```
7 def fonction2():
8     """Docstring de la fonction"""
9     ... Code de la fonction ...
```

18.2 La déclaration d'importation

Pour utiliser du code contenu dans un module il est nécessaire de réaliser les imports dans les divers fichiers où les modules sont indispensables, avec les appels correspondant.

```
1 import nom_module  # sans l'extension .py
```

Listing 18.1 – Import le plus classique

Notez que cela ne rend pas le contenu du module directement accessible à l'appelant. Chaque module possède sa propre table de symboles privée. L'instruction import `nom_module` place uniquement `nom_module` dans la table de symboles de l'appelant. Les objets qui sont définis dans le module restent dans la table de symboles privée du module. Depuis l'appelant, les objets du module ne sont accessibles que lorsqu'ils sont préfixés par `nom_module` via la notation par points, comme illustré ci-dessous :

```
1 import nom_module
2
3 mon_module.fonction_du_module()  # appel
```

Listing 18.2 – Appel d'un objet du module

Plusieurs modules séparés par des virgules peuvent être spécifiés dans une seule déclaration d'importation :

```
1 import module1, module2, module3
```

Listing 18.3 – Import de plusieurs modules

Une autre forme de l'instruction import permet d'importer des objets individuels du module directement dans la table de symboles de l'appelant :

```
1 from nom_module import fonction_du_module
2
3 fonction_du_module()  # Simplifie l'appel de l'objet
```

Listing 18.4 – Imports individualisés

Mais il faut bien faire attention au fait que cette forme d'importation place les noms d'objets directement dans la table des symboles de l'appelant, et par conséquent tout objet existant déjà avec le même nom sera écrasé. Ainsi, s'il existe déjà un objet nommé `fonction_du_module` dans la table des symboles de l'appelant, alors celui-ci sera écrasé lors de l'importation et remplacé par l'objet même du module.

Il est même possible d'importer sans discernement tout ce qui se trouve dans un module en une seule fois :

```
1 from nom_module import *
```

Listing 18.5 – Import de tous les objets du module

Cela placera les noms de tous les objets de `nom_module` dans la table de symboles locale, à l'exception de ceux qui commencent par le caractère de soulignement (_).

Mais une telle pratique peut s'avérer dangereuse, notamment si votre code est assez volumineux, car vous entrez des noms en masse dans la table des symboles locaux. Et le risque est d'écraser un nom local existant par inadvertance. Cependant, cette syntaxe reste très pratique lorsque vous vous amusez avec l'interpréteur interactif, à des fins de test ou de découverte.

Il est également possible d'importer des objets individuels mais de les introduire dans la table de symboles locale avec des noms alternatifs :

```
1 from nom_module import fonction_du_module as nom_alternatif
```

Listing 18.6 – Import avec nom alternatif

`nom_alternatif` est alors directement placé dans la table de symboles locale.

Vous pouvez également importer un module entier sous un autre nom :

```
1 import nom_module as autre_nom_module
2
3 autre_nom_module.fonction_du_module()  # Pour l'appel
```

Listing 18.7 – Import du module sous un autre nom

Le contenu d'un module peut être importé à partir d'une définition de fonction. Dans ce cas, l'importation sera uniquement locale et non globale :

```
1 def ma_fonction():
2     from module import fonction_module
3     return fonction_module(1 ,2)  # fonction prenant deux arguments
4
5 resultat = ma_fonction()
```

Listing 18.8 – Importation d'un module dans une fonction

Cependant a sein d'une fonction une instruction du type `from module import *`, et engendrera une exception de type `SyntaxError`.

A noter qu'un bloc `try` avec une clause `except ImportError` peut être utilisée pour se prémunir contre les tentatives d'importation infructueuses.

Au moment d'importer le module, *Python* va lire (ou créer s'il n'existe pas) un fichier `.pyc`. Ce fichier se trouve dans le répertoire `__pycache__` du projet. Ce fichier est directement généré par *Python* et contient le code compilé (ou presque compilé) du module. Il ne s'agit pas réellement de langage machine mais d'un format que *Python* décode un peu plus vite que le code source écrit par le programmeur. Ce dossier peut-être laissé tel quel ou supprimé, cela ne change rien au code.

18.3 La fonction `dir()`

La fonction intégrée `dir()` renvoie une liste de noms définis dans un espace de noms. Sans arguments, elle produit une liste de noms triés par ordre alphabétique dans la table de symboles locale où elle est appelée. Lorsqu'on lui donne en argument le nom d'un module, `dir()` liste les noms définis dans le module.

18.4 Le chemin de recherche des modules

Dans la continuité de l'exemple ci-dessus, regardons ce qui se passe lorsque Python exécute l'instruction :

```
1 import nom_module
```

Listing 18.9 – Importation d'un module

La recherche du fichier `nom_module.py` s'effectue au travers dune liste de répertoires constituée à partir des sources suivantes :

- Le répertoire à partir duquel le script d'entrée a été exécuté ou le répertoire courant si l'interpréteur est exécuté de manière interactive.
- La liste des répertoires contenus dans la variable d'environnement `PYTHONPATH`, si elle est définie.
- Une liste de répertoires dépendante de l'installation, configurée au moment de l'installation de Python.

Le chemin de recherche résultant est accessible dans la variable `Python sys.path`, qui est obtenue à partir d'un module nommé `sys` :

```
>>> import sys
>>> sys.path
['', '/usr/lib/python310.zip', '/usr/lib/python3.10',
'/usr/lib/python3.10/lib-dynload',
'/home/krystof/.local/lib/python3.10/site-packages',
'/usr/local/lib/python3.10/dist-packages', '/usr/lib/python3/dist-packages']
```

Bien entendu, le contenu exact de `sys.path` dépend de l'installation. Il existe aussi une autre possibilité qui consiste à placer le fichier du module dans un répertoire de votre choix, puis de modifier le `sys.path` afin qu'il contienne ce répertoire. Par exemple, vous pouvez placer le module `nom_module.py` dans le répertoire `/home/krystof/`, puis jouter ce répertoire dans le `sys.path` à l'aide de l'instruction suivante :

```
>>> sys.path.append(r'/home/krystof/')
```

Vous pouvez vérifier l'emplacement du module `nom_module` grâce à l'attribut `__file__` du module :

```
>>> nom_module.__file__
'/home/krystof/nom_module.py'
```

18.5 N'exécuter que le module

Tout fichier `.py` qui contient un module est essentiellement un script `Python`, et il n'y a aucune raison pour qu'il ne puisse pas être exécuté comme tel. Les modules peuvent être conçus avec la possibilité de s'exécuter en tant que script autonome afin de tester la fonctionnalité contenue dans le module. C'est ce qu'on appelle les tests unitaires.

Imaginons le module `calcul.py` qui contient le code suivant :

```
1 def carre(nbre):
2     """Retourne le carre du nombre passé en argument."""
3     return nbre*nbre
4
5
6 if __name__=="__main__":
7     # Code de test:
8     print(carre(5))
```

Listing 18.10 – Module `calcul.py`

Si le module est importé, le code de test ne sera pas exécuté. Mais si nous exécutons directement le code du module, alors les lignes, situées après la ligne 6, seront exécutées. En fait, tout repose sur la variable `__name__`, qui existe dès le lancement de l'interpréteur. Pour vérifier cela, saisir `print(__name__` directement dans l'interpréteur. Si `__name__` vaut `__main__` alors le fichier sera considéré, en étant lancé directement, comme un exécutable.

Nous pouvons rendre le fichier du module exécutable, y compris avec des paramètres, en le modifiant ainsi :

```
1 if (__name__ == '__main__'):
2     import sys
3     if len(sys.argv) > 1:
4         print(calcul(int(sys.argv[1])))
```

Listing 18.11 – Module `calcul.py` exécutable

Le module peut alors être exécuté de manière autonome en passant un argument entier sur la ligne de commande pour les tests :

```
>>> python calcul.py 5
25
```

18.6 Les *packages*

S'il est possible de regrouper plusieurs fonctions au sein d'un même module, nous pouvons aussi regrouper les modules dans des *packages*, que l'on insère, à l'instar des modules, dans notre projet. Y accéder, nécessite alors d'indiquer un chemin précis. En pratique les *packages* sont des répertoires. La création d'un *package* est assez simple, puisqu'elle utilise la structure hiérarchique des fichiers inhérente au système d'exploitation :

```
nom_package
    |
```

```
|___nom_module1
|
|___nom_module2
```

A l'intérieur de ces répertoires peuvent se retrouver d'autres répertoires (d'autres *packages*) ou des fichiers (modules). Plus le projet que l'on développe devient conséquent plus, en principe, le nombre de modules augmente, et nous avons alors besoin de recourir à l'emploi de *packages*. De la même manière que les modules permettent d'éviter les collisions entre les noms de variables globales, les *packages* permettent d'éviter les collisions entre les noms de modules.

Les importations et appels de fonctions au sein de *packages* se réalisent, à l'instar des appels de fonctions au sein des modules, de la façon suivante :

```
1 import nom_package
2
3 # Importation à l'aide du point (.) pour modéliser le chemin d'accès
4 mon_package.mon_module.ma_fonction()
```

Si on le souhaite on en peut importer qu'un module précis, ou même qu'une fonction bien précise :

```
1 from mon_package.mon_module import ma_fonction
```

18.7 Les sous-paquets

Les paquets peuvent contenir des sous-paquets imbriqués jusqu'à une profondeur arbitraire.

```
nom_package
|
|___nom_sous_package1
|   |___nom_module1
|   |___nom_module2
|
|___nom_sous_paquet2
    |___nom_module3
    |___nom_moduule4
```

L'importation fonctionne toujours de la même manière que précédemment. La syntaxe est similaire, mais une notation supplémentaire par points est utilisée pour séparer le nom du paquetage du nom du sous-paquet :

```
1 import nom_package.nom_sous_package1.nom_module2
```

Listing 18.12 – Importation module depuis un sous-paquet

18.8 Les fichiers *Python* « compilés »

Pour accélérer le chargement des modules, *Python* cache une version compilée de chaque module dans un fichier nommé `nom_module(version).pyc` au sein du répertoire `__pycache__` du projet. *Python* compare les dates de modification du fichier source et de sa version compilée pour voir si le module doit être à nouveau compilé. Ce processus est entièrement automatique.

Nous pouvons nous reporter au `PEP 3147` pour plus de précision sur cette compilation [1].

Sources pour la rédaction de ce chapitre

- *Python Modules and Packages – An Introduction* (trad. *Modules et paquets Python - Une introduction*, par John Sturtz - `https://realpython.com/python-modules-packages/`

1. `https://www.python.org/dev/peps/pep-3147`

Quatrième partie

La programmation orientée objet

Chapitre 19

Les principes de base

19.1 Comprendre la programmation orientée objet

La programmation orientée objet (**POO**), qui en anglais se dit *Object Oriented Programming (OOP)*, est un *paradigme de programmation* incontournable, et ce peu importe le langage (`Python`, `Java Script`, `PHP`, `Java`, `C++`, `C#`, etc.). La **POO** est une méthode de structuration d'un programme qui consiste à regrouper des propriétés et des comportements connexes dans des objets individuels. Il s'agit d'une approche permettant de modéliser des objets concrets ainsi que les relations entre ces objets. Avec tous les langages les concepts de la `POO` sont toujours les mêmes. On y retrouve les notions de *classe*, d'*instance*, d'*objet*, de *constructeur*, d'*héritage*, de *méthode*. La `POO` va nous permettre de mieux structurer et organiser notre code, d'écrire du code plus modulaire et plus évolutif. Mais cela va aussi nous aider à réduire les dépendances en créant des parties indépendantes et aisément réutilisables au sein d'un même projet ou pour d'autres projets. La `POO` est une méthode permet de regrouper des propriétés et des comportements connexes en faveur de divers objets individuels. Au final, l'objectif est de proposer un code plus qualitatif, pour servir des projets plus ou moins ambitieux et conséquents.

Quand on déroule des lignes d'instructions, qui se succèdent justement en une suite d'instructions, à l'instar d'une recette de cuisine que l'on appliquerait étape par étape, nous sommes dans ce que l'on appelle la *programmation impérative* (ou *procédurale*) qui est un autre *paradigme de programmation* courant. En `POO` le code est découpé en objets. Il sera alors nécessaire de bien définir les responsabilités des différents objets, par le choix architectural dans la gestion des dépendances entre les différents objets. Ce qu'il faut retenir, c'est que les objets sont au centre de la *programmation orientée objet* en `Python`, non seulement pour représenter les données, comme dans la *programmation procédurale*, mais aussi pour la structure globale du programme.

Les différents concepts :

classe : définition et implémentation de l'objet (mot clé `class`). La définition va nous permettre de créer un ou plusieurs objets, on dit alors que l'on instancie l'objet (ou que l'on construit un objet). Dans la classe on va d'ailleurs définir un constructeur, qui est une fonction appelée au moment de la création de l'objet et qui permet donc de l'initialiser.

instance : c'est l'objet.

méthode : chaque classe contient ses propres méthodes. Une méthode est comme une fonction.

héritage : va permettre d'utiliser les caractéristiques d'une classe dans une sous-classe. On va dire que cette sous-classe hérite de la première classe. On parle alors de classe enfant et de classe parent.

Structure et syntaxe générale :

```
class MaClasse:
    def __init__(parametres):
        self.param1 = valeur1
        self.param2 = valeur2

    def action():
        ... code ...

class MaClasseEnfant(MaClasse):
    def __init__():
        super().__init__(parametres)

    def action_autre():
        ... code ...
```

Listing 19.1 – Syntaxe générale

19.2 Définir une classe

Les classes sont utilisées pour créer des structures de données définies par l'utilisateur, et définissent aussi des fonctions, appelées *méthodes*, qui identifient les comportements et les actions applicables à ces données (ou ces objets que sont les données). La classe va donc nous servir à stocker des informations sur les caractéristiques et les comportements d'un objet. C'est donc un modèle de définition d'un objet. Elle ne contient pas réellement de données, ni même le nom de ces données.

Toutes les définitions de classe commencent par le mot-clé `class`, qui est suivi du nom de la classe et de deux points. Tout le code qui est indenté sous la définition de la classe est considéré comme faisant partie du corps de la classe.

```
class Livre:
    pass
```

Listing 19.2 – Définition d'une classe

Par convention, les noms de classes `Python` sont écrits en lettres majuscules. Ainsi la classe portant sur les auteurs de roman pourrait être nommée ainsi : `AuteurDeRomans`.

Notre classe `LIvre`, pour le moment, ne produit rien. Nous allons donc lui donner quelques caractéristiques comme le nom de l'auteur et son année de naissance. Ces caractéristiques (ou propriétés vont se définir dans une méthode appelée `.__init__()`. Chaque fois qu'un nouvel objet `Livre` est créé, `.__init__()` définit l'état initial de l'objet en lui attribuant des valeurs à ses propriétés. En d'autres termes, `.__init__()` initialise chaque nouvelle instance de la classe.

Vous pouvez donner à `.__init__()` un nombre quelconque de paramètres, mais le premier paramètre sera toujours une variable appelée `self`. Lorsqu'une nouvelle instance de classe est créée, l'instance est automatiquement transmise au paramètre `self` de `.__init__()` afin que de nouveaux attributs puissent être définis sur l'objet.

Illustrons cela avec la classe `Livre` :

```
1 class Livre:
2     def __init__(self, titre, nom_auteur, prenom_auteur,
3                  an_pub) :
4         self.titre = titre
5         self.nom_auteur = nom_auteur
6         self.prenom_auteur = prenom_auteur
7         self.an_pub = an_pub
```

Listing 19.3 – Initialisation avec la méthode `.__init__()`

Remarquez que la signature de la méthode `.__init__()` est indentée de quatre espaces. Le corps de la méthode est indenté de huit espaces. Cette indentation est d'une importance vitale. Elle indique à `Python` que la méthode `.__init__()` appartient à la classe `Livre`.

Dans le corps de la méthode `.__init__()`, il y a deux instructions qui utilisent la variable `self` :

- `self.titre = titre` crée un attribut appelé `titre` et lui attribue la valeur du paramètre `titre`.
- `self.nom_auteur = nom_auteur` crée un attribut appelé `nom_auteur` et lui attribue la valeur du paramètre `nom_auteur`.
- `self.prenom_auteur = prenom_auteur` crée un attribut appelé `prenom_auteur` et lui attribue la valeur du paramètre `prenom_auteur`.
- `self.an_pub = an_pub` crée un attribut appelé `an_pub` et lui attribue la valeur du paramètre `an_pub`.

Les attributs créés dans `.__init__()` sont appelés *attributs d'instance*. La valeur d'un attribut d'instance est spécifique à une instance particulière de la classe. Tous les objets `Livre` ont un titre, un nom et un prénom d'auteur, et une date de publication, mais les valeurs des attributs varient en fonction de l'instance `Livre`. En revanche, les *attributs de classe* sont des attributs qui ont la même valeur pour toutes les instances de la classe. Vous pouvez définir un *attribut de classe* en attribuant une valeur à un nom de variable en dehors de la méthode `.__init__()` [1].

19.3 Instancier un objet

Alors que la classe est le modèle, une *instance* est un objet construit à partir d'une classe et contenant des données réelles (un nom et des caractéristiques). Par similitude ont pourrait dire qu'une classe est comme un formulaire ou un questionnaire, et l'instance finalement le formulaire qui a été rempli d'informations. Tout comme de nombreuses personnes peuvent remplir le même formulaire avec leurs propres informations, de nombreuses instances peuvent être créées à partir d'une seule classe.

La création d'un nouvel objet à partir d'une classe est appelée *instanciation d'un objet*. Vous pouvez instancier un nouvel objet à l'aide du nom de la classe, suivi de parenthèses ouvrantes et fermantes, en affectant ces objets à une variable. Voyons cela en mode interactif :

1. Voir infra, *Les attributs d'instance et de classe* page 118.

```
>>> class Auteur():
...     pass
...
>>> auteur1 = Auteur()
>>> auteur2 = Auteur()
>>> auteur1
<__main__.Auteur object at 0x7f649b2aeef0>
>>> auteur2
<__main__.Auteur object at 0x7f649b2af7c0>
>>> auteur1 == auteur2
False
```

Nous venons de créer deux nouveaux objets qui sont deux instances de la classe `Auteur`. Nous voyons que ces deux objets ont bien deux adresses mémoire différente. Il s'agit bien de deux objets distincts, ce qui est confirmé par le résultat de la ligne `auteur1 == auteur2`.

Reprenons maintenant notre exemple de la classe `Livre` telle que définit plus haut :

```
1 class Livre:
2     def __init__(self, titre, nom_auteur, prenom_auteur,
3                  an_pub) :
4         self.titre = titre
5         self.nom_auteur = nom_auteur
6         self.prenom_auteur = prenom_auteur
7         self.an_pub = an_pub
```

Listing 19.4 – La classe `Livre`

Pour instancier des objets de la classe `Livre`, vous devez fournir des valeurs pour chaque attribut. Si vous ne le faites pas, `Python` lève une `TypeError` :

```
1 livre1 = Livre()
```

Listing 19.5 – Instanciation sans valeur

Exception levée :

```
Traceback (most recent call last):
  File "/home/krystof/Python/TESTS/main.py", line 7, in <module>
    livre1 = Livre()
TypeError: __init__() missing 4 required positional arguments: 'titre',
nom_auteur', 'prenom_auteur' and 'an_pub'
```

Pour passer des arguments aux paramètres, il est nécessaire d'insérer des valeurs entre les parenthèses qui suivent le nom de la classe :

```
1 livre1 = Livre("Sotos", "DJIAN", "Philippe", 1993)
2 livre2 = Livre("Sur la route", "KEROUAC", "Jack", 1957)
```

Listing 19.6 – Instanciation avec valeurs

Lorsque vous instanciez un objet, `Python` crée une nouvelle instance et la transmet au premier paramètre de la méthode `.__init__()`. Vous n'avez pas à vous occuper du premier paramètre, `self`, mais uniquement paramètres qui suivent, ici `titre`,`nom_auteur`, `prenom_auteur` et `an_pub`.

Après avoir créé les instances de la classe, vous pouvez accéder à leurs attributs en utilisant la notation avec point :

```
class Livre:
    def __init__(self, titre, nom_auteur, prenom_auteur,
                 an_pub) :
        self.titre = titre
        self.nom_auteur = nom_auteur
        self.prenom_auteur = prenom_auteur
        self.an_pub = an_pub


livre1 = Livre("Sotos", "DJIAN", "Philippe", 1993)
livre2 = Livre("Sur la route", "KEROUAC", "Jack", 1957)
print(f"'{livre1.titre}' a été publié en {livre1.an_pub}")
print(f"{livre2.prenom_auteur} {livre2.nom_auteur} a écrit "
      f"'{livre2.titre}'")
```

Listing 19.7 – Accès aux atributs d'instance

Notre code affichera alors le résultat suivant :

```
'Sotos' a été publié en 1993
Jack KEROUAC a écrit 'Sur la route'
```

L'un des principaux avantages de l'utilisation de classes pour organiser les données est que les instances ont la garantie de posséder les attributs que vous attendez. Toutes les instances de la classe `Livre` possèdent les attributs `.titre`, `.nom_auteur`, `prenom_auteur` et `.an_pub`. Vous pouvez donc utiliser ces attributs en toute confiance, sachant qu'ils renverront toujours une valeur. Mais bien que l'existence des attributs soit garantie, leurs valeurs peuvent être modifiées dynamiquement :

```
livre1 = Livre("Sotos", "DJIAN", "Philippe", 1993)
livre1.titre = "Bleu comme l'enfer"
print(livre1.titre)
```

Listing 19.8 – Modification dynamique des attributs

Le résultat sera :

```
Bleu comme l'enfer
```

Ce qu'il faut retenir ici, c'est que les objets personnalisés sont mutables par défaut. Un objet est mutable s'il peut être modifié dynamiquement.

19.4 Les méthodes d'instance

Les *méthodes d'instance* sont des *fonctions* qui sont définies à l'intérieur d'une classe et qui ne peuvent être appelées que par une instance de cette classe. Tout comme `.__init__()`, le premier paramètre d'une méthode d'instance est toujours `self`.

Reprenons notre classe `LIvre` et ajoutons deux méthodes d'instance :

```
def afficher_infos(self):
    return f"{self.prenom_auteur} {self.nom_auteur} a écrit:\n"\
           f"'{self.titre}'\n" \
           f"Ce {self.genre} a été publié en {self.an_pub}"

def id_livre(self):
    cle_id_livre = f"{self.titre[0:4]}{self.nom_auteur[0:4]}" \
                   f"{self.prenom_auteur[0:4]}{self.an_pub}"
    return f"Sa clé d'identifiation est {cle_id_livre}"
```

Listing 19.9 – Méthodes d'instance

Ajoutons les lignes suivantes à notre code :

```
livre1 = Livre("Sotos", "DJIAN", "Philippe", 1993)
msg1 = livre1.afficher_infos()
msg2 = livre1.id_livre("Sa clé d'identification est: ")
print(f"{msg1}\n{msg2}")
```

Listing 19.10 – Accéder aux méthodes d'instance

Le résultat de ce code sera le suivant :

```
Philippe DJIAN a écrit:
'Sotos'
Ce roman a été publié en 1993
Sa clé d'identification est:  SotoDJIAPhil1993
```

La méthode `.__str__()`

`.afficher_infos()` renvoie une chaîne contenant des informations sur l'instance `Livre`. Lorsque vous écrivez vos propres classes, c'est une bonne idée d'avoir une méthode qui renvoie une chaîne de caractères contenant des informations utiles sur une instance de la classe. Cependant, `.afficher_infos()` n'est pas la façon la plus pythonique de le faire. Nous pouvons pour cela employer une méthode d'instance spéciale appelée `.__str()__` :

Les méthodes employées ici ne servent qu'à l'affichage. Nous pouvons pour cela employer une méthode spéciale qui est la méthode `.__str__()`. Plutôt que d'appliquer une méthode à notre instance, nous solliciterons directement un affichage du résultat attendu par la fonction `print()`, ce qui nous permettrait de coder : `print(notre_instance)`. Illustrons cela toujours depuis notre classe `Livre` :

```
def __str__(self):
    return f"{self.prenom_auteur} {self.nom_auteur} a écrit:\n"\
```

```
3             f"'{self.titre}'\n" \
4             f"Ce {self.genre} a été publié en {self.an_pub}"
```

Listing 19.11 – La méthode `.__str__()`

Et les lignes d'appel des méthodes seront les suivantes :

```
1 livre_1 = Livre("Sotos", "DJIAN", "Philippe", 1993)
2 print(livre_1)
3 print(livre_1.id_livre("Sa clé d'identification est: "))
```

Listing 19.12 – Appel de la méthode `.__str__()`

Cela nous affichera en sortie un résultat identique au précédent.

Des méthodes comme `.__init__()` et `.__str__()` sont appelées méthodes *dunder* (Le mot *dunder* est un raccourci de *Double UNDERscore*) parce qu'elles commencent et se terminent par un double trait de soulignement. Il existe de nombreuses méthodes *dunder* dont la compréhension est un élément important de la maîtrise de la programmation orientée objet en Python [2].

19.5 Une illustration basique de l'utilisation de la POO

```
1  """POO - Un exemple simple"""
2
3
4  # --- DEFINITION ---
5  class Personne:
6      def __init__(self, nom: str, date: int):
7          self.nom = nom
8          self.date = date
9
10     def __str__(self) -> str:
11         msg = f"{self.nom} est né en {self.date}, et il est"
12         if self.majeur_ou_mineur():
13             return f"{msg} majeur."
14         return f"{msg} mineur."
15
16     def majeur_ou_mineur(self) -> bool:
17         if (2022 - self.date) >= 18:
18             return True
19         return False
20
21
22 # --- MAIN ---
23 liste_personnes_avec_annee_naissance = (
24     ("Philippe", 1949),
25     ("Pierre", 1985),
```

2. Il s'agit d'un sujet avancé de la POO.

```
    ("Krystof", 2012),
    ("Antoine", 2000),
    ("Yann", 2017)
)

for personne in liste_personnes_avec_annee_naissance:
    individu = Personne(personne[0], personne[1])
    print(individu)
```

Listing 19.13 – Quel est l'âge de chacun ?

La ligne `def __init__(self):` vient définir notre constructeur. C'est au sein de ce constructeur que l'on peut y insérer des données. C'est ici que l'objet est créé en mémoire. `self` représente l'objet créé, et c'est ce `self` qui est retourné par le constructeur.

Dans le constructeur du code ci-dessus la ligne `self.nom = nom` crée la variable d'instance `nom`. Les variables d'instance doivent toujours être créées au niveau du constructeur. Avec le constructeur il est possible de spécifier le type attendu :

```
def __init__(self, nom: str, date: int):
```

`individu` est une instance de la classe `Personne`.

Voyons un second exemple :

```
class Livre:
    genre = "roman"

    def __init__(self, mon_livre):
        self.titre = mon_livre[0]
        self.nom_auteur = mon_livre[1]
        self.prenom_auteur = mon_livre[2]
        self.an_pub = mon_livre[3]

    def __str__(self):
        return f"{self.prenom_auteur} {self.nom_auteur} a " \
               f"écrit:\n'{self.titre}'\nCe {self.genre} " \
               f"a été publié en {self.an_pub}"

    def id_livre(self, msg):
        cle_id_livre = f"{self.titre[0:4]}" \
                       f"{self.nom_auteur[0:4]}" \
                       f"{self.prenom_auteur[0:4]}" \
                       f"{self.an_pub}"
        return f"{msg} {cle_id_livre}\n"


liste_livres = (
    ("Sotos", "DJIAN", "Philippe", 1993),
    ("Sur la route", "KEROUAC", "Jack", 1957),
    ("Wendy", "RAVALEC", "Vincent", 1996)
)
```

```
28
29 for livre in liste_livres:
30     infos_livres = Livre(livre)
31     print(infos_livres)
32     print(infos_livres.id_livre("Sa clé d'identification est: "))
```

Listing 19.14 – La classe Livre complétée

19.6 Débogage et classes

`Step into` pour entrer dans le constructeur ou les méthodes. Faire des tests avec le code suivant :

```
1  # --- DEFINITION ---
2  class Auteur:
3      def __init__(self, nom="", date=0):
4          self.nom = nom
5          self.date = date
6          if self.nom == "":
7              self.demander_infos_auteur()
8
9      def presentation(self):
10         msg_principal = f"{self.nom} est né"
11         if self.date != 0:
12             print(f"{msg_principal} en {self.date}")
13             if self.plusde100ans():
14                 print(f"{msg_principal} il y a plus de 100 ans.")
15             else:
16                 print(f"{msg_principal} il y a moins de 100 ans.")
17         else:
18             print(f"{self.nom} (date de naissance inconnue).")
19
20     def plusde100ans(self) -> bool:
21         return (2022 - self.date) > 100
22
23     def demander_infos_auteur(self):
24         self.nom = input("Nom de l'auteur: ")
25         self.date = int(input("Année de naissance (0 si inconnue): "))
26
27
28 # --- UTILISATION ---
29 auteur1 = Auteur("Jack KEROUAC")  # Création (on dit instanciation)
30 auteur2 = Auteur("Philippe DJIAN", 1949)
31 auteur3 = Auteur("Enest HEMINGWAY", 1899)
32 auteur4 = Auteur()
33 annonce = "Les auteurs:"
34 print(annonce)
35 print("-" * len(annonce))
36 auteur1.presentation()
37 auteur2.presentation()
```

```
auteur3.presentation()
auteur4.presentation()
```

Listing 19.15 – Pour tester le débogage

19.7 Les attributs de classe

Une variable commune à toutes les instances de la classe est une variable de classe.

```
class Personne:
    pays = "France"  # Attribut de classe

    def __init__(self, nom, age)  # Constructeur
        # Avec deux variables d'instance:
        self.nom = nom
        self.age = age

    def afficher(self):
        print(f"nom = {self.nom}")
        print(f"age = {self.age}")

    def pays_origine():
        print(f"Pays d'origine = {Personne.pays}")

Personne("Jean", 30)
Personne.afficher()
Personne.pays_origine()
```

Listing 19.16 – Attribut de classe

Ce qui donne en sortie :

```
nom = Jean
age = 30
Pays d'origine = France
```

`pays_origine` est une méthode de classe (ou méthode statique). Elle peut aussi s'écrire de la façon suivante :

```
def pays_origine(self):
    print(f"Pays d'origine = {self.pays}")
```

Et pour l'appel : `Personne.pays_origine()`.

Les *attributs de classe* sont définis directement sous la première ligne du nom de la classe et sont indentés de quatre espaces. Une valeur initiale doit toujours leur être attribuée. Lorsqu'une *instance de la classe* est créée, les *attributs de classe* sont automatiquement créés et affectés à leurs valeurs initiales. Utilisez les *attributs de classe* pour définir les propriétés qui doivent avoir la même valeur pour chaque *instance de classe*. Utilisez les *attributs d'instance* pour les propriétés qui varient d'une instance à l'autre.

Il est toutefois possible (pour une personne précise par exemple) de venir modifier la valeur de la variable de classe. Cependant, en pratique, on ne le fait pas. L'intérêt de la variable de classe est bien d'être commune à toutes les valeurs. Il faut considérer la variable de classe davantage comme une donnée constante (on peut alors coder le nom de la variable en majuscules, à l'instar des constantes).

Sources pour la rédaction de ce chapitre

- *Object-Oriented Programming (OOP) in Python 3* (trad. *Object-Oriented Programming (OOP) in Python 3*, par David Amos, 6 juillet 2020 - `https://realpython.com/python3-object-oriented-programming/`

Chapitre 20

La notion d'héritage

L'héritage permet de définir une classe ayant des caractéristiques communes avec une autre classe. Cette classe ne redéfinit qu'une partie du code de l'autre classe. La classe possédant les caractéristiques communes se nomme la *classe parent* et les autres les *classes enfants* (ou *classes dérivées*). Les *classes enfant* peuvent remplacer ou étendre les *attributs* et les *méthodes* des *classes parent*. En d'autres termes, les *classes enfant* héritent de tous les *attributs* et *méthodes* de la *classe parent*, mais peuvent également spécifier des *attributs* et *méthodes* qui leur sont propres.

Pour remplacer une *méthode* définie dans la *classe parent*, il faut définir une *méthode* portant le même nom dans la *classe enfant*.

```
1 class Poesie(Livre):
2     pass
3
4
5 class PieceDeTheatre(Livre):
6     pass
```

Listing 20.1 – Définir deux textitclasses enfant

Vous pouvez maintenant instancier les ouvrages selon leur genre littéraire :

```
1 livre_1 = ("Sotos", "DJIAN", "Philippe", 1993)
2 livre_2 = ("Les Fleurs du Mal", "BAUDELAIRE", "Charles", 1857)
3 livre_3 = ("En attendant Godot", "BECKETT", "Samuel", 1948)
4 oeuvre_1 = Livre(livre_1)
5 oeuvre_2 = Poesie(livre_2)
6 oeuvre_3 = PieceDeTheatre(livre_3)
```

Listing 20.2 – Les instanciations

Pour déterminer la classe à laquelle appartient un objet donné, vous pouvez utiliser la fonction intégrée `type()` :

```
1 print(type(oeuvre_1))
2 print(type(oeuvre_2))
3 print(type(oeuvre_3))
```

Listing 20.3 – La fonction `type()`

Ce qui donne en sortie :

```
<class '__main__.Livre'>
<class '__main__.Poesie'>
<class '__main__.PieceDeTheatre'>
```

Déterminer si un objet est aussi une instance d'une classe données à l'aide de la fonction intégrée `isinstance()` :

```
1 print(isinstance(oeuvre_2, Livre))
```

Listing 20.4 – La fonction `isinstance()`

En sortie :

```
True
```

Remarquez que `isinstance()` prend deux arguments, un objet et une classe.

Tous les objets créés à partir d'une classe enfant sont des instances de la classe parent, et ne sont pas des instances d'autres classes enfants.

Une chose à garder à l'esprit à propos de l'héritage des classes est que les changements apportés à la classe mère se propagent automatiquement aux classes enfants. Cela se produit tant que l'attribut ou la méthode modifiés ne sont pas remplacés dans la classe enfant.

Illustrons cela par le code suivant :

```
1  class Livre:
2
3      def __init__(self, mon_livre):
4          self.titre = mon_livre[0]
5          self.nom_auteur = mon_livre[1]
6          self.prenom_auteur = mon_livre[2]
7          self.an_pub = mon_livre[3]
8          self.cle_id_livre = self.id_livre()
9
10     def __str__(self):
11         return f"{self.prenom_auteur} {self.nom_auteur} a " \
12                f"écrit:\n'{self.titre}'\nCe {self.genre} " \
13                f"a été publié en {self.an_pub}"
14
15     def id_livre(self):
16         return f"{self.titre[0:4]}{self.nom_auteur[0:4]}" \
17                f"{self.prenom_auteur[0:4]}{self.an_pub}"
18
19
20 class Roman(Livre):
21     def __str__(self):
22         return f"{super().__str__()}\n" \
23                f"Son identifiant est: R:{self.cle_id_livre}\n"
24
25
26 class Poesie(Livre):
```

```
27      def __str__(self):
28          return f"{super().__str__()}\n" \
29                 f"Son identifiant est: P:{self.cle_id_livre}\n"
30
31
32  class PieceDeTheatre(Livre):
33      def __str__(self):
34          return f"{super().__str__()}\n" \
35                 f"Son identifiant est: T:{self.cle_id_livre}\n"
36
37
38  liste_livres = (
39      ("Sotos", "DJIAN", "Philippe", 1993, "R"),
40      ("Les Fleurs du Mal", "BAUDELAIRE", "Charles", 1857, "P"),
41      ("En attendant Godot", "BECKETT", "Samuel", 1948, "T"),
42      ("Les Essais", "MONTAIGNE", "Michel de", 1580, None)
43
44  )
45
46  for livre in liste_livres:
47      infos_livres = Livre(livre)
48      if livre[4]:
49          if livre[4] == "R":
50              infos_livres = Roman(livre)
51          elif livre[4] == "P":
52              infos_livres = Poesie(livre)
53          elif livre[4] == "T":
54              infos_livres = PieceDeTheatre(livre)
55          print(infos_livres)
56      else:
57          print(infos_livres)
58          print(f"Son identifiant est: X:"
59                f"{infos_livres.id_livre()}")
```

Listing 20.5 – Encore des livres...

Voyez la sortie de ce code :

```
Philippe DJIAN a écrit:
'Sotos'
Ce livre a été publié en 1993
Son identifiant est: R:SotoDJIAPhil1993

Charles BAUDELAIRE a écrit:
'Les Fleurs du Mal'
Ce livre a été publié en 1857
Son identifiant est: P:Les BAUDChar1857

Samuel BECKETT a écrit:
'En attendant Godot'
Ce livre a été publié en 1948
Son identifiant est: T:En aBECKSamu1948
```

```
Michel de MONTAIGNE a écrit:
'Les Essais'
Ce livre a été publié en 1580
Son identifiant est: X:Les MONTMich1580
```

20.1 La fonction `super()`

La fonction `super()` permet de faire appel à la classe parente sans avoir besoin de la nommer explicitement. Elle est donc utilisée, notamment dans la méthode `__init__()`, pour faire appel aux objets d'une classe parente.

Un exemple :

```
1  class Auteurs:
2      def __init__(self, name, surname):
3          self.name = name
4          self.surname = surname
5
6
7  class Poetes(Auteurs):
8      def __init__(self, name, surname):
9          # Sans super() il faudrait coder :
10         # Auteurs.__init__(self, name=name, surname=surname)
11         # Avec super() :
12         super().__init__(name=name, surname=surname)
13
14
15 auteur = Auteurs(name="Kerouac", surname="Jack")
16 poete = Poetes(name="Rimbaud", surname="Arthur")
17 print(f"Poète : {poete.name} {poete.surname}")
18 print(f"Romancier : {auteur.name} {auteur.surname}")
```

Listing 20.6 – Fonction `super()` et méthode `.__init__()`

Cette fonction peut s'utiliser en dehors de la *méthode* `__init__()` et employée dans une autre *méthode* elle va nous servir à *surcharger* la *méthode* de la *classe parente*.

```
1  class Auteurs:
2      def __init__(self, name, surname):
3          self.name = name
4          self.surname = surname
5
6      def affiche_texte(self):
7          print(f"{self.name} {self.surname} est un auteur.")
8
9
10 class Poetes(Auteurs):
11     def __init__(self, name, surname):
12         super().__init__(name=name, surname=surname)
```

```
13
14      def affiche_texte(self):
15          # Au lieu de Auteurs.affiche_texte(self):
16          super().affiche_texte()
17          print("Plus précisément un poète.")
18
19
20  auteur = Auteurs(name="Kerouac", surname="Jack")
21  poete = Poetes(name="Rimbaud", surname="Arthur")
22
23  poete.affiche_texte()
24  auteur.affiche_texte()
```

Listing 20.7 – Surcharge d'une *méthode* de la *classe parente*

La fonction `super()` fait bien plus que rechercher une *méthode* ou un *attribut* dans la *classe parente*. Elle parcourt toute la hiérarchie des classes à la recherche de la *méthode* ou de l'*attribut* correspondant. Si vous ne faites pas attention, la fonction `super()` peut avoir des résultats surprenants.

Sources pour la rédaction de ce chapitre

- *Object-Oriented Programming (OOP) in Python 3* (trad. *Object-Oriented Programming (OOP) in Python 3*, par David Amos, 6 juillet 2020 - `https://realpython.com/python3-object-oriented-programming/`

Chapitre 21

En mode expert

Nous allons aborder ici les notions avancées de la POO, que sont :
— le polymorphisme ;
— l'héritage multiple ;
— les méthodes statiques ;
— les modificateurs d'accès ;
— la comparaison/copie ;
— la représentation texte.

Penser de façon orientée objet implique de savoir mener un raisonnement. Quelles entités créer ? Quelle est la réalité de chacune de ces entités ? Quelles évolutions possibles concernant le projet ? Avec la POO il faut savoir que nous obtiendrons une meilleure gestion des données. La POO est aussi très utile quand notre projet inclue des interfaces graphiques.

Illustrons de suite notre propos par le codage en POO d'un questionnaire :

```
"""PROJET QUESTIONNAIRE - V3 - POO."""


###############
### Classes ###
###############

class Question:
    def __init__(self, question, choix_reponses, bonne_reponse):
        self.question = question
        self.choix_reponses = choix_reponses
        self.bonne_reponse = bonne_reponse

    @staticmethod  # Pour passer le code à partir de données brutes
    def from_data(the_data):
        quest = Question(the_data[0], the_data[1], the_data[2])
        return quest

    def poser_question(self):
        """La fonction appelée pour afficher les questions
            et si la réponse est bonne ou non."""
        print(self.question)
        # Lister les réponses possibles
        for i in range(len(self.choix_reponses)):
            print(f"{i + 1} - {self.choix_reponses[i]}")
```

```
26          reponse_user = Question.verif_saisie(1,
27                                               len(self.choix_reponses))
28          #  Vérifier la réponse et ajuster le score
29          if reponse_user == self.bonne_reponse:
30              print("Réponse correcte")
31              return True
32          print("Mauvaise réponse")
33          return False
34
35      @staticmethod
36      def verif_saisie(mini: int, maxi: int) -> int:
37          """Vérifie la saisie de l'utilisateur (de type int
38          et bien comprise dans le choix possible des réponses).
39          Utilisation d'une fonction récursive si la valeur saisie
40          est incorrecte."""
41          reponse_str = input(f"Votre choix parmi ces "
42                              f"{maxi} réponses: ")
43          try:
44              reponse_int = int(reponse_str)
45              if mini <= reponse_int <= maxi:
46                  return reponse_int
47              print(f"Erreur de saisie: la réponse doit être comprise"
48                    f" entre {mini} et {maxi}.")
49          except ValueError:
50              print("Erreur de saisie: veuillez saisir un nombre...")
51          # Rappel de la fonction
52          return Question.verif_saisie(mini, maxi)
53
54
55  class Questionnaire:
56      def __init__(self, ensemble_questions):
57          self.ensemble_questions = ensemble_questions
58
59      def lancer_questionnaire(self):
60          score = 0
61          for question in self.ensemble_questions:
62              if question.poser_question():
63              # car 'question' = 'Question(tuple)'
64                  score += 1
65          print(f"Score final = {score}/{len(self.ensemble_questions)}")
66
67
68  #############
69  ### DATAS ###
70  #############
71  # Le questionnaire sous la forme d'un tuple :
72  # le tuple = (question: str, réponses: tuple de str,
73  #             bonne réponse : int)
74  questionnaire = [
75      Question("Quelle est la capitale de l'Espagne ?",
76               ("Valence", "Séville", "Barcelone", "Madrid"), 4),
```

```
    Question("Quelle est la capitale de l'Italie ?",
             ("Venise", "Rome", "Turin", "Naples"), 2),
    Question("Quelle est la capitale de l'Allemage ?",
             ("Berlin", "Bonn", "Munich", "Frankfort"), 1),
    Question("Quelle est la capitale de la France ?",
             ("Paris", "Bordeaux", "Marseille", "Lyon"), 1),
    Question("Quelle est la capitale du Portugal ?",
             ("Porto", "Coimbra", "Lisbonne", "Viseu"), 3)
]

############
### MAIN ###
############
# A partir d'une donnée brute
print("Question de test :")
data = ("Quelle est la couleur du cheval blanc d'Henri IV ?",
        ("Blanc", "Noir"), 1)
qu = Question.from_data(data)
qu.poser_question()
print("\nC'est parti avec les capitales européennes...\n")

# Depuis un ensemble de questions
Questionnaire(questionnaire).lancer_questionnaire()
```

21.1 `isinstance` : vérifier les types

```
isinstance(var, type)
```

21.2 Polymorphisme

Le polymorphisme consiste à employer des méthodes identiques sur des objets de types différents.

```
"""Le polymorphisme en POO."""


class EtreVivant:

    def afficher_infos(self):
        print("Je suis un être vivant")


class Chat(EtreVivant):

    def afficher_infos(self):
        print("Je suis un chat")


```

```
16 class Personne(EtreVivant):
17
18     def afficher_infos(self):
19         print("Je suis une personne")
20
21
22 ma_liste = (EtreVivant(), Chat(), Personne())
23 for e in ma_liste:
24     e.afficher_infos()
```

21.3 L'héritage multiple

Se dit *inheritance* en anglais. Peu de langages de programmation permettent l'héritage multiple. Cela est par exemple possible en `C++` mais impossible en `C#`. Il s'agit d'une fonctionnalité très peu utilisée.

```
1  """Héritage multiple en POO."""
2
3
4  class EtreVivant:
5      def afficher_infos(self):
6          print("Je suis un être vivant")
7
8
9  class AnimalDomestique:
10     def mange_souris(self):
11         print("J'aime les souris")
12
13
14 class Chat(EtreVivant, AnimalDomestique):  # Héritage mutiple
15     def afficher_infos(self):
16         print("Je suis un chat")
17
18
19 chat = Chat()
20 chat.afficher_infos()
21 chat.mange_souris()
```

21.4 La comparaison d'objets

`is` va comparer les instances (les objets). `==` va comparer les valeurs.

```
1 def __eq__(self, other):  # eq = equality
2     return True
```

L'emploi du comparateur == va systématiquement retourner `True` pour les objets de la classe.

```
"""Comparaison d'objets.
'is' n'est pas '=='."""


class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def afficher_infos(self):
        print(f"Nom : {self.nom} - Age : {self.age} ans")

    def __eq__(self, other):
        if self.nom == other.nom and self.age == other.age:
            return True
        return False


pers1 = Personne("Fanny", 40)
pers1.afficher_infos()

pers2 = Personne("Fanny", 40)
pers2.afficher_infos()

print(pers1 == pers2)
print(pers1 is pers2)

# Comparaison quand on ne peut avoir accès à la méthode __eq__ :
print(pers1.__dict__ == pers2.__dict__)
```

21.5 La copie d'objets

Nécessite l'import du module `copy`. Il faut distinguer la `shallow copy` qui est une copie non profonde, de la `deep copy` qui est une copie profonde.

21.6 Affichage et représentation

```
"""'__str__' et '__repr__'."""


class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

```

```
    def afficher_infos(self):
        print(f"Nom : {self.nom} - Age : {self.age} ans")

    # def __str__(self):
    #     """Représentation sous forme d'une chaîne de caractères"""
    #     return f"{self.nom.upper()} a {self.age} ans"

    def __repr__(self):
        """Affiché si par de fontion '__str__' présente dans
        la classe.
        Représentation pour le développeur"""
        return f"{__class__.__name__} {self.__dict__}"


pers1 = Personne("Fanny", 40)
pers1.afficher_infos()

print(pers1)
```

```
print(mon_objet)
```

Cela retourne :

```
<__main__.mon_objet object at 0x7fc6bc26b730>
```

Pour modifier le comportement du `print()`, on utilise la méthode `__str__()` qui va retourner une chaîne de caractères.

Pour retourner le nom de la classe :

```
__class__.__name__
```

21.7 Méthodes d'instance, de classe et statiques

Une méthode d'instance utilise le `self` car cela nous permet d'accéder aux données de l'objet, aussi bien les variables d'instance que les variables de classe.

Une méthode statique n'utilise pas le `self`. Par contre nous allons utiliser un décorateur : `@staticmethod`.

Une méthode de classe utilise le décorateur `@classmethod` :

```
@classmethod
def methode_de_classe(cls):
    print("Méthode de classe")

# Appel:
NomClasse.methode_de_classe
```

On utilise peu les méthodes de classe. On a plus souvent recours à la méthode statique. Une méthode statique peut être appelée depuis une méthode de classe :

```
self.methode_statique(paramètre)
```

21.8 Les modificateurs d'accès

Les trois types d'accès :

public : accès depuis l'intérieur et l'extérieur de la classe ;

privé : accès uniquement depuis l'intérieur de la classe ;

protégé : accès depuis l'intérieur de la classe et les classes dérivées.

Par défaut, au sein d'une classe tout est public. On va utiliser une convention de nommage pour indiquer que nous souhaitons avoir des éléments privés ou protégés.

```
self.__nom = nom  # double underscore pour privé
self._nom = nom  # simple underscore pour protégé
```

Cinquième partie

Conserver ses données

Chapitre 22

Les fichiers texte

22.1 Les principes de l'écriture d'un fichier texte

Dans un fichier texte (*text file* en anglais) nous avons la possibilité de lire et d'écrire des objets des objets de type `str`. Le principe général (ou étapes dans le code) est le suivant :

1. Ouvrir un fichier texte
2. Lire ou écrire dans ce fichier
3. Fermer le fichier (pour garantir la fin de son utilisation)

Les mots clés associés à ce principe général sont : `open`, `write`, `read` et `close`. Rien de bien compliqué en fait.

22.2 L'ouverture d'un fichier texte

Syntaxe de l'ouverture d'un tel fichier :

```
open("nom_fichier.txt", "mode")
```

Si le fichier ne peut pas être ouvert, une `OSError` est levée. `mode` est une chaîne de caractère optionnelle qui spécifie le mode dans lequel le fichier est ouvert. La valeur par défaut est `'r'` et elle indique que le fichier doit être ouvert en lecture et en mode texte. Les modes disponibles sont :

Caractère du mode	**A quoi cela correspond**
`'r'`	Ouvre en lecture (par défaut - synonyme de `'rt'`). Si le fichier n'existe pas une exception est levée.
`'w'`	Ouverture en écriture. Écrase le contenu du fichier s'il existe ou en crée un s'il n'existe pas.
`'x'`	Ouvre le fichier si celui-ci n'existe pas et échoue s'il existe déjà (ouverture exclusive).
`'a'`	Ouverture pour écriture, et permet d'ajouter du contenu à la suite du contenu du fichier si celui-ci existe.
`'b'`	Mode binaire
`'t'`	Mode texte (par défaut)
`'+'`	ouvre en modification (lecture et écriture)

TABLE 22.1 – Les modes d'ouverture d'un fichier texte

Les options

Il est possible d'ajouter des options à la fonction `open()` :

```
open("nom_fichier.txt", "mode", options...)
```

Voici une liste des ces options :

`encoding` est le nom de l'encodage utilisé pour encoder ou décoder le fichier. Il doit seulement être utilisé en mode texte. L'encodage par défaut dépend de la plateforme, mais n'importe quel encodage de texte pris en charge par `Python` peut être utilisé.

`errors` est une chaîne facultative spécifiant comment les erreurs d'encodage et de décodage sont gérées (non utilisable en mode binaire). Voir la liste des gestionnaires d'erreur [1].

`newline` détermine la manière d'analyser les caractères de nouvelle ligne à partir du flux.

La fonction `open()` retourne un objet fichier qu'il faut alors associer à un nom de variable :

```
1 var = open("mon_fichier.txt", "w")
2 var.close()  # Fermeture du fichier
```

Ces deux lignes vont simplement créer le fichier `mon_fichier.txt`. Si nous voulons écrire une chaîne de caractères dans un fichier, voici le code correspondant :

```
1 var = open("mon_fichier.txt", "w")
2 var.write("Phrase de test")
3 var.close()  # Fermeture du fichier
```

Exemple d'une ligne de chiffres insérée dans un fichier, extraite d'une liste (le code suivant permet aussi la lecture du fichier) :

```
1 f = open("fichier_nbres.txt", "w")
2 for i in range(10):
3     f.write(f"{i + 1}\n")
4 f.close()
5
6 f_lire = open("fichier_nbres.txt", "r", encoding="utf-8")
7 texte = f_lire.read()
8 print(texte)
9 f_lire.close()
```

A noter qu'il est possible d'ouvrir plusieurs fichiers en même temps.

22.3 Lecture d'un fichier

Cela se fait à l'aide de la méthode `read()` :

1. https://docs.python.org/fr/3.10/library/codecs.html#error-handlers

```
f = open("mon_fichier.txt", "r", encoding="utf-8")
texte = f.read()
print(texte)
f.close()
```

Méthodes pour lire le contenu d'un fichier :

`read(10)` : lit les dix premiers caractères.

`readine()` : lit une seule ligne.

`readlines()` : lit toutes les lignes.

Illustrations :

- Code pour récupérer une collection avec chacune des lignes contenues dans une liste :

```
    texte = f.readlines()
    print(texte)
```

- Boucle pour lire chacune des lignes :

```
    for ligne in f:
        print(ligne, end="")
```

22.4 Erreurs : fichiers et répertoires qui existent ou pas - Création et suppressions de répertoires

Une exception `FileNotFoundError` est alors levée, qu'il est possible de gérer avec un bloc `try` / `except`.

Il y a une fonction qui permet de vérifier si un fichier existe :

```
import os.path

if os.path.exists("nom_fichier"):
    filename = os.path.join("rep", "nom_fichier")  #  rep/nom_fichier
    print(filename)  # Affiche rep/nom_fichier
    os.mkdir("nom_rep")  # Pour créer un répertoire
```

A la seconde relance, la ligne `os.mkdir("nom_rep")` va lever une exception du type `FileExists`, qui peut cependant être gérée par un bloc `try`/`except`. Nous pouvons aussi faire appel à une condition :

```
if not os.path.exists("nom_rep"):
    os.mkdir("nom-rep")
```

Pour supprimer un répertoire, c'est assez similaire, à l'aide de la méthode `os.rmdir()`, `rm` pour *Remove directory*. Une instruction du type `os.rmdir("nom_rep")` lève une exception au second passage.

```
1 if os.path.exists("nom_rep"):
2     os.rmdir("nom_rep")
```

Pour la suppression de fichier, on va utiliser la méthode `os.remove("nom_fichier")`.

Chapitre 23

Manipuler des données au format JSON

23.1 Introduction

Le format JSON (JavaScript Object Notation [1]) permet de manipuler des données structurées (liste, dictionnaire) et de les sauvegarder dans un fichier. Un des avantages de ce format est qu'il est utilisable avec tous les langages de programmation. Même s'il la norme de JSON est inspirée du langage JavaScript elle demeure cependant aisément compréhensible. De plus, sa syntaxe est très proche de celle du langage Python s'agissant des collections (cf. JSON Editor Online [2]).

Voyons rapidement un premier exemple :

```
1 import json
2
3 auteur = {"nom": "Djian", "prenom": "Philippe",
4           "titre": "Sotos", "date_pub": 1993}
5
6 with open("liste_auteurs.json", "w") as f:
7     json.dump(auteur, f, indent=4)  # Avec indentation.
```

Listing 23.1 – Créer un fichier .json

Comme nous pouvons le voir, le fichier .json (liste_auteur.json) alors généré est aisément compréhensible. On y lit finalement un dictionnaire, avec l'identation telle quelle prévue par l'option indent.

```
1 {
2     "nom": "Djian",
3     "prenom": "Philippe",
4     "titre": "Sotos",
5     "date_pub": 1993
6 }
```

Listing 23.2 – Contenu d'un fichier .json

1. https://www.json.org/json-en.html et https://fr.wikipedia.org/wiki/JavaScript_Object_Notation
2. JSON Editor Online est un outil Web permettant de manipuler des données JSON - https://jsoneditoronline.org

La norme JSON prend en charge tous les types primitifs, comme les chaînes de caractères et les nombres, ainsi que les listes et les objets imbriqués.

Python est livré avec un paquetage intégré appelé json[3] pour encoder et décoder les données JSON. Il suffit simplement de l'importer : `import json`.

Le processus d'encodage de JSON est généralement appelé sérialisation (fonction `dumps()`). Ce terme fait référence à la transformation de données en une série d'octets (d'où sérialisation) pour être stockées ou transmises sur un réseau. Naturellement, la désérialisation (fonction `loads()`) est le processus réciproque de décodage des données qui ont été stockées ou transmises dans le standard JSON. En réalité, il ne s'agit que de lecture et d'écriture. Pensez-y comme suit : l'encodage sert à écrire des données sur le disque, tandis que le décodage sert à lire des données en mémoire.

23.2 La sérialisation

En utilisant le gestionnaire de contexte de Python, vous pouvez créer un fichier appelé JSON (comme vu précédemment page 141, avec le fichier `liste_auteur.json`) et l'ouvrir en mode écriture. Les fichiers JSON se terminent commodément par une l'extension `.json`.

```
1 with open("liste_auteurs.json", "w") as f:
```

Listing 23.3 – Créer un fichier `.json`

Noter que la méthode `.dump()` prend deux arguments positionnels : l'objet de données à sérialiser, et l'objet de type fichier dans lequel ces données seront écrites.

```
1     json.dump(auteur, f)
```

Listing 23.4 – Ecriture des données

Mais insérées de façon aussi brute les données seront difficilement lisibles. Nous pouvons donc utiliser l'argument `indent` qui va nous permettre de mieux les présenter (comme vu dans précédemment dans le fichier `.json`). `indent` permet de spécifier la taille de l'indentation pour les structures imbriquées.

```
1     json.dump(auteur, f, indent=4)
```

Listing 23.5 – Ecriture de données avec indentation

23.3 Désérialisation

Techniquement, cette conversion n'est pas l'inverse parfait de la sérialisation. Cela signifie essentiellement que si vous encodez un objet maintenant et que vous le décodez à nouveau plus tard, vous ne récupérerez peut-être pas exactement le même objet.

Exemple :

3. https://docs.python.org/fr/3/library/json.html

```python
import json

auteurs = (
    {"auteur1": {
        "nom": "Djian", "prenom": "Philippe",
        "titre": "Sotos", "date_pub": 1993}},
    {"auteur2": {
        "nom": "Kerouac", "prenom": "Jack",
        "titre": "Les Anges Vaganbonds", "date_pub": 1957}}
)

# Etape 1 : Sérialisation
with open("data_auteurs.json", "w") as f:
    json.dump(auteurs, f, indent=4)

# Etape 2 : Désérialisation
with open("data_auteurs.json", "r") as f:
    datas_auteurs = json.load(f)

# Etape 3 : Affichage des données désérialisées
print(datas_auteurs)
```

Listing 23.6 – Sérialisation et désérialisation

Le fichier `data_auteurs.json` généré :

```json
[
    {
        "auteur1": {
            "nom": "Djian",
            "prenom": "Philippe",
            "titre": "Sotos",
            "date_pub": 1993
        }
    },
    {
        "auteur2": {
            "nom": "Kerouac",
            "prenom": "Jack",
            "titre": "Les Anges Vaganbonds",
            "date_pub": 1957
        }
    }
]
```

Listing 23.7 – Le fichier `.json` obtenu grâce à la sérialisation

Et la sortie obtenue par la fonction `print()` :

```
[{'auteur1': {'nom': 'Djian', 'prenom': 'Philippe', 'titre': 'Sotos',
'date_pub': 1993}}, {'auteur2': {'nom': 'Kerouac', 'prenom': 'Jack',
```

```
'titre': 'Les Anges Vaganbonds', 'date_pub': 1957}}]
```

Noter qu'au départ, avant la sérialisation, nous avions un *tuple* contenant deux dictionnaires. Par la suite nous obtenons une liste contenant deux dictionnaires.

Pour la désérialisation le fichier est ouvert en lecture.

23.4 Ajouter des données à un fichier JSON

A la suite du script de désérialisation page 143 nous pouvons ajouter les lignes qui suivent afin d'ajouter des données au fichier `data_auteurs.json` existant. Il s'agira tout d'abord de récupérer le contenu du fichier `.json`, de modifier ce contenu et d'écrire ce nouveau contenu dans le fichier `.json` en écrasant le contenu précédent. Nous devons procéder ainsi car nous ne pouvons directement modifier les données contenus dans un fichier `.json`.

```
1 # Etapes suivantes : Ajouter des données
2 nouvel_auteur = {"auteur3": {
3     "nom": "Hemingway", "prenom": "Ernest",
4     "titre": "Pour Qui Sonne Le Glas", "date_pub": 1940}}
5 datas_auteurs.append(nouvel_auteur)
6 with open("data_auteurs.json", "w") as f:
7     json.dump(datas_auteurs, f, indent=4)
```

Listing 23.8 – Ajouter des données à un fichier `.json` existant.

Contenu du nouveau fichier `data_auteurs.json` :

```
1  [
2      {
3          "auteur1": {
4              "nom": "Djian",
5              "prenom": "Philippe",
6              "titre": "Sotos",
7              "date_pub": 1993
8          }
9      },
10     {
11         "auteur2": {
12             "nom": "Kerouac",
13             "prenom": "Jack",
14             "titre": "Les Anges Vaganbonds",
15             "date_pub": 1957
16         }
17     },
18     {
19         "auteur3": {
20             "nom": "Hemingway",
21             "prenom": "Ernest",
22             "titre": "Pour Qui Sonne Le Glas",
23             "date_pub": 1940
24         }
```

```
25      }
26  ]
```

Listing 23.9 – Contenu du fichier `data_auteurs.json` augmenté des nouvelles données.

23.5 Quelques erreurs à éviter

Une des erreurs à éviter avec les fichiers `.json` est de tenter de lire un fichier sans contenu. En effet, une exception de type `JSONDecodeError` sera levée. Pour éviter cela il faut s'assurer que le fichier `.json` contienne au moins une liste vide (`[]`) ou une chaîne de caractères vide (`""`).

```
1  import json
2
3  with open("mon_fichier.json", "w") as fichier:
4      json.dump([], fichier)
```

Listing 23.10 – Créer un fichier `.json` contenant une liste vide.

Pour ce qui est des caractères accentués nous pouvons utiliser l'argument `ensure_ascii` défini à `False`.

```
1  json.dump("élève", f, ensure_ascii=False)
```

Listing 23.11 – L'argument `ensure_ascii`.

23.6 Les bases de données au format `JSON` avec `TinyDB`

`TinyDB`[4] est une bibliothèque qui ne pèse pas beaucoup sur le disque (1800 lignes de code dont 40% de documentation), et écrite à 100% en `Python`. C'est une bibliothèque relativement simple d'utilisation. Cependant pour des projets importants, qui ont besoin de performances élevées, la bibliothèque `TinyDB` n'est pas recommandée, et également s'il y a besoin de créer des relations entre les modèles de données.

Pour utiliser cette bibliothèque, il va falloir l'installer :

```
$ pip3 install tinydb
```

Puis pour l'import :

```
1  # Import classique de la classe 'TinyDB' :
2  from tinydb import TinyDB
3  # Import pour créer une base de données uniquement en mémoire :
4  from tinydb.storages import MemoryStorage
5
6  # Pour une base de données en mémoire :
7  db = TinyDB(Storage=MemoryStorage)
```

Listing 23.12 – Import de `TinyDB`

4. `https://tinydb.readthedocs.io/en/latest/index.html`

Un premier exemple simple à comprendre

```python
from tinydb import TinyDB

# Création d'une instance de notre base de données :
db_book = TinyDB('data.json', indent=4)  # Fichier sera créé
                                         # dans le répertoire du
                                         # projet

# Insertion d'une donnée sous forme d'un dictionnaire :
db_book.insert({'Name': 'Djian', 'book': "Sotos"})

# Insertion de pluseurs données en une seule fois
# (ici, une liste de dictionnaires)
db_book.insert_multiple([
    {'Name': 'Hemingway', 'book': "Pour Qui Sonne Le Glas"},
    {'Name': 'Kerouac', 'book': "Sur La Route"}
])
```

Listing 23.13 – Le fichier `Python`

```json
{
    "_default": {
        "1": {
            "Name": "Djian",
            "book": "Sotos"
        },
        "2": {
            "Name": "Hemingway",
            "book": "Pour Qui Sonne Le Glas"
        },
        "3": {
            "Name": "Kerouac",
            "book": "Sur La Route"
        }
    }
}
```

Listing 23.14 – Le fichier `.json` généré

`"_default"` est le nom par défaut de notre dictionnaire. `"1"`, `"2"` et `"3"` sont des identifiants gérés automatiquement.

Effectuer des recherches dans la base de données

```python
kerouac = db_book.search(auteur.Name == "Kerouac")
print(kerouac)  # Obtenir une liste qui contient un dictionnaire
kerouac_dico_seul = db_book.get(auteur.Name == "Kerouac")
```

```
print(kerouac_dico_seul)  # Uniquement le dictionnaire

# Titres à partir de 'Q' comme première lettre
titre_apres_Q = db_book.search(where('book') > "Q")
print(titre_apres_Q)
# Auteurs à partir de 'I' comme première lette > 'I'
auteur_apres_i = db_book.search(where('Name') > "I")
print(auteur_apres_i)

print(f"Il a {len(db_book)} éléments dans la base de données.")
# Vérifier qu'un élément est dans la base (retourne un booléen):
print(db_book.contains(auteur.Name == "Rimbaud"))
# Compter le nombre d'éléments :
print(db_book.count(auteur.Name == "Djian"))
```

Listing 23.15 – Quelques recherches dans la base de données

Mettre à jour et supprimer des éléments

```
# Modifier le titre d'un livre :
db_book.update({"book": "Iles à la dérive"},
               where("name")=="Hemingway")

# Ajouter une nouvelle clé à tous les élements de la base de
# données :
db_book.update({"auteur": ["Romancier"]})

# Méthode pour insérer des nouvelles données :
db_book.insert()

# Méthode pour supprimer des données :
db_book.remove()

# Méthode pour supprimer toutes les données :
db_book.truncate()
```

Listing 23.16 – Modifier la base de données

Utiliser plusieurs tables

Nous allons pour cela utiliser la méthode `.table("Nom_table"` :

```
table1 = db.table("Nom1")
table2 = db.table("Nom2")
```

Listing 23.17 – Créer plusieurs tables

Et par la suite on utilise les variables `table1` et `table2` avec les méthodes d'insertion, de modification, de suppression, etc.

Sources pour la rédaction de ce chapitre

- *Working With JSON Data in Python* (trad. *Travailler avec des données JSON en Python*), par Lucas Lofaro , 16 avril 2018 - `https://realpython.com/python-json/`
- *`TinyDB: une base de données adaptée à vos besoins`*, par Thibault Houdon - `https://www.docstring.fr/blog/tinydb-une-base-de-donnees-adaptee-vos-projets/`

Chapitre 24

Les bases de données

24.1 Introduction

Les bases de données permettent de manipuler (stocker, organiser, accéder) une large quantité de données. Les valeurs y sont associées à des noms. L'accès à ces données est relativement puissante, et se réalise à l'aide de requêtes et de filtres. Il est alors possible à plusieurs utilisateurs d'y accéder simultanément quand les données sont stockées sur des serveurs.

Le principe des bases de données est d'organiser les données sous forme de tables. Les associations seront réalisées grâce à des identifiants uniques, et par la création d'une relation. Il existe divers types de relations :

One-to-many : un auteur a écrit plusieurs livres, mais chaque livre ne peut être associé qu'à un seul auteur.

Many-to-Many : un étudiant peut s'inscrire à plusieurs cours et chaque cours est suivi par plusieurs étudiants.

One-to-one : une personne a un identifiant unique (Numéro de Sécurité Sociale, IBAN) et chaque identifiant est unique.

24.2 Le langage `SQL`

Nous allons manipuler des bases des données à l'aide du module `sqlite` qui utilise le langage `SQL`. Ce n'est pas un langage de programmation mais il sert à créer des bases de données et à adresser des requêtes dans cette même base de données.

Prenons l'exemple d'une base de données simple, contenant un auteur, un ouvrage, une date de publication. Nous pourrions créer les tables suivantes :

```
auteur
- auteur_id (un identifiant)
- nom

ouvrage
- ouvrage_id
- auteur_id
- titre
- date_publication
```

Les tables ne sont créées qu'une seule fois. La commande `SQL` qui permet de créer ces tables est `CREATE TABLE nom_de_la_table`, accompagnées des renseignements de type :

```
INTEGER: int
VARCHAR: str
NOT NULL: Ne peut être un champ vide
```

Le code en Python serait alors le suivant :

```
CREATE TABLE auteur (
    auteur_id INTEGER NOT NULL PRIMARY KEY,
    nom VARCHAR);

CREATE TABLE ouvrage (
    ouvrage_id INTEGER NOT NULL PRIMARY KEY,
    auteur_id INTEGER REFERENCES auteur,
    titre VARCHAR,
    date_publication INTEGER);
```

L'insertion des données dans la table se réaliser à l'aide de l'instruction `INSERT INTO` (A noter que les `id` sont ajoutés automatiquement).

```
INSERT INTO auteur (nom) VALUES ("DJIAN")
INSERT INTO ouvrage (titre, date_publication) VALUES ("Sotos", 1992)
```

Il est ensuite possible de modifier ces données (de les mettre à jour si l'on préfère) :

```
# Modification de la date de publication pour l'ouvrage "Sotos"
UPDATE ouvrage SET date_publication = 1991 WHERE titre = "Sotos"
```

Nous pouvons également supprimer une entrée de la base de données :

```
# Suppression de "DJIAN" de la table "auteur"
DELETE FROM auteur WHERE nom = "DJIAN"
```

Enfin, il est possible de récupérer ces données afin de les stocker dans des variables ou des fichiers :

```
# Récupérer toutes les données de la table "auteur":
SELECT * FROM auteur
#  Récupérer toutes les valeurs du champs "nom" de la table "auteur":
SELECT nom FROM auteur
# récupérer toutes les données des albums sortis après 1990:
SELECT * FROM ouvrage WHERE date_publication > 1990
# A partir de diverses associations:
SELECT nom, titre FROM album a JOIN auteur ar ON a.auteur_id =
ar.auteur_id
SELECT titre FROM album a JOIN auteur ar ON a.auteur_id =
ar.auteur_id AND ar.nom = "HEMINGWAY"
```

24.3 Visualiser des données à l'aide du logiciel `sqlitebrowser`

Son installation :

```
# aptitude install sqlitebrowser
```

`sqlitebrowser`[1] est un logiciel qui permet de créer des bases de données, de les lire et de les modifier (écriture), mais aussi d'exécuter des commandes SQL. Voici comment procéder :

1. Ouvrir le logiciel
2. Cliquer sur « `Nouvelle base de données` »
3. Sauvegarder avec l'extension `.db` (Ex : `ouvrage.db`)
4. Cliquer sur « `Exécuter le SQL` »
5. Copier les codes ci-dessus
6. Cliquer sur « `Exécuter Tout ou seulement le SQL sélectionné` » (bouton « `play` »)
7. Vérifier dans « `Structure de la base de données` » puis « `Parcourir les données` »
8. Insertion de données en copiant/collant le code ci-dessus dans « `Exécuter le SQL` »

24.4 Créer une base `SQLite` avec `Python`

Les étapes

1. Importer le module `sqlite`.
2. Créer une connexion avec la base de données.
3. Exécuter les différentes requêtes `SQL` via un objet que l'on nomme un *curseur*.
4. *Commiter* pour s'assurer que les données soient bien enregistrées dans la base de données.
5. Fermer la base de données.

Exemples de scripts

Tout d'abord un premier script qui va nous permettre de créer notre base de données et d'y insérer des données à l'aide de requêtes `SQL` :

```
1  import sqlite3
2
3  # Connexxion
4  ma_connexion = sqlite3.connect("biblio.db")
5
6  # Exécution requête
7  mon_curseur = ma_connexion.cursor()
8  # Ajout des tables "auteur" et "ouvrage"
9  mon_curseur.execute("""CREATE TABLE auteur (
10     auteur_id INTEGER NOT NULL PRIMARY KEY,
11     nom VARCHAR);""")
12 mon_curseur.execute("""CREATE TABLE ouvrage (
13     ouvrage_id INTEGER NOT NULL PRIMARY KEY,
```

1. `https://sqlitebrowser.org` - Page de documentation : `https://github.com/sqlitebrowser/sqlitebrowser/wiki`

```
    auteur_id INTEGER REFERNCES auteur,
    titre VARCHAR);""")

# Ajout de trois auteurs
mon_curseur.execute("""INSERT INTO auteur (nom) VALUES ("DJIAN");""")
djian = mon_curseur.lastrowid  # Récupère le dernier id
mon_curseur.execute("""INSERT INTO auteur (nom) VALUES """
                    """("KEROUAC");""")
kerouac = mon_curseur.lastrowid
mon_curseur.execute("""INSERT INTO auteur (nom) VALUES """
                    """("HEMINGWAY");""")
hemingway = mon_curseur.lastrowid
# Ajout de quatre ouvrages, dont deux du même auteur
mon_curseur.execute(f"""INSERT INTO ouvrage (auteur_id, titre) """
                    f"""( VALUES {djian}, "Sotos");""")
mon_curseur.execute(f"""INSERT INTO ouvrage (auteur_id, titre) """
                    f"""VALUES ({hemingway}, """
                    f"""'Pour qui sonne le glas');""")
mon_curseur.execute(f"""INSERT INTO ouvrage (auteur_id, titre) """
                    f"""VALUES ({kerouac}, "Sur la route");""")
mon_curseur.execute(f"""INSERT INTO ouvrage (auteur_id, titre) """
                    f"""VALUES ({djian}, "Zone érogène");""")
# commit
ma_connexion.commit()
ma_connexion.close()
```

Puis nous réalisons un second script (les deux scripts doivent figurer dans le même projet) qui va nous permettre de consulter la base de données, toujours à l'aide de requêtes `SQL` :

```
import sqlite3

# Connexxion
ma_connexion = sqlite3.connect("biblio.db")

# Exécution requête
mon_curseur = ma_connexion.cursor()
# Récupération de la lite des auteurs :
# méthode 1
#mon_curseur.execute("""SELECT * FROM auteur""")
#les_auteurs = mon_curseur.fetchall()  # Liste dans une variable
#print(les_auteurs)
# méthode 2
for chaque_auteur in mon_curseur.execute("""SELECT * FROM auteur"""):
    print(chaque_auteur)

ma_connexion.close()
```

Chapitre 25

Archiver ses fichiers

25.1 Compresser des fichiers au format `.zip`

Étudions le code suivant :

```
1  import zipfile
2
3
4  # Créer un fichier .zip
5  mon_zip = zipfile.ZipFile("nom_archive.zip", "w",
6                            zipfile.ZIP_DEFLATED)
7  # Insérer les fichiers
8  mon_zip.write("fichier_01.ods")
9  mon_zip.write("fichier_02.ods")
10 mon_zip.write("fichier_03.ods")
11 # Fermeture du fichier .zip
12 mon_zip.close
```

1. On importe le module `zipfile` qui permet de travailler avec les fichiers `.zip`.
2. Ligne 5 on crée le fichier `.zip`. Les options `w` et `zipfile.ZIP_DEFLATED` permettent réciproquement l'écriture et la compression (par défaut les fichiers sont juste insérés dans l'archive sans compression).
3. On insère ensuite les fichiers.
4. On ferme ensuite le fichier.

Chapitre 26

La gestion des chemins des fichiers et des répertoires

26.1 Les modules `os.path`, `shutil` et `glob`

Classiquement quand on parle de gestion de fichiers avec `Python` on évoque d'emblée ces trois modules :

Le module `os.path` : Ce module fournit un certain nombre de fonctions qui permettent de gérer et manipuler les fichiers et répertoires du présents dans le système de fichiers (concaténations des chemins, création de fichiers et répertoires, modifications, etc.) [1].

Le module `shutil` : Ce module shutil fournit des fonctions de haut niveau pour travailler avec les systèmes de fichiers. Il est construit à partir de fonctionnalités que l'on retrouve dans le module `os` et fournit des fonctions supplémentaires pour copier, déplacer, supprimer et travailler avec des fichiers et des répertoires [2].

Le module `glob` : Ce module fournit des fonctions pour trouver (en scannant les systèmes de fichiers) des fichiers et des répertoires à l'aide de motifs et de noms, ce qui est particulièrement utile pour travailler avec des ensembles de fichiers volumineux ou pour automatiser des tâches de manipulation de fichiers [3].

La difficulté avec ces trois modules est qu'il a falloir que l'on fasse appel à des chaînes de caractères, ce qui peut rapidement alourdir le code et accentuer les possibilités d'erreurs. Et c'est pour ces raisons que nous allons plutôt faire appel au module `pathlib` qui va nous permettre de manipuler des objets en lieu et place de chaînes de caractères, objets sur lesquels nous emploierons des méthodes beaucoup plus intuitives que la kyrielle de fonctions qui compose le module `os.path`.

26.2 Présentation du module `pathlib`

Ce module [4], implémenté dans la bibliothèque standard depuis la version `3.4` de `Python`, propose une hiérarchie simple de classes permettant la représentation d'un système de fichiers, afin de réaliser diverses opérations sur les chemins de ces systèmes de fichiers, en usant d'une sémantique appropriée pour chaque système d'exploitation.

1. Diverses interfaces pour le système d'exploitation - `https://docs.python.org/fr/3/library/os.path.html`
2. Opérations de haut niveau sur les fichiers - `https://docs.python.org/fr/3/library/shutil.html`
3. Recherche de chemins de style Unix selon certains motifs - `https://docs.python.org/fr/3/library/glob.html`
4. Code source : `https://github.com/python/cpython/blob/3.11/Lib/pathlib.py`

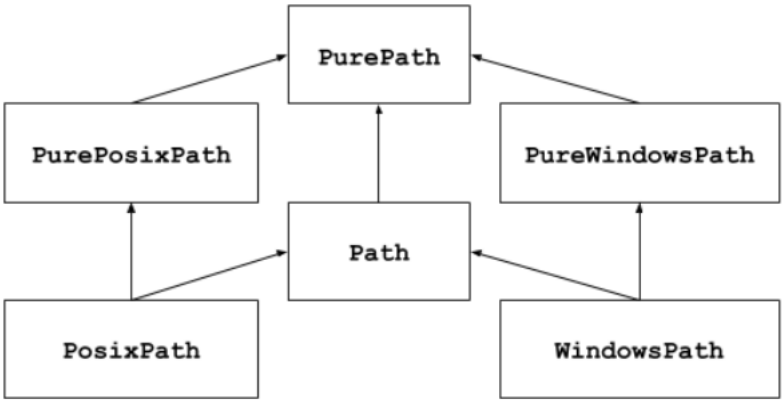

FIGURE 26.1 – Hiérarchie des classes du module `pathlib`

Le principe est le suivant : on crée un objet de type *chemin* avec lequel nous disposons de diverses méthodes cohérentes et utiles nous permettant d'interagir avec cet objet. Le raisonnement clé du module est le fait de ne pas hériter de la classe `str` afin d'éviter tout potentiel incident lié à des opérations avec des chaînes de caractères représentant un chemin. Les objets `Path` sont des objets immuables. De par sa philosophie le module `pathlib` se place à l'opposée d'`os.path` et de ses multiples fonctions.

Illustrons cela avec un exemple aisément compréhensible :

```
>>> from pathlib import Path
>>> p = Path("/Chemin/vers/fichier.py")  # Instanciation de l'objet
>>> p. parent  # Récupération du répertoire parent
PosixPath('/Chemin/vers')
>>> p.suffix  # Récupération de l'extension
'.py'
```

26.3 La classe `Path`

Héritière de la classe `PurePath`, c'est la classe que nous utilisons le plus fréquemment, et c'est d'ailleurs pour cela qu'elle porte le nom le plus court. La classe `Path` permet de créer un objet représentant un chemin qui pointe sur un fichier ou un répertoire [5]. Un tel objet est un objet de type `PosixPath` qui représente les chemins ds systèmes *GNU/Linux* et *MacOS*. Sur les systèmes *Windows* sera créé un objet de type `WindowsPath`. A noter qu'il n'est pas possible d'instancier un objet `PosixPath` sur un objet `WindowsPath` et vice-et-versa.

Il nous faut tout d'abord importer cette classe :

```
1 from pathlib import Path
```

Listing 26.1 – Import de la classe `Path`

Illustrons par un exemple simple :

5. En fait, elle permet d'instancier un chemin concret de fichier ou de répertoire, présent ou non sur le système.

```python
from pathlib import Path

user_dir = Path.home()  # Répertoire de l'utilisateur
current_dir = Path.cwd()  # Répertoire courant

# Création d'un objet :
way = Path("/Chemin/Vers/Fichier.py")
way_p = way.parent  # Retourne le répertoire parent

print(f"Répertoire de l'utilisateur : {user_dir}")
print(f"Répertoire courant : {current_dir}")
print(f"Répertoire parent de l'objet '{way}' : {way_p}")
```

Listing 26.2 – Un premier script à l'aide du module `pathlib`

La sortie de ce script est alors la suivante :

```
Répertoire de l'utilisateur : /home/krystof
Répertoire courant : /home/krystof/Documents/DEVELOPPEMENT/Python/Module_
    pathlib/Presentation
Répertoire parent de l'objet '/Chemin/Vers/Fichier.py' : /Chemin/Vers
```

26.4 Concaténer des chemins

Pour la concaténation nous n'avons pas à nous soucier, avec les objets récupérés, du sens des *slashs* et *antislashs* (Les barres obliques / et \) puisque le système d'exploitation utilisé sera reconnu.

```
>>> from pathlib import Path
>>> p = Path.home()
>>> p / "New_dir"
PosixPath('/home/krystof/New_dir')
>>> p / "New_dir" / "fichier.py"  # Concaténation mutliple
PosixPath('/home/krystof/New_dir/fichier.py')
```

Il est aussi possible d'user de la méthode `.joinpath()` :

```
>>> p.joinpath("New_dir", "fichier.py")
PosixPath('/home/krystof/New_dir/fichier.py')
>>> p.joinpath("New_dir", "fichier.py").suffix
'.py'
```

Avec la méthode `.joinpath()` et à l'aide du symbole de dépaquetage (*) nous pouvons réaliser une concaténation via une liste de noms :

```
>>> p = Path.home()
>>> reps_et_sous_reps = ['Projets', 'Bibpy', 'Files']
>>> p.joinpath(*reps_et_sous_reps)
```

L'objet en sortie sera alors :

```
PosixPath('/home/krystof/Projet/Bibpy/Files']
```

26.5 Récupérer des informations sur un chemin

Illustrons cela par un petit script :

```
1  from pathlib import Path
2
3  p = Path.home()
4  chemin = p / "Dir" / "main.py"
5  chemin_complet = chemin.parent / chemin.name
6  print(f"Nom du fichier : {chemin.name}")
7  print(f"Nom du fichier (sans l'extension) : {chemin.stem}")
8  print(f"Extension du fichier : {chemin.suffix}")
9  print(f"Découpage du chemin : {chemin.parts}")
10 print(f"Vérification de l'existence su fichier : "
11       f"{chemin.exists()}")
12 print(f"Le chemin pointe-t-il vers un répertroire ? "
13       f"{chemin.is_dir()}")
14 print(f"Le chemin pointe-t-il vers un fichier ? "
15       f"{chemin.is_file()}")
16 print(f"Répertoire parent: {chemin.parent}")
17 print(f"Chemin complet: {chemin_complet}")
```

Listing 26.3 – Script avec recueils d'informations

Ce qui nous donne en sortie :

```
Nom du fichier : main.py
Nom du fichier (sans l'extension) : main
Extension du fichier : .py
Découpage du chemin : ('/', 'home', 'krystof', 'Dir', 'main.py')
Vérification de l'existence su fichier : False
Le chemin pointe-t-il vers un répertroire ? False
Le chemin pointe-t-il vers un fichier ? False
Répertoire parent: /home/krystof/Dir
Chemin complet: /home/krystof/Dir/main.py
```

Il existe la méthode `.suffixes` qui nous permet d'obtenir toutes les extensions d'un fichier, comme par exemple pour un fichier du type `archive.tar.gz`. Ce qui nous donnerait en sortie la liste suivante : `['.tar', '.gz']`.

26.6 Créer et supprimer des répertoires

Nous allons tout d'abord créer un répertoire dans lequel nous pourrons réaliser divers tests, puis à la fin de cette sections nous supprimerons ce répertoire ainsi que tout son contenu. Créons ce répertoire sur notre *Bureau* à l'aide du script suivant :

```
1  from pathlib import Path
2
3  test_rep = Path.home() / "Bureau" / "TestRep"
4  test_rep.mkdir()  # Méthode .mkdir() pour créer le répertoire
```

Listing 26.4 – Créer un répertoire sur le bureau de l'utilisateur

Si le répertoire existe déjà, la méthode `.mkdir()` va lever une exception de type `FileExistsError` (relancer le script ci-dessus pour le vérifier). Pour palier à cette levée d'exception il est nécessaire d'utiliser le paramètre `exist_ok` ayant pour valeur `True` :

```
1 test_rep.mkdir(exist_ok=True)
```

Listing 26.5 – Avec `exist_ok=True`

Il est possible de créer un répertoire contenant une arborescence de répertoires. Ajouter les lignes suivantes au script ci-dessus :

```
1 test_rep = test_rep / "SousRep1" / "SousRep2" / "SousRep3"
2 test_rep.mkdir(exist_ok=True, parents=True)
3 print(test_rep)
```

Listing 26.6 – Créer une arborescence de répertoires

La ligne `print(test_rep` vient afficher l'arborescence alors créée :

```
/home/krystof/Bureau/TestRep/SousRep1/SousRep2/SousRep3
```

La création d'un fichier obéit au même principe mais avec la méthode `.touch()`. Ajouter les lignes suivantes au script :

```
1 # Créer un fichier dans le répertoire 'SousRep3'
2 fichier = test_rep / "README.txt"
3 fichier.touch()
```

Listing 26.7 – Créer un fichier

Pour supprimer un fichier il nous faut utiliser la méthode `.unlink()`. Enfin, pour supprimer un répertoire nous disposons de la méthode `.rmdir()`. Cependant, si le répertoire contient des éléments, l'emploi de cette méthode va lever une exception de type `OSError`. Pour supprimer un répertoire avec l'ensemble de son contenu nous devons faire appel au module `shutil`. Ajoutons ces lignes à la fin de notre script, en s'assurant au préalable de l'import du module :

```
1  import shutil
2
3  # ... code du script ...
4
5  # Suppression ou pas
6  o_n, supprime = ["o", "n"], ""
7  while supprime not in o_n:
8      supprime = input("Voulez-vous supprimer le répertoire de "
9                       "test créé précédemment (o/n) : ").lower()
10     if supprime == "o":
11         shutil.rmtree(Path.home() / "Bureau" / "TestRep")
```

Listing 26.8 – La méthode `.rmtree()` du module `shutil`

26.7 Lire et écrire dans un fichier

Avec le module `pathlib` il est possible d'écrire dans un fichier et de lire son contenu sans en passer par un *context manager* avec lequel nous ouvrons et fermons classiquement un fichier. Voyons cela par le script ci-dessous :

```
from pathlib import Path


def oui_ou_non(question: str) -> bool:
    """L'utilisateur doit saisir 'o' ou 'n'."""
    o_n, reponse = ["o", "n"], ""
    while reponse not in o_n:
        reponse = input(f"{question}(o/n) ").lower()
        if reponse not in o_n:
            print("Veuillez répondre par 'o' ou 'n'...")
            return oui_ou_non(question)
        elif reponse == "n":
            return False
    return True


# Création du fichier test :
file_text = Path.home() / "Bureau" / "file_test.txt"
file_text.touch(exist_ok=True)

# Ecrire dans fichier
if oui_ou_non(f"Voulez-vous écrire dans le fichier "
              f"{file_text.name} ? "):
    # Méthode .write_text(texte_à_insérer) :
    file_text.write_text(input("Veuillez saisir le texte à "
                               "saisir : "))
if oui_ou_non(f"Voulez-vous supprimer le fichier "
              f"{file_text.name} ?"):
    file_text.unlink()  # Méthode de suppression d'un fichier
if file_text.exists():  # Si le fichier existe
    # Méthode de lecture du contenu du fichier.
    print(f"Texte du fichier : {file_text.read_text()}")
```

Listing 26.9 – Ecriture dans un fichier de type `.txt`

26.8 Scanner un répertoire

Voyons comment récupérer tous les répertoires et fichiers, contenus dans un répertoire :

```
from pathlib import Path

for f in Path.home().iterdir():  # Répertoire de l'utilisateur
    print(f.name)
```

Listing 26.10 – La méthode `.iterdir()`

L'emploi de la fonction `.iterdir()` retourne un objet de type *générateur* (*generator*) sur lequel il est possible d'itérer.

Le script ci-dessus nous affiche l'ensemble des fichiers et répertoires, y compris ceux cachés (préfixés par un point), contenus dans le répertoire sur lequel la méthode `.iterdir()` est appliquée :

```
Téléchargements
.gitconfig
.cinnamon
Hasard.py
.cache
.vscode
.profile
.mozilla
.ipython
.var
.bashrc
.python_history
.virtualenvs
.java
.themes
.pyenv
Images
...etc
```

A l'aide de compréhensions de listes nous pouvons affiner les recherches :

```
1 # Retourne la liste des répertoires :
2 print([f for f in Path.home().iterdir() if f.is_dir()])
3 # Retourne la liste des fichiers :
4 print([f for f in Path.home().iterdir() if f.is_file()])
```

Listing 26.11 – La méthode `.iterdir()` avec des compréhensions de liste

Effectuer une recherche selon les types de fichiers avec la méthode `.glob(motif)`, où `motif` représente le type de fichiers recherchés :

```
1 # Affiche l'ensemble des fichiers de type .png :
2 for f in Path.Home().glob("*.png"):
3     print(f.name)
```

Listing 26.12 – La méthode `.glob(motif)`

Il existe aussi la méthode `.rglob(motif)` qui permet la récursivité, et qui va donc effectuer la recherche dans l'ensemble des sous-répertoires.

26.9 Quelques exemples concrets de l'utilisation du module pathlib

Modifier le nom d'un fichier

```
>>> from pathlib import Path
>>> file_img = Path.home() / "Images" / "5452.jpg"
>>> file_img.parent / (file_img.stem + "-Photo_id" + file_img.suffix)
PosixPath('/home/krystof/Images/5452-Photo_id.jpg')
```

A partie de là nous pouvons imaginer un script qui permettrait de renommer en masse des fichiers...

Trier des fichiers selon l'extension

Le script qui suit permet de trier les fichiers présents dans le répertoire `Téléchargements` de l'utilisateur.

```
"""Tri des fichiers selon leur extension."""

from pathlib import Path  # Import de la classe 'Path'

dirs = {  # "Extensions": "Répertoire cible"
    ".png": "Images",
    ".jpeg": "Images",
    ".jpg": "Images",
    ".gif": "Images",
    ".pdf": "Documents_texte",
    ".docx": "Documents_texte",
    ".txt": "Documents_texte",
    ".odt": "Documents_texte",
    ".ods": "Documents_texte",
    ".deb": "Archives_Deb",
    ".zip": "Archives",
    ".gz": "Archives",
    ".xz": "Archives",
    ".tgz": "Archives",
    ".rpm": "Archives",
    ".gpx": "Divers",
    ".xlsx": "Divers",
    ".exe": "Divers",
    ".tex": "Divers",
    ".gpg": "Divers",
    ".ics": "Divers",
    ".eml": "Divers",
    ".py": "Fichiers_Python",
    ".iso": "Fichiers_ISO"
}

rep_a_trier = Path.home() / "Téléchargements"
# Récupérer l'ensemble des chemins qui correspondent uniquement
# à des fichiers:
```

```
all_files = [f for f in rep_a_trier.iterdir() if f.is_file()]
# Boucle pour traiter chacun des fichiers de la liste:
for f in all_files:
    rep_de_redirection = rep_a_trier / dirs.get(f.suffix,
                                                "Extentions_autres")
    rep_de_redirection.mkdir(exist_ok=True)
    # Déplacement du fichier :
    f.rename(rep_de_redirection / f.name)
```

Listing 26.13 – Trier des fichiers

26.10 Les méthodes de la classe `Path`

Pour connaître l'ensemble des méthodes applicables avec la classe `Path` :

```
>>> dir(Path)
['__bytes__', '__class__', '__delattr__', '__dir__', '__doc__',
'__enter__', '__eq__', '__exit__', '__format__', '__fspath__', '__ge__',
'__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rtruediv__',
'__setattr__', '__sizeof__', '__slots__', '__str__', '__subclasshook__',
'__truediv__', '_cached_cparts', '_cparts', '_drv',
'_format_parsed_parts', '_from_parsed_parts', '_from_parts', '_hash',
'_make_child', '_make_child_relpath', '_parse_args', '_parts', '_pparts',
'_root', '_scandir', '_str', 'absolute', 'anchor', 'as_posix', 'as_uri',
'chmod', 'cwd', 'drive', 'exists', 'expanduser', 'glob', 'group',
'hardlink_to', 'home', 'is_absolute', 'is_block_device',
'is_char_device', 'is_dir', 'is_fifo', 'is_file', 'is_mount',
'is_relative_to', 'is_reserved', 'is_socket', 'is_symlink', 'iterdir',
'joinpath', 'lchmod', 'link_to', 'lstat', 'match', 'mkdir', 'name',
'open', 'owner', 'parent', 'parents', 'parts', 'read_bytes',
'read_text', 'readlink', 'relative_to', 'rename', 'replace', 'resolve',
'rglob', 'rmdir', 'root', 'samefile', 'stat', 'stem', 'suffix',
'suffixes', 'symlink_to', 'touch', 'unlink', 'with_name', 'with_stem',
'with_suffix', 'write_bytes', 'write_text']
```

Pour un descriptif de certaines de ces méthodes, voir les tableaux des pages 165 et 166.

Pour un tableau des correspondances avec les outils du module `os` on se reportera à l'adresse suivante :
`https://docs.python.org/fr/3/library/pathlib.html#correspondance-to-tools-in-the-os-module`

26.11 Sources pour la rédaction de ce chapitre

- *Gérer des chemins de fichiers avec* ***pathlib***, par Thibault Houdon - `https://www.docstring.fr/blog/gere-des-chemins-de-fichiers-avec-pathlib`
- ***pathlib*** *- Chemins de système de fichiers orientés objet* - `https://docs.python.org/fr/3/library/pathlib.html`

- *PEP 428 – The `pathlib` module – object-oriented filesystem paths* (trad. *PEP 428 - Le module `pathlib` - Chemins d'accès au système de fichiers orientés objet*), par Antoine Pitrou, 30 juillet 2012 - `https://peps.python.org/pep-0428/`

Documentation complémentaire

- *Python 3's `pathlib` Module : Taming the File System* (trad. *Le module `pathlib` de Python 3 : Apprivoiser le système de fichiers*), par Geir Arne Hjelle, 23 avril 2018 - `https://realpython.com/python-pathlib/`
- *Working With Files in Python* (trad. *Travailler avec des fichiers en Python*), par Vuyisile Ndlovu, 21 janvier 2019 - `https://realpython.com/working-with-files-in-python/`
- *How to Get a List of All Files in a Directory With Python* (trad. *Comment obtenir la liste de tous les fichiers d'un répertoire avec Python ?*), par Ian Currie, 28 novembre 2022 - `https://realpython.com/get-all-files-in-directory-python/`

Méthode	Action
`Path.chmod(mode, *, follow_symlinks=True)`	Change les permissions du fichier à l'instar de `os.chmod`. Exemple : `fichier.chmod(0o777)`
`Path.cwd()`	Retourne un nouveau chemin représentant le dossier courant, tel que renvoyé par `os.getcwd()`.
`Path.exists()`	Permet de vérifier si le chemin existe. Retourne `True` ou `False`.
`Path.expanduser()`	Un `PosixPath('~/Documents')` auquel on applique la méthode `expanduser()` retournera l'objet suivant : `PosixPath('/home/krystof/ Documents')`
`Path.glob(motif)`	Permet une itération dans les répertoires afin de lister les fichiers correspondant au motif. Ainsi l'instruction `list(chemin.glob('**/*.py))` va lister l'ensemble des fichiers `.py` présent dans le répertoire ciblé.
`Path.home()`	Retourne un nouveau chemin représentant le répertoire racine de l'utilisateur, tel que renvoyé par `os.path.expanduser('~')`. `PosixPath('/home/krystof')` sera affiché en sortie, tandis qu'avec `os.path.expanduser('~')` nous obtiendrons `'/home/krystof'`.
`Path.is_dir()`	Vérifie si le chemin est celui d'un répertoire. Retourne une valeur booléenne.
`Path.is_file()`	Vérifie si le chemin est celui d'un fichier. Retourne une valeur booléenne.
`Path.iterdir()`	Permet d'itérer sur l'ensemble des fichiers du répertoire (avec une boucle `for` pour une utilisation classique). Exemple de la compréhension de liste `[x for x in chemin.iterdir() if x.is_dir()]` qui permet de lister les répertoires du répertoire cible.
`Path.mkdir(mode=0o777, parents=False, exist_OK=False)`	Pour créer un répertoire.
`Path.open(mode='r', buffering=-1, encoding=None, errors=None, newline=None)`	Permet d'ouvrir un fichier à l'instar de la fonction native `open()`. Utilisations : `chemin_fichier.open()` ou `with chemin_fichier.open() as f:`.
`Path.rmdir()`	Supprime le répertoire donné. Celu-ci doit être vide.
`Path.samefile(autre_path)`	Retourne `True` ou `False` si le chemin pointe vers le même fichier que `autre_path`, qui peut être un chemin ou une chaîne de caractères. La sémantique est similaie à `os.path.samefile()` et `os.path.samestat()`.
`Path.stat(*, follow_symlinks=True)`	Retourne un objet contenant des informations sur le chemin.
`Path.touch(mode=0o666, exist_ok=True)`	Créer un fichier, s'il n'existe pas, avec le mode d'accès donné.
`Path.unlink(missing_ok=False)`	Supprime le fichier (ou lien symbolique). Utiliser `Path.rmdir()` si le chemin pointe vers un répertoire. Si `missing_ok` est à `True` (`False` étant la valeur par défaut) l'exception `FileNotFoundError` est ignorée (même comportement que la commande *posix* `rm -f`.

TABLE 26.1 – Les méthodes de la classe `Path` (1).

Méthode	Action
`Path.write_text( donnée, encoding=None, errors=None, newline=None)`	Ouvre le fichier pointé en mode texte, y inscrit `donnée` puis ferme le fichier. S'il existe, le fichier du même nom est écrasé. Les paramètres optionnels ont la même signification que la fonction native `open()`.
`Path.owner()`	`chemin.owner()` donne en sortie `'krystof'`.
`Path.rename(cible)`	Renomme le chemin avec le nom 'cible'.
`Path.replace(cible)`	Comme `Path.rename(cible)`, mais écrase le chemin renommé s'il existe.
`Path.resolve(strict= False)`	Retourne le chemin absolu en résolvant tous les liens symboliques afin de normaliser ce chemin.
`Path.rglob(motif)`	Similaire à la méthode `Path.glob()`. Retourne de façon récursive (en explorant les sous-répertoires) tous les fichiers correspondant au motif donné.
`Path.symlink_to(cible , target_is_directory= False`	Faire de ce chemin un lien symbolique qui point vers `cible`.
`Path.write_bytes(don née)`	Ouvre le fichier pointé en mode binaire, y écrit `donnée` puis ferme le fichier. S'il existe, le fichier du même nom est écrasé.

TABLE 26.2 – Les méthodes de classe la `Path` (2).

Sixième partie

Python en profondeur

Chapitre 27

Ecrire du code Python selon le PEP8

PEP signifie *Python Enhancement Proposal* (*Proposition d'amélioration de Python*), et il en existe plusieurs. Un PEP est un document qui décrit les nouvelles fonctionnalités proposées pour Python et en documente les aspects. Le PEP8[1] fournit des directives et des bonnes pratiques sur la façon d'écrire du code Python. Il a été rédigé en 2001 par Guido van Rossum, Barry Warsaw et Nick Coghlan. L'objectif principal du PEP8 est d'améliorer la lisibilité et la cohérence du code Python.

En tant que débutant, suivre les règles de la PEP 8 peut rendre l'apprentissage de Python beaucoup plus agréable. Suivre le PEP 8 est particulièrement important si vous cherchez un emploi dans le domaine du développement. Écrire un code clair et lisible est une preuve de professionnalisme. Cela indique à un employeur que vous savez comment bien structurer votre code. Si vous avez plus d'expérience dans l'écriture de code Python, vous devrez peut-être collaborer avec d'autres personnes. L'écriture d'un code lisible est ici cruciale.

27.1 Les conventions de nommage

Dans le codage il est nécessaire de définir des variables, des fonctions, des classes, bref des objets qu'il nous faudra nommer. Le choix de ces noms devra rendre explicite le contenu de l'objet. Ainsi, si une valeur doit représenter un âge, autant nommer la variable age plutôt que a, ce qui rendra votre code plus lisible et facilitera son débogage.

Les styles de nommage selon l'objet :

Le choix des noms :

Il est pertinent de choisir des noms descriptifs pour nommer ses objets, qui décrivent clairement l'objet. Cela participe d'ailleurs à rendre le code plus lisible, notamment cela aide le développeur quand il reprend son code plusieurs jours ou plusieurs semaines après. Par exemple choisir comme nom de variable les termes nom ou name pour stocker le nom d'une personne, ou bien comme nom de fonction multiplie_par_deux ou multiply_by_two.

27.2 Présentation du code

Les lignes vides :

Les lignes vides, peuvent améliorer considérablement la lisibilité de votre code. Un code peu aéré peut être écrasant et difficile à lire, de même, trop de lignes vides lui donne un aspect très clairsemé. Voici quelques règles pour améliorer la lisibilité de vote code :

1. *Style Guide for Python Code* - https://peps.python.org/pep-0008/

!h

Type	Convention de nommage	Exemples
Fonction	Nom en minuscules, et si plusieurs mots utilisés, ils sont séparés par un underscore.	`fonction`, `ma_fonction`
Variable	Comme pour les fonctions. Possibilité de n'utiliser qu'une seule lettre.	`x`, `variable`, `ma_variable`
Classe	Chaque mot du nom débute par une majuscule, et ils ne sont pas séparés. C'est le style *camel case*.	`NomDeClasse`
Méthodes (de classe)	Comme pour les fonctions.	`methode_de_classe`
Constante	Nom en majuscules, et si plusieurs mots utilisés, ils sont séparés par un underscore.	`X`, `CONST`, `MA_CONSTANTE`
Module et package	Comme pour les fonctions.	`module.py`, `mon_module.py`, `nom_package.py`

TABLE 27.1 – Les styles de nommage en `Python`.

— Deux lignes vides séparent les fonctions et les classes de haut niveau.
— Une seule ligne sépare les méthodes d'une classe.
— Il est possible d'utiliser une ligne de séparation entre diverses étapes d'une fonction, si cette fonction dispose d'un code relativement long.

Si les lignes vides sont utilisées avec soin, elles peuvent améliorer considérablement la lisibilité de votre code. Cela aide le lecteur à comprendre visuellement comment est structuré le code.

Longueur maximale des lignes et saut de ligne

Le `PEP 8` suggère que les lignes soient limitées à 79 caractères, ce qui permet d'avoir plusieurs fichiers ouverts l'un à côté de l'autre. Bien sûr, il n'est pas toujours possible de limiter les déclarations à 79 caractères, et le `PEP 8` propose des moyens de permettre aux instructions de s'étendre sur plusieurs lignes.

Voici quelques exemples recommandés par le `PEP 8` :

Avec des parenthèses :

```
1     def fonction(arg_un, arg_deux,
2                  arg_trois, arg_quatre):
3         return arg_un
```

utilisation des *antislashes* pour couper les lignes :

```
1     from mon_package import meth1, \
2         meth2, meth3
```

Saut de ligne avant les opérateurs :

```
1     total = (valeur_un
```

```
2               + valeur_deux
3               - valeur_trois)
```

27.3 Indentation

L'indentation est extrêmement importante en `Python`. Le niveau d'indentation des lignes de code en `Python` détermine la manière dont les instructions sont regroupées. Les principales règles d'indentation énoncées par la `PEP 8` sont les suivantes :

— Utiliser 4 espaces consécutifs pour indiquer l'indentation.

— Préférer les espaces aux tabulations.

Lorsque vous utilisez des sauts de ligne pour limiter les lignes à moins de 79 caractères, il est utile d'utiliser l'indentation pour améliorer la lisibilité. Elle permet au lecteur de faire la distinction entre deux lignes de code et une seule ligne de code qui s'étend sur deux lignes. Il existe deux styles d'indentation que vous pouvez utiliser.

— Le premier consiste à aligner le bloc indenté avec le délimiteur d'ouverture :

```
1    def fonction(arg_un, arg_deux,
2                 arg_trois, arg_quatre):
3        return arg_un
```

— Un autre style d'indentation est l'indentation suspendue. Il s'agit d'un terme typographique qui signifie que toutes les lignes d'un paragraphe ou d'une déclaration, sauf la première, sont mises en retrait :

```
1    var = fonction(
2        arg_un, arg_deux,
3        arg_trois, arg_quatre)
```

Il arrive parfois que 4 espaces seulement soient nécessaires pour s'aligner sur le délimiteur d'ouverture. Cela se produit souvent dans les instructions `if` qui s'étendent sur plusieurs lignes, car le `if`, l'espace et la parenthèse ouvrante représentent 4 caractères. Dans ce cas, `PEP 8` fournit deux alternatives pour aider à améliorer la lisibilité :

— Ajouter un commentaire après la condition finale.

```
1    x = 5
2    if (x > 3 and
3        x < 10):
4        # Les deux conditions sont remplies
5        print(x)
```

— Ajouter une indentation supplémentaire sur la suite de la ligne.

```
1    x = 5
2    if (x > 3 and
3            x < 10):
4        print(x)
```

Avec les fonctions, il est préférable d'utiliser une double indentation sur la suite de la ligne. Cela permet de faire la distinction entre les arguments de fonction et le corps de la fonction, ce qui améliore la lisibilité :

```
def fonction(
        arg_un, arg_deux,
        arg_trois, arg_quatre):
    return arg_un
```

Les continuations de ligne permettent de rompre les lignes à l'intérieur des parenthèses, des crochets ou des accolades. Il est facile d'oublier l'accolade fermante, mais il est important de la placer à un endroit judicieux. Sinon, cela peut perturber le lecteur. Le `PEP 8` fournit deux options pour la position de l'accolade fermante dans le cadre des continuations de ligne implicites :

— Alignez l'accolade fermante avec le premier caractère de la ligne précédente.

```
    liste_de_nombres = [
        1, 2, 3,
        4, 5, 6,
        7, 8, 9
        ]
```

— Alignez l'accolade fermante avec le premier caractère de la ligne qui commence la construction.

```
    liste_de_nombres = [
        1, 2, 3,
        4, 5, 6,
        7, 8, 9
    ]
```

27.4 Les commentaires

Un commentaire doit permettre de comprendre facilement le code auquel le commentaire s'applique. Voici quelques points essentiels à retenir lorsque vous ajoutez des commentaires à votre code :

— Limitez la longueur de ligne des commentaires et des *docstrings* à 72 caractères.
— Utilisez des phrases complètes, commençant par une majuscule.
— Veillez à mettre à jour les commentaires si vous modifiez votre code.

Les blocs de commentaires

Utilisez les commentaires de bloc pour documenter une petite section de code. Ils sont importants car ils aident les autres à comprendre l'objectif et la fonctionnalité d'un bloc de code donné. `PEP 8` fournit les règles suivantes pour la rédaction des commentaires de bloc :

— Indenter les blocs de commentaires au même niveau que le code qu'ils décrivent.
— Commencez chaque ligne par un `#` suivi d'un simple espace.
— Séparer les paragraphes par une ligne contenant un seul `#`.

Parfois, si le code est très technique, il est nécessaire d'utiliser plus d'un paragraphe dans un commentaire de bloc.

```
# Voici un premier paragraphe
# pour mon commentaire de bloc
#
# Et après un saut de ligne (avec #)
# je m'attaque au second paragraphe
```

Commentaires de ligne unique de code

Ils sont destinés à expliciter le code d'une seule ligne. Le `PEP 8` propose les règles suivantes :

— Ils doivent être utilisés avec parcimonie.
— Ils doivent être écrits sur la même ligne que le code auquel ils font références.
— Les séparer par deux espaces ou plus du code.
— Ils débutent par `#` suivi d'un espace unique.
— Ils ne s'utilisent pas pour expliquer ce qui est aisément compréhensible.

Les commentaires de ligne unique de code peuvent rapidement encombrer votre code, les commentaires de bloc sont alors à privilégier.

Les *docstrings*

Les *docstrings*, sont des chaînes entre guillemets doubles (`"""`) ou simples (`'''`) qui apparaissent sur la première ligne de toute fonction, classe, méthode ou module. Le `PEP 257` [2] leur est consacré.

27.5 Espaces blancs dans les expressions et les déclarations

Les espaces blancs peuvent être très utiles dans les expressions et les déclarations lorsqu'ils sont utilisés correctement. S'il n'y a pas assez d'espace blanc, le code peut être difficile à lire, car tout est regroupé. S'il y a trop d'espaces blancs, il peut être difficile de combiner visuellement des termes connexes dans une déclaration.

Espaces blancs autour des opérateurs binaires

Les opérateurs d'affectation (`=`, `+=`, `-=`, etc.), de comparaison (`==`, `!=`, `>`, `<`. `>=`, `<=`, `is`, `is not`, `in`, `not in`) et booléens (`and`, `not`, `or`) doivent être entourés d'un simple espace de chaque côté.

Cepedanr lorsque le signe `=` est utilisé pour attribuer une valeur par défaut à un argument de fonction, ne l'entourez pas d'espaces.

Lorsqu'il y a plus d'un opérateur dans une instruction, l'ajout d'un espace unique avant et après chaque opérateur peut prêter à confusion. Il est préférable de n'ajouter des espaces qu'autour des opérateurs ayant la priorité la plus faible, en particulier lors de manipulations mathématiques :

2. `https://peps.python.org/pep-0257/` - Voir aussi un guide assez complet sur la documentation du code à l'adresse suivante : `https://realpython.com/documenting-python-code/` - Voir aussi le chapitre *Documenter son code et son projet* page 181.

```
y = x**2 + 5
z = (x+y) * (x-y)
```

Cela s'applique aussi avec les instructions comme `if` :

```
if x>5 and x%2==0:
    print('x est plus grand que 5 et est un nombre pair')
```

Vous êtes bien entendu libre de choisir ce qui est le plus clair.

Les mêmes règles s'appliquent aux *slices* :

```
liste1[x+1 : x+2]
liste2[x+1 : x+2 : x+3]
liste3[x+1 : x+2 :]
```

Quand éviter d'ajouter des espaces blancs

Dans certains cas, l'ajout d'espaces blancs peut rendre le code plus difficile à lire. Trop d'espaces blancs peuvent rendre le code trop clairsemé et difficile à suivre. La `PEP 8` donne des exemples très clairs de cas où les espaces blancs sont inappropriés. L'endroit le plus important pour éviter d'ajouter des espaces blancs est la fin d'une ligne. C'est ce qu'on appelle les espaces blancs de fin de ligne. Il est invisible et peut produire des erreurs difficiles à repérer.

Quelques exemples de recommandation :

```
ma_liste = [1, 2, 3]
print(x, y)  # x et y étant des variables
ma_fonction(3)  # Appel de fonction
ma_liste[3]
mon_tuple = (1,)
var1 = 5
var2 = 6
un_long_nom_de_variable = 7
```

Il existe d'autres cas où la `PEP 8` déconseille l'ajout d'espaces supplémentaires, notamment à l'intérieur des parenthèses, ainsi qu'avant les virgules et les deux-points.

27.6 Recommandations en matière de programmation

Le *Zen de* ***Python*** dit : « *Le simple est préférable au complexe* ».

Voici les principales recommandations :

— Ne pas comparer pas les valeurs booléennes (`True` ou `False` en utilisant un opérateur d'équivalence :

```
    if mon_booleen:
```

— Utiliser le fait que les séquences vides sont fausses quand elles sont évaluer avec l'instruction `if` :

```
ma_liste = []
if not ma_liste:
    print("La liste est vide")
```

— Utiliser `is not` plutôt que `not ... is` dans les instructions `if` :

```
if not x is None:
    return 'x existe !'
```

— Ne pas utiliser `if x :` lorsque l'on souhaite déclarer `if x is not None :` car parfois on peut avoir une fonction avec des arguments qui sont `None` par défaut, ce qui peut être source d'erreur.

```
if arg is not None:
    # Faire quelque chose avec arg...
```

— Utiliser `.startswith()` et `.endswith()` au lieu du découpage en tranches :

```
if nom_fichier.endswith('jpg'):
    print("Le fichier est un fichier image de type JPEG")
```

27.7 Un code conforme à la PEP 8

Si vous suivez la `PEP 8` à la lettre, vous pouvez garantir que vous aurez un code propre, professionnel et lisible. Vous en tirerez profit, ainsi que les développeurs avec qui vous collaborer sur un projet commun.

Pour aider le développeur à rendre son code conforme à la `PEP 8`, il existe deux catégories d'outils : les *linters* et les outils de formatage automatique.

Les *linters*

Les *linters* sont des programmes qui analysent le code et signalent les erreurs. Ils fournissent des suggestions sur la manière de corriger l'erreur. Les *linters* sont particulièrement utiles lorsqu'ils sont installés comme extensions de votre éditeur de texte, car ils signalent les erreurs et les problèmes stylistiques pendant que vous écrivez.

`pycodestyle` est un outil permettant de vérifier votre code *Python* par rapport à certaines des conventions de style du `PEP 8`[3].

```
$ pip install pycodestyle  # Installation
$ pycodestyle mon_code.py  # Exécution
mon_code.py:1:17: E231 missing whitespace after ','
mon_code.py:2:21: E231 missing whitespace after ','
mon_code.py:6:19: E711 comparison to None should be 'if cond is None:'
```

3. https://pypi.org/project/pycodestyle/

`flake8` est un outil qui combine un débogueur (`pyflakes`) avec `pycodestyle` qui, via la ligne de commandes, va analyser le code et en indiquer les erreurs de formatage [4].

```
$ pip install flake8
$ flake8 mon_code.py
mon_code.py:1:17: E231 missing whitespace after ','
mon_code.py:2:21: E231 missing whitespace after ','
mon_code.py:3:17: E999 SyntaxError: invalid syntax
mon_code.py:6:19: E711 comparison to None should be 'if cond is None:'
```

Les outils de formatage automatique

On retrouve ce type d'outils sous forme de programmes qui refactorisent automatiquement votre code pour le rendre conforme à la `PEP 8`, comme par exemple le programme *black*. Sur la page du projet [5] il est dit que *black* « *rend la révison du code plus rapide* ».

Installation de *black* de façon globale sur le système :

```
$ pip install black
```

Utilisation (s'utilise en ligne de commandes) :

```
$ black mon_code.py
reformatted mon_code.py
All done!
```

Par défaut *black* limite la longueur des lignes à 88 caractères, au lieu de 79, mais nous pouvons modifier un tel comportement avec l'option `--line-length flag` :

```
$ black --line-length=79 mon_code.py
reformatted mon_code.py
All done!
```

Deux autres outils réalisent des formatages automatiques, similaires à ceux réalisés par *black* : *autopep8* [6] et *yapf* [7]. Et bien entendu à l'aide de `Pycharm` :

```
Code > Reformat Code
```

D'autre encore proposent des fonctionnalités plus spécialisées :

`docformatter` [8] pour formater les *docstrings*. A installer avec `pip install`.

```
$docformatter --in-place fichier.py
```

isort : Organise les imports comme il se doit. installer avec `pip install`.

```
$isort fichier.py
```

4. `https://pypi.org/project/flake8/`
5. `https://pypi.org/project/black/`
6. `https://pypi.org/project/autopep8/`
7. `https://pypi.org/project/yapf/`
8. Voir la `PEP 257` : `https://peps.python.org/pep-0257/`

Chapitre 28

Qualité du code *Python* : Outils et bonnes pratiques

28.1 Qu'est-ce qu'un code de qualité ?

Un code de qualité, selon ce qui fait consensus dans la communauté des développeurs, répond à trois principes cruciaux :

- Un code doit faire ce qu'il est censé faire.
- C'est un code qui ne contient pas de défaut ou problème.
- Il doit demeurer facile à lire, à maintenir et à étendre.

Ne pas respecter ces trois dimensions expose à ce que le code :

- **ne fasse pas ce qu'il est censé faire :** Le respect des exigences est à la base de tout produit, logiciel ou autre. Nous créons des logiciels pour faire quelque chose. Si, au bout du compte, il ne le fait pas, alors il ne s'agit certainement pas d'un produit de qualité.
- **présente des défauts et des problèmes :** Si les défauts provoquent des comportements indésirables, là aussi nous n'avons pas un produit de qualité.
- **soit difficile à lire, à maintenir ou à étendre :** Si le code est complexe et alambiqué, il sera difficile pour une autre personne (voire vous-même) de le comprendre pour en corriger les éventuels défauts. Il est également souhaitable qu'il soit facile d'ajouter de nouvelles fonctionnalités sans perturber les fonctionnalités précédentes. Si le code n'est pas facile à étendre, votre nouvelle fonctionnalité risque de perturber d'autres éléments.

Pour améliorer la qualité d'un code il est souhaitable d'apporter une grande attention au style. Un guide de style a pour but de définir une manière cohérente d'écrire votre code. En règle générale, il s'agit d'une démarche cosmétique, qui ne modifie pas le résultat logique du code. Toutefois, certains choix stylistiques permettent d'éviter des erreurs logiques courantes. En ce qui concerne *Python*, il existe une norme bien acceptée. Elle a été rédigée, en partie, par l'auteur du langage de programmation Python lui-même. Le `PEP 8` fournit des conventions de codage pour le code *Python*. Il est assez courant que le code *Python* suive ce guide de style. C'est un bon point de départ car il est déjà bien défini (voir chapitre précédent).

Il existe aussi le `PEP 257`, qui décrit les conventions pour les *docstrings* de *Python*, qui sont des chaînes destinées à documenter les modules, les classes, les fonctions et les méthodes. En prime, il existe des outils capables de générer de la documentation directement à partir du code (voir prochain chapitre).

28.2 Les *linters* : des outils puissants pour améliorer son code

Les petites erreurs, les incohérences stylistiques et les logiques dangereuses ne rendent pas un code agréable à regarder et à modifier. Les *linters* permettent d'identifier ces problèmes. Il faut savoir que la plupart des éditeurs et des IDE ont la possibilité d'exécuter des *linters* en arrière-plan, et ce pendant que vous saisissez votre code. Les *linters* analysent alors le code pour détecter différentes catégories de problèmes :

- **d'un point de vue logique :** erreurs de code, code dont les résultats sont potentiellement inattendus, modèles de code dangereux.
- **d'un point de vue stylistique :** code non conforme aux conventions définies.

Il existe également des outils d'analyse de code qui permettent d'en améliorer la qualité :

`Flake8` **:** Capable de détecter à la fois les défauts logiques et stylistiques. Il ajoute les vérifications de style et de complexité de `pycodestyle` à la détection logique de `PyFlakes`. Il combine les *linters* suivants : `PyFlakes`, `pycodestyle` (`PEP 8`) et `Mccabe`.

`Pylama` **:** C'est un outil d'audit de code composé d'un grand nombre de *linters* et d'autres outils d'analyse de code. Il combine les éléments suivants : `pycodestyle`, `pydocstyle` (`PEP 257`), `PyFlakes`, `Mccabe`, `Pylint`, `Radon`, `gjslint`.

Voici un tableau qui présente les divers *linters* utilisables avec *Python* :

linter	**Catégorie**	**Description**	**Sites Internet**
`Pylint`	Logique et style	Vérifie les erreurs, tente d'appliquer une norme de codage, recherche les défauts du code.	`https://www.pylint.org/` et `https://pylint.pycqa.org/en/latest/user_guide/usage/output.html`
`PyFlakes`	Logique	Analyse les programmes et détecte les différentes erreurs. Il se concentre sur les problèmes de code logique et les erreurs potentielles. L'avantage ici est la rapidité. L'inconvénient est que les différents problèmes et erreurs ne sont pas étiquetés ou organisés par type.	`https://github.com/PyCQA/pyflakes`
`pycodestyle`	Style	Vérifie certaines conventions de style de la `PEP 8`. Les erreurs sont étiquetées par catégorie. Vous pouvez choisir d'ignorer certaines erreurs si vous ne souhaitez pas adhérer à une convention spécifique.	`https://github.com/PyCQA/pycodestyle`

TABLE 28.1 – Les différents *linters* - 1

linter	Catégorie	Description	Sites Internet
`pydocstyle`	Style	Vérifie la conformité avec les conventions de la *docstring* de *Python*, selon le `PEP 257`. `pydocstyle` étiquette et catégorise les différentes erreurs qu'il trouve. La liste n'entre pas en conflit avec celle de `pycodestyle` puisque toutes les erreurs sont préfixées par un `D` pour *docstring*.	`https://github.com/PyCQA/pydocstyle` - Liste des codes d'erreur : `http://www.pydocstyle.org/en/latest/error_codes.html`
`Bandit`	Logique	Analyse le code pour détecter les problèmes de sécurité courants.	`https://github.com/PyCQA/bandit`
`MyPy`	Logique	Vérifie l'existence de types statiques optionnels.	`https://mypy-lang.org/`
`Mccabe`	Analytique	Vérification de la complexité de `McCabe`.	`https://github.com/PyCQA/mccabe`
`Radon`	Analytique	Analyse le code en fonction de diverses mesures (lignes de code, complexité, etc.).	`https://radon.readthedocs.io/en/latest/`
`Black`	Formateur	Formate le code *Python* sans compromis.	`https://github.com/psf/black`
`Isort`	Formateur	Formate les importations en les triant par ordre alphabétique et en les séparant en sections.	`https://github.com/PyCQA/isort`

TABLE 28.2 – Les différents *linters* - 2

Il est utile de faire exécuter fréquemment des tests de *linters* sur votre code afin d'en vérifier régulièrement la qualité.

1. Au fur et à mesure de l'écriture : à l'aide de *linters* implémentables sur votre IDE.
2. Avant de valider le code : bien que cela puisse sembler drastique, le fait de forcer chaque morceau de code à passer un examen de contrôle est une étape importante pour garantir une qualité continue.
3. Lors de l'exécution des tests.

En résumé, nous pouvons dire qu'un code de qualité fait ce qu'il est censé faire sans plantage. Il est aussi facile à lire, à maintenir et à étendre. Il fonctionne sans problème ni défaut et est écrit de manière à être facile à utiliser pour la personne qui le reprend. L'amélioration de la qualité du code est finalement un processus à acquérir.

Sources pour la rédaction de ce chapitre

- *Python Code Quality : Tools & Best Practices* (trad : « *Qualité du code Python : Outils et bonnes pratiques* »), par Alexander Van Tol - 30 juillet 2018 - `https://realpython.com/python-code-quality/`

Chapitre 29

Documenter son code et son projet

29.1 Documenter votre base de code Python à l'aide de *Docstrings*

Dans cette section, nous allons nous plonger dans la documentation d'une base de code `Python` en réabordant plus en profondeur les *docstrings*, et nous verrons comment utiliser ces derniers en vue d'obtenir une documentation conforme aux préconisations. Nous traiterons du fonctionnement interne des *docstrings* dans `Python`, des différents types de *docstrings* selon la nature de l'objet (fonction, classe, méthode de classe, module), ainsi que des différents formats de *docstrings* (`Google`, `reStructuredText`).

29.2 Le fonctionnement interne des *docstrings*

Les *docstrings* insérées dans votre code, s'ils sont correctement configurés, peuvent aider vos utilisateurs et vous-même à documenter votre projet. `Python` dispose également de la fonction intégrée `help()` qui affiche la *docstring* des objets sur la console. Voyons par exemple la brève documentation intégrée dans la *docstring* de la classe `int` :

```
$ python3
Python 3.10.8 (main, Nov  4 2022, 09:21:25) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> help(int)
Help on class int in module builtins:

class int(object)
|  int([x]) -> integer
|  int(x, base=10) -> integer
|
|  Convert a number or string to an integer, or return 0 if no arguments
|  are given.  If x is a number, return x.__int__().  For floating point
|  numbers, this truncates towards zero.
|
|  If x is not a number or if base is given, then x must be a string,
|  bytes, or bytearray instance representing an integer literal in the
|  given base.  The literal can be preceded by '+' or '-' and be surrounded
|  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
|  Base 0 means to interpret the base from the string as an integer literal.
|  >>> int('0b100', base=0)
|  4
```

```
|
|  Built-in subclasses:
|      bool
|
|  Methods defined here:
|
|  __abs__(self, /)
|      abs(self)
|
|  __add__(self, value, /)
|      Return self+value.
|
```

Examinons cela en affichant la documentation de l'objet avec la méthode `.__doc__` :

```
>>> print(int.__doc__)
int([x]) -> integer
int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments
are given.  If x is a number, return x.__int__().  For floating point
numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string,
bytes, or bytearray instance representing an integer literal in the
given base.  The literal can be preceded by '+' or '-' and be surrounded
by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
Base 0 means to interpret the base from the string as an integer literal.
>>> int('0b100', base=0)
4
```

Vous savez dorénavant trouer les *docstrings* stockées dans l'objet. Et si cela fonctionne avec des objets de type *builtin* (interne à `Python`, cela fonctionne également avec tout objet personnalisé :

```
>>> def multiplie(nbre1, nbre2):
...     print(f"{nbre1} X {nbre2} = {nbre1*nbre2}")
...
>>> multiplie.__doc__ = "Une fonction qui multiplie deux nombres passés en
    paramètres"
>>> help(multiplie)
Help on function multiplie in module __main__:

multiplie(nbre1, nbre2)
    Une fonction qui multiplie deux nombres passés en paramètres
```

`Python` possède une autre fonctionnalité qui simplifie la création d'une documentation à l'aide d'une chaîne de caractères. Le placement stratégique de la chaîne littérale directement sous la définition de l'objet définira automatiquement ce qui sera affiché à l'aide de la méthode `__doc__`. Voici ce qui se passe avec le même exemple que ci-dessus :

```
>>> def multiplie(nbre1, nbre2):
...     """Une fonction qui multiplie deux nombres passés en paramètres"""
```

```
...     print(f"{nbre1} X {nbre2} = {nbre1*nbre2}")
...
>>> help(multiplie)
Help on function multiplie in module __main__:

multiplie(nbre1, nbre2)
    Une fonction qui multiplie deux nombres passés en paramètres
```

29.3 Les types de *docstring*

Les conventions des *docstring* sont décrites dans le `PEP 257` [1]. Leur but est de fournir un bref aperçu de l'objet. Elles doivent être suffisamment concises pour être faciles à maintenir, mais suffisamment élaborées pour en comprendre leur but et savoir comment utiliser l'objet documenté. Elle indique l'effet de la fonction ou de la méthode en tant que commande (« Fais ceci », « Renvoie cela ») et n'est pas une description (ne pas écrire, par exemple : « Rentourne le nom de chemin... ». Une *docstring* est une chaîne de caractères littérale qui apparaît comme la première déclaration dans un module, une fonction, une classe ou une définition de méthode. Dans tous les cas, les *docstrings* sont sous la forme de chaîne de caractères insérées entre des tripes quotes (`"""`) [2], que la *docstring* soit multi-lignes ou non.

Si la chaîne documentaire tient sur une ligne, les guillemets fermants doivent être sur la même ligne que les guillemets ouvrants. Des guillemets triples sont utilisés même si la chaîne tient sur une seule ligne. Cela permet de l'étendre facilement par la suite. Il n'y a pas de ligne vierge avant ou après la chaîne de caractères. La *docstring* est une phrase qui se termine par un point. Au minimum, une *docstring* doit être un résumé rapide de ce que vous décrivez et doit être contenue sur une seule ligne :

```
1 """Voici une ligne de description rapide de l'objet."""
```

Les *docstrings* multi-lignes sont utilisés pour développer la description de l'*objet*, au-delà du résumé. Dans toutes les *docstrings* multi-lignes ont retrouve les caractéristiques suivantes :

- une ligne de résumé sur une seule ligne ;
- une ligne vide précédant le résumé ;
- des explications complémentaires ;
- une autre ligne vierge.

```
1  """La première ligne de résumé.
2
3  Ici débute l'élaboration plus poussée de la docstring. Dans
4  cette section, vous pouvez développer davantage les détails en
5  fonction de la situation.
6  Notez que le résumé et le développement sont séparés par une
7  nouvelle ligne vierge.
8  """
9
10 # Ligne vide ci-dessus : le code doit continuer sur cette ligne.
```

1. `https://peps.python.org/pep-0257/`
2. Pour des raisons de cohérence, utilisez toujours des guillemets doubles ou triples autour des chaînes de caractères. Utilisez les guillemets triples si vous utilisez des antislashs dans vos chaînes de caractères.

La *docstring* entière est indentée au même niveau que les guillemets de la première ligne.

Toutes les *docstrings* doivent avoir la même longueur maximale de caractères : soit 72 caractères. Les *docstrings* peuvent être subdivisées en quatre grandes catégories.

Les *docstrings* de classe

Les *docstrings* de classe sont placées immédiatement après la définition de la classe ou de la méthode de la classe, indentées d'un niveau :

```
class SimpleClasse :
    """La docstring de la classe est à insérer ici."""

    def dire_bonjour(self, nom : str) :
        """La docstring de la méthode de classe se place ici."""

        print(f'Bonjour {nom}')
```

Les *docstrings* de classe doivent contenir les informations suivantes :
- un bref résumé de son objectif et de son comportement ;
- toute méthode publique, accompagnée d'une brève description ;
- toutes les propriétés de la classe (attributs) ;
- tout ce qui concerne l'interface pour les sous-classes.

Les paramètres du constructeur de la classe doivent être documentés dans la *docstring* de la méthode de classe `__init__`. Les méthodes individuelles doivent être documentées à l'aide de leurs propres *docstrings*. Les *docstrings* des méthodes de classe doivent contenir les éléments suivants :
- une brève description de la méthode et de son utilité ;
- tous les arguments (obligatoires et facultatifs) qui sont transmis, y compris les arguments de type mot-clé ;
- l'étiquetage de tous les arguments considérés comme facultatifs ou ayant une valeur par défaut ;
- tout effet secondaire qui se produit lors de l'exécution de la méthode ;
- toutes les exceptions qui sont levées ;
- toute restriction concernant le moment où la méthode peut être appelée.

Prenons l'exemple simple d'une classe de données qui représente un animal. Cette classe contiendra quelques propriétés de classe, des propriétés d'instance, un `__init__` et une seule méthode d'instance :

```
class Animal :
    """
    Une classe utilisée pour représenter un animal

    ...

    Attributs
    ----------
    dit_str : str
        une chaîne formatée pour imprimer ce que dit l'animal
```

```
11      nom : str
12          le nom de l'animal
13      son : str
14          le son que fait l'animal
15      nbre_pattes : int
16          le nombre de pattes de l'animal (par défaut 4)
17
18      Méthodes
19      -------
20      dit(son=None)
21          Affiche le nom de l'animal et le son qu'il émet.
22      """
23
24      dit_str = "Un {nom} dit {son}"
25
26      def __init__(self, nom, son, nbre_pattes=4) :
27          """
28          Paramètres
29          ----------
30          nom : str
31              Le nom de l'animal
32          son : str
33              Le son que fait l'animal
34          nombre_pattes : int, facultatif
35              Le nombre de pattes de l'animal (4 par défaut)
36          """
37
38          self.nom = nom
39          self.son = son
40          self.nbre_pattes = nbre_pattes
41
42      def dit(self, son=None):
43          """Affiche le nom de l'animal et le son qu'il émet.
44
45          Si l'argument `son` n'est pas passé, le son par défaut
46          de l'animal est utilisé.
47
48          Parametres
49          ----------
50          son : str, optionnel
51              Le son émis par l'animal (par défaut, aucun - None)
52
53          Exceptions
54          ------
55          NotImplementedError
56              Si aucun son n'est défini pour l'animal ou passé en
57              tant que paramètre.
58          """
59
60          if self.son is None and son is None:
61              raise NotImplementedError("Les animaux silencieux ne
```

```
62                                          sont pas supportés !")
63
64          out_son = self.son if son is None else son
65          print(self.dit_str.format(nom=self.nom,, son=out_son))
```

Les *docstrings* de fonctions

La *docstring* d'une fonction ou d'une méthode doit résumer son comportement et documenter ses arguments, sa (ses) valeur(s) de retour, ses effets secondaires, les exceptions soulevées et les restrictions sur le moment où elle peut être appelée. Les arguments facultatifs doivent être indiqués. Il convient de documenter si les arguments des mots-clés font partie de l'interface.

Les *docstrings* des paquets et des modules

Les *docstrings* des paquets doivent être placées en haut du fichier `__init__.py` du paquet. Cette *docstring* doit énumérer les modules et les sous-paquets exportés par le paquet. Les *docstrings* de modules sont similaires aux *docstrings* de classes. Au lieu de documenter les classes et leurs méthodes, c'est maintenant le module et toutes les fonctions qu'il contient qui sont documentés. Les *docstrings* des modules sont placées en haut du fichier, avant même toute importation. Les *docstrings* des modules doivent inclure les éléments suivants :

- une brève description du module et de son objectif ;
- une liste des classes, des exceptions, des fonctions et de tout autre objet exporté par le module.

La *docstring* d'une fonction de module doit inclure les mêmes éléments qu'une méthode de classe.

Les *docstrings* des scripts

Les scripts sont considérés comme des exécutables à fichier unique lancés depuis la console. Les *docstrings* des scripts sont placées en haut du fichier et doivent être suffisamment bien documentées pour que les utilisateurs puissent avoir une compréhension suffisante de la façon d'utiliser le script.

29.4 Les formats de *docstring*

Il existe des formats de *docstrings* spécifiques. Certains des formats les plus courants figures dans le tableau *Les divers types de formatage les plus courants* page 187.

Le choix du format de la *docstring* est laissé à votre discrétion, mais vous devez conserver le même format pour l'ensemble de votre document/projet.

Exemple avec `Google docstrings` [3] :

```
1 """Affiche les colonnes d'en-tête d'une feuille de calcul.
2
3 Args :
```

3. https://github.com/google/styleguide/blob/gh-pages/pyguide.md

Type de formatage	Description	Supporté par Sphynx	Spécification formelle
Google docstrings	Format de documentation recommandé par Google	Oui	Non
reStructuredText	Documentation standard officielle de Python - Non conviviale pour les débutants, mais riche en fonctionnalités.	Oui	Oui
NumPy/SciPy docstrings	La combinaison de NumPy avec reStructuredText et Google Docstrings.	Oui	Oui

TABLE 29.1 – Les divers types de formatage les plus courants

```
4       file_loc (str) : Emplacement du fichier de la feuille de
5           calcul
6       print_cols (bool) : Un indicateur utilisé pour afficher les
7           colonnes sur la console (la valeur par défaut est False)
8
9   Returns :
10  liste : une liste de chaînes de caractères représentant les
11      en-têtes des colonnes.
12  """
```

Exemple avec reStructuredText [4] :

```
1   """Affiche les colonnes d'en-tête de la feuille de calcul.
2
3   :param file_loc: L'emplacement du fichier de la feuille de
4       calcul
5   :type file_loc: str
6   :param print_cols: Indicateur permettant d'imprimer les colonnes
7       sur la console. (la valeur par défaut est False)
8   :type print_cols: bool
9   :returns: une liste de chaînes de caractères représentant les
10      en-têtes de colonnes
11  :rtype: liste
12  """
```

Exemple avec NumPy/SciPy docstrings [5] :

```
1   """Affiche les colonnes d'en-tête de la feuille de calcul.
2
```

4. https://docutils.sourceforge.io/rst.html
5. https://numpydoc.readthedocs.io/en/latest/format.html

```
Parameters
----------
file_loc : str
    Emplacement du fichier de la feuille de calcul
print_cols : bool, facultatif
    Un indicateur permettant d'imprimer les colonnes sur la
    console (par défaut, False)

Returns
-------
liste
    une liste de chaînes de caractères représentant les colonnes
    d'en-tête.
"""
```

29.5 Documenter ses projets Python

Tous les modules doivent normalement avoir une *docstring*, et toutes les fonctions et classes exportées par un module doivent également avoir une *docstring*. Les méthodes publiques (y compris le constructeur `__init__`) doivent également avoir des *docstrings*. Un paquet peut être documenté dans la *docstring* du module du fichier `__init__.py` dans le répertoire du paquet.

La façon dont vous documentez votre projet doit être adaptée à au projet lui-même. Gardez à l'esprit qui seront les utilisateurs de votre projet et adaptez-vous à leurs besoins. Selon le type de projet, certains aspects de la documentation sont recommandés. La présentation générale du projet et de sa documentation doit être la suivante :

```
repertoire_racine_du_projet/
|
|--- project/  # Code source
|--- docs/
|--- README
|--- HOW_TO_CONTRIBUTE
|--- CODE_OF_CONDUCT
|--- examples.py
```

Les projets peuvent généralement être subdivisés en trois grands types : privés, partagés et publics/ouverts.

29.6 Les projets privés

Les projets privés sont des projets destinés à un usage personnel uniquement et ne sont généralement pas partagés avec d'autres utilisateurs ou développeurs. La documentation peut être assez légère sur ces types de projets. Il y a quelques parties recommandées à ajouter si nécessaire :

Readme : Un bref résumé du projet et de son objectif. Incluez toute exigence particulière pour l'installation ou le fonctionnement du projet.

examples.py : Un fichier script Python qui donne des exemples simples d'utilisation du projet.

N'oubliez pas que, même si les projets privés vous sont destinés personnellement, vous êtes également considéré comme un utilisateur. Pensez à tout ce qui pourrait être source de confusion pour vous à l'avenir et veillez à le mentionner dans les commentaires, les *docstrings* ou le *readme*.

29.7 Les projets partagés

Les projets partagés sont des projets dans lesquels vous collaborez avec quelques autres personnes pour le développement et/ou l'utilisation du projet. Le « *client* » ou l'utilisateur du projet reste vous-même et les quelques personnes qui utilisent également le projet.

La documentation doit être un peu plus rigoureuse qu'elle ne doit l'être pour un projet privé, principalement pour aider à intégrer de nouveaux membres au projet ou alerter les contributeurs/utilisateurs des nouvelles modifications apportées au projet. Voici quelques-unes des parties qu'il est recommandé d'ajouter au projet :

Readme : Un bref résumé du projet et de son objectif. Incluez toute exigence particulière pour l'installation ou l'utilisation du projet. De plus, ajoutez tout changement majeur depuis la version précédente.

examples.py : Un fichier script Python qui donne des exemples simples sur la façon d'utiliser les projets.

How to Contribute : Cette section doit indiquer comment les nouveaux contributeurs au projet peuvent commencer à contribuer.

29.8 Les projets publics et open source

Les projets publics et open source sont des projets destinés à être partagés avec un grand groupe d'utilisateurs et peuvent impliquer de grandes équipes de développement. Ces projets doivent accorder une priorité aussi élevée à la documentation du projet qu'au développement du projet lui-même. Voici quelques-unes des parties qu'il est recommandé d'ajouter au projet :

Readme : Un bref résumé du projet et de son objectif. Incluez toute exigence particulière pour l'installation ou le fonctionnement du projet. De plus, ajoutez tout changement majeur depuis la version précédente. Enfin, ajoutez des liens vers d'autres documents, des rapports de bogues et toute autre information importante pour le projet [6].

How to Contribute : Cette section doit indiquer comment les nouveaux contributeurs au projet peuvent aider. Cela inclut le développement de nouvelles fonctionnalités, la correction de problèmes connus, l'ajout de documentation, l'ajout de nouveaux tests ou le signalement de problèmes.

Code of Conduct : Définit la manière dont les autres contributeurs doivent se comporter lorsqu'ils développent ou utilisent votre logiciel. Il indique également ce qui se passera si ce code n'est pas respecté [7].

Licence : Un fichier en texte clair qui décrit la licence utilisée par votre projet.

docs : Un répertoire qui contient de la documentation supplémentaire.

6. Voici un excellent tutoriel sur ce qui doit être inclus dans votre fichier `readme` : `https://dbader.org/blog/write-a-great-readme-for-your-github-project`

7. Voir *Ajouter un code de conduite à son projet* : `https://docs.github.com/fr/communities/setting-up-your-project-for-healthy-contributions/adding-a-code-of-conduct-to-your-project`

29.9 Outils et ressources pour aider à la rédaction de la documentation

Documenter votre code, en particulier les projets de grande envergure, peut être décourageant. Heureusement, il existe des outils et des références pour vous aider (voir tableau « *Outils pour la rédaction de la documentation* » page 190).

Outils	Description
`Sphinx`	Une collection d'outils pour générer automatiquement de la documentation dans plusieurs formats. – `https://www.sphinx-doc.org/en/master/`
`Epydoc`	Outil permettant de générer de la documentation API pour les modules `Python` à partir de leurs *docstrings*. – `https://epydoc.sourceforge.net/`
`Read The Docs`	Création, versionnement et hébergement automatiques de vos documents pour vous. – `https://readthedocs.org/`
`Doxygen`	Un outil pour générer de la documentation qui supporte Python ainsi que de nombreux autres langages. – `https://www.doxygen.nl/manual/docblocks.html`
`MkDocs`	Un générateur de site statique pour aider à construire la documentation d'un projet en utilisant le langage `Markdown`. – `https://www.mkdocs.org/`
`pycco`	Un générateur de documentation « *rapide et sale* » qui affiche le code et la documentation côte à côte. – `https://pycco-docs.github.io/pycco/`
`doctest`	Un module de la bibliothèque standard pour exécuter des exemples d'utilisation en tant que tests automatisés. – `https://docs.python.org/3/library/doctest.html`

TABLE 29.2 – Outils pour la rédaction de la documentation

29.10 Par où commencer ?

La documentation des projets a une progression simple en fonction de l'avancée du projet :

1. Aucune documentation
2. Un peu de documentation
3. Documentation complète
4. Bonne documentation
5. Excellente documentation

De prime abord la quantité de travail nécessaire à la documentation du code peut apparaître monstrueuse, mais une fois que vous avez commencé à documenter votre code, il devient plus facile de continuer.

Sources pour la rédaction de ce chapitre

- *Documenting Python Code : A Complete Guide* (trad : « *Documenter le code Python : Un guide complet* »), par James MERTZ - 25 juillet 2018 - `https://realpython.com/documenting-python-code/`

- *Defining Your Own Python Function* (trad. *Définir votre propre fonction* `Python`, par John Sturtz - 9 mars 2020 - `https://realpython.com/defining-your-own-python-function/`
- *PEP 257 – Docstring Conventions* (trad : « *Conventions relatives à la docstring* »), par David Goodger et Guido van Rossum - 29 mai 2001 - `https://peps.python.org/pep-0257/`

Chapitre 30

Structurer une application Python

Lorsque votre projet évolue au point de s'étendre sur plusieurs centaines de lignes de code, il devient alors judicieux de découper celui-ci en plusieurs fichiers. De tels fichiers sont appelés des modules qui peuvent être importées dans un autre module ou dans le module main (le module principal du projet) [1]. Chaque module devra porter l'extension `.py`. la règle à suivre est de classer ces fichiers selon leur thématique. Par exemple on placera toutes les fonctions est classe qui traitent de la gestion d'une base de données ensemble, ou bien s'agissant de l'interface graphique.

Les imports de ces modules seront réalisés de la façon suivante :

```
import module
from module import objet
```

Voyons un exemple fictif de structuration de projet :

```
> .venv
> mon_projet
    - main.py
    - gui.py
    - database.py
> tests
    - test1.py
    - test2.py
- LICENSE.txt
- README.md
- setup.cfg
- etc.
```

Les fichiers contenus dans le répertoire `mon_projet` sont répartis selon des thématiques bien précises. Le fichier `setup.cfg` est le fichier de configuration du projet.

En `Python`, un répertoire est considéré comme un paquet qui va contenir plusieurs modules [2]. A chaque niveau de répertoire on trouve souvent un module nommé `__init__.py`. Depuis la version `3.4` de `Python` il n'est plus obligatoire d'avoir un tel fichier, mais ne pas l'insérer peut avoir des effets de bord néfastes au bon déroulement du code. Attention toutefois à ne pas insérer un tel fichier à la racine du projet.

1. Voir à ce sujet le chapitre consacré à la modularité page 99 ainsi que la documentation officielle à l'adresse suivante : `https://docs.python.org/fr/3/tutorial/modules.html`
2. Voir : `https://docs.python.org/fr/3/tutorial/modules.html#packages`

L'instruction `from paquet import module` exécute le fichier `__init__.py` et l'éventuel code qui se trouve à l'intérieur. Ce code ne s'exécutera qu'une seule fois au cours de l'exécution du programme.

30.1 Pour une bonne pratique, en résumé

La structuration d'un projet en `Python` est importante pour faciliter sa maintenance, son évolution et sa compréhension. Voici, en résumé, un guide de bonne pratique pour bien structurer un projet :

1. Créer un répertoire pour stocker l'ensemble du projet, en le nommant de manière aisément identifiable.
2. Diviser son code en modules qui doivent avoir des fonctions ou des classes spécifiques, cohérentes entre elles. Il s'agit de les organiser dans des répertoires selon leur fonctionnalité.
3. Utiliser des packages (répertoires contenant un fichier `__init__.py`) qui permettent d'organiser des modules connexes et d'importer facilement les classes et fonctions du projet.
4. Utiliser des environnement virtuels (`virtualenv`) qui seront des environnements `Python` isolés permettant de travailler sur plusieurs projets avec des versions de `Python` et des bibliothèques différentes. Cela évitera les conflits de bibliothèques entre les projets et facilitera la gestion des dépendances.
5. Utiliser un fichier `requirements.txt` ou équivalent qui listera toutes les dépendances du projet. Il permet aux autres développeurs de savoir quelles bibliothèques sont nécessaires pour faire fonctionner le projet et ainsi installer les mêmes versions.
6. Utiliser des tests unitaires, fonctions permettant de vérifier que le code fonctionne correctement. Écrire des tests pour chaque module et les exécuter régulièrement.
7. Utiliser un système de contrôle de version, tel que `Git` pour suivre les modifications apportées au code. Cela permet de revenir à une version antérieure si nécessaire et de travailler en collaboration avec d'autres développeurs.

Chapitre 31

Les versions de Python

Chaque année sortent de nouvelles versions de `Python`, chaque nouvelle version se voit dotée d'un certains nombres d'améliorations par rapport à la précédente. Nous pouvons prendre connaissance de ces améliorations via les PEP (*Python Enhancement Proposals* ou *Propositions d'amélioration du langage* ***`Python`***). Connaître ces évolutions fait partie des impératifs pour un développeur afin de demeurer au fait des spécificités du langage et de son évolution. Certaines de ces évolutions peuvent en effet devenir majeurs, mais d'autres, plus mineures, seront finalement peu utilisées.

Quand une formation recommande une version précise de `Python`, cela signifie que l'on se doit d'utiliser la dite version ou bien les suivantes. La formation étant enregistrée un jour et dispensée durant plusieurs années, nous pouvons utiliser des versions ultérieurs car `Python` garantit une excellente compatibilité ascendante. D'ailleurs, pour apprendre le langage on recommande en généralement d'utiliser la version stable la plus récente, (ou éventuellement l'avant-dernière si la dernière est très récente), car au cours de notre apprentissage nous serons amenés à consulter des ressources qui ne seront pas compatibles avec une version trop ancienne.

Documenter votre code, en particulier les projets de grande envergure, peut être décourageant. Heureusement, il existe des outils et des références pour vous aider à démarrer :

31.1 Version `3.9` (sortie en octobre 2020)

Les principales améliorations :

- l'union des dictionnaires [1] ;
- les annotations de type [2] ;
- la suppression des préfixes et suffixes des chaînes de caractères [3].

31.2 Version `3.10`

Les principales améliorations :

- la correspondance structurelle [4] ;

1. Voir page 75 pour la fusion des dictionnaires et la page 74 pour l'ajout d'éléments à l'aide du « *pipe* ».
2. Voir les types de paramètres et de retour des fonctions page 55.
3. Voir les fonctions `removesuffix()` et `removeprefix()` à la page 79.
4. Voir la correspondance structurelle page 62.

- l'amélioration des messages d'erreur de syntaxe, qui sont beaucoup plus détaillés et explicites [5] ;
- l'union de types [6] ;
- la longueur des `zip` [7].

31.3 Version `3.11` (sortie le 24 octobre 2022)

Les principales améliorations :
- les messages d'erreurs ;
- les groupes de tâches asynchrones et d'exceptions ;
- `TOML` et `tomlib` ;
- la vérification des types ;
- un démarrage et une exécution du code plus rapides ;
- les exceptions à coût nul.

5. Cf. PEP 626 – Precise line numbers for debugging and other tools - `https://peps.python.org/pep-0626/`
6. Cf. PEP 604 – Allow writing union types as `X | Y` - https ://peps.python.org/pep-0604/
7. Cf. PEP 618 – Add Optional Length-Checking To zip - `https://peps.python.org/pep-0618/`

Chapitre 32

Les environnements Python

32.1 Les environnements avec Python

Les projets dépendent de la version de `Python` ainsi que des versions des packages (modules) externes (`Django`, `Kivy`, `Pyside`, etc.). Il devient alors nécessaire d'utiliser des environnements virtuels (outil `venv` et de faire appel à la gestion des configurations (outil `pyenv`. L'environnement virtuel offre une configuration isolée pour un projet, permettant alors de garantir une maîtrise des versions à la fois des modules et de `Python`. La gestion des configurations c'est le fait d'avoir plusieurs configurations de présentes sur sa machine.

De façon générale deux à trois configurations sont utilisée sur une même machine, et on utilise un environnement virtuel par projet. Les environnements virtuels sont relativement aisés à mettre en place avec `Pycharm` et `Visual Studio Code`.

Dans ce chapitre nous allons aborder les divers outils indispensables dans la gestions des environnements avec `Python`, tels que :

le terminal : la ligne de commande ;

`pip` : pour installer des packages ;

`venv` : pour gérer les environnements virtuels ;

`pyenv` : pour gérer les configurations de `Python`.

32.2 Le terminal et l'outil pip

Exécuter un fichier `Python` depuis un terminal :

```
$ python main.py
```

Vérifier l'installation de `pip` ainsi que la version :

```
$ pip --version
```

Lister l'ensemble des packages *installés* sur la machine :

```
$ pip list
```

Installer un module avec `pip` :

```
$ pip install nom_module
```

Mise à jour de `pip` :

```
$ python -m pip install --upgrade pip
```

Désinstaller un module avec `pip` :

```
$ pip uninstall nom_module
```

32.3 Les environnements virtuels (`venv`)

Au préalable, s'assurer de la présence des paquets `python3-venv` ou bien `python3.10-venv` (pour la version `3.10`.

Avec le terminal

1. Créer un répertoire pour notre projet.
2. Créer un fichier `main.py`.
3. Se placer dans le répertoire du projet.
4. Lancer la commande de création de l'environnement virtuel (cf. infra).
5. Activer l'environnement virtuel : `$ source venv/bin/activate` (`$ deactivate`, pour sortir de l'environnement virtuel).
6. Vérifier que nous sommes bien dans l'environnement virtuel et lister les modules utilisés par ce même environnement : `(venv)$ pip list`.

Commande de création de l'environnement virtuel :

```
$ python -m venv venv
```

Le second `venv` de cette commande correspond au répertoire crée qui contiendra la configuration de l'environnement virtuel. Par convention il est souvent nommé `venv`.

A noter que si un environnement virtuel est compatible sous *GNU/Linux* est compatible avec *Mac*, il ne l'est pas avec *Windows*.

Avec `Visual Studio Code`

Ouvrir un projet avec son environnement virtuel :

```
Fichier > Ouvrir le dossier
```

A l'exécution de la commande `Run` ce sera bien l'environnement virtuel qui sera utilisé. Vérifier toutefois le nom de l'interpréteur utilisé.

Avec `Pycharm`

C'est au moment de la création du projet que l'on définit l'environnement virtuel qui sera celui du projet, à l'aide de l'interface `New Project`.

1. Choisir l'emplacement (`Location`)
2. Choisir l'interpréteur en cochant `New environnement using` et sélectionner `Virtualenv` qui utilise `venv`. `Location` permet de définir le répertoire où se situera la configuration virtuelle. Avec `Base interpreter` on choisit la version de `Python` à utiliser pour notre environnement virtuel.

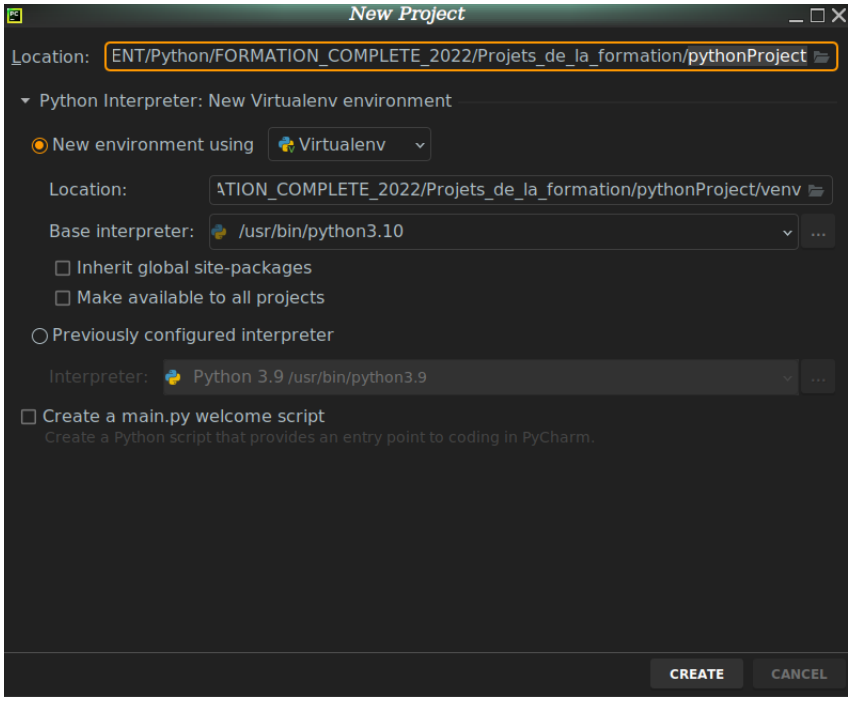

FIGURE 32.1 – Interface *New Project*

3. Choix optionnels :

   Inherit global site-packages : permet d'utiliser tous les paquets présents sur notre système.

   More available to all projects : permet de partager cet environnement virtuel avec d'autres projets.

Vérifier que nous nous situons bien dans cet environnement virtuel en ouvrant le terminal dans `Pycharm`

32.4 Gérer plusieurs configurations (pyenv)

Installation de `pyenv` sous `Debian GNU/Linux`

```
# apt-get install-y make build-essential libssl-dev zliblg-dev libbz2-dev
 libreadline-dev libsqlite3-dev wget curl llvm libcurses5--dev libncurses-dev
 xz-utils tk-dev libffi-dev liblzma-dev python3-openssl
$ curl -L https://raw.githubusercontent.com/yyuu/pyenv-installer/master/
 binpyenv-installer | bash
```

Il est possible qu'à l'issue de l'installation on vous demande d'ajouter `pyenv` à votre `Path`, il suffit alors de suivre les instructions fournies. Un ajout des lignes proposées dans les fichiers `~/.profile` et `~/.bash.rc` ou `~/.zshrc` devrait suffire. Pour vérifier l'installation et la version de `pyenv` :

```
$ pyenv
```

Les principales commandes pour `pyenv`

Voir quelle version de `Python` est utilisée :

```
$ pyenv versions
```

Exemple de retour :

```
* system (set by /home/krystof/.pyenv/version)
3.8.0
3.11.0b5
```

L'étoile (`*`) indique la version utilisée. Ici, il s'agit de la version utilisée par défaut par le système. Suivi de la commande `$ python --version` nous connaîtrons la version de `Python` par défaut du système.

Lister toutes les versions de `Python` qu'il est possible d'installer avec `pyenv` :

```
$ pyenv install -l
```

Pour installer une version de `Python` (ici la `3.8.0`) :

```
$ pyenv install 3.8.0
```

Mais pour que cette version puisse être utilisée par défaut :

```
$ pyenv 3.8.0
$ pyenv rehash
```

Les deux commandes qui permettent d'effectuer des changements de version :

```
$ pyenv rehash
$ pyenv global [version]
```

Chapitre 33

Approfondir ses connaissances en algorithmie

33.1 Codingame

Le site Codingame offre des exercices sous forme d'énigmes algorithmiques [1]. Il est possible de s'y inscrire via son profil GitHub.

Pour afficher des sorties à l'aide de la fonction `print()` :

```
1 import sys
2 print("Ce_que_l'on_souhaite_afficher", file=sys.stderr, flush=True)
```

1. https://www.codingame.com/home

Chapitre 34

Les messages d'erreur

34.1 `IndexError`

34.2 `KeyError`

34.3 `SyntaxError`

34.4 `TypeError`

34.5 `ValueError`

Septième partie

Python applicatif

Chapitre 35

Manipuler des fichiers .pdf

35.1 La bibliothèque PyPDF2

La bibliothèque PyPDF2 est l'une des meilleures bibliothèques pour manipuler des fichiers .pdf. Elle permet à la fois de lire et d'écrire de tels fichiers, d'en extraire du texte et de combiner ensemble plusieurs fichiers. Il est aussi possible de réaliser une rotation des pages et des les superposer. Par contre, on ne peut rajouter du texte ou des marges.

Installation :

```
$ pip install PyPDF2==1.26.0
```

35.2 Un premier script : combiner deux fichiers .pdf ensemble

```
"""Combiner deux fichiers .pdf.
Ce code suppose que vous disposiez de deux fichiers .pdf,
respectivement nommés 'fichier1.pdf' et 'fichier2.pdf'.
"""

from PyPDF2 import PdfFileWriter, PdfFileReader

# Création d'un fichier .pdf vide (en mémoire) :
contenu_sortie = PdfFileWriter()

# "rb" ouvrir en lecture un fichier binaire :
fichier_pdf_1 = open("fichier1.pdf", "rb")
fichier_pdf_2 = open("fihcier2.pdf", "rb")
# Lire les informations des fichiers
reader_pdf1 = PdfFileReader(fichier_pdf_1)
reader_pdf2 = PdfFileReader(fichier_pdf_2)
print(f"Nbre de pages du fichier recap : "
      f"{reader_pdf2.getNumPages()}")
# Combiner
# 1 - le fichier "fichier1.pdf" (0 pour première page) :
contenu_sortie.addPage(reader_pdf1.getPage(0))
# Et si nous voulions tourner le fichier de 90 degrés
# dans le sens des aiguilles d'une montre :
# contenu_sortie.addPage(reader_pdf1.getPage(0)).rotateClockwise(90)
```

```
25 # 2 - "fichier2.pdf" (toutes les pages via une boucle for) :
26 for i in range(reader_pdf2.getNumPages()):
27     contenu_sortie.addPage(reader_pdf2.getPage(i))
28
29 # Ecrire notre fichier combiné :
30 fichier_combine = open("fichier_combine.pdf", "wb")
31 contenu_sortie.write(fichier_combine)
32
33 # Fermeture des fichiers
34 fichier_pdf_1.close()
35 fichier_pdf_2.close()
36 fichier_combine.close()
```

35.3 Extraire le texte d'un fichier `.pdf`

L'extraction de base est assez simple :

```
1 from PyPDF2 import PdfFileWriter, PdfFileReader
2
3 f = open("mon_fichier.pdf", "rb")
4 reader = PdfFileReader(f)
5 page0 = reader.getPage(0)  # Extraction de la première page
6 texte = page0.extractText()  # Extraction du texte de la première page
7 f.close()
8
9 print(texte)  # Affichage du texte extrait
```

Nous pouvons voir, qu'en sortie, le texte est extrait de manière brute, que nous avons perdu le style du fichier `.pdf` (on ne fait plus de distinction entre les titres et le corps du texte, par exemple). Ce qui est normal, puisque nous extrayons des chaînes de caractères. Cependant nous avons un autre problème concernant l'affichage d'un certain nombre de caractères spéciaux, ainsi qu'avec les caractères accentués. De plus, si le fichier `.pdf` contient du texte dans les *headers* celui-ci apparaît à la fin et non-plus en en-tête.

Concernant les caractères spéciaux et caractères accentués, il va falloir étudier la sortie brute et à l'aide de la méthode replace(), venir remplacer un à un chaque caractère (user du copier/coller depuis la sortie). Notre code pourrait être alors le suivant :

```
1 ...
2 texte = page0.extractText()  # Extraction du texte de la première page
3 # On remplace chaque caractère un à un:
4 texte = texte.replace("Û", '"').replace("'", "è")  # etc...
5 f.close()
6 ...
```

Chapitre 36

Gérer des courriels

36.1 Envoyer un courriel

Exemple avec un compte `Gmail`

Il va tout d'abord nous falloir configurer le compte pour cela. Se rendre sur la `Gestion de compte`, puis à la section `Sécurité`. A `Connexion à Google`, activer la `Validation en deux étapes`. Ensuite :

`Mots de passe des applications > Sélectionner une application > Autre (personnalisé)`

Donner un nom (Exemple : « *Avec Python* ») et cliquer sur `Générer`. Un mot de passe de seize caractères est alors généré, et c'est ce mot de passe que nous utiliserons dans nos codes.

Chercher ensuite (via un moteur de recherche) la configuration `SMTP`[1] de `Gmail`. Vous devriez trouver les éléments de configuration suivants :

SMTP Server : `smtp.gmail.com`

Server port : `587`

Envoyer un courriel simple

Créer en premier lieu un projet contenant deux fichiers : `emails.py` (celui de notre programme principal) et config_mail.py (fichier qui contiendra les paramètres et les informations confidentielles comme le mot de passe).

Nous allons, pour mener à bien notre objectif, utiliser la bibliothèque `smtplib`.

Voyons le code de nos deux fichiers :

```
1 """Envoyer des mails avec gmail."""
2
3 import smtplib
4 import config_mail
5
6
7 def envoyer_mail(email_destinataire, message):
8     serveur_mails = smtplib.SMTP(config_mail.config_server,
9                                  config_mail.config_server_port)
```

1. *Simple Mail Transfert Protocol* (ou *Protocole simple de transfert de courrier* : protocole de communication utilisé pour transférer le courrier électronique vers les serveurs de messagerie électronique.

```
10      serveur_mails.starttls()  # Protocole de sécurité
11      # Se connecter :
12      serveur_mails.login(config_mail.config_email,
13                          config_mail.config_password)
14      # Envoyer mail :
15      serveur_mails.sendmail(config_mail.config_email,
16                             email_destinataire, message)
17      serveur_mails.quit()  # Fermeture de la connexion
18
19
20  envoyer_mail("adresse_mail@destinataire.com",
21               "Ceci est un test avec Python")
```

Listing 36.1 – `mails.py`

```
1  config_email = "mail_émeteur@gmail.com"
2  config_password = "les_16_caractères_du_mot_de_passe_généré"
3  config_server = "smtp.gmail.com"
4  config_server_port = 587
```

Listing 36.2 – `config_mail.py`

Pour envoyer un message contenu sur plusieurs lignes, il suffit d'attribuer le corps de notre message à une variable que l'on passera en paramètre lors de l'appel de la fonction `envoyer_email()` :

```
1  corps_du_message = """Bonjour,
2  Ceci est un message rédigé à l'aide de Python.
3  A bientôt.
4  """
5
6  envoyer_mail("adresse_mail@destinataire.com", corps_du_message)
```

Listing 36.3 – Message sur plusieurs lignes

Un message *mulitpart*

Il s'agit d'un message auquel on ajoute des paramètres comme le sujet du courriel par exemple. Pour cela nous allons importer deux méthodes : `MIMEMultipart` et `MIMEText`.

```
1  from email.mime.multipart import MIMEMultipart
2  from email.mime.text import MIMEText
```

Listing 36.4 – Importer `MIMEMultipart` et `MIMEText`

`MIME`, pour « *Multipurpose Internet Mail Extensions* » (ou *Extensions multifontions du courrier Internet*), est un standard Internet qui étend le format de données des courriels pour supporter des textes en différents codages des caractères autres que l'*ASCII*, des contenus non textuels, des contenus multiples et des informations d'en-tête. Les courriels, envoyés via le protocole `SMPTP` au format `MIME` sont appelés courriels `SMTP/MIME`.

```
"""Envoyer des mails avec gmail, et plusieurs paramètres."""

import smtplib
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
import config_mail


def envoyer_mail(email_destinataire, sujet, message):
    multipart_message = MIMEMultipart()
    multipart_message["Subject"] = sujet
    multipart_message["From"] = config_mail.config_email
    multipart_message["To"] = email_destinataire
    # Attaché du texte plein
    # ("html" est possible - Intéressant pour les styles) :
    multipart_message.attach(MIMEText(message, "plain"))
    serveur_mails = smtplib.SMTP(config_mail.config_server,
                                 config_mail.config_server_port)
    serveur_mails.starttls()  # Protocole de sécurité
    # Se connecter :
    serveur_mails.login(config_mail.config_email,
                        config_mail.config_password)
    # Envoyer mail :
    serveur_mails.sendmail(config_mail.config_email,
                           email_destinataire,
                           multipart_message.as_string())
    serveur_mails.quit()  # Fermeture de la connexion


message_email = """Bonjour,

Voici un message généré avec Python.

Au revoir..."""
objet_email = "Email depuis Python"
mail_destinataire = input("Adresse mail du destinataire : ")
envoyer_mail(mail_destinataire, objet_email, message_email)
```

Listing 36.5 – Un courriel avec divers paramètres (*multipart*)

Chapitre 37

Le *scraping*

37.1 Qu'est-ce que le *scraping* ?

Le *scraping* permet d'extraire des données depuis des sites Web. Pour cela deux bibliothèques sont nécessaires : `requests` et `Beautiful Soup`.

- `requests` [1] permet de réaliser ses requêtes `HTTP` et de récupérer le code source des pages `HTML` [2].
- `Beautiful Soup` va nous permettre de parcourir le code `HTML` afin d'y récupérer les données que l'on souhaite (comme le titre, du texte, la source d'une image, etc.).

Tout ce qui est visible sur le Web peut potentiellement fournir des informations que l'on peut extraire.

Depuis un navigateur on accède à un serveur via une requête `HTTP`. Ce serveur nous retourne des informations au format `HTML`, et le navigateur affiche alors une page Internet. Avec `Python` il est possible de développer une application qui va utiliser la bibliothèque `requests` afin de lancer des requêtes `HTTP` et ainsi de pouvoir récupérer les données `HTML` afin de les analyser et d'y extraire les informations voulues, cela à l'aide la bibliothèque `Beautiful Soup`. Il s'agit d'une méthode classique, mais il faut savoir qu'il existe aussi le *framework* `Scrapy` [3] qui permet à la fois de lancer des requêtes `HTTP`, de récupérer les données du serveur et puis de les analyser. Avec `requests` et `Beautiful Soup` on demeure dans une démarche algorithmique.

Il est à noter que certains sites se prémunissent contre le *scraping*. Il est toutefois possible de contourner ces protections, et c'est ce que l'on nomme le *Headless Browsing* [4], mais cela s'apparente à du hacking.

37.2 La bibliotèque `requests` : faire des appels réseaux

Ce module fournit des fonctionnalités permettant de lancer des requêtes `HTTP`.

Installer le module :

```
$ pip install requests
```

1. `https://requests.readthedocs.io/en/latest/`
2. L'`HTML` est un langage de balisage qui permet de décrire du contenu, notamment pour réaliser des pages Web. Cf `https://fr.wikipedia.org/wiki/Hypertext_Markup_Language`
3. `https://scrapy.org/`
4. `https://fr.wikipedia.org/wiki/Navigateur_headless`

```
# FAIRE DES APPELS RESEAUX AVEC REQUESTS

# https://codeavecjonathan.com/res/programmation.txt
# https://codeavecjonathan.com/res/pizzas1.json
# https://codeavecjonathan.com/res/exemple.html

import requests
import json

# Récupérer du texte :
reponse = \
    requests.get(
        "https://codeavecjonathan.com/res/programmation.txt")
reponse.encoding = "utf-8"
print(reponse.text)

# Récupérer des données au format .json :
reponse = \
    requests.get(
        "https://codeavecjonathan.com/res/pizzas1.json")
# Pour une gestion de l'erreur si la page web
# ne devait pas exister :
if reponse.status_code == 200:
    reponse.encoding = "utf-8"
    print(reponse.text)
    pizzas = json.loads(reponse.text)
    print(f"Nombre de pizzas : {len(pizzas)}")
else:
    print(f"ERREUR Code : {reponse.status_code}")

# Récupérer le code source d'un site :
reponse = \
    requests.get(
        "https://codeavecjonathan.com/res/exemple.html")
reponse.encoding = "utf-8"
print(reponse.text)
```

Listing 37.1 – Un premier exemple avec le module `requests`

La fonction `.get()` permet de récupérer du code `HTML`.

```
# TELECHARGER UNE IMAGE AVEC REQUESTS

# https://codeavecjonathan.com/res/papillon.jpg

import requests

reponse = \
    requests.get(
        "https://codeavecjonathan.com/res/papillon.jpg")
```

```
10 if reponse.status_code == 200:
11     # Ouverture en écriture binaire :
12     f = open("papillon.jpg", "wb")
13     f.write(reponse.content)
14     f.close()
15     print("Ecriture terminée")
16 else:
17     print(f"ERREUR : {reponse.status_code}")
```

Listing 37.2 – Télécharger une image avec le module `requests`

37.3 La bibliothèque `Beautiful Soup` : extraire des données depuis un simple fichier `.html`

```
$ pip install beautifulsoup4
```

Commençons par créer un projet, un répertoire contenant à la fois le fichier .py qui contiendra notre code et un fichier .html qui contiendra les éléments à extraire.

Le fichier `livre.html` :

```
<h1>Pour Qui sonne le Glas</h1>

<table border="1">
    <tr>
        <th>Auteur</th>
        <th>Date de publication</th>
    </tr>
    <tr>
        <td>Ernest Hemingway</td>
        <td>1940</td>
    </tr>
</table>

<p>'Pour qui sonne le glas' est un roman d'Ernest Hemingway fortement
inspiré de son vécu de journaliste pendant la guerre civile espagnole. </p>

<img src="https://upload.wikimedia.org/wikipedia/commons/thumb/2/28/ErnestHe
mingway.jpg/220px-ErnestHemingway.jpg">
```

Il est parfois cependant difficile de lire le code source d'un fichier `.html`. Nous pouvons alors faire appel à un formateur de code `html` [5].

De même le code récupéré à l'aide du module `requests` est illisible. Il va donc falloir *parser* [6] ce qui a été récupéré à l'aide du module `BeautifulSoup`. On *parse* du `HTML` avec les attributs `HTML class` et `id`.

Le fichier `extraction.py` :

5. Voir par exemple : `https://www.site24x7.com/fr/tools/html-formateur.html`
6. Parcourir le contenu d'un fichier en l'analysant pour vérifier sa syntaxe ou en extraire des éléments.

```
# Import de la classe 'BeautifulSoup':
from bs4 import BeautifulSoup

# Lecture des données HTML :
f = open("livre.html", "r")  # Ouverture fichier .html en local
html_content = f.read()  # Lecture du fichier .html
f.close()
soup = BeautifulSoup(html_content, "html.parser")

# Extractions:
titre_h1 = soup.find("h1")  # Chercher l'occurence "h1"
description_p = soup.find("p")  # Un texte
image_src = soup.find("img")  # Source de l'image
texte_titre = titre_h1.text  # Récupérer le texte

# Affichage des extractions:
print(f"Titre de la page HTML: {texte_titre}")
# Ou bien -> print(f"Titre de la page HTML: {titre_h1.text}")
print(f"Texte descritptif: {description_p.text}")
print(f"Le fichier source de l'image est: {image_src['src']}")
```

Listing 37.3 – Exemple simple d'extraction avec `Beautiful Soup`

La ligne :

```
soup = BeautifulSoup(html_content, "html.parser")
```

permet de créer un objet « *soup* » à partir du `HTML` récupéré. La variable `soup` possède alors toutes les fonctions qui facilitent l'obtention de données à partir de l'`HTML`.

37.4 Transformer les données à l'aide du module `csv`

Il existe différentes façons de transformer des données. Cela va dépendre du type de données et du format souhaité en sortie.

Voir la lecture et l'écriture de fichiers `.csv`[7]

La méthode `.reader`

Va analyser tout le texte dans un fichier `.csv`, le *parser* ligne par ligne et convertir chaque ligne dans une liste de chaîne de caractères. Pour le délimiteur de colonnes, on peut choisir n'importe quel signe, mais la virgule est communément employée.

Prenons comme exemple le fichier `.csv` suivant, contenant les noms de communes du canton de Lescar[8], avec les distances qui séparent chaque commune de celle de Lescar :

```
commune,distance_km_depuis_Lescar
Arbus,8.8
Artiguelouve,4.6
```

7. https://docs.python.org/fr/3/library/csv.html
8. https://fr.wikipedia.org/wiki/Canton_de_Lescar

```
Aussevielle,5.4
Beyrie-en-Béarn,7.3
Bougarber,9
Caubios-Loos,11
Denguin,7.5
Lescar,0
Lons,4.4
Momas,15
Poey-de-Lescar,4.8
Sauvagnon,10
Siros,4.8
Uzein,9.7
```

```
1 import csv
2
3 with open('communes_canton_Lescar.csv') as fichier_csv:
4     reader = csv.reader(fichier_csv, delimiter=',')
5     for ligne in reader:
6         print(ligne)
```

Listing 37.4 – La méthode `.reader()`

En sortie, cela donnera :

```
['commune', 'distance_km_depuis_Lescar']
['Arbus', '8.8']
['Artiguelouve', '4.6']
['Aussevielle', '5.4']
['Beyrie-en-Béarn', '7.3']
['Bougarber', '9']
['Caubios-Loos', '11']
['Denguin', '7.5']
['Lescar', '0']
['Lons', '4.4']
['Momas', '15']
['Poey-de-Lescar', '4.8']
['Sauvagnon', '10']
['Siros', '4.8']
['Uzein', '9.7']
```

La méthode `.DictReader()`

Avec la méthode `.reader` la ligne d'en-tête est considérée à l'instar des autres. Avec la méthode `.DictReader()` la ligne d'en-tête va être utilisée pour générer les clés du dictionnaire qui contiendra les valeurs de chaque ligne.

```
1 import csv
2
3 with open('communes_canton_Lescar.csv') as fichier_csv:
```

```
4       reader = csv.DictReader(fichier_csv, delimiter=',')
5       for ligne in reader:
6           print(ligne) # Sous forme d'un dictionnaire
7           # Générer un texte:
8           print(f"La commune de {ligne['commune']} est distante "
9                 f"de {ligne['distance_km_depuis_Lescar']} km "
10                f"depuis Lescar")
```

Listing 37.5 – La méthode `.DictReader()`

En sortie :

```
{'commune': 'Arbus', 'distance_km_depuis_Lescar': '8.8'}
La commune de Arbus est distante de 8.8 km depuis Lescar
{'commune': 'Artiguelouve', 'distance_km_depuis_Lescar': '4.6'}
La commune de Artiguelouve est distante de 4.6 km depuis Lescar
{'commune': 'Aussevielle', 'distance_km_depuis_Lescar': '5.4'}
La commune de Aussevielle est distante de 5.4 km depuis Lescar
{'commune': 'Beyrie-en-Béarn', 'distance_km_depuis_Lescar': '7.3'}
La commune de Beyrie-en-Béarn est distante de 7.3 km depuis Lescar
{'commune': 'Bougarber', 'distance_km_depuis_Lescar': '9'}
La commune de Bougarber est distante de 9 km depuis Lescar
{'commune': 'Caubios-Loos', 'distance_km_depuis_Lescar': '11'}
La commune de Caubios-Loos est distante de 11 km depuis Lescar
{'commune': 'Denguin', 'distance_km_depuis_Lescar': '7.5'}
La commune de Denguin est distante de 7.5 km depuis Lescar
{'commune': 'Lescar', 'distance_km_depuis_Lescar': '0'}
La commune de Lescar est distante de 0 km depuis Lescar
{'commune': 'Lons', 'distance_km_depuis_Lescar': '4.4'}
La commune de Lons est distante de 4.4 km depuis Lescar
{'commune': 'Momas', 'distance_km_depuis_Lescar': '15'}
La commune de Momas est distante de 15 km depuis Lescar
{'commune': 'Poey-de-Lescar', 'distance_km_depuis_Lescar': '4.8'}
La commune de Poey-de-Lescar est distante de 4.8 km depuis Lescar
{'commune': 'Sauvagnon', 'distance_km_depuis_Lescar': '10'}
La commune de Sauvagnon est distante de 10 km depuis Lescar
{'commune': 'Siros', 'distance_km_depuis_Lescar': '4.8'}
La commune de Siros est distante de 4.8 km depuis Lescar
{'commune': 'Uzein', 'distance_km_depuis_Lescar': '9.7'}
La commune de Uzein est distante de 9.7 km depuis Lescar
```

37.5 Les problèmes éthiques de l'extraction *Web*

Pour visualiser ce qui est extractables des sites *Web* il faut se référer à leur fichier `robots.txt` [9] s'il existe. Il faut alors ajouter l'extension `/robots.txt` après l'*URL* d'un site. Exemple :

```
https://www.docstring.fr/robots.txt
```

9. `https://robots-txt.com/`

37.6 Les défis de l'extraction *Web*

- La variété des sites *Web* avec des instructions `HTML` différentes.
- La durabilité des scripts, en lien avec les changements des structures `HTML` des sites.

Chapitre 38

Télécharger des fichiers audio et vidéo depuis *Youtube*

Nous allons coder un petit projet permettant de télécharger une vidéo depuis *Youtube*, en sélectionnant la meilleure qualité qui soit.

38.1 Création du projet et installation des modules et paquets nécessaires

Tout d'abord il nous sera nécessaire de créer un projet à l'aide d'un environnement virtuel. Ensuite nous installerons dans cet environnement les modules suivants :

Pytube [1] : c'est le module incontournable pour télécharger des fichiers audio et vidéo depuis *Youtube*. Ce module se télécharge à partir des sources [2].

ffmpeg.python [3] : ce module permet de regrouper des fichiers audio et vidéo séparés. Son utilisation nécessite l'installation sur le système du package *ffmpeg*.

Installons donc tout cela :

```
# pip3 install git+https://github.com/pytube/pytube
# aptitude install ffmpeg
# pip3 install ffmpeg-python
```

38.2 Le code du projet

Le projet se compose de deux fichiers : *main.py* et *yt_ downloader.py*.

```
1 import yt_downloader
2
3 urls = ("https://www.youtube.com/watch?v=BNdcOOsrHOs&list=
4             PL44A0E4C7FFA5B749&index=1",
5         "https://www.youtube.com/watch?v=NxjuOJkXAq0&list=
6             PL44A0E4C7FFA5B749&index=2",
7         "https://www.youtube.com/watch?v=qdYwrFRxKH0&list=
```

1. Voir la page *GitHub* du projet : `https://github.com/pytube/pytube`. Pour la documentation on se référera à la page suivante : `https://pytube.io/en/latest/api.html`
2. Voir : `https://github.com/pytube/pytube#installation`
3. Voir la page *Github* du projet : `https://github.com/kkroening/ffmpeg-python`

```
            PL44A0E4C7FFA5B749&index=3",
        "https://www.youtube.com/watch?v=_lZPlQiV4ps&list=
            PL44A0E4C7FFA5B749&index=4",
        "https://www.youtube.com/watch?v=Sg-FFmwT2jE&list=
            PL44A0E4C7FFA5B749&index=5",
        "https://www.youtube.com/watch?v=4jCxP0d5BtI&list=
            PL44A0E4C7FFA5B749&index=6")

for url in urls:
    yt_downloader.download_video(url)
```

Listing 38.1 – Le fichier *main.py*

Vous pouvez bien entendu changer les *url* des pages *Youtube*. Il s'agit ici de six titres du groupe *The Cure* en version acoustique.

```
import os
from pytube import YouTube
import ffmpeg


def progression_telechargement(the_stream, chunk,
                               bytes_remainnig):
    """Affichage en pourcentage de la progression du
    téléchargement.
    """
    # Nbre octets au total - nbre octets restant :
    bytes_telecharges = the_stream.filesize - bytes_remainnig
    pourcentage = bytes_telecharges * 100 / the_stream.filesize
    # int() pour avoir des nbres entiers :
    print(f"Progession du téléchargement : "
          f"{int(pourcentage)}%")


def download_video(url):
    ma_video_yt = YouTube(url)  # Création objet 'YouTube'.
    # Affichage informations :
    print(f"TITRE : {ma_video_yt.title}")
    # Utilisation filtre pour avoir les meilleures résolutions :
    streams_video = ma_video_yt.streams.filter(
        progressive=False, file_extension='mp4',
        type="video").order_by("resolution").desc()
    streams_audio = ma_video_yt.streams.filter(
        progressive=False, file_extension='mp4',
        type="audio").order_by("abr").desc()
    meilleur_stream_video = streams_video[0]
    meilleur_stream_audio = streams_audio[0]
    # Pour afficher la progression du téléchargement :
    ma_video_yt.register_on_progress_callback(
        progression_telechargement)
    # Téléchargement :
```

```
36      print(f"Téléchargement de la vidéo "
37            f"'{ma_video_yt.title}'...")
38      meilleur_stream_video.download("Video")
39      meilleur_stream_audio.download("Audio")
40      # Combiner le fichier vidéo et le fichier audio :
41      audio_filename = os.path.join(
42          "Audio", meilleur_stream_video.default_filename)
43      video_filename = os.path.join(
44          "Video", meilleur_stream_video.default_filename)
45      output_filename = meilleur_stream_video.default_filename
46      ffmpeg.output(ffmpeg.input(audio_filename),
47                    ffmpeg.input(video_filename),
48                    output_filename, vcodec="copy",
49                    acodec="copy",
50                    loglevel="quiet").run(overwrite_output=True)
51      print(f"Fin du téléchargement de la vidéo "
52            f"'{ma_video_yt.title}'")
53      # Suppression des fichiers temporaires :
54      os.remove(audio_filename)
55      os.remove(video_filename)
56      os.rmdir("Audio")
57      os.rmdir("Video")
```

Listing 38.2 – Le fichier *yt_ downloader.py*

38.3 Les extractions

Pour visualiser ce que nous pouvons extraire de l'objet « *Youtube* » (ici, notre objet se nomme `ma_video_yt`), à l'aide du debogger nous pouvons placer un point d'arrêt juste après la création de l'objet (situer le point d'arrêt ligne 22). Nous pouvons alors voir que nous disposons de divers éléments d'informations que nous pouvons obtenir de la façon suivante :

```
ma_video_yt.méthode
```

Voyons quelques uns de ces champs (ou méthodes) :

`itag` : identifiant du *stream*.

`mine_type` : type de fichier.

`res` : résolution (pour les fichiers vidéo).

`abr` : qualité sonore (pour les fichiers audio).

`fps` : nombre d'images par seconde (plus ce nombre est important, plus la vidéo est fluide).

`vcodec` et `acodec` : différents codecs.

`progressive` : `True` pour les fichiers contenant à la fois le flux audio et le flux vidéo. `False` ne prendra en compte que les fichiers audio et vidéo séparés (il sera nécessaire de combiner ces fichiers pour avoir les deux flux dans un même fichier, c'est ce que permet le module `ffmpeg-python`).

38.4 Enregistrer la progression d'un téléchargement

A noter que si le fichier souhaité est déjà téléchargé, relancer le programme ne lancera pas le téléchargement.

L'enregistrement de la progression se situe au niveau de la fonction `progression_telechargement` du fichier `yt_downloader.py`. Voir aussi la ligne 33 pour l'appel de cette fonction.

Voir la documentation idoine [4].

La section *API* de la documentation liste toutes les fonctions du module. La méthode `register_on_progress_callback` permet d'enregistrer une fonction *callback* (fonction appelée par la fonction) relative à la progression du téléchargement après l'initialisation. Ce qui va nous intéresser c'est la valeur de `bytes_remaining` qui correspond au nombre d'octets à télécharger, et à l'aide de cette valeur nous pourrons calculer la durée restante de téléchargement.

38.5 Faire évoluer ce projet

A partir de ce projet de base nous pouvons imaginer divers évolutions : permettre l'insertion d'*url* par l'utilisateur, choix du type de fichier à télécharger, choix des répertoires d'enregistrement, implémentation d'une interface graphique, renommer les fichiers téléchargés, et j'en passe.

4. `https://pytube.io/en/latest/api.html#pytube.YouTube.register_on_progress_callback` et `https://pytube.io/en/latest/_modules/pytube/__main__.html#YouTube.register_on_progressive_callback`

Huitième partie

Python en mode graphique

Chapitre 39

Jouons avec la tortue

39.1 Introduction

Nous allons donc déplacer la tortue (*turtle*) à l'aide des quatre commandes de base qui sont : `forward` (avancer), `backward` (reculer), `right` (tourner à droite) et *left* (tourner à gauche). On va pouvoir écrire un programme qui utilise de façon séquentielle les commandes citées.

La première chose à faire est d'importer le module `turtle` et d'initialiser notre tortue.

```
1  import turtle
2  # Initialisation de notre 'turtle' avec la variable 't':
3  t = turtle.Turtle()
4  # De façon à garder la fenêtre active:
5  turtle.done()
```

Listing 39.1 – Les trois lignes d'initialisation de notre fenêtre graphique

Rien qu'avec ce code, une fenêtre graphique s'ouvre avec notre tortue au centre.

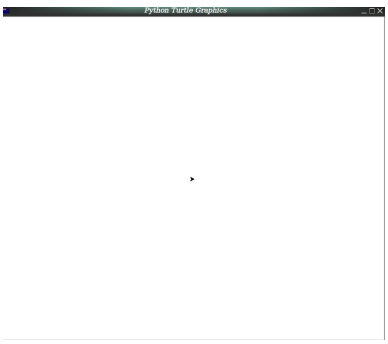

FIGURE 39.1 – La fenêtre graphique et la tortue

Faisons maintenant avancer et tourner notre tortue, puis avancer à nouveau, etc.

```
1  import turtle
2  t = turtle.Turtle()
3  # Faisons bouger notre tortue
4  t.forward(100)  # Avancer de 100 pixels
```

```
  t.left(90)  # Tourner à gauche de 90 degrés
  t.forward(50)
  t.backward(100)  # Reculer de 100 pixels
  t.right(45)  # Tourner à droite de 45 degrés
  t.forward(200)
  turtle.done()
```

Listing 39.2 – La tortue se déplace

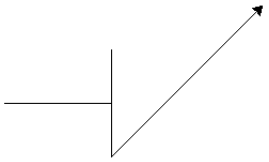

FIGURE 39.2 – Premiers déplacements de la tortue

39.2 Un escalier

Dessin d'un escalier avec des marches de 30 pixels de largeur et de hauteur, avec 5 marches au total.

```
"""Dessin d'un escalier avec le module 'turtle' :
- 5 marches de 30 pixels."""

import turtle

t = turtle.Turtle()
nbre_pixels = 30
angle = 90
nbre_marches = 5


def escalier(nb_px, ang, nb_m):
    """Dession de l'escalier"""
    for i in range(0, nb_m):
        t.left(ang)
        t.forward(nb_px)
        t.right(ang)
        t.forward(nb_px)


# Pour la première marche
t.forward(nbre_pixels)
# Pour toutes les autres marches
```

```
escalier(nbre_pixels, angle, nbre_marches)

turtle.done()
```

39.3 Dessin de plusieurs carrés de tailles différentes

```
"""Dessin d'un carré de 100 pixels de côté"""

import turtle

nbre_pixels = 50
nbre_carres = 8
t = turtle.Turtle()


def carre(nb_px):
    for i in range(0, 4):
        t.forward(nb_px)
        t.left(90)


def carres(taille1, nb_carres):
    for i in range(0, nb_carres):
        taille1 += 20
        carre(taille1)


carres(nbre_pixels, nbre_carres)

turtle.done()
```

Neuvième partie

Une bibliothèque riche et dense

Chapitre 40

Générer des données aléatoires avec le module `random`

40.1 Les *PRNGs* en *Python*

Il s'agit en fait de données *pseudo-aléatoires*. Cela signifie qu'elles sont générées par un générateur de nombres pseudo-aléatoires (*PRNG*), c'est-à-dire tout algorithme permettant de générer de telles données, apparemment aléatoires, est néanmoins reproductible. Les « vrais » nombres aléatoires sont quant à générés par un véritable générateur de nombres aléatoires (*TRNG*). Les *PRNG*, sont généralement réalisés avec un logiciel plutôt qu'avec du matériel, et ils fonctionnent de manière légèrement différente des *TRNG*[1]. C'est pour cette raison que dans la documentation officielle du module[2] on y trouve l'avertissement suivant : « *Les générateurs pseudo-aléatoires de ce module ne doivent pas être utilisés à des fins de sécurité* »[3].

L'outil le plus connu pour générer des données aléatoires en *Python* est probablement le module `random`, qui utilise l'algorithme *PRNG* `Mersenne Twister`[4] comme générateur de base.

40.2 `random.random()` et `random.seed()`

Tout d'abord, construisons des données aléatoires sans recourir à la technique de l'ensemencement (`random.seed()`). La fonction `random.random()` renvoie un nombre flottant aléatoire dans l'intervalle `[0.0, 1.0]`. Le résultat sera toujours inférieur à l'extrémité droite `(1,0)`. C'est ce qu'on appelle un intervalle semi-ouvert :

```
>>> import random
>>> random.random()
0.6179433636795252
>>> random.random()
0.2533837287218852
```

On voir ici que les résultats obtenus sont différents. Avec `random.seed()`, il est possible de rendre les résultats reproductibles :

1. Pour plus d'informations à ce sujet : `https://learn.microsoft.com/fr-fr/windows/win32/api/wincrypt/nf-wincrypt-cryptgenrandom?redirectedfrom=MSDN`
2. Module `random` : https ://docs.python.org/fr/3/library/random.html
3. Pour générer des nombres aléatoires de façon sécurisée, on se tournera vers le module `secrets` : `https://docs.python.org/fr/3.11/library/secrets.html`
4. `https://github.com/python/cpython/blob/main/Modules/_randommodule.c`

```
>>> random.seed(444)
>>> random.random()
0.3088946587429545
>>> random.random()
0.01323751590501987
>>> random.seed(444)
>>> random.random()
0.3088946587429545
>>> random.random()
0.01323751590501987
```

Avec `random.seed()` la séquence de nombres aléatoires devient déterministe, ou complètement déterminée par la valeur de départ (ici `444`).

40.3 `random.randint()` et `random.randrange()`

Il est possible de générer un nombre entier aléatoire, compris entre deux points d'extrémité, à l'aide de la fonction `random.randint()`. Cette fonction couvre l'ensemble de l'intervalle `[x, y]` :

```
>>> random.randint(0, 10)
10
>>> random.randint(500, 50000)
49829
```

Avec `random.randrange()`, nous excluons le côté droit de l'intervalle, ce qui signifie que le nombre généré se situe toujours à l'intérieur de `[x, y]` et sera toujours plus petit que la valeur `y` :

```
>>> random.randrange(0, 3)
1
>>> random.randrange(0, 3)
1
>>> random.randrange(0, 3)
0
>>> random.randrange(0, 3)
2
>>> random.randrange(0, 3)
1
```

40.4 `random.uniform()`

Si nous avons besoin de générer des valeurs flottantes aléatoires comprises dans un intervalle `[x, y]` spécifique, nous utiliserons `random.uniform()` :

```
>>> random.uniform(10, 15)
10.598997942362535
>>> random.uniform(10, 15)
13.969776294398033
```

40.5 `random.choice()`, `random.choices()`, `random.sample()` et `random.shuffle()` : des outils pour les séquences

Pour choisir un élément aléatoire dans une séquence non vide (comme une liste ou un tuple), nous pouvons utiliser `random.choice()`. Il existe également `random.choices()` pour choisir plusieurs éléments d'une séquence (à noter que les doublons sont possibles) :

```
>>> items = ['un', 'deux', 'trois', 'quatre', 'cinq']
>>> random.choice(items)
'un'
>>> random.choice(items)
'quatre'
>>> random.choices(items, k=2)
['un', 'cinq']
>>> random.choices(items, k=3)
['un', 'un', 'cinq']
>>> random.choices(items, 3)  # Avec absence du mot-clé de l'argument
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.11/random.py", line 500, in choices
    raise TypeError(
TypeError: The number of choices must be a keyword argument: k=3
```

Pour éviter les doublons il sera préférable de faire appel à `random.sample()` :

```
>>> random.sample(items, 4)
['trois', 'cinq', 'un', 'quatre']
>>> random.sample(items, 4)
['un', 'trois', 'deux', 'quatre']
>>> random.sample(items, 4)
['trois', 'deux', 'un', 'quatre']
>>> random.sample(items, 4)
['cinq', 'un', 'quatre', 'deux']
```

Nous pouvons modifier l'ordre d'une séquence sur place à l'aide de `random.shuffle()`. Cette méthode modifie l'objet séquence et modifie l'ordre des éléments de manière aléatoire :

```
>>> random.shuffle(items)
>>> items
['cinq', 'un', 'quatre', 'deux', 'trois']
```