Python3 - Support de formation

David Gayerie

15 septembre 2019

Liens utiles

Documentation officielle https://docs.python.org/3.6/

Tutoriel officiel https://docs.python.org/fr/3.6/tutorial/index.html

Référence du langage https://docs.python.org/3.6/reference/index.html

Bibliothèque standard https://docs.python.org/3.6/library/index.html

Packt publishing https://www.packtpub.com/tech/Python

CHAPITRE 1

Introduction

Python est un langage de programmation créé par Guido Van Rossum alors qu'il travaillait sur le système d'exploitation Amœba et qu'il souhaitait disposer d'un langage simple et puissant pour interagir avec le système.

Note : Contrairement à ce que le logo du langage et le nom de certains outils pourraient suggérer, le langage Python ne tire pas son nom d'un reptile mais des *Monty Python* : le célèbre groupe d'humoristes anglais qui ont débuté leur carrière au début des années 1970. Guido Van Rossum étant fan des Monty Python, il a suggéré que les exemples et les documentations devraient s'inspirer des sketchs du groupe.

1.1 Python 2 et Python 3

Comme tous les langages de programmation, Python a connu, et connaît toujours, des évolutions qui conduisent à utiliser des numéros de version. La particularité de ce langage réside dans le fait qu'il existe toujours une version 2 et une version 3 du langage en cours de développement et d'exploitation. En effet, la version 3 a conduit à impacter profondément le langage tel qu'il existait jusqu'à présent. Il a donc été décidé de développer la version 3 tout en maintenant activement la version 2.

Cette décision a parfois obligé les développeurs à faire preuve d'inventivité et de virtuosité pour écrire des bibliothèques et des systèmes compatibles à la fois pour Python 2 et pour Python 3.

Il a finalement été décidé que Python 2 ne sera plus supporté après 2020. Pour les systèmes existants, il est donc recommandé d'effectuer la migration du code vers la version 3. Pour tous les nouveaux développements et sauf contraintes majeures, il faut choisir Python 3.

Note : Ce support est donc uniquement dédié à Python 3. Il ne sera pas fait mention de Python 2 ni des différences entre les deux versions.

1.2 Licence logiciel

Python est distribué sous la *Python Software Foundation License*. Cette licence est très proche de la licence BSD tout en étant compatible avec la licence GPL de la *Free Software Foundation*.

Ainsi Python peut être utilisé sans difficulté aussi bien dans des logiciels libres que dans des logiciels à licence propriétaire.

1.3 Les principales caractéristiques du langage

1.3.1 Orienté objet

Python est un langage de programmation orientée objet. Mais comparé à d'autres langages similaires (comme Java et C#), Python n'impose pas aux développeurs de créer spécifiquement des applications en utilisant les principes de la programmation objet. Il est même possible de créer des programmes en Python en suivant une approche uniquement procédurale voire même fonctionnelle.

1.3.2 Courbe d'apprentissage linéaire

Cette plasticité dans l'approche du développement permet à Python de posséder une courbe d'apprentissage très linéaire. Il est possible d'apprendre rapidement les bases du langage et de réaliser des applications sans connaître nécessairement toutes les subtilités de la programmation objet. Ainsi, Python est très souvent proposé comme langage d'apprentissage de la programmation ou bien il est utilisé dans des domaines d'ingénierie autres que l'informatique.

1.3.3 Compilé et interprété

Python est un langage compilé et interprété. Cela signifie que le code source doit être compilé, c'est-à-dire transformé dans un langage binaire. Cependant ce langage n'est pas compréhensible directement par le processeur de la machine. Les fichiers compilés doivent donc être interprétés, c'est-à-dire lus et transformés en instructions exécutables par le processeur de la machine. Cette transformation est réalisée logiciellement par un programme que l'on nomme **l'interpréteur**.

Il existe des langages de programmation qui sont uniquement compilés (comme C++) et l'étape de compilation permet d'obtenir le programme binaire. Il existe des

langages de programmation qui sont uniquement interprétés (comme JavaScript). Si le concepteur de Python a choisi de combiner les deux approches, c'est pour pouvoir disposer des avantages des deux approches tout en limitant leurs inconvénients respectifs. Ainsi, le programme compilé reste portable, c'est-à-dire exécutable sur des machines aux architectures différentes puisqu'il doit encore être interprété. Mais le code à interpréter est plus rapide à convertir en instructions binaires que le fichier source, ce qui permet des performances supérieures à des langages uniquement interprétés.

Note : Java et C# sont également des langages compilés et interprétés. Mais dans le cas de Python, la compilation est faite au moment du premier lancement du programme. Il n'y a pas d'étape spécifique de compilation imposée aux développeurs comme en Java et C#. Cela accroît encore la simplicité de prise en main de ce langage.

1.3.4 Typage dynamique

Python est un langage à typage dynamique. Cela signifie que le type d'une variable (c'est-à-dire le format des données qui peuvent être représentées par cette variable) est choisi au moment où le programme assigne une valeur à la variable. Si le programme affecte plus tard une valeurs d'un type différent à la même variable, celle-ci donne l'impression de changer de type.

Python se distingue de langages tels que C, C++, Java ou C# qui sont des langages dit à typage fort. Il se rapproche de JavaScript qui fonctionne également suivant le principe du typage dynamique. Le typage dynamique offre une plus grande souplesse dans le développement et permet de conserver une relative simplicité dans la grammaire du langage. Cependant, cela se fait au détriment de la possibilité pour le compilateur d'aider le développeur à détecter des problèmes éventuels et ainsi éviter des bugs à l'exécution. En pratique, cela ne signifie absolument pas que les applications développées en Python souffrent de plus de bugs, il s'agit juste d'une approche différente du développement qui permet aux développeurs de moins faire porter leurs efforts sur la rigueur du typage mais un peu plus sur la rigueur de l'utilisation des variables.

1.4 Les domaines d'application

Python est partout ! Comme il n'est pas nécessaire d'être un expert en programmation pour pouvoir écrire des applications complexes, Python est utilisé bien au-delà de son domaine de départ (l'interaction avec un système d'exploitation avec la réalisation d'un *shell*). Ainsi, on trouve des programmes en Python pour le calcul scientifique, les mathématiques, l'électronique, la robotique, l'intelligence artificielle…

Mais il ne faut surtout pas en déduire que Python est un langage pour les non informaticiens, en effet dans ces usages les plus avancés, on le trouve également au cœur de certains systèmes d'information dans les entreprises, dans la réalisation de sites

Web et d'API Web, dans les services exécutés par les systèmes d'exploitation ou encore dans le domaine de la sécurité informatique. Comme l'interpréteur Python peut s'intégrer dans des applications écrites avec d'autres langages, Python est aussi très vite devenu un langage très utile pour offrir une interface de programmation (API) afin de créer plus facilement des extensions pour des logiciels propriétaires ou pour des logiciels libres.

1.5 Les différentes implémentations de Python

Il existe plusieurs implémentations de l'interpréteur Python.

CPython Probablement le plus célèbre et le plus utilisé, il s'agit de l'interpréteur d'origine écrit en langage C. Il est surtout choisi pour ses performances d'exécution et pour sa simplicité d'installation et d'intégration.

PyPy Il s'agit d'un interpréteur Python écrit en… Python ! Ce qui peut paraître comme un défi ou une plaisanterie de développeurs est en fait extrêmement utile à la communauté. En effet, il est plus facile pour un développeur Python de modifier le code source d'un interpréteur écrit en Python que celui d'un interpréteur écrit en C. L'objectif de l'interpréteur PyPy n'est donc pas la performance mais l'extrême portabilité et la possibilité d'expérimenter de façon relativement simple des évolutions du langage.

Jython Cet interpréteur s'exécute dans une machine virtuelle Java. Son principal atout est de permettre l'interaction, au sein d'une même machine virtuelle Java, de code écrit en Java avec du code écrit en Python.

IronPython Cet interpréteur génère du code compatible avec .Net. Son principal atout est de permettre d'écrire des programmes en Python pour la plate-forme .Net de Microsoft et d'interagir facilement avec du code écrit en C#.

1.6 Communauté et organisation

Comme beaucoup de communautés dans les logiciels libres, la communauté Python est hétéroclite et foisonnante. Il existe cependant une organisation dédiée à la gestion de l'évolution du langage.

Le créateur du langage, Guido Van Rossum, possède le titre de BDFL : *Benevolant Dictator For Life* (dictateur bienveillant à vie). Cela signifie qu'il veille au respect de l'esprit dans lequel le langage a été conçu mais il n'est plus en charge de l'évolution du langage.

La communauté a mis en place le système des *Python Enhancement Proposals* (PEP). Il s'agit de documents qui sont soumis par n'importe qui et qui font l'objet d'une relecture par le BDFL lui-même mais également les membres les plus influents de la communauté et notamment ceux qui, par leur expérience et leurs contributions, ont le droit de modifier le code source considéré comme faisant référence. Une PEP est identifiée par son numéro. Il peut s'agir d'un document informationnel ou d'une proposition d'évolution du langage. Par exemple, la **PEP 8** est bien connue des développeurs car elle décrit le format (notamment l'indentation et les espaces) à respecter

dans un fichier source Python. Mais on trouve aussi des documents plus généraux comme la **PEP 20**, le *Zen de Python*, qui résume les principes qui devraient guider tous les développeurs Python. Toutes les évolutions du langage passent par la rédaction et la validation d'une PEP.

CHAPITRE 2

Console et exécution de fichiers source

Pour exécuter du code Python, nous avons la possibilité d'écrire des fichiers sources qui seront ensuite pris en charge par l'interpréteur (qui se chargera également de les compiler). Mais nous pouvons aussi utiliser la console. Cette dernière permet d'entrer une instruction qui est immédiatement compilée et interprétée. On peut ainsi obtenir directement le résultat d'une instruction.

Avec CPython, l'interpréteur et la console sont un seul est même programme appelé `python` (ou `python.exe` pour Windows)

Important : Les systèmes MacOS et *nix ont besoin d'installer l'interpréteur Python 2 et l'interpréteur Python 3 (qui sont des programmes distincts). Dans ce cas, le programme `python` correspond la plupart du temps à l'interpréteur Python 2 et l'interpréteur Python 3 se nomme alors `python3`. Dans la suite de ce support nous considérerons que l'interpréteur Python 3 se nomme simplement `python`.

2.1 La console

La console Python se lance simplement avec la commande `python`

```
$ python
Python 3.6.7
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Important : Pour les exemples de ce support, nous utilisons le caractère `$` pour

représenter le prompteur de l'invite de commandes et ainsi indiquer qu'il s'agit de commandes à taper dans une invite de commandes (ou *shell*).

De même, nous utilisons la séquence de caractères >>> pour représenter le prompteur de la console et ainsi indiquer qu'il s'agit d'un exemple de code exécuté dans la console Python.

Tous les autres exemples correspondent à du code Python qui peut être placé dans un fichier source ou écrit directement sur la console.

Pour quitter la console, il suffit soit de fermer la fenêtre, soit d'utiliser le raccourci clavier `Ctrl+D` ou encore d'appeler les fonctions `quit()` ou `exit()`.

```
>>> quit()
$
```

Il existe d'autres consoles en Python, citons :

bpython Cette console est très pratique pour les utilisateurs occasionnels de Python ou pour les personnes en cours d'apprentissage. En effet, la console vous propose automatiquement les choix possibles au fur et à mesure que vous saisissez du code. Il faut utiliser la touche `TAB` pour naviguer parmi les choix proposées. De plus, la console affiche systématiquement la documentation de l'élément courant.

ipython Cette console offre des fonctionnalités avancées comme la coloration syntaxique et la complétion de code.

Exercice en console

Essayez de taper l'instruction suivante dans la console :

```
>>> print("Bonjour à tout le monde")
```

2.2 Les fichiers sources

Un programme Python est représenté par un fichier (ou pour les programmes les moins triviaux par plusieurs fichiers) source. Un fichier source Python est un fichier texte dont le nom se termine par l'extension `.py` ou `.pyw`. La première fois que l'interpréteur Python exécute le fichier source, il va le compiler sous la forme d'une fichier avec l'extension `.pyc`. Les fichiers de compilation étant automatiquement gérés par l'interpréteur, ils peuvent largement être ignorés par les développeurs. D'ailleurs l'interpréteur Python 3 place les fichiers compilés dans un répertoire `__pycache__` pour ne pas les mélanger avec le code source.

Pour lancer un programme à partir d'un fichier source, il suffit de taper sur une invite de commande `python` suivi du chemin vers le fichier source.

```
$ python mon_fichier_source.py
```

En Python 3, les fichiers source **doivent être encodés en UTF-8**. Normalement, un éditeur de code adapté au développement en Python 3 devrait gérer ce réglage sans l'intervention du développeur. Mais méfiance…

Pour plus de sécurité, vous pouvez ajouter en début de fichier la pseudo instruction :

```
# -*- coding: utf-8 -*-
```

Elle permet de spécifier manuellement l'encodage.

Exercice avec un fichier source

Créez le fichier source `premier_programme.py` et écrivez dedans :

```
print("Bonjour à tout le monde")
```

Exécutez le programme depuis une invite de commandes en vous plaçant dans le même répertoire de travail que le fichier et en tapant :

```
$ python premier_programme.py
```

2.3 Lancement en mode interactif

Lors du développement, il est parfois utile d'exécuter un fichier et ensuite de basculer en mode console pour pouvoir écrire directement du code en fonction des déclarations et des traitements réalisés par ce fichier source. On appelle cela le lancement en mode interactif. Il suffit d'ajouter le paramètre de lancement `-i` dans l'invite de commandes avant de préciser le chemin du fichier à exécuter.

```
$ python -i mon_fichier_source.py
```

2.4 Lancement d'un programme hors de la console

Tous les systèmes autorisent le lancement d'un programme Python en cliquant sur le fichier depuis un explorateur de fichiers. Pour cela, il faut tout de même s'assurer que le fichier possède les droits d'exécution pour l'utilisateur.

Pour les systèmes *nix, il faut ajouter au début du fichier le shebang :

```
#!/usr/bin/python
```

Lorsque vous exécutez de cette manière un fichier Python, le système ouvrira un terminal pour permettre au programme d'afficher des informations. Si votre programme Python est un programme graphique qui interagit avec l'utilisateur uniquement avec des fenêtres, vous pouvez indiquer au système que votre programme n'a pas besoin d'un terminal au démarrage en donnant à votre fichier l'extension `.pyw`).

2.5 Notion de bloc et d'indentation

Comme beaucoup de langages de programmation, la grammaire du langage Python décompose un programme en bloc. Mais à la différence des autres langages, Python n'utilise pas un caractère ou une instruction pour délimiter les blocs mais uniquement l'indentation du code, c'est-à-dire le nombres d'espaces avant une instruction. Cela incite les développeurs à présenter correctement le code source à la fois pour le compilateur et pour les éventuels lecteurs. En Python il est recommandé d'utiliser **uniquement des espaces** (pas de tabulation) et d'indiquer la présence d'un bloc par **quatre espaces**.

```
Ceci est un exemple
    Avec un premier bloc qui n'est pas du Python mais qui permet de ce
    rendre compte du principe d'indentation
        Voici un sous bloc dans le bloc

        qui peut continuer avec éventuellement des lignes vides
    Après le bloc reprend

On démarre une nouvelle instruction
Et ainsi de suite...
```

2.6 Les commentaires

Il est possible d'écrire des commentaires dans le code source, c'est-à-dire du texte qui sera ignoré par le compilateur. En Python les commentaires commencent par le caractère # (sauf dans une chaîne de caractères où ce caractère est considéré comme faisant partie de la chaîne).

```
# Ceci est un commentaire
print("Bonjour à tout le monde") # Et ceci est un autre commentaire
print("# Attention ceci n'est pas un commentaire !")
```

CHAPITRE 3

Types et variables

3.1 Nombres et calcul arithmétique

Python permet de réaliser facilement des calculs arithmétiques. Il est par exemple possible de saisir dans l'interpréteur :

```
>>> 2 + 3
5
```

```
>>> 2 * 3
6
```

```
>>> 10 / 2
5.0
```

Les opérateurs arithmétiques sont :

Tableau 1 – Les opérateurs arithmétiques

+	L'addition
-	La soustraction
*	La multiplication
**	La puissance
/	La division
//	La division arrondie à l'inférieur
%	Le reste de la division (modulo)

Note : Il existe également l'opérateur `@` pour la multiplication matricielle. Cepen-

dant, Python n'a pas de représentation par défaut des matrices. Cet opérateur n'est donc pas utilisé dans l'environnement Python par défaut.

Par défaut, Python définit une priorité pour les opérateurs. Par exemple, la multiplication et la division sont prioritaires sur l'addition et la soustraction. Pour les expressions mathématiques plus complexes, il est possible d'utiliser les parenthèses.

```
>>> 10 + 2 * 3 + 6 / 2
19.0
```

```
>>> (10 + 2) * (3 + 6) / 2
54.0
```

Comme beaucoup d'autres langages de programmation, Python fait une différence entre les nombres entiers (**int**) et les nombres réels appelés également les nombres à virgule flottante (**float**).

Les nombres entiers sont par défaut écrits en base 10. Mais il est également possible d'écrire la valeur d'un nombre en binaire (base 2) en la préfixant par `0b`, en octal (base 8) en la préfixant par `0o` et en hexadécimal (base 16) en la préfixant par `0x`.

```
>>> 0b1000
8
>>> 0o10
8
>>> 0xFF
255
```

Les nombres réels s'écrivent avec un `.` pour séparer la partie entière de la partie décimale. Il est également possible d'utiliser la notation scientifique de la forme x * 10^{y}

```
>>> .25
0.25
>>> 12.35
12.35
>>> 2.3e3
2300
>>> 2.3e-3
0.0023
```

Les entiers et les nombres naturels sont traités différemment par Python. Il est néanmoins possible de passer de l'un à l'autre de manière transparente pour le programmeur. Par exemple la division de deux entiers produit un nombre réel :

```
>>> 5 / 2
2.5
```

Tandis que la division avec arrondi à l'inférieur de deux nombres entiers produit un entier :

```
>>> 5 // 2
2
```

Note : Python supporte également les nombres complexes. La partie imaginaire est suffixée par la lettre `j` :

```
>>> 1 + 1j
(1+1j)
>>> (1 + 1j) * (3 + 2j)
(1+5j)
```

Important : Les nombres à virgule flottante ne permettent pas de représenter les nombres de manière précise. Les opérations arithmétiques sur ces nombres peuvent conduire à des arrondis par la machine. Si l'application doit produire des résultats précis, alors il est conseillé de s'appuyer sur le module `decimal`... une fois que nous aurons vu les modules et la programmation objet.

3.2 Les variables

Une variable est une zone nommée contenant une information. Une variable possède un **identifiant** (son nom) et un **type** (la nature de l'information qu'elle contient). Pour déclarer une variable en Python, il suffit simplement de donner son nom et de lui affecter une valeur grâce à **l'opérateur d'affectation** `=` :

```
>>> x = 1
>>> y = 2.2
```

Python est un langage de programmation à typage dynamique. Cela signifie que le type de la variable est déterminé par le type de la valeur qu'il contient. Dans l'exemple ci-dessus, le type de la variable `x` est **int** (un nombre entier) et le type de la variable `y` est **float** (un nombre réel).

L'identifiant (ou le nom) d'une variable s'écrit par convention en lettres minuscules. L'identifiant peut contenir des chiffres à condition qu'ils ne soient pas en premier pour ne pas les confondre avec un nombre. Si le nom est formé de plusieurs mots, on les sépare par un trait inférieur (*underscore*) suivant la notation dite du snake case.

```
>>> v = 1
>>> v2 = 2
>>> ma_variable = 3
>>> une_autre_variable = 4
```

Le nom d'une variable peut commencer par un trait inférieur (*underscore*) mais cela est réservé pour des usages particuliers.

Note : Python ne supporte pas directement la notion de constante. Pour signaler que le contenu d'une variable ne devrait pas être modifié par le programme, les programmeurs ont coutume d'écrire son nom en lettres majuscules. Il n'y a cependant rien qui interdit effectivement de modifier sa valeur :

```
>>> MA_CONSTANTE = 4
>>> PI = 3.1416
```

Une variable peut être utilisée dans une expression conforme à son type. Ainsi, il est possible d'utiliser les variables de type **int** ou **float** dans des opérations arithmétiques :

```
>>> x = 1
>>> y = 2.2
>>> x + y * 2
5.4
```

Il est possible d'inverser le signe d'une variable grâce à l'opérateur unaire - :

```
>>> x = 2
>>> -x
-2
>>> y = -4
>>> -y
4
```

Il est possible de modifier le contenu d'une variable en utilisant l'opérateur d'affectation = :

```
>>> x = 1
>>> x = 2
>>> x = 2 * x
>>> x
4
```

Note : Pour ajouter 1 à une variable, on peut écrire :

```
>>> x = x + 1
```

Il existe une forme compacte qui s'écrit :

```
>>> x += 1
```

Cette forme est disponible pour tous les opérateurs arithmétiques :

```
>>> x -= 3
>>> x *= 5
>>> x //=2
>>> x **=3
>>> x
343
```

Une variable peut avoir la valeur spéciale `None`. Ce mot-clé indique que la variable n'a aucune valeur ni aucun type (ou plus exactement la variable a le type spécial `NoneType`).

```
>>> nothingness = None
```

Prudence : Pour pouvoir être utilisée dans une expression, une variable doit avoir été déclarée :

```
>>> x = 1
>>> x += variable_non_declaree
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'variable_non_declaree' is not defined
```

3.3 La chaîne de caractères

En plus des nombres, un autre type de données couramment utilisé est la chaîne de caractères (*character string* ou plus simplement *string*), c'est-à-dire une portion de texte. En Python, une chaîne de caractères est encadrée par le caractère apostrophe ' ou le caractère guillemet " que l'on appelle les **délimiteurs** :

```
>>> "hello the world"
>>> 'bonjour le monde'
```

Les chaînes de caractères utilisent l'encodage UTF-8. Si l'on désire faire apparaître certains caractères spéciaux, il faut utiliser un caractère échappé, c'est-à-dire une séquence commençant par \ afin de la différencier des caractères normaux ou des délimiteurs :

retour à la ligne	`\n`
tabulation	`\t`
guillemet	`\'`
apostrophe	`\"`
antislash	`\\`

Si vous ne souhaitez pas utiliser de caractères d'échappement, alors vous pouvez

déclarer une chaîne de caractères brut (*raw string*) en préfixant la chaîne avec la lettre `r`. Ainsi la chaîne de caractères :

```
"\t\n"
```

correspond à deux caractères : une tabulation suivie d'un retour à la ligne. Tandis que la chaîne de caractères :

```
r"\t\n"
```

correspond à quatre caractères : antislash, t, antislash et n.

Si vous voulez écrire une chaîne de caractères sur plusieurs lignes, vous pouvez utiliser le caractère antislash \ suivi immédiatement d'un retour à la ligne. Ces caractères ne seront pas considérés comme faisant partie de la chaîne finale.

```
"Une chaîne de caractères\
  sur plusieurs lignes en\
  utilisant l'antislash"
```

correspond à :

```
"Une chaîne de caractères sur plusieurs lignes en utilisant l'antislash"
```

Sinon vous pouvez utiliser le triple guillemet `"""` qui permet de créer une chaîne de caractères sur autant de lignes que l'on souhaite sans avoir besoin de spécifier l'antislash. Mais **attention**, les retours à la ligne seront présents dans la chaîne de caractères.

```
"""Une chaîne de caractères
sur plusieurs lignes en
utilisant le triple guillemet"""
```

correspond à :

```
"Une chaîne de caractères\nsur plusieurs lignes en\nutilisant le triple guillemet"
```

Une variable contenant une chaîne de caractères et de type **str**. L'opérateur +, appelé **opérateur de concaténation**, permet de créer une nouvelle chaîne à partir de deux chaînes accolées l'une à l'autre :

```
>>> h = "hello"
>>> w = "world"
>>> h + " " + w
"hello world"
```

3.3.1 Formatage de chaîne de caractères

Pour afficher une information à l'écran, il est souvent nécessaire de prendre des informations telles que des nombres et les insérer suivant un certain format dans une chaîne de caractères.

On peut indiquer dans une chaîne de caractères l'emplacement où la valeur d'une variable doit être insérée grâce au caractère `%` suivi d'une lettre indiquant le type de la variable :

%d	Nombre entier en base 10
%i	Nombre entier en base 10
%o	Nombre entier en base 8
%x	Nombre entier en base 16 (lettres en minuscule)
%X	Nombre entier en base 16 (lettres en majuscule)
%f	Nombre réel
%s	Chaîne de caractères

On utilise l'opérateur `%` après la chaîne de caractères pour spécifier la ou les variables à utiliser. S'il y en a plusieurs, il faut obligatoirement les signaler dans le bon ordre entre parenthèses et séparées par une virgule.

```
>>> total = 122
>>> "Le montant de la facture est de %d euros avec une TVA de 20.0%%" % total
'Le montant de la facture est de 122 euros avec une TVA de 20.0%'
>>> "%s le %s" % ("hello", "monde")
'hello le monde'
```

Astuce : Pour écrire le caractère % dans la chaîne, il faut écrire %%.

Note : Pour plus d'information sur le formatage, reportez-vous à la documentation complète.

Depuis Python 3.6, il est possible d'utiliser le formatage de chaîne de caractères en préfixant la chaîne par `f`. Le motif défini par la chaîne peut contenir entre accolade `{}` le nom d'une variable dont la valeur sera insérée à cet emplacement.

```
>>> code_facture = "XF32"
>>> montant = 122.55
>>> f"La facture n°{code_facture} a un montant de {montant} euros"
'La facture n°XF32 a un montant de 122.55 euros'
```

Note : Pour plus d'information sur le formatage, reportez-vous à la documentation complète.

3.4 Le type booléen

Un type booléen accepte deux valeurs : `True` et `False`. Les valeurs booléennes peuvent être utilisées avec les opérateurs logiques : `not`, `and` et `or`.

```
>>> a = True
>>> b = False
>>> not a
False
>>> not b
True
>>> a and b
False
>>> a and not b
True
>>> a or b
True
>>> not a or b
False
```

Il existe des opérateurs de comparaison pour les nombres qui produisent un résultat de type booléen :

Tableau 2 – Les opérateurs de comparaison

==	égalité
!=	inégalité
<	inférieur
>	supérieur
<=	inférieur ou égal
>=	supérieur ou égal

```
>>> 1 == 2
False
>>> 1 != 2
True
>>> 1 < 2
True
>>> 1 > 2
True
>>> 1 <= 1
True
>>> 2 > 1
True
>>> 2 >= 2
True
```

Les chaînes de caractères acceptent également les mêmes opérateurs pour réaliser une comparaison caractère par caractère :

```
>>> "Hello World" == "Hello World"
True
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> "Hello World" != "hello world"
True
>>> "Hello" < "Hello World"
True
>>> "Hello World" > "Hello"
True
```

Astuce : La comparaison entre chaînes de caractères est basée sur le nombres de caractères mais également sur le code des caractères. Ainsi :

```
>>> 'a' < 'z'
True
>>> 'a' > 'A'
True
```

Il est possible d'écrire des expressions booléennes complexes :

```
>>> x = 1
>>> y = 2
>>> message = "Bonjour à tous"
>>> x * 2 == y and message > "Bonjour"
True
>>> -1 < x < y
True
```

CHAPITRE 4

Les fonctions

Une fonction permet d'identifier une séquence de traitements par un nom. Une fonction peut alors être appelée par son nom à plusieurs endroits dans un programme pour réaliser ces traitements. Il s'agit ainsi de rendre un programme plus compréhensible tout en évitant de dupliquer une séquence à plusieurs points du programme. Un autre intérêt d'une fonction est qu'elle peut être écrite par un programmeur et être utilisée dans un code source écrit par un autre programmeur qui n'a pas besoin de connaître précisément le code exécuté par la fonction si son nom et sa documentation sont suffisamment explicites.

4.1 Appeler une fonction

Il existe beaucoup de fonctions déjà fournies avec l'environnement d'exécution Python. On les appelle les `builtin functions`. La plus célèbre d'entre-elles est sans doute `print()`. Elle affiche sur la sortie standard du programme une message fourni en paramètre. Pour appeler une fonction dans un programme, il faut préciser son nom et passer entre parenthèses la liste des **paramètres**. La fonction `print()` accepte en paramètre ce qu'elle doit afficher :

```
# appel de la fonction print
print("Bonjour le monde")
```

Il est possible de passer plusieurs paramètres à la fonction `print()`. Dans ce cas, il faut séparer la valeur des paramètres par une virgule.

```
# Cet appel affiche : Bonjour le monde
print("Bonjour", "le", "monde")
```

Il est également possible d'appeler la fonction `print()` sans aucun paramètre. La fonction se contentera d'afficher une ligne vide sur la sortie standard. Même s'il n'y pas de paramètre, les parenthèses sont obligatoires en Python 3.

```
# Cet appel affiche une ligne vide
print()
```

Si un paramètre peut être une valeur directement donnée par le programme, un paramètre peut aussi être une variable. Dans ce cas, c'est la valeur contenue dans la variable qui est passée à la fonction :

```
msg = "hello"
# Cet appel affiche : hello
print(msg)
```

Un paramètre peut aussi être une expression. Dans ce cas, c'est le résultat de l'évaluation de l'expression qui est passé à la fonction :

```
x = 2
# Cet appel affiche : Le résultat est 8
print("Le résultat est", x**3)
```

Une fonction peut retourner une valeur, c'est-à-dire fournir le résultat de son traitement. Cette valeur peut être affectée à une variable. Par exemple, la fonction `abs()` retourne la valeur absolue du nombre qui lui est passé en paramètre :

```
x = -2
valeur_absolue = abs(x)
# affiche 2
print(valeur_absolue)
```

L'appel d'une fonction peut être passé en paramètre d'une autre fonction. Dans ce cas, c'est la valeur retournée par l'appel de la fonction qui est passée à l'autre fonction.

```
# afficher 2
print(abs(-2))
```

Ce type d'écriture permet une plus grande concision dans l'expression.

Note : Une fonction qui ne produit aucun résultat est souvent appelée une **procédure**. On dit également qu'elle agit par *effet de bord*.

En Python, essayer d'affecter le résultat de l'appel d'une procédure à une variable ne produit pas d'erreur. La valeur de la variable est simplement positionnée à `None`.

```
# print est une procédure, elle ne retourne aucune valeur
x = print()
```

(suite sur la page suivante)

(suite de la page précédente)

```
# affiche None
print(x)
```

4.2 Les fonctions utiles dans la console Python

Parmi les `builtin functions`, il existe cinq fonctions qui sont très pratiques lorsqu'on exécute des instructions Python dans une console :

`exit()` et `quit()` Ces fonctions permettent d'arrêter la console Python

```
>>> quit()
```

`repr(o)` Affiche des informations sur l'élément passé en paramètre.

```
>>> repr(print)
'<built-in function print>'
```

L'interpréteur appelle implicitement cette fonction à chaque fois que vous tapez la touche `Entrer`. L'interpréteur évalue l'expression saisie et, si elle produit un résultat, il affiche la chaîne de caractères retournée par l'appel à `repr(o)` auquel est passé en paramètre le résultat de l'expression.

```
>>> print
'<built-in function print>'
```

`help(o)` Affiche la documentation de l'élément passé en paramètre. La console passe généralement en mode affichage de l'aide. Pour sortir de ce mode, il faut presser la touche `Q` pour *quit*.

```
>>> help(print)
```

`dir(o)` Retourne une liste des attributs et des méthodes de l'élément passé en paramètre. Nous verrons beaucoup plus tard que, dans le langage Python, tout est un objet. Un objet se définit par ses attributs (les données) et par ses méthodes (son comportement).

```
>>> dir(print)
['__call__', '__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '
 __init__', '__init_subclass__', '__le__', '__lt__', '__module__',
 '__name__', '__ne__', '__new__', '__qualname__', '__reduce__',
 '__reduce_ex__', '__repr__', '__self__', '__setattr__', '__sizeof__',
 '__str__', '__subclasshook__', '__text_signature__']
```

4.3 Les fonctions de conversion de type

Les fonctions `int()`, `float()`, `bool()` et `str()` permettent de réaliser des conversions d'un type à l'autre.

4.3.1 Convertir en nombre

La conversion entre entiers et nombres à virgule flottante :

```
>>> r = 3.5
>>> int(r)
3
>>> n = 2
>>> float(n)
2.0
```

Il est également possible de convertir une chaîne de caractères :

```
>>> int("3")
3
>>> float("5.25")
5.25
```

Dans ce cas, la fonction `int()` accepte un deuxième paramètre indiquant la base du nombre (par défaut la base est 10) :

```
>>> int("10", 8)
8
>>> int("0xFF", 16)
255
```

Prudence : Pour la conversion en nombre, la chaîne de caractères peut contenir des espaces avant et après mais si elle ne représente pas un nombre valide, les fonctions `int()` et `float()` produisent une erreur `ValueError`.

4.3.2 Convertir en valeur booléenne

La fonction `bool()` permet de convertir en valeur booléenne. Elle produit la valeur `True` pour un nombre différent de 0 et pour une chaîne de caractères non vide. La valeur `None` produit toujours la valeur `False`.

```
>>> bool(42)
True
>>> bool(0)
False
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> bool("hello")
True
>>> bool("False")
True
>>> bool("")
False
>>> bool(None)
False
```

4.3.3 Convertir en chaîne de caractères

La fonction `str()` permet de convertir en chaîne de caractères :

```
>>> str(1)
'1'
>>> str(True)
'True'
>>> str(None)
'None'
```

4.4 Exercices

Exercice : comparaison à la moyenne

Écrivez un programme qui demande à l'utilisateur sa taille puis qui indique l'écart en centimètres par rapport à la moyenne. On pourra utiliser comme moyenne 1m75.

Pour réaliser ce programme, vous allez avoir besoin de la fonction `input()`. Cette fonction accepte en paramètre le message à afficher à l'utilisateur. L'appel à cette fonction bloque le programme et attend que l'utilisateur saisisse une valeur puis tape sur `Entrer`. La fonction retourne alors la valeur saisie sous la forme d'une chaîne de caractères.

Exemples d'utilisation du programme :

```
Quelle est votre taille ? 1.70
Vous avez un écart de -5 centimètres par rapport à la moyenne
```

```
Quelle est votre taille ? 1.75
Vous avez un écart de 0 centimètres par rapport à la moyenne
```

Exercice : conversion Fahrenheit / Celsius

Écrivez un programme qui demande à l'utilisateur de saisir une température en degré Fahrenheit et qui affiche le résultat de la conversion en degré Celsius avec un seul chiffre après la virgule.

La formule à appliquer :

$$d(f) = \frac{f - 32}{1.8}$$

Exemple d'utilisation du programme :

```
Température Fahrenheit ? 42
Une température de 42.0°F correspond à 5.5°C
```

Exercice : QCM en invite de commande

Écrivez un programme qui pose des questions pour lesquelles l'utilisateur doit fournir une réponse. Pour chaque réponse correcte, l'utilisateur marque un point. À la fin, le programme indique le nombre de réponses justes et le nombre de réponses fausses.

Exemple d'utilisation du programme :

```
Quel est ton nom ?
Sir Robin de Camelot.
Quelle est ta quête ?
Trouver le Saint Graal.
Quelle est la capitale de la Syrie ?
Je sais pas ça !

Réponses correctes : 2
Réponses incorrectes : 1
```

Astuce : Il faut avoir à l'esprit que :

```
>>> int(True)
1
>>> int(False)
0
```

4.5 Déclarer une fonction

L'intérêt principal des fonctions est de pouvoir organiser un programme en le décomposant en traitements de granularité plus faible. Ces traitements deviendront des fonctions dont le nom et la documentation amélioreront la lisibilité et la maintenabilité.

Pour déclarer sa propre fonction, on commence par préciser sa **signature**. Cette signature est délimitée par le mot-clé `def` et deux-points `:`. Elle donne le nom de la fonction et la liste des paramètres. Puis vient le **corps** de la fonction, c'est-à-dire le code à exécuter lorsque cette fonction est appelée. Pour délimiter le corps d'une fonction, Python utilise **l'indentation**. Chaque ligne de la fonction commence par un certain nombre d'espaces (généralement quatre). Cela donne, à la fois, une indication visuelle au lecteur et à l'interpréteur de code.

Note : Le nom d'une fonction suit la même convention que le nom des variables. Il s'écrit en lettres minuscules et les mots sont séparés par un trait inférieur (*underscore*) suivant la notation dite du snake case.

```
def dire_bonjour_a(nom):
    print("Bonjour", nom)
```

Un paramètre est simplement indiqué par son nom qui peut être ensuite utilisé dans le corps de la fonction. Un paramètre a pour valeur celle qui est donnée à chaque appel de la fonction.

Il est ensuite possible d'appeler la fonction qui a été déclarée :

```
# affiche Bonjour David
dire_bonjour_a("David")
```

Note : Le corps d'une fonction Python doit contenir au moins une ligne. Si pour une raison ou une autre, vous voulez déclarer une fonction mais que vous voulez laisser le corps vide (votre fonction ne réalise aucun traitement), alors vous devez utiliser le mot-clé `pass` :

```
def ma_fonction_qui_ne_fait_rien():
    pass
```

4.6 Documenter une fonction

Pour écrire la documentation d'une fonction, il suffit d'ajouter une chaîne de caractères directement à la suite de la déclaration de la signature de la fonction. On appelle ce texte la `docstring`. C'est ce texte qui est affiché lorsqu'on passe le nom de la fonction à la fonction `help()`.

```
def dire_bonjour_a(nom):
    """Affiche un message qui dit bonjour au nom qui est passé en
       paramètre"""
    print("Bonjour", nom)
```

```
>>> help(dire_bonjour_a)
```

4.7 Les valeurs de retour

Pour qu'une fonction retourne une valeur, il faut utiliser le mot-clé `return` suivi de la valeur à retourner.

```
def creer_nom(prenom, nom):
    nom_complet = prenom + " " + nom
    return nom_complet
```

La valeur à retourner peut être le résultat d'une expression. Cela evite souvent d'avoir recours à des variables intermédiaires :

```
def creer_nom(prenom, nom):
    return prenom + " " + nom
```

Le mot-clé `return` interrompt immédiatement le traitement de la fonction et retourne la valeur au programme appelant. Si vous écrivez du code après un `return`, il ne sera jamais exécuté.

```
def creer_nom(prenom, nom):
    return prenom + " " + nom
    print("Cette instruction ne sera jamais executée")
```

Une fonction peut retourner plusieurs valeurs. C'est le cas, par exemple, de la fonction standard `divmod()` qui retourne à la fois le résultat de la division et le reste de la division. Python permet de récupérer ces valeurs dans deux variables différentes :

```
resultat, reste = divmod(10, 3)
print("Le resultat de la division de 10 par 3 est", resultat, "et le reste est",␣
↪reste)
# Le resultat de la division de 10 par 3 est 3 et le reste est 1
```

Nous pourrions réécrire sans difficulté la fonction `divmod()` :

```
def divmod(numerateur, denominateur):
    resultat = numerateur // denominateur
    reste = numerateur % denominateur
    return resultat, reste
```

Le mot-clé `return` peut également être employé seul pour signaler la fin d'une procédure. Dans cas, l'exécution de `return` interrompt immédiatement la procédure sans retourner de valeur.

4.8 Les paramètres nommés

Python supporte les **paramètres nommés** pour une fonction. Cela signifie que vous pouvez, lors de l'appel d'une fonction, préciser le nom et la valeur du paramètre.

```
def dire_bonjour_a(nom):
    print("Bonjour", nom)

dire_bonjour_a(nom="David")
```

Cela peut s'avérer utile pour une fonction qui accepte plusieurs paramètres. En effet, il n'est pas toujours facile de se souvenir de l'ordre exact des paramètres ni de leur signification. En nommant les paramètres au moment de l'appel, leur rôle devient plus explicite et nous ne sommes plus obligés de respecter l'ordre de leur déclaration :

```
def dire_bonjour_a(prenom, nom):
    print("Bonjour", prenom, nom)

dire_bonjour_a(nom="Gayerie", prenom="David")
```

Note : Vous pouvez appeler un fonction en passant des paramètres et des paramètres nommés. Dans ce cas, les paramètres nommés doivent être placés à la fin de la liste des paramètres.

4.9 Valeur par défaut des paramètres

Python autorise une fonction à avoir une valeur par défaut pour un ou plusieurs de ses paramètres. Ainsi le programme qui appelle la fonction peut omettre de donner une valeur pour ces paramètres. Cela permet de rendre certains paramètres optionnels. On précise la valeur d'un paramètre après le signe =.

```
def dire_bonjour_a(nom, message="Bonjour"):
    print(message, nom)

# affiche Bonjour David
dire_bonjour_a("David")
# affiche Bonsoir David
dire_bonjour_a(message="Bonsoir", nom="David")
```

4.10 Fonction et portée des variables

Une fonction peut accepter des paramètres et une fonction peut déclarer des variables :

```
def calcul_prix_ttc(prix_ht, taux_tva = 20):
    # déclaration de la variable tva
    tva = prix_ht * taux_tva / 100
    tva = round(tva, 2)
    return prix_ht + tva
```

Astuce : La fonction `round()` arrondie un nombre (le premier paramètre) avec une précision après la virgule donnée par le second paramètre.

Dans l'exemple ci-dessus, la fonction `calcul_prix_ttc()` utilise la variable `tva` pour stocker le calcul de la TVA et réaliser un arrondi. Cette variable est liée à l'exécution de la fonction `calcul_prix_ttc()`. Cela signifie qu'elle n'existe pour l'interpréteur qu'au moment de l'exécution de la fonction. Dès que la fonction se termine, la variable et sa valeur ne sont plus accessibles. On dit que la **portée** de la variable `tva` est limitée à la fonction `calcul_prix_ttc()`. Donc cette variable n'existe pas dans le reste du programme.

```
# Génère une erreur NameError car la variable tva n'existe pas
print(tva)

calcul_prix_ttc(100)

# Génère une erreur NameError car la variable tva n'existe pas
print(tva)
```

Mais que se passe-t-il si le programme déclare et utilise une variable `tva` avant et après l'appel à `calcul_prix_ttc()`

```
tva = "ceci n'est pas vraiment le montant d'une tva"

calcul_prix_ttc(100)

# affiche: ceci n'est pas vraiment le montant d'une tva
print(tva)
```

Avec l'exemple précédent, on voit bien que la variable `tva` n'est pas modifiée par l'appel à la fonction `calcul_prix_ttc()` car même si elle a un nom identique à la variable utilisée dans cette fonction, il s'agit de variables différentes pour l'interpréteur. On peut déclarer et utiliser des variables qui ont le même nom à condition qu'elles n'appartiennent pas à la même portée.

4.11 Exercices

Exercice : comparaison à la moyenne

Reprenez l'exercice qui permet de comparer une taille saisie avec la moyenne.

Créez trois fonctions dans le programme :

demander_taille() Cette fonction demande sa taille à l'utilisateur et retourne la valeur saisie sous la forme d'un nombre.

comparer_taille() Cette fonction compare une première taille à une seconde et donne la différence en centimètres. Le second paramètre a une valeur par défaut qui correspond à la moyenne de la taille.

afficher_difference_moyenne() Cette fonction prend en paramètre un écart de taille en centimètres et affiche l'information à l'utilisateur.

Exercice : fonction de conversion Fahrenheit / Celsius

Écrivez une fonction pour convertir une température en degré Fahrenheit en une température en degré Celsius.

4.12 Variable globale et fonction

Une variable qui est déclarée dans le fichier principal est aussi appelée **variable globale** car elle existe dans la mémoire jusqu'à la fin de l'exécution du programme.

Imaginons le programme suivant :

```
message="Bonjour"

def dire_bonjour_a(nom):
    print(message, nom)

# Affiche Bonjour David
dire_bonjour_a("David")
```

La fonction `dire_bonjour_a()` utilise la paramètre `nom` mais aussi la variable globale `message`. Dans les deux cas, la fonction ne fait que lire la valeur du paramètre et de la variable globale. Mais comment faire pour modifier la valeur d'une variable globale dans le corps d'une fonction ?

Nous avons vu à la section précédente qu'il n'est pas possible de modifier directement une variable globale dans une fonction. Nous pouvons simplement **créer une nouvelle variable avec le même nom ayant une portée limitée à la fonction**. Pour nous en convaincre, essayons le programme suivant :

```
message = "Un message global"

def traiter_message():
    message = "Un message local"
    print(message)

print(message)
```

(suite sur la page suivante)

(suite de la page précédente)

```
traiter_message()
print(message)
```

Ce programme produit le résultat suivant :

```
Un message global
Un message local
Un message global
```

Cela montre bien que la variable globale `message` et la variable `message` initialiser dans la fonction `traiter_message()` sont bien distinctes.

En Python, il est tout de même possible de modifier une variable globale dans une fonction. Pour cela, il faut préciser explicitement grâce au mot-clé `global` que la fonction fait référence à la variable globale. Si nous changeons notre implémentation :

```
message = "Un message global"

def traiter_message():
    global message
    message = "Un message local"
    print(message)

print(message)
traiter_message()
print(message)
```

Ce programme produit le résultat suivant et modifie bien le contenu de la variable globale :

```
Un message global
Un message local
Un message local
```

Prudence : Le recours à des variables globales est souvent le signe d'une mauvaise conception logicielle et est fortement déconseillé. On se limite généralement à utiliser ce type de variables pour représenter des constantes, c'est-à-dire des valeurs qui ne doivent pas changer pendant l'exécution du programme. Il n'existe pas de méthode directe pour déclarer des constantes en Python, on se contente d'écrire le nom des constantes en majuscules pour signaler notre intention aux autres développeurs :

```
TAUX_TVA = .2

def calculer_prix_ttc(prix_ht):
    tva = round(prix_ht * TAUX_TVA, 2)
    return prix_ht + tva
```

Notez qu'il n'est pas nécessaire d'utiliser le mot-clé `global` pour lire et utiliser la valeur d'une variable globale.

4.13 Notion de méthode

En plus des fonctions, Python connaît la notion de **méthode**. Une méthode est une séquence de traitement que l'on effectue sur un type. Une méthode est appelée pour une valeur, une variable ou un paramètre en utilisant l'opérateur `.`. Par exemple, une chaîne de caractères possède les méthodes `upper()` et `lower()` permettant de créer une nouvelle chaîne de caractères en lettres minuscules et respectivement en lettres majuscules :

```
msg = "Testons les méthodes."
msg_lower = msg.lower()
msg_upper = msg.upper()
print(msg_lower)
print(msg_upper)
msg_upper = "hello".upper()
print(msg_upper)
```

Note : Nous verrons plus en détail la notion de méthode lorsque nous aborderons le paradigme de la programmation orientée objet.

CHAPITRE 5

La liste (*list*)

Une liste (souvent appelée tableau dans d'autres langages de programmation) est une séquence modifiable de données ordonnées. Cela signifie que chaque élément de la liste est associé à une position (un **index**). Le premier élément d'une liste possède l'index 0 et le dernier élément possède l'index n - 1 pour une liste contenant n éléments.

5.1 Créer une liste

Pour créer une liste, on écrit la liste des éléments entre crochets `[]` et séparés par des virgules :

```
liste_vide = []
ma_liste = [10, 20, 30, 40]
liste_depareillee = [None, 1, "shrubbery", True]
```

Comme le montre la `liste_depareillee` ci-dessus, une liste peut contenir des éléments de types différents. Elle peut même contenir un élément ou des éléments `None` ce qui signifie qu'elle contient un élément qui n'est rien.

5.2 Accéder aux éléments

Pour accéder à un élément d'une liste, il faut utiliser les crochets et préciser l'index de l'élément.

```
ma_liste = [10, 20, 30, 40]
premier_element = ma_liste[0]
second_element = ma_liste[1]
```

On peut utiliser un index négatif. Cela signifie que l'on souhaite partir de la fin de la liste. Ainsi le dernier élément d'une liste est aussi accessible avec l'index **-1**.

```
ma_liste = [10, 20, 30, 40]
dernier_element = ma_liste[-1]
```

Attention, si un programme veut accéder à un indice qui n'existe pas, cela produit une erreur de type `IndexError`.

```
>>> ma_liste = [10, 20, 30, 40]
>>> ma_liste[10000]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

L'accès à un élément permet également de modifier la valeur d'un élément de la liste s'il est placé à gauche de l'opérateur d'affectation :

```
ma_liste = [10, 20, 30, 40]
ma_liste[0] = 0
print(ma_liste)
# affiche [0, 20, 30, 40]
```

5.3 Les opérations sur les listes

5.3.1 Connaître la taille d'une liste

Pour connaître le nombre d'éléments d'une liste, il faut utiliser la fonction `len()` (qui est la contraction de *length*) :

```
ma_liste = [10, 20, 30, 40]
taille_liste = len(ma_liste)
print(taille_liste)
# affiche 4
```

5.3.2 Supprimer un élément d'une liste

Pour supprimer un élément d'une liste, on utilise le mot-clé `del` :

```
ma_liste = [10, 20, 30, 40]
del ma_liste[1]
print(ma_liste)
# affiche [10, 30, 40]
```

5.3.3 Ajouter deux listes ensemble

L'opérateur + permet d'ajouter deux listes ensemble pour former une troisième liste :

```
ma_liste = [0, 1, 2] + [10, 20, 30]
print(ma_liste)
# affiche [0, 1, 2, 10, 20, 30]
```

Astuce : Pour augmenter la taille d'une liste, vous pouvez utiliser l'opérateur +=. Cet opérateur ne crée pas de nouvelle liste, la liste existante est simplement étendue avec les nouveaux éléments.

```
ma_liste = [0, 1, 2]
ma_liste += [10, 20, 30]
print(ma_liste)
# affiche [0, 1, 2, 10, 20, 30]
```

5.3.4 Créer une liste étendue

L'opérateur * permet de créer une liste en répétant le contenu d'une liste autant de fois que spécifié :

```
ma_liste = ['ni'] * 5
print(ma_liste)
# affiche ['ni', 'ni', 'ni', 'ni', 'ni']
```

5.3.5 Savoir si un élément est présent dans une liste

Pour savoir si un élément est présent dans une liste, on peut utiliser le mot-clé `in` :

```
>>> ma_liste = [1, 2, 3]
>>> 2 in ma_liste
True
>>> 12 in ma_liste
False
>>> 12 not in ma_liste
True
```

5.3.6 Comparer deux listes

On peut utiliser tous les opérateurs de comparaison entre deux listes.

```
>>> ma_liste = [1, 2, 3]
>>> ma_liste == [1, 2, 3]
True
>>> ma_liste != []
True
>>> ma_liste < [1, 2, 3, 4]
True
>>> ma_liste > [1, 2]
True
```

Pour que deux listes soient égales, elles doivent avoir la même taille et contenir des éléments deux à deux identiques.

5.3.7 Connaître le plus grand élément d'une liste

Pour connaître le plus grand élément d'une liste, il faut utiliser la fonction `max()` :

```
ma_liste = [10, 20, 30, 40]
element = max(ma_liste)
print(element)
# affiche 40
```

Cela suppose néanmoins que tous les éléments d'une liste sont comparables et donc de type similaire sinon on obtient une erreur de type `TypeError`.

```
>>> ma_liste = [10, "coconut"]
>>> element = max(ma_liste)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'str' and 'int'
```

5.3.8 Connaître le plus petit élément d'une liste

Pour connaître le plus petit élément d'une liste, il faut utiliser la fonction `min()` :

```
ma_liste = ["a", "b", "c"]
element = min(ma_liste)
print(element)
# affiche a
```

Comme pour la méthode `max()`, cela suppose que tous les éléments d'une liste sont comparables et donc de type similaire.

5.3.9 Découper une liste

Il est possible de découper (*slice*) une liste. Attention, cette opération crée une nouvelle liste et copie les éléments de la liste originelle vers la nouvelle liste. Pour découper, on utilise les crochets [ ] avec l'une des notations suivantes :

[index départ : indice supérieur]

[index départ : indice supérieur : pas d'incrément]

```
ma_liste = [10, 20, 30, 40]
autre_liste = ma_liste[1:3]
print(autre_liste)
# affiche [20, 30]
```

```
ma_liste = [10, 20, 30, 40]
derniers_elements = ma_liste[-2:4]
print(derniers_elements)
# affiche [30, 40]
```

```
ma_liste = [1, 2, 3, 4, 5, 6, 7, 8, 9]
nombres_pairs = ma_liste[1:9:2]
print(nombres_pairs)
# affiche [2, 4, 6, 8]
```

Par défaut, l'indice de départ est 0, l'indice de fin est le nombre d'éléments + 1 et le pas est 1. Il est donc possible d'omettre une ou plusieurs de ces valeurs.

```
ma_liste = [10, 20, 30, 40]
liste_sauf_dernier_element = ma_liste[:-1]
print(liste_sauf_dernier_element)
# affiche [10, 20, 30]

liste_sauf_premier_element = ma_liste[1:]
print(liste_sauf_premier_element)
# affiche [20, 30, 40]

un_sur_deux = ma_liste[::2]
print(un_sur_deux)
# affiche [10, 30]
```

Astuce : Pour le *slicing* de liste, vous pouvez indiquer comme indice de départ ou comme indice supérieur un indice en dehors de la liste. Cela ne génère pas d'erreur. Python s'arrêtera au dernier élément de la liste :

```
ma_liste = [10, 20, 30, 40]
nouvelle_liste = ma_liste[2:1000]
print(nouvelle_liste)
# affiche [30, 40]
```

(suite sur la page suivante)

(suite de la page précédente)

```
nouvelle_liste = ma_liste[1000:]
print(nouvelle_liste)
# affiche []
```

Le *slicing* peut être également utilisé pour modifier une liste. Par exemple pour supprimer des éléments à la fin ou au début d'une liste :

```
ma_liste = ['a', 'b', 'c', 'd']
del ma_liste[-2:]
print(ma_liste)
# affiche ['a', 'b']
```

5.3.10 Copier une liste

Prenons l'exemple suivant :

```
ma_liste = [10, 20, 30, 40]
une_autre_liste = ma_liste
```

On pourrait croire que le code ci-dessus copie la liste contenue depuis la variable `ma_liste` dans la variable `une_autre_liste`. Il n'en est rien ! Avec ce code, les deux variables **référencent** simplement la même liste.

Pour s'en convaincre, il suffit de modifier la liste à travers une variable et constater que les modifications sont visibles à partir de l'autre variable :

```
ma_liste = [10, 20, 30, 40]
une_autre_liste = ma_liste

ma_liste[0] = 0
print(une_autre_liste[0])
# affiche 0
```

Donc les variables sont comme des poignées qui permettent d'accéder à des données et plusieurs poignées peuvent accéder aux mêmes données. Les listes en Python sont des séquences mutables (*mutable sequences*), cela signifie que l'on peut modifier leur contenu. Ainsi toutes les variables qui référencent la même liste partagent les modifications effectuées sur cette liste.

Pour copier une liste, on peut utiliser la fonction `list` :

```
ma_liste = [10, 20, 30, 40]
une_autre_liste = list(ma_liste)

ma_liste[0] = 0
print(une_autre_liste[0])
# affiche 10
```

Astuce : Pour copier une liste, on peut aussi utiliser le *slicing* :

```
ma_liste = [10, 20, 30, 40]
nouvelle_liste = ma_liste[:]
```

`ma_liste[:]` crée une copie de la liste à partir de l'indice 0 et jusqu'à l'indice correspondant au nombre d'éléments plus 1 (non inclus). Donc, il s'agit d'un copie intégrale de la liste.

5.4 Quelques méthodes pour les listes

Les listes possèdent beaucoup de méthodes.

`append(e)` Pour ajouter un élément à la liste.

```
ma_liste = [1]
ma_liste.append(2)
ma_liste.append(3)
print(ma_liste)
# affiche [1, 2, 3]
```

`extend(l)` Pour ajouter une séquence à une liste.

```
ma_liste = [1]
ma_liste.extend([2, 3, 4])
print(ma_liste)
# affiche [1, 2, 3, 4]
```

`insert(i, e)` Pour insérer un nouvel élément à un index.

```
ma_liste = ['a', 'c', 'd']
ma_liste.insert(1, 'b')
print(ma_liste)
# affiche ['a', 'b', 'c', 'd']
```

`remove(e)` Pour supprimer un élément de la liste.

```
ma_liste = ['a', 'b', 'c', 'd']
ma_liste.remove('b')
print(ma_liste)
# affiche ['a', 'c', 'd']
```

`clear()` Pour supprimer tous les éléments d'une liste.

```
ma_liste = [10, 20, 30]
ma_liste.clear()
print(ma_liste)
# affiche []
```

`count(e)` Pour compter le nombre d'occurrences d'un élément.

```
ma_liste = ["John", "Eric", "Michael", "John", "Eric", "John"]
occurrence = ma_liste.count("John")
print(occurrence)
# affiche 3
```

`sort()` Pour trier une liste.

```
ma_liste = [3, 2, 6, 4]
ma_liste.sort()
print(ma_liste)
# affiche [2, 3, 4, 6]
```

`reverse()` Pour inverser l'ordre des éléments dans la liste.

```
ma_liste = [1, 2, 3]
ma_liste.reverse()
print(ma_liste)
# affiche [3, 2, 1]
```

On peut obtenir le même effet en utilisant le paramètre nommé `reverse` avec la fonction `sort()` :

```
ma_liste = [1, 2, 3]
ma_liste.sort(reverse=True)
print(ma_liste)
# affiche [3, 2, 1]
```

CHAPITRE 6

Le n-uplet (*tuple*)

Un n-uplet (appelé *tuple* en anglais) est une séquence non modifiable de données ordonnées. Cela signifie que chaque élément du tuple est associé à une position (un **index**). Le premier élément d'un tuple possède l'index 0 et le dernier élément possède l'index n - 1 pour une liste contenant n éléments. Contrairement aux listes, les tuples **ne sont pas modifiables**.

6.1 Créer un tuple

Pour créer un tuple, on écrit la séquence des éléments entre parenthèses `()` et séparés par des virgules :

```
tuple_vide = ()
mon_tuple = (10, 20, 30, 40)
tuple_depareille = (None, 1, "shrubbery", True)
```

Comme le montre le `tuple_depareille` ci-dessus, un tuple peut contenir des éléments de types différents. Il peut même contenir un élément ou des éléments `None`, ce qui signifie qu'il contient un élément qui n'est rien.

Astuce : Les parenthèses sont utilisés également pour modifier l'ordre de précellence des opérateurs arithmétiques et logiques. Il y a donc une ambiguïté lorsqu'une seule valeur est mise entre parenthèses :

```
a = (1)
```

Pour l'interpréteur Python, `a` va contenir **l'entier 1**. Pour indiquer que l'on veut créer un tuple d'un seul élément qui vaut 1, il faut écrire :

```
a = (1,)
```

6.2 Accéder aux éléments

Comme pour les listes, pour accéder à un élément d'un tuple, il faut utiliser les crochets et préciser l'index de l'élément.

```
mon_tuple = (10, 20, 30, 40)
premier_element = mon_tuple[0]
second_element = mon_tuple[1]
```

Comme pour les listes, il est possible d'utiliser un index négatif pour compter à partir de la fin du tuple. Le dernier élément a l'index **-1**.

6.3 Les opérations sur les tuples

On peut effectuer les mêmes opérations sur les tuples que sur les listes pour la consultation. Par définition, il n'est pas possible d'effectuer une opération modifiant le contenu d'un tuple.

6.3.1 Connaître la taille d'un tuple

Pour connaître le nombre d'éléments d'un tuple, il faut utiliser la fonction `len()` (qui est la contraction de *length*) :

```
mon_tuple = (10, 20, 30, 40)
taille_tuple = len(mon_tuple)
print(taille_tuple)
# affiche 4
```

6.3.2 Ajouter deux tuples ensemble

L'opérateur + permet d'ajouter deux tuples ensemble pour former un troisième tuple :

```
mon_tuple = (0, 1, 2) + (10, 20, 30)
print(mon_tuple)
# affiche (0, 1, 2, 10, 20, 30)
```

Astuce : Pour augmenter la taille d'un tuple, vous pouvez utiliser l'opération +=. Techniquement, le tuple n'est pas modifié, un nouveau tuple est créé et assigné à la variable à gauche

```
mon_tuple = (0, 1, 2)
mon_tuple_original = mon_tuple
mon_tuple += (10, 20, 30)
print(mon_tuple)
# affiche (0, 1, 2, 10, 20, 30)
print(mon_tuple == mon_tuple_original)
# affiche False car un nouveau tuple a été créé
print(mon_tuple_original)
# affiche (0, 1, 2)
```

6.3.3 Créer un tuple étendu

L'opérateur * permet de créer un tuple en répétant le contenu d'un tuple autant de fois que spécifié :

```
mon_tuple = ('ni') * 5
print(mon_tuple)
# affiche ('ni', 'ni', 'ni', 'ni', 'ni')
```

6.3.4 Savoir si un élément est présent dans un tuple

Pour savoir si un élément est présent dans un tuple on peut utiliser le mot-clé `in` :

```
>>> mon_tuple = (1, 2, 3)
>>> 2 in mon_tuple
True
>>> 12 in mon_tuple
False
>>> 12 not in mon_tuple
True
```

6.3.5 Comparer deux tuples

On peut utiliser tous les opérateurs de comparaison entre deux tuples.

```
>>> mon_tuple = (1, 2, 3)
>>> mon_tuple == (1, 2, 3)
True
>>> mon_tuple != ()
True
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> mon_tuple < (1, 2, 3, 4)
True
>>> mon_tuple > (1, 2)
True
```

Pour que deux tuples soient égaux, ils doivent avoir la même taille et contenir des éléments deux à deux identiques.

6.3.6 Connaître le plus grand élément d'un tuple

Pour connaître le plus grand élément d'un tuple, il faut utiliser la fonction `max()` :

```
mon_tuple = (10, 20, 30, 40)
element = max(mon_tuple)
print(element)
# affiche 40
```

Cela suppose néanmoins que tous les éléments d'un tuple sont comparables et donc de type similaire sinon on obtient une erreur de type `TypeError`.

```
>>> mon_tuple = (10, "coconut")
>>> element = max(mon_tuple)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'str' and 'int'
```

6.3.7 Connaître le plus petit élément d'une tuple

Pour connaître le plus petit élément d'un tuple, il faut utiliser la fonction `min()` :

```
mon_tuple = ("a", "b", "c")
element = min(mon_tuple)
print(element)
# affiche a
```

Comme pour la méthode `max()`, cela suppose que tous les éléments d'un tuple sont comparables et donc de type similaire.

6.3.8 Découper un tuple

Il est possible de découper (*slice*) un tuple. Comme le tuple n'est pas modifiable, cette opération crée un nouveau tuple et copie les éléments du tuple originel vers le nouveau tuple. Pour découper, on utilise les crochets `[]` avec l'une des notations suivantes :

[index départ : indice supérieur : pas d'incrément]
[index départ : indice supérieur]

```
mon_tuple = (10, 20, 30, 40)
autre_tuple = mon_tuple[1:3]
print(autre_tuple)
# affiche (20, 30)
```

```
mon_tuple = (10, 20, 30, 40)
derniers_elements = mon_tuple[-1:2]
print(derniers_elements)
# affiche (30, 40)
```

```
mon_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9)
nombres_pairs = mon_tuple[1:9:2]
print(nombres_pairs)
# affiche (2, 4, 6, 8)
```

Par défaut, l'indice de départ est 0, l'indice de fin est nombre d'éléments + 1 et le pas est 1. Vous pouvez omettre ces valeurs.

```
mon_tuple = (10, 20, 30, 40)
tuple_sauf_dernier_element = mon_tuple[:-1]
print(tuple_sauf_dernier_element)
# affiche (10, 20, 30)

tuple_sauf_premier_element = mon_tuple[1:]
print(tuple_sauf_premier_element)
# affiche (20, 30, 40)

un_sur_deux = mon_tuple[::2]
print(un_sur_deux)
# affiche (10, 30)
```

6.3.9 Compter les occurrences dans un tuple

Pour compter le nombre d'occurrences d'un élément, on utilise la méthode `count(e)`.

```
mon_tuple = ("John", "Eric", "Michael", "John", "Eric", "John")
occurrence = mon_tuple.count("John")
print(occurrence)
# affiche 3
```

6.4 Listes et tuples

Les listes et les tuples sont des séquences, il est donc possible de créer une tuple à partir d'une liste et réciproquement. Pour cela, on utilise les méthodes `tuple(seq)`

et `list(seq)` :

```
mon_tuple = (1, 2, 3)
ma_liste = list(mon_tuple)
print(ma_liste)
# affiche [1, 2, 3]

# on inverse l'ordre des éléments de la liste
ma_liste.reverse()

mon_tuple = tuple(ma_liste)
print(mon_tuple)
# affiche (3, 2, 1)
```

Attention, ces fonctions ne transforment pas la nature de la séquence, l'interpréteur crée en mémoire des copies de type `list` ou `tuple`.

Pourquoi utiliser des tuples plutôt que des listes ? Il s'agit bien souvent d'une optimisation pour l'exécution du code. En effet, un tuple est plus performant qu'une liste. Comme sa structure ne peut pas être modifiée, son implémentation permet un stockage en mémoire plus compact.

6.5 Les chaînes de caractères comme tuple

Les chaînes de caractères ne sont pas réellement des tuples, néanmoins elles sont des séquences non modifiables de caractères. Il est donc possible de leur appliquer les mêmes opérations.

```
>>> s = "Good night ding ding ding"
>>> s[0]
'G'
>>> "ding" in s
True
>>> "dong" not in s
True
>>> s[-4:]
'ding'
>>> s.count("ding")
3
>>> "Good" != s
True
>>> "Good" < s
True
>>> tuple(s)
('G', 'o', 'o', 'd', ' ', 'n', 'i', 'g', 'h', 't', ' ', 'd', 'i', 'n', 'g', ' ', 'd',
↪'i', 'n', 'g', ' ', 'd', 'i', 'n', 'g')
>>> list(s)
['G', 'o', 'o', 'd', ' ', 'n', 'i', 'g', 'h', 't', ' ', 'd', 'i', 'n', 'g', ' ', 'd',
↪'i', 'n', 'g', ' ', 'd', 'i', 'n', 'g']
```

CHAPITRE 7

L'ensemble (*set*)

Un ensemble est un groupement non ordonné d'éléments uniques.

Prudence : Pour qu'un élément puisse faire partie d'un ensemble, il faut qu'il puisse produire une valeur de hachage (*hash* en anglais). Il s'agit d'une valeur numérique qui permet d'identifier l'élément sans forcément lui être unique. Par défaut en Python, les nombres, les chaînes de caractères et les valeurs booléennes peuvent produire une valeur de hachage. Ils peuvent donc être mis dans un ensemble.

7.1 Créer un ensemble

Pour créer un ensemble, on écrit ses éléments entre accolades `{}` et séparés par des virgules :

```
mon_ensemble = {10, 20, 30, 40}
ensemble_depareillee = {None, 1, "shrubbery", True}
```

Comme le montre la variable `ensemble_depareillee` ci-dessus, un ensemble peut contenir des éléments de types différents. Il peut même contenir un élément ou des éléments `None` ce qui signifie qu'il contient un élément qui n'est rien.

Si vous déclarez plusieurs fois la même valeur dans un ensemble alors la valeur ne sera présente qu'un fois (les doublons sont ignorés).

```
mon_ensemble = {1, 2, 2, 3, 3, 3, 4, 4, 4, 4}
print(mon_ensemble)
# affiche {1, 2, 3, 4}
```

Prudence :

```
v = {}
```

Le code ci-dessus ne crée pas un ensemble vide mais un *dictionnaire* vide. Pour créer une dictionnaire vide, il faut utiliser la fonction `set()` :

```
v = set()
```

7.2 Accéder aux éléments

Il n'est **pas possible** d'accéder aux éléments d'un ensemble avec l'opérateur `[]`. En revanche, nous verrons avec les structures de contrôle qu'il est possible de parcourir un ensemble.

7.3 Les opérations sur les ensembles

7.3.1 Connaître la taille d'un ensemble

Pour connaître le nombre d'éléments d'un ensemble, il faut utiliser la fonction `len()` (qui est la contraction de *length*) :

```
mon_ensemble = {10, 20, 30, 40}
taille_ensemble = len(mon_ensemble)
print(taille_ensemble)
# affiche 4
```

7.3.2 Savoir si un élément est présent dans un ensemble

Pour savoir si un élément est présent dans un ensemble, on peut utiliser le mot-clé `in` :

```
>>> mon_ensemble = {1, 2, 3}
>>> 2 in mon_ensemble
True
>>> 12 in mon_ensemble
False
```

(suite sur la page suivante)

(suite de la page précédente)

```
>>> 12 not in mon_ensemble
True
```

7.3.3 Comparer deux ensembles

On peut utiliser tous les opérateurs de comparaison entre deux ensembles.

```
>>> mon_ensemble = {1, 2, 3}
>>> mon_ensemble == {1, 2, 3}
True
>>> mon_ensemble != {}
True
>>> mon_ensemble < {1, 2, 3, 4}
True
>>> mon_ensemble > {2}
True
```

Pour que deux ensembles soient égaux, il faut qu'ils contiennent les mêmes éléments.

7.3.4 Connaître le plus grand élément d'un ensemble

Pour connaître le plus grand élément d'un ensemble, il faut utiliser la fonction `max()` :

```
mon_ensemble = {10, 20, 30, 40}
element = max(mon_ensemble)
print(element)
# affiche 40
```

Cela suppose néanmoins que tous les éléments d'un ensemble sont comparables et donc de type similaire sinon on obtient une erreur de type `TypeError`.

```
>>> mon_ensemble = {10, "coconut"}
>>> element = max(mon_ensemble)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'str' and 'int'
```

7.3.5 Connaître le plus petit élément d'un ensemble

Pour connaître le plus petit élément d'un ensemble, il faut utiliser la fonction `min()` :

```
mon_ensemble = {"a", "b", "c"}
element = min(mon_ensemble)
print(element)
# affiche a
```

Comme pour la méthode `max()`, cela suppose que tous les éléments d'un ensemble sont comparables et donc de type similaire.

7.4 Quelques méthodes pour les ensembles

`copy()` Copie un ensemble pour créer un nouvel ensemble

```
mon_ensemble = {10, 20, 30}
nouvel_ensemble = mon_ensemble.copy()
```

`add()` Ajoute un élément à un ensemble

```
mon_ensemble = {10, 20, 30}
mon_ensemble.add(40)
print(mon_ensemble)
# affiche {10, 20, 30, 40}
```

`remove()` Supprime un élément d'un ensemble

```
mon_ensemble = {10, 20, 30}
mon_ensemble.remove(20)
print(mon_ensemble)
# affiche {10, 30}
```

`clear()` Supprime tous les éléments d'un ensemble

```
mon_ensemble = {10, 20, 30}
mon_ensemble.clear()
print(mon_ensemble)
# affiche {}
```

`union(e)` Crée un ensemble représentant l'union entre deux ou plusieurs ensembles :

```
e1 = {1, 2}
e2 = {2, 3}
e3 = e1.union(e2)
print(e3)
# affiche {1, 2, 3}
```

Astuce : L'union peut également se représenter avec l'opérateur `|` :

```
e1 = {1, 2}
e2 = {2, 3}
e3 = e1 | e2
print(e3)
# affiche {1, 2, 3}
```

---

intersection(e) Crée un ensemble représentant l'intersection entre deux ou plusieurs ensembles :

```
e1 = {1, 2}
e2 = {2, 3}
e3 = e1.intersection(e2)
print(e3)
# affiche {2}
```

Astuce : L'intersection peut également se représenter avec l'opérateur & :

```
e1 = {1, 2}
e2 = {2, 3}
e3 = e1 & e2
print(e3)
# affiche {2}
```

---

difference(e) Crée un ensemble représentant la différence entre deux ou plusieurs ensembles :

```
e1 = {1, 2}
e2 = {2, 3}
e3 = e1.difference(e2)
print(e3)
# affiche {1}
```

Astuce : La différence peut également se représenter avec l'opérateur - :

```
e1 = {1, 2}
e2 = {2, 3}
e3 = e1 - e2
print(e3)
# affiche {1}
```

---

symmetric_difference(e) Crée un ensemble contenant les éléments présents dans un des ensembles mais pas dans les deux :

```
e1 = {1, 2}
e2 = {2, 3}
e3 = e1.symmetric_difference(e2)
print(e3)
# affiche {1, 3}
```

Astuce : La différence symétrique peut également se représenter avec l'opérateur ^ :

```
e1 = {1, 2}
e2 = {2, 3}
e3 = e1 ^ e2
print(e3)
# affiche {1, 3}
```

7.5 Ensemble, liste et tuple

Les ensembles, listes et tuples sont des collections d'éléments, il est possible de passer de l'un à l'autre grâce aux méthode `set`, `list`, `tuple`.

```
ma_liste = [1, 1, 2, 2]
mon_ensemble = set(ma_liste)
mon_tuple = tuple(mon_ensemble)
```

Prudence : Comme un ensemble n'est pas ordonné, si vous créez une liste ou un tuple à partir d'un ensemble, vous n'avez **aucune garantie sur l'ordre des éléments** dans la liste ou le tuple qui sera créé.

7.6 Ensemble figé

On peut créer un ensemble non modifiable grâce à la fonction `frozenset()`. Cette fonction attend en paramètre une séquence (liste, tuple ou ensemble) :

```
mon_ensemble = (1, 2, 3)
ensemble = frozenset(mon_ensemble)
```

Un ensemble figé se comporte comme un ensemble sauf que l'on obtient une erreur pour toute tentative de modification.

CHAPITRE 8

Le dictionnaire

Un dictionnaire (*dictionnary* ou, en abrégé en Python, *dict*) est une collection qui associe une clé à une valeur. Par exemple, il est possible d'associer la clé `"nom"` à un nom et la clé `"prenom"` à un prénom.

Prudence : Pour qu'une donnée puisse être utilisée comme une clé dans un dictionnaire, il faut qu'elle puisse produire une valeur de hachage (*hash* en anglais). Il s'agit d'une valeur numérique qui permet d'identifier la clé sans forcément lui être unique. Par défaut en Python, les nombres, les chaînes de caractères et les valeurs booléennes peuvent produire une valeur de hachage. Ils peuvent donc être utilisés comme clé dans un dictionnaire. On peut aussi utiliser comme clé un tuple d'éléments produisant une valeur de hachage.

8.1 Créer un dictionnaire

Pour créer un dictionnaire, on associe une clé à une valeur en les séparant par `:`, le tout entre accolades `{}` :

```
mon_dict = {"nom": "Gayerie", "prenom": "David"}
```

8.2 Accéder aux éléments

Pour accéder à un élément d'une liste, il faut utiliser les crochets et préciser la valeur de la clé.

```
mon_dict = {"nom": "Gayerie", "prenom": "David"}
nom = mon_dict["nom"]
prenom = mon_dict["prenom"]
```

Attention, si un programme veut accéder à une valeur à partir d'une clé qui n'existe pas, l'erreur `KeyError` est produite.

```
>>> mon_dict = {"nom": "Gayerie", "prenom": "David"}
>>> surnom = mon_dict["surnom"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'surnom'
```

Il est possible de changer la valeur pour une clé donnée ou ajouter une nouvelle valeur pour une nouvelle clé :

```
mon_dict = {"nom": "Gayerie", "prenom": "Eric"}
mon_dict["prenom"] = "David"
mon_dict["surnom"] = "Spoonless"
print(mon_dict)
# affiche {'nom': 'Gayerie', 'prenom': 'David', 'surnom': 'Spoonless'}
```

8.3 Les opérations sur les dictionnaires

8.3.1 Connaître la taille d'un dictionnaire

Pour connaître le nombre de valeurs dans un dictionnaire, il faut utiliser la fonction `len()` (qui est la contraction de *length*) :

```
mon_dict = {"nom": "Gayerie", "prenom": "Eric"}
taille_dict = len(mon_dict)
print(taille_dict)
# affiche 2
```

8.3.2 Supprimer une clé d'un dictionnaire

Pour supprimer une clé dans un dictionnaire, on utilise le mot-clé `del` :

```
mon_dict = {"nom": "Gayerie", "prenom": "Eric"}
del mon_dict["prenom"]
print(mon_dict)
# affiche {'nom': 'Gayerie'}
```

8.3.3 Savoir si une clé est présente dans un dictionnaire

Pour savoir si une clé est présente dans un dictionnaire, on peut utiliser le mot-clé `in` :

```
>>> mon_dict = {"nom": "Gayerie", "prenom": "David"}
>>> "nom" in mon_dict
True
>>> "surnom" in mon_dict
False
>>> "surnom" not in mon_dict
True
```

8.3.4 Comparer deux dictionnaires

On peut tester si deux dictionnaires sont ou non identiques.

```
>>> dict1 = {"nom": "Gayerie", "prenom": "Eric"}
>>> dict2 = {"nom": "Gayerie", "prenom": "David"}
>>> dict1 == dict2
False
>>> dict1 != dict2
True
```

8.4 Quelques méthodes pour les dictionnaires

`get(key[, default])` Retourne la valeur associée à la clé fournie en paramètre. Cette méthode diffère de l'utilisation des crochets `[]` car elle retourne la valeur du paramètre `default` (par défaut `None`) si la clé n'est pas présente dans le dictionnaire.

```
dict = {"nom": "Gayerie", "prenom": "David"}
print(dict.get("surnom"))
# affiche None
print(dict.get("surnom", "Pas de valeur"))
# affiche Pas de valeur
```

`clear()` Supprime tous les éléments du dictionnaire.

`items()` Retourne une vue du dictionnaire sous la forme d'une séquence itérable de tuples contenant le couple clé, valeur. Les modifications réalisées sur la vue sont répercutées sur le dictionnaire.

```
dict = {"nom": "Gayerie", "prenom": "David"}
vue_dict = dict.items();
print(vue_dict)
# Affiche dict_items([('nom', 'Gayerie'), ('prenom', 'David')])
```

keys() Retourne une vue du dictionnaire sous la forme d'une séquence itérable contenant les clés. Les modifications réalisées sur la vue sont répercutées sur le dictionnaire.

```
dict = {"nom": "Gayerie", "prenom": "David"}
vue_dict = dict.keys();
print(vue_dict)
# Affiche dict_keys(['nom', 'prenom'])
```

values() Retourne une vue du dictionnaire sous la forme d'une séquence itérable contenant les valeurs. Les modifications réalisées sur la vue sont répercutées sur le dictionnaire.

```
dict = {"nom": "Gayerie", "prenom": "David"}
vue_dict = dict.values();
print(vue_dict)
# Affiche dict_values(['Gayerie', 'David'])
```

copy() Crée une copie d'un dictionnaire.

CHAPITRE 9

Les structures de contrôle

Les structures de contrôle permettent de créer des embranchements dans le flot d'exécution d'un programme. Python, comme la plupart des langages de programmation impératifs, prévoit des instructions pour représenter des décisions et des itérations.

9.1 L'indentation du code

Une structure de contrôle est associée à un bloc de code qui doit être ou non exécuté (structure de décision) ou répété (structure itérative). Comme pour le corps des fonctions, le code associée à une structure de contrôle est indiqué par l'indentation (généralement de quatre espaces supplémentaires à gauche).

Astuce : Une structure de contrôle doit contenir obligatoirement un bloc de code. Si vous voulez laisser le bloc vide, vous pouvez utiliser le mot-clé `pass`.

```
if a < 0:
    pass
```

9.2 Structures de décision

9.2.1 L'instruction *if*

La structure de décision `if` exécute un bloc de code si l'expression associée est évaluée à `True`.

```
if 'b' in 'bonjour':
    print ("La lettre b est présente dans le mot bonjour")

if 'z' not in 'bonjour':
    print ("La lettre z est présente dans le mot bonjour")
```

```
a = 0

if a < 2:
    a += 1

if a < 2:
    a += 1

if a < 2:
    a += 1

if a < 2:
    a += 1

print(a)
# affiche 2
```

Pour simplifier l'écriture, beaucoup de types peuvent directement s'évaluer comme une expression booléenne en python. Par exemple :

- une variable ou un paramètre est équivalent à `False` s'il vaut `None`
- un nombre est équivalent à `True` s'il est différent de 0
- une chaîne de caractères est équivalente à `True` si elle n'est pas la chaîne vide `""`
- une liste, un ensemble, un dictionnaire ou un tuple est équivalent à `True` s'il n'est pas vide

```
a = None
if not a:
    print(a, "est équivalent à False")

a = 1
if a:
    print(a, "est équivalent à True")

a = "bonjour"
if a:
    print(a, "est équivalent à True")

a = []
if not a:
    print(a, "est équivalent à False")
```

9.2.2 L'instruction *else*

Il est possible d'ajouter un bloc `else` qui sera exécuté si la condition est évaluée à `False` :

```
a = 2
if a % 2:
    print(a, "est un nombre impair")
else:
    print(a, "est un nombre pair")
```

9.2.3 L'instruction *elif*

Si on souhaite représenter plusieurs alternatives, il est possible d'utiliser le mot-clé `elif` :

```
def get_mention(note):
    if note < 10:
        return "insuffisant"
    elif note <= 12:
        return "passable"
    elif note <= 14:
        return "assez bien"
    elif note <= 16:
        return "bien"
    else:
        return "trés bien"
```

Il est possible de créer des expressions booléennes complexes en les combinant grâce aux mots-clés `and` ou `or` ou en chaînant les opérateurs de comparaison :

```
a = 50
b = -2
if 0 <= a <= 100 and a % 2 and b < 2:
    pass
```

9.2.4 L'opérateur ternaire

Il est possible d'affecter une variable à partir d'une décision en l'exprimant sur une seule ligne :

```
a = 1
# msg vaut "impair" si a % 2 est différent de 0 sinon msg vaut "pair"
msg = "impair" if a % 2 else "pair"
```

9.3 Exercice

Exercice : comparaison à la moyenne

Écrivez un programme qui demande à l'utilisateur sa taille et son sexe (homme/femme) puis qui indique l'écart en centimètres par rapport à la moyenne en fonction de son sexe. On pourra utiliser comme moyenne 1m75 pour les hommes et 1m65 pour les femmes.

9.4 Structures itératives

Les structures itératives permettent de répéter un bloc de code.

9.4.1 L'instruction *while*

Avec l'instruction `while`, le bloc de code associé est répété tant que la condition est vraie.

```
reponse = None
while reponse != "oui":
    reponse = input("Voulez-vous arrêter ? (oui/non) ")
```

Prudence : Si la condition n'est jamais fausse, le bloc se répète à l'infini.

```
while True:
    print("ceci va continuer encore et encore...")
```

9.4.2 L'instruction *for*

L'instruction `for` permet de parcourir (itérer) tous les éléments d'une séquence (tuple, ensemble, liste, dictionnaire, chaîne de caractères).

```
ma_liste = ["premier", "deuxième", "troisième"]
for element in ma_liste:
    print(element)
```

Dans l'exemple ci-dessus la portée de la variable `element` est limitée au bloc de code associé à l'instruction `for`. La variable `element` prend successivement la valeur des éléments de la liste.

Utiliser une chaîne de caractères comme séquence permet de parcourir un à un les caractère qui la compose :

```
message = "Hello world"
for l in message:
    print(l)
```

9.4.3 La fonction *range*

La fonction `range` permet de parcourir une liste de nombres. La méthode accepte :

— un paramètre pour indiquer la borne supérieure (en commençant la liste à 0 et sans inclure la borne supérieure)

```
for i in range(5):
    print(i)
# Affiche
#   0
#   1
#   2
#   3
#   4
```

— deux paramètres pour indiquer l'intervalle avec le début et la fin (exclue)

```
for i in range(3, 5):
    print(i)
# Affiche
#   3
#   4
```

— trois paramètres pour indiquer l'intervalle avec le début, la fin (exclue) et le pas d'incrément entre chaque valeur

```
for i in range(1, 20, 2):
    print(i)
# Affiche
#   1
#   3
#   5
#   7
#   9
#   11
#   13
#   15
#   17
#   19
```

Il est possible de donner un pas négatif pour obtenir une suite décroissante :

```
for i in range(5, 0, -1):
    print(i)
# Affiche
#   5
#   4
#   3
#   2
#   1
```

9.4.4 Interrompre un bloc d'itération

Il est possible d'utiliser les mots-clés `break` et `continue` pour interrompre l'exécution d'un bloc d'itération.

`break` interrompt immédiatement le bloc et le flot d'exécution sort de la boucle et reprend après le bloc.

```
while True:
    if input("Voulez-vous arrêter ? (oui/non) ") == "oui":
        break
print("C'est fini.")
```

`continue` interrompt immédiatement le bloc et on passe à l'itération suivante.

```
for l in "Python":
    if l in "aeiouy":
        continue
    print(l)
```

9.4.5 Structure itérative avec *else*

Il est possible d'utiliser une clause `else` après un `while` ou un `for`. Cette clause est exécutée lorsque l'expression de la condition `while` est fausse ou dès que la fin de la séquence à parcourir avec l'instruction `for` est atteinte.

L'intérêt du bloc `else` réside dans le fait qu'il **n'est pas exécuté** si l'itération a été interrompue par un `break`.

```
for i in range(10):
    if input("choisissez-vous le nombre %d ? (oui/non) " % i) == "oui":
        break
else:
    print("il fallait choisir un nombre... dommage")
```

9.5 Exercice

Exercice : calcul de la moyenne

Écrivez un programme qui demande à l'utilisateur de saisir une suite de nombres jusqu'à ce qu'il appuie directement sur `Entrer`.

Puis le programme affiche le tableau des chiffres donnés et ensuite le programme calcule et affiche la moyenne.

Exercice : le code de César

Le code de César est un algorithme de chiffrement classique, utilisé par Jules César dans ses correspondances secrètes. Il se base sur le principe du décalage. Une lettre est remplacée par son équivalent dans l'alphabet en effectuant un décalage. Le programme ci-dessous utilise un décalage de 23. Cela donne comme équivalence :

- a → x
- b → y
- c → z
- d → a
- e → b
- f → c
- g → d
- h → e
- i → f
- j → g
- k → h
- l → i
- m → j
- n → k
- o → l
- p → m
- q → n
- r → o
- s → p
- t → q
- u → r
- v → s
- w → t
- x → u
- y → v
- z → w

Ainsi la phrase :

> In cryptography, a Caesar cipher is one of the simplest and most widely known encryption techniques.

sera encodée avec un décalage de 23 comme ceci :

> Fk zovmqldoxmev, x Zxbpxo zfmebo fp lkb lc qeb pfjmibpq xka jlpq tfabiv hkltk bkzovmqflk qbzekfnrbp.

Écrivez un programme qui demande à l'utilisateur d'entrer une phrase. Puis le programme doit afficher la phrase chiffrée avec un décalage de 23 lettres.

Astuce : Vous pouvez, par exemple, vous aider des fonctions `ord()` et `chr()`. La première retourne le code d'un caractère et la seconde retourne un caractère à partir de son code.

Exercice : le code de César (pythonique)

Écrivez à nouveau le programme précédent mais en utilisant la fonction `str.maketrans()` pour créer une table de traduction. Utilisez ensuite la méthode `str.translate()` pour chiffrer la phrase saisie par l'utilisateur.

Vous pouvez trouver un exemple d'utilisation dans ce tutoriel.

CHAPITRE 10

Les fonctions (notions avancées)

10.1 décompactage des paramètres

Lors de l'appel d'une fonction, il est possible d'utiliser le contenu d'un tableau pour désigner les paramètres. On utilise pour cela l'opérateur * parfois appelé *unpack operator*.

Par exemple, la fonction `divmod()` attend deux paramètres, le numérateur et le dénominateur et retourne le résultat de la division entière et le reste. Il est possible d'appeler cette fonction en décompactant un tableau de deux éléments.

```
t = [20, 4]
resultat, reste = divmod(*t)
print("Le résultat de la division est", resultat, "avec comme reste", reste)
# Affiche Le résultat de la division est 5 avec comme reste 0
```

Dans l'exemple ci-dessus, l'appel à `divmod(*t)` est un exemple de décompactage. Le premier élément du tableau est passé en premier paramètre et le second élément en deuxième paramètre.

Prudence : Pour que le décompactage fonctionne, il faut que l'ordre des éléments du tableau, leur type et leur nombre correspondent à ce que la fonction attend comme paramètres.

Astuce : Le décompactage est possible pour toutes les structures de données itérables. Donc cela fonctionne également avec les tuples ou les ensembles.

```
t = (20, 4)
resultat, reste = divmod(*t)
print("Le résultat de la division est", resultat, "avec comme reste", reste)
# Affiche Le résultat de la division est 5 avec comme reste 0
```

Nous avons vu que Python accepte également le passage de paramètres par leur nom. Il donc également possible de décompacter un dictionnaire. La clé donne le nom du paramètre et la valeur, la valeur du paramètre.

Si nous reprenons l'exemple que nous avions pris dans un chapitre précédent :

```
def dire_bonjour_a(prenom, nom):
    print("Bonjour", prenom, nom)
```

Nous pouvons décompacter une dictionnaire lors de l'appel avec l'opérateur ** :

```
d = {"nom": "Gayerie", "prenom": "David"}
dire_bonjour_a(**d)
# Affiche Bonjour David Gayerie
```

Astuce : Vous pouvez utiliser l'opérateur de décompactage * sur un dictionnaire mais cela signifie que vous passez à la fonction les clés du dictionnaire comme paramètres.

Astuce : Vous pouvez combiner le décompactage à partir d'une séquence et d'un dictionnaire pour passer à la fois des paramètres et des paramètres nommés lors de l'appel d'une fonction.

```
s = ["David"]
d = {"nom": "Gayerie"}
dire_bonjour_a(*s, **d)
# Affiche Bonjour David Gayerie
```

10.2 compactage des paramètres

Symétriquement, il est possible de déclarer une fonction qui accepte un nombre quelconque de paramètres. Pour cela on compacte les paramètres sous la forme d'un tuple grâce à l'opérateur *. Par convention, on appelle généralement ce tuple `args` :

```
def moyenne(x, *args):
    """Calcule la moyenne d'un nombre quelconque de valeurs passées en paramètres."""
```

(suite sur la page suivante)

(suite de la page précédente)

```
    nb = 1 + len(args)
    somme = x
    for y in args:
        somme += y
    return somme / nb
```

La fonction ci-dessus oblige à passer au moins un paramètre appelé x. Elle utilise le compactage pour représenter le reste des paramètres sous la forme d'un tuple que la fonction peut ensuite parcourir pour calculer la somme et faire la moyenne. Donc, les appels suivants à la fonction sont valides :

```
resultat = moyenne(5)
print(resultat)
# Affiche 5.0

resultat = moyenne(5, 2)
print(resultat)
# Affiche 3.5

resultat = moyenne(5, 2, 10, 20, 12, 22)
print(resultat)
# Affiche 11.833333333333334
```

Une fonction peut également accepter un nombre quelconque de paramètres nommés en compactant les paramètres sous la forme d'un dictionnaire grâce à l'opérateur **. Par convention, on appelle généralement ce dictionnaire `kw` ou `kwargs` :

```
def afficher_params(**kwargs):
    """affiche tous les paramètres nommés passé en paramètres"""
    for k, v in kwargs.items():
        print("Paramètre", k, "qui a comme valeur", v)
```

Il est possible d'appeler cette fonction avec n'importe quels paramètres nommés :

```
afficher_params(prenom="David", taille=174)
```

Enfin, il est possible d'associer les deux formes de compactage est ainsi créer une fonction qui accepte n'importe quels paramètres :

```
def fonction_libre(*args, **kwargs):
    pass
```

10.3 Appel récursif de fonction

Une fonction peut bien évidemment appeler une autre fonction dans le corps de son traitement. Mais une fonction peut aussi s'appeler elle-même. On parle alors **d'appel récursif**. Un appel récursif implique **toujours** une condition d'arrêt sinon la fonction

va s'appeler sans fin... ou plus exactement il y aura une fin appelée **débordement de pile d'appel** (*call stack overflow*) qui fait échouer le programme avec une erreur de type `RecursionError` :

```
def appel_recursif():
    """Si vous appelez cette fonction, elle fera échouer votre programme !"""
    appel_recursif()
```

Une condition d'arrêt empêche la fonction de s'appeler elle-même à l'infini. Un exemple classique d'appel récursif et le calcul de la factorielle d'un nombre. Sachant que :

$$n ! = n \times (n-1) !$$
$$0 ! = 1$$

On peut écrire la fonction `factorielle(n)` de manière récursive :

```
def factorielle(n):
    if n > 0:
        return n * factorielle(n-1)
    else:
        return 1
```

10.4 Déclaration de fonction dans une fonction

Il est possible de déclarer une fonction dans une fonction.

```
def outter(message):
    def inner():
        print(message)
    inner()

outter("hello")
# Afficher hello
```

Dans l'exemple ci-dessus, la fonction `outter(message)` prend un paramètre, elle définit la fonction `inner()` qui affiche la valeur du message passé à `outter(message)`. Puis la fonction `outter(message)` appelle la fonction `inner()`. On voit que ces deux fonctions possèdent un lien entre-elles puisque `inner()` peut utiliser le paramètre passé à `outter(message)`. On désigne ce phénomène sous le nom de fermeture (*closure*) pour indiquer que, lors de sa déclaration, un fonction décrit un espace fermé de valeurs auxquelles elle a accès. En Python, une fonction déclarée dans une autre fonction inclut dans sa fermeture les paramètres et les variables de la fonction englobante.

Note : Dans des cas avancés, la fonction interne peut avoir besoin de modifier une variable déclarée dans la fonction englobante. Il faut pour cela utiliser le mot-clé `nonlocal` pour indiquer que la variable n'est pas déclarée dans la fonction interne mais dans la fonction englobante :

```
1 def outter(prenom):
2     message = None
3     def inner():
4         nonlocal message
5         message = "Bonjour %s" % prenom
6     inner()
7     print(message)
```

À la ligne 4, on précise que la variable `message` n'est pas locale à la fonction `inner()`, ce qui signifie que cette variable fait référence à celle déclarée dans la méthode `outter(message)`.

Note : Cet usage avancé de la déclaration de fonction sera très utile lorsque nous créerons des décorateurs de fonction.

10.5 La fonction comme type Python

Une fonction est un type Python comme un autre. Elle n'a pas un statut plus particulier dans le langage que les entiers, les chaînes de caractères ou les séquences. Elle est simplement appelable (*callable*). Mais nous verrons avec la programmation orientée objet en Python que n'importe quel type peut être appelable.

Donc si une fonction n'est pas différente des autres types, cela signifie qu'il est possible d'affecter une fonction à une variable, de passer une fonction en paramètre d'une autre fonction ou encore de retourner une fonction comme résultat à l'appel d'une fonction.

```
# on affecte la fonction min à une variable
ma_fonction_min = min
x = ma_fonction_min(2,3)
print(x)
# Affiche 2
```

Note : Il est même possible de faire des choses un peu dangereuses car vite incompréhensibles, telles que :

```
min, max = max, min
```

Le code ci-dessus affecte `min()` à `max()` et `max()` à `min()`. Donc maintenant :

```
print(max(1,9))
# Affiche 1
```

(suite sur la page suivante)

(suite de la page précédente)

```
print(min(1,9))
# Affiche 9
```

10.6 Les fonctions anonymes (lambdas)

Une lambda est une fonction anonyme (c'est-à-dire une fonction qui est déclarée sans être associée à un nom). Le terme *lambda* est emprunté à la méthode formelle du lambda-calcul. Les fonctions lambda (ou plus simplement les lambdas) sont utilisées dans la programmation fonctionnelle.

En Python pour déclarer une fonction anonyme, on utilise le mot-clé `lambda` :

```
neg = lambda x: -x
print(neg(1))
# Affiche -1

somme = lambda x, y: x + y
print(somme(2, 3))
# Affiche 5
```

Les lambdas sont utilisées pour déclarer des traitements très simples comme ci-dessus et qui peuvent s'écrire sur une seule ligne. Contrairement aux fonctions nommés, on considère que la fonction lambda retourne toujours le dernier résultat produit. Ainsi :

```
lambda x, y: x + y
```

retourne implicitement le résultat de l'addition des paramètres `x` et `y`. Par soucis de simplification d'écriture, la liste des paramètres d'une lambda ne s'écrivent pas entre parenthèses.

Les fonctions lambdas sont très utiles lorsqu'une fonction attend en paramètre une autre fonction. On peut ainsi passer en paramètre une fonction anonyme.

CHAPITRE 11

Liste en compréhension

Une liste en compréhension (*comprehension list*) permet de créer une liste à partir d'une itération.

```
liste = [x for x in range(5)]
print(liste)
# affiche [0, 1, 2, 3, 4]
```

Un intérêt des listes en compréhension est aussi de permettre de construire une nouvelle liste à partir d'une liste existante :

```
liste = [1, 2, 3, 4]
nouvelle_liste = [2 * x for x in liste]
print(nouvelle_liste)
# affiche [2, 4, 6, 8]
```

```
liste = ["hello", "the", "world"]
nouvelle_liste = [len(x) for x in liste]
print(nouvelle_liste)
# affiche [5, 3, 5]
```

Il est également possible d'appliquer un filtre lors de la création d'une liste en compréhension afin de ne pas prendre un compte certains éléments de la liste ou de l'itération de départ.

```
liste = ["hello", "the", "world"]
nouvelle_liste = [x for x in liste if len(x) < 4]
print(nouvelle_liste)
# affiche ['the']
```

En combinant la liste en compréhension avec des méthodes telles que `zip()`, il est possible de construire une liste comme le résultat d'opérations sur plusieurs listes :

```
liste1 = [1, 2, 3, 4]
liste2 = [10, 20, 30, 40]
nouvelle_liste = [x * y for x, y in zip(liste1, liste2)]
print(nouvelle_liste)
# affiche [10, 40, 90, 160]
```

Il est également possible de combiner deux itérations dans une liste en compréhension. Cela permet de réaliser une itération sur chaque élément de la liste ou de l'itération de départ :

```
liste = ["hello", "the", "world"]
nouvelle_liste = [c for mot in liste for c in mot]
print(nouvelle_liste)
# affiche ['h', 'e', 'l', 'l', 'o', 't', 'h', 'e', 'w', 'o', 'r', 'l', 'd']
```

Dans l'exemple ci-dessus, la liste est construite en itérant sur chaque mot de la liste et donc de créer un tableau de l'ensemble des lettres.

11.1 Dictionnaire en compréhension

Il est également possible de créer un dictionnaire en compréhension en construisant un couple clé : valeur à partir d'une itération :

```
liste = ["liste", "avec", "des", "mots"]
dictionnaire = {len(e) : e for e in liste}
print(dictionnaire)
# Affiche {5: 'liste', 4: 'mots', 3: 'des'}
```

Par exemple, on peut ainsi inverser la clé et la valeur

```
dictionnaire = {"pomme": 8, "poire": 3, "orange": 7}
nouveau_dict = {v : k for k, v in dictionnaire.items()}
print(nouveau_dict)
# Affiche {8: 'pomme', 3: 'poire', 7: 'orange'}
```

CHAPITRE 12

Les modules

Un programme Python est écrit dans un fichier portant l'extension **.py**. Cependant, pour la plupart des programmes, le recours à un seul fichier n'est pas très pratique car le nombre de lignes de code augmente très rapidement au fur et à mesure de l'ajout de fonctionnalités dans l'application.

Les modules permettent de découper logiquement le code source de l'application à travers plusieurs fichiers. Un module peut même être partagé entre plusieurs applications afin de créer une bibliothèque logiciel réutilisable. Par exemple, Python fournit déjà une bibliothèque standard utilisable par toutes les applications sous la forme de plusieurs modules.

12.1 Le module pour prévenir la collision de nom

Un problème récurrent en programmation est celui de la **collision de noms**. Écrire un programme implique de nommer les différents éléments du programme (variables, paramètres, fonctions, objets…). Si on désire réutiliser des fonctions développées par d'autres personnes, il est possible que l'on soit amené à incorporer des fonctions qui portent le même nom. En Python, si on déclare une fonction avec un nom déjà utilisé pour désigner une autre fonction, cette dernière ne sera plus accessible. On se retrouve confronté à un phénomène de collision des noms.

Pour résoudre cette situation, la solution la plus couramment utilisée consiste à créer des **espaces de noms** différents pour les deux fonctions. En Python, un module porte un nom qui identifie un espace dans lequel on peut définir des fonctions. Ainsi si les modules `module1` et le `module2` définissent tous les deux une fonction s'appelant `abs()`, il sera possible de les distinguer à l'appel en préfixant le nom de la fonction par le nom du module :

```
module1.abs()
module2.abs()
abs(2)
```

Dans l'exemple ci-dessus, il est même possible d'appeler la méthode standard `abs()` qui permet de calculer la valeur absolue d'un nombre.

12.2 La bibliothèque standard Python

La bibliothèque standard Python contient déjà beaucoup de modules. La liste ci-dessous énumère les modules les plus utiles à connaître pour débuter en Python :

`sys`	variables et fonctions pour interagir avec l'interpréteur Python
`os`	fonctions élémentaires pour interagir avec le système d'exploitation
`math`	fonctions mathématiques avancées
`random`	bibliothèque pour la génération de nombres aléatoires
`datetime`	représentation des dates et du temps
`calendar`	gestion du calendrier
`collections`	structures de données supplémentaires pour les séquences et les dictionnaires

12.3 Importer un module

Pour accéder à un élément d'un module, il faut d'abord importer le module grâce au mot-clé `import`. Par exemple, le module `sys` définit la variable `version` qui contient la version de l'interpréteur Python. Pour pouvoir y accéder, il faut préalablement importer le module `sys` :

```
import sys

print(sys.version)
```

De même, le module `random` définit la fonction `random()` qui retourne un nombre compris entre 0 et 1 :

```
import random

nombre = random.random()
print(nombre)
```

L'instruction `import` peut être placée à n'importe quel endroit dans un programme (y compris dans le corps d'une fonction). Cela permet d'importer le module uniquement lorsqu'il est réellement nécessaire d'accéder à un élément qu'il définit.

Note : Si vous souhaitez importer plusieurs modules, vous pouvez utiliser une seule instruction `import` en séparant le nom des modules par une virgule :

```
import os, sys, random
```

Il est possible de donner un alias au moment de l'import pour référencer le module sous un autre nom.

```
import random as r

nombre = r.random()
print(nombre)
```

12.4 Importer directement un nom

Parfois, vous ne souhaitez pas importer tout un module et il peut être fastidieux de préfixer systématiquement un nom par le module qui le définit. Il existe la syntaxe `from ... import` pour résoudre cette situation :

```
from sys import version

print(version)
```

Grâce à cette syntaxe, la variable `version` du module `sys` est directement accessible par son nom.

Cette syntaxe est également utilisable pour les fonctions :

```
from math import factorial

f = factorial(5)
print(f)
# Affiche 120
```

Note : Si vous souhaitez importer plusieurs noms du même module, vous pouvez utiliser une seule instruction `from ... import` en séparant les noms par une virgule :

```
from sys import version, platform, exit
```

Il est possible de donner un alias au moment de l'import pour référencer l'élément importé.

```python
from math import factorial as f
resultat = f(3)
print(resultat)
# Affiche 6
```

Note : Vous pouvez importer tous les noms définis dans un module en utilisant * :

```python
from math import *

v = factorial(5)
f = floor(10.25)
l = log(2)

print(v, f, l)
# Affiche 120 10 0.6931471805599453
```

Cette syntaxe n'est pas conseillée car il est difficile de connaître avec certitude la liste des noms importés.

12.5 Créer un module à partir d'un fichier

Tout fichier source Python (fichier avec l'extension *.py*) peut être utilisé comme un module. Il est donc très simple de concevoir une application Python composée de plusieurs fichiers : un fichier sert de fichier principal pour le lancement de l'application tandis que les autres fichiers sont utilisés comme des modules.

Supposons qu'une application python est composée de deux fichiers : `app.py` qui contient le programme principal et `entree_sortie.py` qui regroupe les fonctions pour interagir avec l'utilisateur.

Code source 1 – Le fichier `entree_sortie.py`

```python
def demander_nom():
    nom = input("Comment vous appelez-vous ? ")
    return nom

def afficher(*args):
    print(*args)
```

Code source 2 – Le fichier `app.py`

```python
from entree_sortie import demander_nom, afficher

nom = demander_nom()
afficher("Bonjour", nom)
```

Le nom du module correspond au nom du fichier. Le programme principal dans

le fichier `app.py` peut importer et utiliser les fonctions définies dans le module `entree_sortie`, c'est-à-dire dans le fichier `entree_sortie.py`.

12.6 Module exécutable

Si un module déclare directement du code en dehors de toute fonction, ce code sera exécuté lors du premier import du module. Si vous le souhaitez, un module peut avoir une comportement différent s'il est exécuté directement par l'interpréteur (comme programme principal) ou s'il est importé depuis un autre fichier. Cela permet de créer des modules exécutables de manière autonome. Pour cela, il suffit de tester la valeur de l'attribut du module appelé `__name__`. Si cet attribut vaut `"__main__"` alors cela signifie que le fichier est lancée directement par l'interpréteur. Il n'est pas importé et n'agit donc pas comme un module.

```
if __name__ == "__main__":
    # Il est possible d'exécuter du code pour un module qui est
    # directement appelé depuis l'interpréteur Python
    pass
```

12.7 Créer un module à partir d'un répertoire

Parfois un module est tellement complexe qu'il peut être organisé dans plusieurs fichiers, chacun agissant comme un sous-module du module principal. Dans ce cas, il faut placer tous les fichiers du module dans un répertoire portant le nom du module. Ce répertoire **doit** contenir un fichier nommé `__init__.py` qui représente le point d'entrée du module.

Si nous reprenons notre exemple précédent, nous pouvons créer le module `entree_sortie` en créant un répertoire du même nom avec, par exemple, les fichiers suivants à l'intérieur :

Code source 3 – Le fichier `entree_sortie/entree.py`

```
1 def demander_nom():
2     nom = input("Comment vous appelez-vous ? ")
3     return nom
```

Code source 4 – Le fichier `entree_sortie/sortie.py`

```
1 def afficher(*args):
2     print(*args)
```

Code source 5 – Le fichier `entree_sortie/__init__.py`

```
from entree_sortie.entree import demander_nom
from entree_sortie.sortie import afficher
```

Le module `entree_sortie` est maintenant représenté par une répertoire qui contient plusieurs fichiers. Le fichier `__init__.py` se limite à importer les fonctions `demander_nom()` et `afficher()` dans son propre espace de noms qui est celui du module `entree_sortie`. Le programme principal n'a pas a être modifié par rapport à l'exemple de la section précédente :

Code source 6 – Le fichier `app.py`

```
from entree_sortie import demander_nom, afficher

nom = demander_nom()
afficher("Bonjour", nom)
```

12.8 Module répertoire exécutable

Si un module est représenté par un répertoire, il est possible de le rendre exécutable en ajoutant un fichier `__main__.py` dans le répertoire. Ce fichier contient le code a exécuter uniquement si le module est lancé directement à partir de l'interpréteur Python grâce à l'option `-m` :

```
python3 -m mon_module
```

12.9 Chemin des modules

Comme un module est représenté par un fichier ou par un répertoire, l'interpréteur doit définir quel est l'emplacement par défaut des modules. L'ensemble des répertoires pouvant contenir des modules est appelé le **path**. Il est accessible directement par un programme avec `sys.path`.

```
import sys
print(sys.path)

# Affiche sur un système Linux
# ['', '/usr/lib/python36.zip', '/usr/lib/python3.6',
#  '/usr/lib/python3.6/lib-dynload', '/usr/local/lib/python3.6/dist-packages',
#  '/usr/lib/python3/dist-packages', '/usr/lib/python3.6/dist-packages']
```

Pour un système Linux, la variable `sys.path` indique tous les répertoires systèmes pouvant contenir des modules Python sous la forme d'un tableau. Le premier chemin indiqué est `''`, ce qui signifie que l'interpréteur cherche en priorité un module directement à partir du répertoire de travail dans lequel l'interpréteur a été lancé.

Il est possible de modifier le contenu de `sys.path` par programmation. On peut également utiliser la variable d'environnement système `PYTHONPATH` pour ajouter des chemins supplémentaires au moment du lancement de l'interpréteur. Si on souhaite spécifier plusieurs chemins, il faut les séparer par `:`.

```
$ export PYTHONPATH="/monrepertoire/python:/monrepertoire2"
$ python3
```

```
import sys
print(sys.path)

# Affiche sur un système Linux
# ['', '/monrepertoire/python', '/monrepertoire2', '/usr/lib/python36.zip',
#  '/usr/lib/python3.6', '/usr/lib/python3.6/lib-dynload',
#  '/usr/local/lib/python3.6/dist-packages', '/usr/lib/python3/dist-packages',
#  '/usr/lib/python3.6/dist-packages']
```

12.10 Les *packages*

Les packages en Python désignent le fait de pouvoir structurer l'organisation des fichiers d'un projet dans une arborescence de répertoires. Par exemple si un projet a l'arborescence suivante :

```
stock/
        __init__.py
        ecran/
                __init__.py
                accueil.py
                detail_produit.py
        gestion
                __init__.py
                produit.py
                commande.py
```

Il est possible d'importer le module `commande` en utilisant son chemin de packages :

```
import stock.gestion.commande

stock.gestion.commande.traiter_commande()
```

Si on veut éviter de préciser systématiquement le chemin de packages pour accéder à élément du module, on peut utiliser la syntaxe du `from ... import` :

```
from stock.gestion import commande

commande.traiter_commande()
```

Note : La présence des fichiers `__init__.py` est obligatoire dans l'arborescence des packages pour que l'interpréteur Python traite chaque répertoire comme un module. Ces fichiers peuvent être vides.

Lorsque vous importez un module dans un chemin de package, le fichier `__init__.py` de chaque répertoire est appelé pour initialiser chacun des modules. Un module n'est initialisé qu'une fois ! Les imports successifs de ce module ou d'un sous module n'exécuteront plus son fichier `__init__.py`.

12.10.1 Import relatif

Dans une arborescence de packages, il est parfois plus simple d'utiliser un import relatif. Dans l'exemple d'arborescence précédent, supposons que dans le module `stock.gestion.produit`, on souhaite importer des fonctions présentes dans le fichier `produit.py`, on peut écrire :

Code source 7 – Import depuis le fichier `commande.py`

```
import stock.gestion.produit
```

Cela a comme inconvénient de rendre le contenu du fichier dépendant de l'arborescence des packages. On peut également recourir à un import relatif. Puisque les fichiers `commande.py` et `produit.py` sont dans le même répertoire, on peut écrire :

Code source 8 – Import relatif depuis le fichier `commande.py`

```
from . import produit
```

On peut également remonter dans l'arborescence en utilisant `..` :

Code source 9 – Import relatif depuis le fichier `commande.py`

```
from ..ecran import accueil
```

12.11 Exercices

Exercice en forme d'astuce

Essayez la commande :

```
python -m http.server
```

Exercice : Trouver un nombre

Utilisez la fonction `randint()` du module `random` pour tirer un nombre aléatoirement entre 1 et 20.

L'utilisateur a droit à trois tentatives pour deviner ce nombre.

Exercice : Trouver un nombre

Utilisez la fonction `datetime.now()` du module `datetime` pour connaître la date et l'heure et l'afficher.

Exercice : Organiser un projet en module

Reprenez le projet pour l'affichage de l'écart avec la taille. Découpez ce projet avec un module sous la forme d'un répertoire qui s'appelle `taille`.

Le contenu du module doit être :

```
taille
    __init__.py
    __main__.py
    entree_sortie.py
    calcul.py
```

Le module `entree_sortie` doit regrouper les fonctions pour interagir avec l'utilisateur. Le module `calcul` doit regrouper la fonction de calcul de différence. Le fichier principal du module `__main__` doit permettre d'exécuter le programme.

L'application peut maintenant s'exécuter directement en ligne de commande en invoquant le module :

```
$ python -m taille
```

CHAPITRE 13

Les décorateurs

Un décorateur est une fonction qui prend en paramètre une fonction et qui retourne une fonction. Ce type particulier de fonctions est également appelé une fonction d'ordre supérieur (*higher order function*). L'intérêt d'un décorateur est qu'il permet de transformer le comportement de la fonction passée en paramètre pour exécuter des traitements supplémentaires avant ou après les traitement normal de la fonction passée en paramètre.

Les décorateurs ont des usages multiples. Ils permettent de :

- vérifier des conditions particulières lors de l'appel à une fonction
- acquérir une ressource le temps de l'appel à la fonction (ouverture de fichier, accès à un service distant de base de données…)
- tracer les appels de fonctions pour des raisons de débogage.
- …

Nous verrons également qu'ils sont très utiles pour la programmation orientée objet.

13.1 Un premier décorateur

Prenons l'exemple d'un décorateur qui se contente d'écrire sur la sortie standard le début de l'appel à une fonction et la fin de l'appel :

```
def trace(func):
    def decorateur():
        print("Début d'appel à", func)
        func()
        print("Fin d'appel à", func)
    return decorateur
```

Il est possible maintenant de décorer un appel à une fonction :

```
def do_something():
    print("doing something")

do_something = trace(do_something)

do_something()
# Affiche
# Début d'appel à <function do_something at 0x7f210f642730>
# doing something
# Fin d'appel à <function do_something at 0x7f210f642730>
```

À la ligne 4, on décore la fonction `do_something()` en créant une nouvelle fonction que nous affectons directement à `do_something` pour changer le comportement initial de la fonction.

Les décorateurs agissent comme un composé de fonctions en mathématiques. Pour deux fonctions `g(x)` et `f(x)`, cela équivaut à :

g O f(x)

En Python, le *pie operator* `@` permet de réaliser la même opération de manière beaucoup plus simple :

```
@trace
def do_something():
    print("doing something")

do_something()
# Affiche
# Début d'appel à <function do_something at 0x7f210f642730>
# doing something
# Fin d'appel à <function do_something at 0x7f210f642730>
```

À la ligne 1, on décore la fonction do_something grâce à la notation `@trace`

13.2 Décorer des fonctions avec des paramètres et des valeurs de retour

Un décorateur est une fonction qui appelle la fonction décorée. Si cette dernière attend des paramètres et/ou retourne des valeurs, alors le décorateur doit en tenir compte :

```
def trace(func):
    def decorateur(*args, **kwargs):
        print("Début d'appel à", func)
        resultat = func(*args, **kwargs)
        print("Fin d'appel à", func)
        return resultat
    return decorateur
```

```
@trace
def moyenne(x, *args):
    """Calcule la moyenne d'un nombre quelconque de valeurs passées en paramètres."""
    nb = 1 + len(args)
    somme = x
    for y in args:
        somme += y
    return somme / nb

resultat = moyenne(5, 2, 2)
# Affiche
# Début d'appel à <function moyenne at 0x7f210a4a7a60>
# Fin d'appel à <function moyenne at 0x7f210a4a7a60>

print("La moyenne est", resultat)
# Affiche La moyenne est 3.0
```

Comme un décorateur est une simple fonction, il est possible de décorer des fonctions même sans utiliser le *pie operator* `@`. On peut donc décorer des méthodes natives de Python comme `max()`

```
max = trace(max)
resultat = max(1, 2, 3)
# Affiche
# Début d'appel à <built-in function max>
# Fin d'appel à <built-in function max>

print("Le maximum est", resultat)
# Affiche Le maximum est 3
```

13.3 Décorateur avec des paramètres

Il est possible de créer des décorateurs qui acceptent eux-mêmes des paramètres. En fait il suffit de créer une fonction qui retourne un décorateur. Ce dernier devant à son tour retourner la fonction de décoration.

```
1  def repeter(nb_fois):
2      def repeter_decorateur(func):
3          def decorateur(*args, **kwargs):
4              for i in range(nb_fois):
5                  func(*args, **kwargs)
6          return decorateur
7      return repeter_decorateur
8
9  @repeter(3)
10 def do_something():
11     print("do something")
12
13 do_something()
14
15 # Affiche
```

(suite sur la page suivante)

(suite de la page précédente)

```
16      # do something
17      # do something
18      # do something
```

13.4 Conserver la documentation malgré l'utilisation d'un décorateur

L'utilisation d'un décorateur n'est pas totalement transparente. En effet, le décorateur produit généralement une fonction qui replace la fonction d'origine. Reprenons l'exemple précédent :

```
def trace(func):
    def decorateur(*args, **kwargs):
        print("Début d'appel à", func)
        resultat = func(*args, **kwargs)
        print("Fin d'appel à", func)
        return resultat
    return decorateur

@trace
def moyenne(x, *args):
    """Calcule la moyenne d'un nombre quelconque de valeurs passées en paramètres."""
    nb = 1 + len(args)
    somme = x
    for y in args:
        somme += y
    return somme / nb
```

Si nous voulons afficher la document de la fonction `moyenne()` :

```
help(moyenne)
```

Nous verrons s'afficher :

```
Help on function decorateur in module __main__:

decorateur(*args, **kwargs)
```

En effet, la méthode `moyenne()` a été décorée. C'est donc la documentation de la méthode interne `decorateur(*args, **kwargs)` qui s'affiche et nous avons perdu la véritable documentation d'origine de la fonction `moyenne()`.

Le module `functools` fournit un certain nombre de décorateur dont `functools.wraps()` qui permet de corriger ce problème :

```
from functools import wraps
```

(suite sur la page suivante)

(suite de la page précédente)

```
def trace(func):
    @wraps(func)
    def decorateur(*args, **kwargs):
        print("Début d'appel à", func)
        resultat = func(*args, **kwargs)
        print("Fin d'appel à", func)
        return resultat
    return decorateur
```

Maintenant nous pouvons utiliser ce décorateur et la documentation de la méthode décorée est conservée :

```
@trace
def moyenne(x, *args):
    """Calcule la moyenne d'un nombre quelconque de valeurs passées en paramètres."""
    nb = 1 + len(args)
    somme = x
    for y in args:
        somme += y
    return somme / nb

help(moyenne)
# Affiche
# moyenne(x, *args)
#   Calcule la moyenne d'un nombre quelconque de valeurs passées en paramètres.
```

13.5 Appliquer plusieurs décorateurs

Il est possible d'ajouter plusieurs décorateurs sur une fonction :

```
@decorateur1
@decorateur2
def do_something():
    pass
```

Cela est équivalent à réaliser les appels suivants :

```
do_something = decorateur1(decorateur2(do_something))
```

L'ordre de déclaration des décorateurs est important puisqu'ils seront appliqués dans cet ordre.

CHAPITRE 14

La programmation orientée objet

La programmation orientée objet (POO) est un paradigme, c'est-à-dire une façon particulière de concevoir des systèmes logiciels. La POO envisage un logiciel comme des interactions entre une myriade d'entités élémentaires que l'on nomme des objets. Un objet est défini par un état qui est constitué de **propriétés** et par des comportements que l'on appelle des **méthodes**.

Un des apports importants de la POO est de permettre de concevoir un système logiciel en des termes directement issus de la problématique que le système est sensé résoudre. Par exemple, si le système est sensé gérer des comptes utilisateur, il est possible de créer des objets qui modélisent le comportement et les états d'un compte utilisateur. Par exemple, l'état d'un compte utilisateur pourra inclure un numéro de compte et une date de création. Il pourra avoir des comportements permettant de le verrouiller ou de le valider.

La POO est donc une façon de répondre à une problématique avec un modèle qui est plus proche des utilisateurs que de la représentation interne de l'information. Il est également plus aisé d'appréhender des systèmes complexes comme étant des composés d'objets aux comportements plus simples.

14.1 La classe et les instances

Un objet suit un modèle (un patron) qui définit ses propriétés et ses comportements. Ce modèle est appelé une **classe** et se définit en Python avec le mot-clé `class`.

```
class MaClasse:
    pass
```

Une classe est définie par un nom et un bloc précisant la définition de la classe. Si on ne veut rien préciser de particulier dans le bloc, il faut utiliser le mot-clé `pass`.

Note : En Python, le nom des classes commence par convention par une lettre en majuscule. Les différents mots sont également indiqués par une lettre majuscule suivant le principe de l'écriture en dromadaire (*camel case*). Cette convention a évolué au cours du temps et donc il est possible de trouver des anciennes classes dont le nom commence par une lettre minuscule.

Pour créer un objet, il suffit d'appeler la classe :

```
mon_objet = MaClasse()
```

La variable `mon_objet` référence l'objet crée. On dit que l'objet est une **instance** de `MaClasse`. Une classe définit un nouveau type dans le langage. La POO est d'abord un modèle de programmation qui permet d'ajouter dans le langage des nouveaux types d'objet. En Python, il existe deux fonctions standards intéressantes pour connaître et tester le type d'un objet :

`type(o)` Retourne le type de l'objet passé en paramètre.

```
>>> type(mon_objet)
<class '__main__.MaClasse'>
```

`isinstance(o, cls)` Retourne `True` si l'objet passé en premier paramètre est une instance de la classe passée en deuxième paramètre.

```
>>> isinstance(mon_objet, MaClasse)
True
```

Note : Nous verrons dans un chapitre ultérieur, que pour Python, tout est objet. Ainsi les fonctions `type(o)` et `isinstance(o, cls)` sont utilisables même avec des nombres :

```
>>> nombre = 1
>>> type(nombre)
<class 'int'>
>>> isinstance(nombre, int)
True
```

14.2 Les attributs

Un objet est défini par son état interne formé par l'ensemble de ses **attributs**. Supposons que nous voulions représenter la notion de vecteur dans un espace à deux dimensions dans notre système. Nous allons déclarer une classe `Vecteur`.

```
class Vecteur:
    pass
```

Nous pouvons maintenant créer autant de vecteurs que nous le souhaitons en leur attribuant à chacun une valeur pour ses attributs x et y :

```
vec1 = Vecteur()
vec1.x = 2
vec1.y = 3

vec2 = Vecteur()
vec2.x = -12
vec2.y = vec1.x

print(f"Vecteur 1 : ({vec1.x}, {vec1.y})")
# Affiche Vecteur 1 : (2, 3)
print(f"Vecteur 2 : ({vec2.x}, {vec2.y})")
# Affiche Vecteur 2 : (-12, 2)
```

L'opérateur `.` permet d'accéder à un attribut, soit pour le consulter, soit pour le modifier. Chaque objet possède ses propres valeurs pour ses attributs. C'est pour cela que l'on dit que les attributs permettent de définir l'état interne de l'objet.

Prudence : Si vous essayez d'accéder à un attribut qui n'existe pas pour en consulter la valeur, l'opération échoue avec une erreur de type `AttributeError`.

```
>>> vec3 = Vecteur()
>>> vec3.x = 1
>>> vec3.y = 2
>>> x = vec3.x
>>> y = vec3.y
>>> z = vec3.z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Vecteur' object has no attribute 'z'
```

On peut supprimer un attribut grâce au mot-clé `del`.

```
>>> vec = Vecteur()
>>> vec.x = 1
>>> vec.y = -3
>>> del vec.x
>>> vec.x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Vecteur' object has no attribute 'x'
```

Il est possible pour un programme de découvrir et de manipuler les attributs d'un objet grâce aux fonctions `hasattr(o, n)`, `getattr(o, n)`, `setattr(o, n, v)` et `delattr(o, n)` qui permettent respectivement de savoir si un objet dispose d'un attribut, d'obtenir la valeur de cet attribut, de positionner la valeur d'un attribut et

de supprimer un attribut.

```
>>> vec = Vecteur()
>>> hasattr(vec, "x")
False
>>> setattr(vec, "x", 1)
>>> hasattr(vec, "x")
True
>>> getattr(vec, "x")
1
>>> delattr(vec, "x")
>>> hasattr(vec, "x")
False
```

L'intérêt de ces fonctions est qu'elles permettent de manipuler un attribut en donnant son nom sous la forme d'une chaîne de caractères. Cela ouvre la possibilité à un programme d'accéder aux attributs sans avoir réellement besoin de disposer de la déclaration de la classe au moment de l'écriture du programme. En Python, il est donc possible de manipuler les types comme des données et on parle alors de programmation réflexive ou de **réflexivité**.

Note : La fonction `getattr(o, n)` accepte également un troisième paramètre qui donne la valeur à retourner si l'attribut n'existe pas plutôt que de produire une erreur de type `AttributeError`.

```
>>> vec = Vecteur()
>>> getattr(vec, "attribut_qui_n_existe_pas", "Valeur par défaut")
'Valeur par défaut'
```

14.3 Les attributs implicites

Il existe des attributs implicites (c'est-à-dire pour lesquels il n'est pas nécessaire de fournir une valeur) pour chaque objet.

`__doc__`

Cet attribut contient la documentation de la classe. Pour renseigner cet attribut, il suffit d'ajouter une chaîne de caractères directement après la ligne de déclaration du nom de la classe :

```
class Vecteur:
    """Un vecteur à deux dimensions"""
    pass
```

La documentation peut être affichée dans la console Python grâce à la fonction `help()`. Pour sortir du mode d'aide, il faut généralement appuyer sur la touche `Q` (pour *quit*).

```
>>> v = Vecteur()
>>> help(v)
```

On peut aussi simplement accéder à l'attribut `__doc__` :

```
>>> v.__doc__
'Un vecteur à deux dimensions'
```

`__class__`
: Cet attribut contient l'objet qui décrit la classe d'un objet. En Python même les classes sont représentées par des objets ! C'est cet attribut qui est notamment utilisé pour les fonctions `type()` et `isinstance()`.

`__module__`
: Cet attribut contient le nom du module auquel l'objet appartient.

`__dict__`
: Cet attribut est le dictionnaire des attributs de l'objet.

```
>>> v = Vecteur()
>>> v.x = 2
>>> v.y = 3
>>> v.__dict__
{'y': 3, 'x': 2}
```

Cet attribut est notamment utilisé par la fonction `vars()` qui permet de créer un dictionnaire à partir d'un objet :

```
>>> vars(v)
{'y': 3, 'x': 2}
```

Cet attribut permet également de mettre à jour tous les attributs à partir d'un dictionnaire.

```
>>> v = Vecteur()
>>> v.__dict__ = {'y': 8, 'x': 3}
>>> v.x
3
>>> v.y
8
```

En Python, il est donc assez facile de passer d'un objet à un dictionnaire et d'un dictionnaire à un objet.

14.4 Les méthodes

Les méthodes représentent les comportements des objets. Elles sont décrites dans la classe en suivant les mêmes règles d'écriture que les fonctions.

Si on désire ajouter la possibilité de calculer la norme du vecteur, on peut ajouter la méthode `calculer_norme`.

```
import math

class Vecteur:

    def calculer_norme(self):
        return math.sqrt(self.x**2 + self.y**2)
```

Pour calculer la norme d'un vecteur, nous avons besoin de la fonction `sqrt()` que nous obtenons en important le module `math`. La méthode `calculer_norme` prend un premier paramètre qui est appelé par convention `self`. `self` représente l'objet dans le corps de la méthode. Il est donc possible d'accéder aux attributs `self.x` et `self.y` pour réaliser le calcul :

```
>>> v = Vecteur()
>>> v.x = 2
>>> v.y = 3
>>> v.calculer_norme()
3.605551275463989
```

Pour appeler une méthode, on utilise l'opérateur `.` entre la variable qui désigne l'objet et la méthode.

Note : Pour simplifier, nous pourrions dire qu'une méthode est une fonction qui appartient à l'espace de nom de la classe et dont le premier paramètre est l'objet lui-même. Ainsi l'écriture avec l'opérateur `.` est une manière plus simple et raccourcie d'appeler une méthode mais nous pouvons tous aussi bien écrire en Python :

```
>>> v = Vecteur()
>>> v.x = 2
>>> v.y = 3
>>> Vecteur.calculer_norme(v)
3.605551275463989
```

On comprend mieux ainsi la présence obligatoire du paramètre `self` dans la déclaration d'une méthode pour représenter l'objet.

En plus du paramètre `self`, une méthode peut avoir d'autres paramètres exactement comme une fonction. Nous pouvons ainsi ajouter une méthode pour calculer le produit scalaire entre le vecteur courant et un autre vecteur passé en paramètre :

```
import math

class Vecteur:

    def calculer_norme(self):
        return math.sqrt(self.x**2 + self.y**2)

    def calculer_produit_scalaire(self, v):
        return self.x * v.x + self.y * v.y
```

```
>>> v1 = Vecteur()
>>> v1.x = 2
>>> v1.y = 3
>>> v2 = Vecteur()
>>> v2.x = 1
>>> v2.y = -1
>>> v1.calculer_produit_scalaire(v2)
-1
```

Une méthode peut se contenter de modifier l'état interne d'un objet (ses attributs) et ainsi agir comme une procédure. Par exemple, la méthode `normaliser` pour la classe `Vecteur` ne produit pas de résultat mais modifie les attributs x et y pour garantir que la norme du vecteur est 1.

```
1  import math
2
3  class Vecteur:
4
5      def calculer_norme(self):
6          return math.sqrt(self.x**2 + self.y**2)
7
8      def calculer_produit_scalaire(self, v):
9          return self.x * v.x + self.y * v.y
10
11     def normaliser(self):
12         norme = self.calculer_norme()
13         self.x /= norme
14         self.y /= norme
```

Note : Notez comment la méthode `normaliser()` appelle la méthode `calculer_norme()` à la ligne 12.

```
>>> v = Vecteur()
>>> v.x = 2
>>> v.y = 3
>>> v.normaliser()
>>> v.x
0.5547001962252291
>>> v.y
0.8320502943378437
>>> v.calculer_norme()
1.0
```

14.5 Le constructeur

L'implémentation de notre classe `Vecteur` a un inconvénient majeur. Toutes ses méthodes utilisent les attributs x et y pour réaliser leur traitement. Que se passe-t-il si on oublie de donner des valeurs pour les attributs x et y ?

```
>>> v = Vecteur()
>>> v.calculer_norme()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    return math.sqrt(self.x**2 + self.y**2)
AttributeError: Vecteur instance has no attribute 'x'
```

Dans l'exemple ci-dessus, l'appel à la méthode `calculer_norme()` produit une erreur de type `AttributeError` car l'objet n'a pas les attributs attendus. Cela signifie que si un développeur veut utiliser correctement la classe `Vecteur`, il doit affecter une valeur pour les attributs `x` et `y` pour chacun des objets de type `Vecteur`.

Un constructeur est une méthode spéciale qui est appelée **au moment de la création de l'objet**. Il permet de garantir que l'objet est dans un état cohérent dès sa création. En Python, le constructeur s'appelle `__init__()` est prend comme premier paramètre l'objet en cours de création.

```
import math

class Vecteur:

    def __init__(self):
        self.x = 0
        self.y = 0

    def calculer_norme(self):
        return math.sqrt(self.x**2 + self.y**2)

    def calculer_produit_scalaire(self, v):
        return self.x * v.x + self.y * v.y

    def normaliser(self):
        norme = self.calculer_norme()
        self.x /= norme
        self.y /= norme
```

Le constructeur de notre classe `Vecteur` positionne à 0 les attributs nécessaires aux objets de ce type. Nous avons maintenant corrigé notre problème.

```
>>> v = Vecteur()
>>> v.x
0
>>> v.y
0
>>> v.calculer_norme()
0
```

Même si un constructeur est une méthode un peu particulière, il peut accepter autant de paramètres que nécessaire. La valeur de ces paramètres est donnée au moment de la création de l'objet.

```
import math

class Vecteur:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def calculer_norme(self):
        return math.sqrt(self.x**2 + self.y**2)

    def calculer_produit_scalaire(self, v):
        return self.x * v.x + self.y * v.y

    def normaliser(self):
        norme = self.calculer_norme()
        self.x /= norme
        self.y /= norme
```

```
>>> v = Vecteur()
>>> v.x, v.y
(0, 0)
>>> v = Vecteur(1, 2)
>>> v.x, v.y
(1, 2)
>>> v = Vecteur(y=6, x=12)
>>> v.x, v.y
(12, 6)
```

14.6 Les propriétés

Les attributs permettent de définir l'état d'un objet mais il existe d'autres données qui peuvent être représentatives de cet état. Pour reprendre notre exemple du vecteur, la norme d'un vecteur est également une donnée qui est simplement calculée à partir des coordonnées x et y. L'ensemble de ces données sont appelées les propriétés de l'objet. Si on souhaite créer une propriété calculée, on peut le faire grâce au décorateur `@property`.

```
import math

class Vecteur:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    @property
    def norme(self):
        return math.sqrt(self.x**2 + self.y**2)

    def calculer_produit_scalaire(self, v):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return self.x * v.x + self.y * v.y

    def normaliser(self):
        norme = self.norme
        self.x /= norme
        self.y /= norme
```

En transformant le calcul de la norme en une propriété `norme`, il est possible d'y accéder comme s'il s'agissait d'un attribut :

```
>>> v = Vecteur(2, 3)
>>> v.norme
3.605551275463989
```

Par défaut, une propriété créée grâce à ce décorateur n'est pas modifiable. Pour indiquer les méthodes qui doivent être appelées si on souhaite modifier ou supprimer la propriété, il faut utiliser le même nom de méthode et ajouter des décorateurs de la forme `<nom>.setter` et `<nom>.deleter`.

```
class Personne:

    def __init__(self, prenom, nom):
        self.prenom = prenom
        self.nom = nom

    @property
    def nom_complet(self):
        """Prénom et nom de la personne"""
        return f"{self.prenom} {self.nom}"

    @nom_complet.setter
    def nom_complet(self, nom_complet):
        self.prenom, self.nom = nom_complet.split(maxsplit=1)

    @nom_complet.deleter
    def nom_complet(self):
        self.nom = self.prenom = ""
```

```
>>> p = Personne("David", "Gayerie")
>>> p.nom_complet
'David Gayerie'
>>> p.nom_complet = "Eric Gayerie"
>>> p.prenom
'Eric'
>>> del p.nom_complet
>>> p.nom_complet
' '
```

14.7 Les attributs de classe

Une classe peut également avoir des attributs. Pour cela, il suffit de les déclarer dans le corps de la classe. Les attributs de classe sont accessibles depuis la classe elle-même et sont partagés par tous les objets. Si un objet modifie un attribut de classe, cette modification est visible de tous les autres objets.

```
class ClasseAvecCompteur:

    nb_instances = 0

    def __init__(self):
        ClasseAvecCompteur.nb_instances += 1
```

```
>>> ClasseAvecCompteur.nb_instances
0
>>> c1 = ClasseAvecCompteur()
>>> c2 = ClasseAvecCompteur()
>>> c3 = ClasseAvecCompteur()
>>> ClasseAvecCompteur.nb_instances
3
```

Dans cet exemple simple, l'attribut de classe `nb_instances` est incrémenté à chaque fois que le constructeur de la classe est appelé. Donc cela permet de compter le nombre d'objets de ce type créés par le programme.

Prudence : Notez comment l'attribut de classe `nb_instances` est modifié dans le constructeur de la classe `ClasseAvecCompteur`. On utilise le nom de la classe et non `self` car sinon il s'agirait de l'attribut de l'objet. Il est possible d'utiliser `self` pour lire le contenu d'une variable de classe mais cela est fortement déconseillé car cela peut donner lieu à une ambiguïté à la lecture du code.

Les attributs de classe sont le plus souvent utilisés pour représenter des constantes.

14.8 Méthodes de classe

Tout comme il est possible de déclarer des attributs de classe, il est également possible de déclarer des méthodes de classe. Pour cela, on utilise le décorateur `@classmethod`. Comme une méthode de classe appartient à une classe, le premier paramètre correspond à la classe. Par convention, on appelle ce paramètre `cls` pour préciser qu'il s'agit de la classe et pour le distinguer de `self`.

```
class ClasseAvecCompteur:

    nb_instances = 0
```

(suite sur la page suivante)

(suite de la page précédente)

```
    @classmethod
    def reinitialiser_compteur(cls):
        cls.nb_instances = 0

    def __init__(self):
        ClasseAvecCompteur.nb_instances += 1
```

```
>>> c1 = ClasseAvecCompteur()
>>> c2 = ClasseAvecCompteur()
>>> c3 = ClasseAvecCompteur()
>>> ClasseAvecCompteur.nb_instances
3
>>> ClasseAvecCompteur.reinitialiser_compteur()
>>> ClasseAvecCompteur.nb_instances
0
```

14.9 Méthodes statiques

Une méthode statique est une méthode qui appartient à la classe mais qui n'a pas besoin de s'exécuter dans le contexte d'une classe. Autrement dit, c'est une méthode qui ne doit pas prendre le paramètre `cls` comme premier paramètre. Pour déclarer une méthode statique, on utilise le décorateur `@staticmethod`. Les méthodes statiques sont des méthodes utilitaires très proches des fonctions mais que l'on souhaite déclarer dans le corps d'une classe.

```
class UneClasse:

    @staticmethod
    def une_methode():
        print("appel de la méthode")
```

```
>>> UneClasse.une_methode():
# Affiche appel de la méthode
```

14.10 Visibilité des attributs et des méthodes

Certains langages de programmation proposent des mécanismes pour gérer la visibilité des attributs et des méthodes. Il arrive souvent qu'une classe déclare des méthodes et des attributs qui ne sont pas destinés à être directement appelés par un programme. On dit qu'il s'agit de méthodes et d'attributs privés, c'est-à-dire destinés à être utilisés uniquement par l'objet lui-même. Par exemple, un objet peut stocker dans un attribut une connexion vers une système externe (base de données, serveur

web...) avec pour objectif d'offrir une interaction simplifiée avec ce système. Cet attribut n'est pas sensé être accédé depuis l'extérieur de l'objet. En le rendant privé, on évite au reste du programme d'y accéder inutilement. Cela permet également une meilleure évolutivité du programme en évitant de trop exposer les données internes d'un objet. C'est ce que l'on appelle le **principe d'encapsulation**.

En Python, il n'existe pas de mécanisme dans le langage qui nous permettrait de gérer la visibilité. Par contre, il existe une convention dans le nommage. Une méthode ou un attribut dont le nom commence par _ (*underscore*) est considéré comme privé. Il est donc déconseillé d'accéder à un tel attribut ou d'appeler une telle méthode depuis l'extérieur de l'objet.

```
class ClientSystemeExterne:

    def ouvrir_connexion(self):
        self._connexion = ConnexionInterne()

    def fermer_connexion(self):
        if self._connexion is not None:
            self._connexion.fermer()
            self._connexion = None
```

14.11 Fermer la liste des attributs

Nous avons vu au début de ce chapitre que nous pouvons créer autant d'attributs que nous le souhaitons pour un objet. Cela peut conduire à des bugs ou des difficultés de compréhension du code. Si nous reprenons notre classe `Vecteur`, nous pouvons écrire :

```
1 >>> v = Vecteur(1, 0)
2 >>> v.z = 4
3 >>> v.norme
4 1.0
```

À la ligne 2, le programme positionne la valeur d'un nouvel attribut `z`. Un lecteur pourrait penser que le vecteur est en fait un vecteur à trois dimensions. Mais il n'en est rien. La classe `Vecteur` n'a pas été conçue pour prendre en charge cette troisième dimension et la norme du vecteur sera bien de 1.

Pour éviter ce genre de situation, il est possible déclarer la liste finie des attributs. Pour cela, on déclare une attribut de classe appelé `__slots__`.

```
import math

class Vecteur:

    __slots__ = ('x', 'y')

    def __init__(self, x=0, y=0):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        self.x = x
        self.y = y

    @property
    def norme(self):
        return math.sqrt(self.x**2 + self.y**2)

    def calculer_produit_scalaire(self, v):
        return self.x * v.x + self.y * v.y

    def normaliser(self):
        norme = self.norme
        self.x /= norme
        self.y /= norme
```

Si maintenant un programme essaie de positionner la valeur d'un attribut qui ne fait pas partie de la définition de la classe, l'interpréteur génère une erreur `AttributeError`.

```
>>> v = Vecteur(1, 0)
>>> v.z = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Vecteur' object has no attribute 'z'
```

Note : L'attribut `__slots__` a également une utilité pour l'optimisation du code. Comme on indique à l'interpréteur le nombre exact d'attributs, il peut optimiser l'allocation mémoire pour un objet de ce type.

14.12 Créer des objets avec des attributs constants

Lorsque l'on veut créer des objets qui contiennent uniquement des attributs en lecture seule, il est possible d'utiliser la classe `collections.namedtuple`. Elle fournit une solution à la déclaration de constante en Python.

```
>>> from collections import namedtuple
>>> Intervalle = namedtuple('Intervalle', ('min', 'max'))
>>> intervalle = Intervalle(min=0, max=100)
>>> intervalle.min
0
>>> intervalle.max
100
>>> intervalle.min = 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

La classe `namedtuple` attend en paramètre le nom du type à créer ainsi que la liste

des noms des attributs. L'objet construit a les mêmes comportements qu'un tuple, c'est-à-dire que ses données ne peuvent pas être modifiées mais elles peuvent être accessibles comme des attributs. Si on tente de modifier la valeur d'un des attributs, cela produit une erreur de type `AttributeError`.

CHAPITRE 15

Héritage et polymorphisme

Une application Python est composée d'un ensemble d'objets. Un des intérêts de la programmation orientée objet (POO) réside dans les relations que ces objets entretiennent les uns avec les autres. Ces relations sont construites par les développeurs et constituent ce que l'on appelle l'architecture d'une application. Il existe deux relations fondamentales en POO :

a un(e) (*has-a*) Cette relation permet de créer une relation de dépendance d'une classe envers une autre. Une classe a besoin des services d'une autre classe pour réaliser sa fonction. On parle également de relation de composition pour désigner ce type de relation.

La relation « a un » implique simplement qu'un objet possède un attribut qui référence un autre objet. Par exemple, si une programme déclare une classe `Personne`, cette classe peut avoir une propriété pour stocker l'adresse sous la forme d'une classe adresse :

```
1  class Personne:
2
3      def __init__(self, nom, prenom, adresse):
4          self.nom = nom
5          self.prenom = prenom
6          self.adresse = adresse
7
8
9  class Adresse:
10
11     def __init__(self, rue, code_postal, ville):
12         self.rue = rue
13         self.code_postal = code_postal
14         self.ville = ville
15
16
17 adresse = Adresse(rue="4 rue d'ici", code_postal="78000", ville="Paris")
18 personne = Personne(prenom="Jean", nom="Dumond", adresse=adresse)
```

Le code ci-dessus illustre la relation de composition et on peut bien dire qu'une personne **a une** adresse.

est un(e) (*is-a*) Cette relation permet de créer une chaîne de relation d'identité entre des classes. Elle indique qu'une classe peut être assimilée à une autre classe qui correspond à une notion plus abstraite ou plus générale. On parle **d'héritage** pour désigner le mécanisme qui permet d'implémenter ce type de relation. Le reste de ce chapitre est consacré à ce type particulier de relation entre classes.

15.1 L'héritage

L'héritage est donc le mécanisme qui permet de traduire une relation de type « est un(e) ».

Prenons l'exemple d'une classe `Voiture` et d'une classe `Vehicule`. Il est possible de dire qu'une voiture **est un** véhicule. Je peux traduire cette relation en Python. On dit que `Vehicule` est la classe parente de `Voiture` ou la généralisation. Symétriquement, on dit que `Voiture` est la classe enfant (ou fille) de `Vehicule` ou la spécialisation. On dit aussi que la classe `Vehicule` est la **super classe** de `Voiture`.

```
class Vehicule:
    pass


class Voiture(Vehicule):
    pass
```

Cette relation se traduit en précisant le nom de la classe parente entre parenthèses après le nom de la classe.

Si nous créons un objet à partir de la classe `Voiture`, cet objet est à la fois de type `Vehicule` et de type `Voiture`.

```
>>> v = Voiture()
>>> isinstance(v, Voiture)
True
>>> isinstance(v, Vehicule)
True
```

Le mot de héritage vient du fait qu'une classe fille hérite des attributs et des comportements de sa classe mère.

```
class Vehicule:

    def __init__(self):
        self._vitesse = 0

    @property
    def vitesse(self):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return self._vitesse

    def accelerer(self, delta_vitesse):
        self._vitesse += delta_vitesse

    def decelerer(self, delta_vitesse):
        self._vitesse -= delta_vitesse


class Voiture(Vehicule):
    pass
```

```
>>> v = Voiture()
>>> v.vitesse
0
>>> v.accelerer(80)
>>> v.vitesse
80
```

Les objets de type `Voiture` disposent des mêmes comportements, propriétés et attributs déclarés dans la classe `Vehicule`.

La classe fille peut fournir ses propres comportements et ses propres attributs :

```
class Voiture(Vehicule):

    def klaxonner(self):
        print("tût tût !")
```

15.2 Constructeur et héritage

Le constructeur est, comme toutes les méthodes, hérité. Cependant que ce passe-t-il si nous souhaitons ajouter une constructeur dans notre classe `Voiture` ?

```
class Voiture(Vehicule):

    def __init__(self, klaxon="tût tût !"):
        self.klaxon = klaxon

    def klaxonner(self):
        print(self.klaxon)
```

```
>>> v = Voiture()
>>> v.klaxonner()
tût tût !
>>> v.vitesse
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return self._vitesse
AttributeError: 'Voiture' object has no attribute '_vitesse'
```

Cela ne fonctionne plus car le constructeur déclaré dans la classe `Voiture` remplace et cache celui hérité de la classe `Vehicule` et l'initialisation de l'attribut `_vitesse` n'est plus faite. Quand une classe hérite d'une autre classe, elle a la responsabilité de s'assurer que le constructeur de la classe parente est appelé pour garantir l'initialisation. Pour cela, Python 3 fournit la fonction `super()` qui retourne le type de la super classe. L'implémentation correcte de la classe `Voiture` doit donc être :

```
class Voiture(Vehicule):

    def __init__(self, klaxon="tût tût !"):
        super().__init__()
        self.klaxon = klaxon

    def klaxonner(self):
        print(self.klaxon)
```

```
>>> v = Voiture()
>>> v.klaxonner()
tût tût !
>>> v.vitesse
0
```

Il est donc facile de passer des paramètres à l'initialisation de la classe parente si cette dernière attend des paramètres de constructeur.

```
class Vehicule:

    def __init__(self, marque=None, vitesse_initiale=0):
        self.marque = marque
        self._vitesse = vitesse_initiale

    @property
    def vitesse(self):
        return self._vitesse

    def accelerer(self, delta_vitesse):
        self._vitesse += delta_vitesse

    def decelerer(self, delta_vitesse):
        self._vitesse -= delta_vitesse


class Voiture(Vehicule):

    def __init__(self, marque=None, vitesse_initiale=0, klaxon="tût tût !"):
        super().__init__(marque, vitesse_initiale)
        self.klaxon = klaxon

    def klaxonner(self):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        print(self.klaxon)
```

```
>>> v = Voiture("De Lorean", 88.0)
>>> v.vitesse
88.0
>>> v.marque
'De Lorean'
>>> v.klaxonner()
tût tût !
```

Note : Pour chaîner l'appel des constructeurs, il est également possible d'appeler directement le constructeur en désignant la classe parente :

```
class Voiture(Vehicule):

    def __init__(self, marque=None, vitesse_initiale=0, klaxon="tût tût !"):
        Vehicule.__init__(marque, vitesse_initiale)
        self.klaxon = klaxon
```

Cette écriture est cependant considérée comme obsolète en Python 3 grâce à la fonction `super()`.

15.3 Héritage et mutualisation de code

Un des intérêts de l'héritage est de permettre la réutilisation de code. Maintenant que nous avons défini une `Vehicule`, nous pouvons imaginer diverses classes appropriées pour notre système et qui sont des véhicules.

```
class Charette(Vehicule):

    def __init__(self, vitesse_initiale=0):
        super().__init__(vitesse_initiale=vitesse_initiale)
```

```
>>> c = Charette(1)
>>> c.accelerer(5)
>>> c.vitesse
6
```

15.4 La classe *object*

Python définit la classe `object`. Toutes les classes hérite directement ou indirectement de cette classe. Si une classe ne déclare aucune classe parente alors sa classe

parente est `object`.

```
class MaClasse:
    pass
```

La déclaration ci-dessus est strictement équivalente à :

```
class MaClasse(object):
    pass
```

Il est possible de créer des instances de la classe `object`. Cela reste d'un usage limité car cette classe n'offre aucune méthode particulière et il n'est pas possible d'ajouter dynamiquement des attributs à ses instances.

```
>>> o = object()
>>> o.nom = "Un nom"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'object' object has no attribute 'nom'
```

15.5 Le polymorphisme

Le polymorphisme est un mécanisme important dans la programmation objet. Il permet de modifier le comportement d'une classe fille par rapport à sa classe mère. Le polymorphisme permet d'utiliser l'héritage comme un mécanisme d'extension en adaptant le comportement des objets.

Prenons l'exemple de la classe `Animal`. Cette classe offre une méthode `crier`. Pour simplifier notre exemple, la méthode se contente d'écrire le cri de l'animal sur la sortie standard.

```
class Animal:

    def crier(self):
        print("un cri d'animal")
```

Nous pouvons créer les classes `Chien` et `Chat` qui héritent toutes deux de la classe `Animal`. Ces classes peuvent être des spécialisations de la classe `Animal` en ce qu'elles peuvent **redéfinir** (*override*) le comportement de la méthode `crier ()`.

```
class Chien(Animal):

    def crier(self):
        print("whouaf whouaf !")


class Chat(Animal):
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def crier(self):
        print("miaou !")
```

```
>>> a = Animal()
>>> animal = Animal()
>>> animal.crier()
un cri d'animal
>>> animal = Chien()
>>> animal.crier()
whouaf whouaf !
>>> animal = Chat()
>>> animal.crier()
miaou !
```

Un chat est bien un animal, un chien est bien un animal. Les objets de ces types disposent bien de la méthode `crier()` mais son comportement est polymorphe. Il dépend du type réel de l'objet.

À noter qu'il est toujours possible d'appeler la méthode parente dans la classe enfant grâce à la fonction `super()`.

```
class Chien(Animal):

    def crier(self):
        print("whouaf whouaf !")

    def crier_comme_un_animal(self):
        super().crier()
```

```
>>> chien = Chien()
>>> chien.crier()
whouaf whouaf !
>>> chien.crier_comme_un_animal()
un cri d'animal
```

15.6 Polymorphisme et *duck typing*

Pour les langages de programmation à typage fort comme le C++ ou le Java, le polymorphisme a des implications très fortes et il permet d'introduire une grande souplesse dans les relations entre types. Pour les langages de programmation à typage dynamique comme Python, le polymorphisme, même s'il reste une notion importante, a moins d'impact.

En Python, on préfère souvent le principe du duck typing (ou typage canard). Cette approche repose sur le principe suivant :

> « Si je vois un oiseau qui vole comme un canard, cancane comme un canard, et

nage comme un canard, alors j'appelle cet oiseau un canard. »

Cela signifie que le type réel est moins important que le comportement attendu. Pour reprendre notre exemple des véhicules : si une classe offre des méthodes pour accélérer, décélérer et des propriétés pour la vitesse et la marque, alors c'est qu'il s'agit d'un véhicule et peu importe que cette classe hérite ou non de la classe `Vehicule`. Tous les objets de cette classe pourront être utilisés dans un contexte qui manipule des véhicules.

15.7 Masquer les attributs et les méthodes pour les classe filles

Parfois, on ne souhaite pas qu'une méthode puisse être redéfinie ou qu'un attribut puisse être modifié dans une classe fille. Pour cela, il suffit que le nom de la méthode ou de l'attribut commence par deux caractères soulignés (*underscores*). Nous avons vu précédemment que Python n'a pas de mécanisme pour contrôler la visibilité des éléments d'une classe. Par convention, les développeurs signalent par un caractère souligné (*underscore*) le statut privé d'un attribut ou d'une méthode. Par contre le recours à deux caractères soulignés à un impact sur l'interpréteur. Ce dernier renomme la méthode ou l'attribut de la forme :

_<nom de la classe>__<nom>

Ainsi si nous modifions le nom de la la classe `Vehicule` :

```
class Vehicule:

    def __init__(self, marque=None, vitesse_initiale=0):
        self.marque = marque
        self.__vitesse = vitesse_initiale

    @property
    def vitesse(self):
        return self.__vitesse

    def accelerer(self, delta_vitesse):
        self.__vitesse += delta_vitesse

    def decelerer(self, delta_vitesse):
        self.__vitesse -= delta_vitesse
```

L'attribut `__vitesse` sera renommé `_Vehicule__vitesse`. Donc, si un attribut `__vitesse` est également utilisé dans une classe fille, l'interpréteur considérera qu'il s'agit d'un attribut différent.

Note : Cette notation est avant tout faite pour éviter des effets de bord involontaire lorsqu'on hérite d'une classe dont on ne connaît pas bien le code source. Après tout, il est toujours possible d'implémenter une classe fille qui utilise un attribut ou déclare une méthode sans que le développeur ait conscience qu'il réutilise un attribut ou qu'il redéfinit une méthode d'une classe parente.

15.8 Héritage multiple et *mixin*

Une classe peut hériter de plusieurs classes. Dans ce cas, elle héritera des méthodes et des attributs de l'ensemble de ces classes et de leurs classes parentes. Il suffit d'indiquer la liste des classes parentes en les séparant par une virgule.

```
class Animal:

    def crier(self):
        pass


class Carnivore:

    def chasser(self):
        pass


class Chien(Animal, Carnivore):
    """Un chien qui est à la fois un animal et un carnivore"""
```

```
>>> c = Chien()
>>> c.chasser()
>>> c.crier()
```

La classe `Chien` hérite des comportements mais aussi des attributs des classes parentes. La variable `c` désigne un objet qui est à la fois un `Chien`, un `Animal` et un `Carnivore`.

```
>>> isinstance(c, Chien)
True
>>> isinstance(c, Animal)
True
>>> isinstance(c, Carnivore)
True
```

15.8.1 L'héritage en diamant

Il peut arriver qu'une classe hérite indirectement plusieurs fois de la même classe. Reprenons notre exemple précédent et supposons que les classes `Animal` et `Carnivore` héritent toutes deux de la classe `EtreVivant` et que la classe `EtreVivant` possède l'attribut `point_de_vie`.

```
class EtreVivant:

    def __init__(self):
        self.point_de_vie = 100

class Animal(EtreVivant):
```

(suite sur la page suivante)

(suite de la page précédente)

```
7
8       def dormir(self):
9           self.point_de_vie += 1
10
11  class Carnivore(EtreVivant):
12
13      def chasser(self):
14          self.point_de_vie -= 1
15
16
17  class Chien(Animal, Carnivore):
18      """Un chien qui est à la fois un animal et un carnivore"""
```

Dans ce cas, la classe `Chien` hérite deux fois de la classe `EtreVivant`. À cause de la représentation graphique d'une telle situation, on appelle ce cas particulier **l'héritage en diamant**.

Python résout cette situation en considérant qu'une classe ne peut pas directement ou indirectement hériter plusieurs fois d'une même classe. Donc, les instances de la classe `Chien` ne posséderont qu'un seul attribut `point_de_vie`.

```
>>> c = Chien()
>>> c.chasser()
>>> c.point_de_vie
99
>>> c.dormir()
>>> c.point_de_vie
100
```

Mais que se passe-t-il si l'héritage en diamant implique des méthodes ? Par exemple, imaginons que la classe `EtreVivant` possède la méthode `se_nourrir()` qui est redéfinie à la fois dans la classe `Animal` et dans la classe `Carnivore`.

```
1   class EtreVivant:
2
3       def __init__(self):
4           self.point_de_vie = 100
5
6       def se_nourrir(self):
7           self.point_de_vie += 1
8
9
10  class Animal(EtreVivant):
11
12      def dormir(self):
13          self.point_de_vie += 1
14
15      def se_nourrir(self):
16          self.point_de_vie += 5
17
18
19  class Carnivore(EtreVivant):
20
```

(suite sur la page suivante)

(suite de la page précédente)

```
21      def chasser(self):
22          self.point_de_vie -= 1
23
24      def se_nourrir(self):
25          self.point_de_vie += 10
26
27
28  class Chien(Animal, Carnivore):
29      """Un chien qui est à la fois un animal et un carnivore"""
```

Que se passe-t-il si nous appelons la méthode `se_nourrir()` depuis une instance de `Chien` ?

```
>>> c = Chien()
>>> c.se_nourrir()
>>> c.point_de_vie
105
```

C'est la méthode de la classe `Animal` qui est exécutée, passant la valeur de `point_de_vie` à 105. Pour choisir quelle méthode est appelée, l'interpréteur se base sur un algorithme appelé **method resolution order** ou **mro**. Il s'agit de chercher dans la liste de l'héritage des classes la première déclaration de la méthode appelée. On peut facilement connaître le *mro* d'une classe car il existe la méthode `class.mro()` qui retourne précisément cette liste. Pour accéder à la classe d'une instance, on peut utiliser la fonction `type`.

```
>>> type(c).mro()
[<class '__main__.Chien'>, <class '__main__.Animal'>,
 <class '__main__.Carnivore'>, <class '__main__.EtreVivant'>,
 <class 'object'>]
```

On voit que pour le type `Chien`, un appel à une méthode conduit à chercher cette méthode d'abord dans la classe `Chien` et ensuite dans les classes `Animal` `Carnivore`, `EtreVivant` et enfin `object`.

Note : Rappelez-vous qu'une classe qui ne précise explicitement aucune classe parente hérite tout de même de la classe `object`.

Si nous intervertissons l'ordre d'héritage de la déclaration de la classe `Chien`.

```
1   class EtreVivant:
2
3       def __init__(self):
4           self.point_de_vie = 100
5
6       def se_nourrir(self):
7           self.point_de_vie += 1
8
9
```

(suite sur la page suivante)

(suite de la page précédente)

```
class Animal(EtreVivant):

    def dormir(self):
        self.point_de_vie += 1

    def se_nourrir(self):
        self.point_de_vie += 5


class Carnivore(EtreVivant):

    def chasser(self):
        self.point_de_vie -= 1

    def se_nourrir(self):
        self.point_de_vie += 10


class Chien(Carnivore, Animal):
    """Un chien qui est à la fois un animal et un carnivore"""
```

```
>>> c = Chien()
>>> c.se_nourrir()
>>> c.point_de_vie
110
```

```
>>> type(c).mro()
[<class '__main__.Chien'>, <class '__main__.Carnivore'>,
 <class '__main__.Animal'>, <class '__main__.EtreVivant'>,
 <class 'object'>]
```

Maintenant, c'est la méthode `se_nourrir()` de la classe `Carnivore` qui est appelée pour un objet de type `Chien`. L'ordre de déclaration de l'héritage multiple est donc primordial !

15.8.2 Appel des constructeurs

Si nous voulons déclarer des constructeurs, il faut impérativement **appeler le constructeur de la super classe**.

```
class EtreVivant:

    def __init__(self, point_de_vie):
        self.point_de_vie = point_de_vie

    def se_nourrir(self):
        self.point_de_vie += 1


class Animal(EtreVivant):
```

(suite sur la page suivante)

(suite de la page précédente)

```
11
12      def __init__(self, nom, point_de_vie):
13          super().__init__(point_de_vie)
14          self.nom = nom
15
16      def dormir(self):
17          self.point_de_vie += 1
18
19      def se_nourrir(self):
20          self.point_de_vie += 5
21
22
23  class Carnivore(EtreVivant):
24
25      def chasser(self):
26          self.point_de_vie -= 1
27
28      def se_nourrir(self):
29          self.point_de_vie += 10
30
31
32  class Chien(Carnivore, Animal):
33      """Un chien qui est à la fois un animal et un carnivore"""
34
35      def __init__(self, point_de_vie, nom):
36          super().__init__(point_de_vie, nom)
```

```
>>> c = Chien("Médor", 60)
>>> c.nom
'Médor'
>>> c.point_de_vie
60
>>> c.se_nourrir()
>>> c.point_de_vie
70
```

La fonction `super()` retourne le type suivant dans la liste du *mro*. La classe `Carnivore` ne proposant pas de constructeur, elle se contente de passer l'appel au constructeur de la classe `Animal` qui est la classe suivante dans la liste *mro*.

15.8.3 Les mixins

Un *mixin* est une classe qui permet d'ajouter des fonctionnalités supplémentaires. Il s'agit simplement d'une classe comme une autre mais qui n'est pas destinée à être utilisée directement pour créer des instances.

Imaginons que nous désirions ajouter à certaines de nos classes la possibilité d'afficher leur état (la valeur des attributs) grâce à une méthode `afficher()`. Nous pouvons créer un *mixin* proposant cette méthode. Ce *mixin* accepte en paramètre le caractère à utiliser comme séparateur ainsi que l'indentation à utiliser lors de l'affichage.

```python
class Affichable:

    def __init__(self, *args, indentation=0, separateur="-", **kwargs):
        self.__indentation = indentation
        self.__separateur = separateur
        super().__init__(*args, **kwargs)

    def afficher(self):
        cls = type(self)
        separateur = self.__separateur * 80
        print(separateur)
        print("Objet de la classe", cls.__name__)
        print(cls.__doc__)
        print("mro : ", cls.mro())
        print()
        indentation = ' ' * self.__indentation
        for attr in self.__dict__:
            if not attr.startswith('_'):
                valeur = getattr(self, attr)
                print(f"{indentation}{attr} = {valeur}")
        print(separateur)
```

Nous pouvons modifier notre implémentation de la classe `Chien` pour la rendre affichable :

```python
class Chien(Affichable, Animal):
    """Un chien qui est à la fois un animal et un carnivore"""

    def __init__(self, point_de_vie, nom, **kwargs):
        super().__init__(point_de_vie, nom, **kwargs)
```

Ainsi nous pouvons afficher l'état interne d'un objet de type `Chien`.

```
>>> c = Chien(40, "Médor", indentation=4)
>>> c.afficher()
--------------------------------------------------------------------------------
Objet de la classe Chien
Un chien qui est à la fois un animal et un carnivore
mro :  [<class '__main__.Chien'>, <class '__main__.Affichable'>,
<class '__main__.Animal'>, <class '__main__.EtreVivant'>, <class 'object'>]

    point_de_vie = 40
    nom = Médor
--------------------------------------------------------------------------------
```

Il y a plusieurs points importants à remarquer dans l'implémentation ci-dessus :

1) La classe `Chien` hérite en premier du *mixin* `Affichable`. À cause de ce que nous avons vu à propos du *mro,* un *mixin* devrait toujours être déclaré avant la classe parente.
2) La classe `Chien` accepte comme dernier paramètre de constructeur le paramètre de compactage `**kwargs`. La classe `Chien` peut ainsi accepter les paramètres nommés `identation` et `separateur` sans avoir besoin de les répéter.
3) La classe `Affichage` a un constructeur assez compliqué.

```
def __init__(self, *args, indentation=0, separateur="-", **kwargs):
    self.__indentation = indentation
    self.__separateur = separateur
    super().__init__(*args, **kwargs)
```

Les paramètres `indentation` et `separateur` sont placés après l'opérateur de compactage `*args` ce qui implique qu'il s'agit de paramètres nommés. Le constructeur utilise à la fin la fonction `super()` pour appeler le constructeur suivant en lui passant tous les paramètres que le *mixin* ne reconnaît pas. On voit qu'il est tout à fait possible de concevoir un *mixin* qui accepte des paramètres de constructeur tout en s'intégrant élégamment à un héritage existant... au prix d'une petite complexité dans la déclaration du constructeur.

Note : Les *mixins* sont une façon de construire de nouvelles classes par agrégat de fonctionnalités. Attention cependant car en programmation orienté objet, il est recommandé de définir ses classes de manière à ce qu'elles aient une responsabilité unique et clairement identifiable dans le système. Cela permet de limiter la complexité des classes et donc de faciliter leur compréhension. L'abus du recours au *mixin* risque de créer des classes tout-en-un difficiles à comprendre et à maintenir.

15.9 La méta-classe

La méta-classe est un concept avancé en Python qui n'est que très très rarement utilisé directement par les développeurs.

En Python, les classes sont elles-mêmes des objets qui héritent de `type`. Il est possible de spécifier le type dont doit hériter l'objet qui représente la classe. On parle alors de **méta-classe**.

Une méta-classe est une classe qui décrit une classe. Cela signifie que tous les attributs et toutes les méthodes d'une méta-classe seront les attributs et les méthodes de la classe.

L'usage de la **méta-classe** permet de réaliser des implémentations qui ne sont pas possibles avec une simple classe. Par exemple, il n'est pas possible en Python de déclarer une propriété de classe avec le décorateur `@property`. Mais cela devient possible par le truchement d'une méta-classe.

```
1  class MetaclasseCompteur(type):
2      """Une méta-classe pour aider à compter les instances créées."""
3
4      def __init__(cls, *args, **kwargs):
5          super().__init__(*args, **kwargs)
6          cls._nb_instances = 0
7
8      @property
9      def nb_instances(cls):
10         return cls._nb_instances
```

(suite sur la page suivante)

(suite de la page précédente)

```
11
12      def plus_une_instance(cls):
13          cls._nb_instances += 1
14
15
16  class MaClasse(metaclass=MetaclasseCompteur):
17
18      def __init__(self):
19          MaClasse.plus_une_instance()
```

Une méta-classe doit hériter directement ou indirectement de `type`. À la ligne 16, on déclare la méta-classe de la classe `MaClasse`. Cette méta-classe va fournir la propriété de classe `nb_instances` ainsi que la méthode de classe `plus_une_instance()`.

```
>>> MaClasse.nb_instances
0
>>> m1 = MaClasse()
>>> m2 = MaClasse()
>>> m3 = MaClasse()
>>> MaClasse.nb_instances
3
```

15.10 Classes et méthodes abstraites

Comme Python est un langage à typage dynamique, nous avons déjà dit plus haut que la plupart des développeurs favorisent le duck typing : le type réel d'un objet importe moins que le fait qu'il produise les comportements attendus (c'est-à-dire les méthodes et les propriétés attendues). Cette approche amène indéniablement plus de souplesse dans la conception des applications. Mais elle rend plus difficile la validation du code et donc cela peut aboutir à la production d'un code moins robuste.

Les langages de programmation à typage fort comme C++, Java ou C# introduisent tous le principe de classes abstraites et/ou d'interfaces. Ce type d'approche insiste sur le fait de pouvoir définir un type particulier contenant un certain nombre de méthodes mais pour lesquelles on ne fournit aucune implémentation. Ces classes abstraites et ces interfaces sont ensuite héritées ou implémentées par d'autres classes qui doivent fournir les implémentations des méthodes attendues. Cela permet de garantir qu'un objet aura bien les comportements attendus, c'est-à-dire implémentera les méthodes attendues. On parle parfois de programmation par contrat, dans le sens où ces classes abstraites et ces interfaces sont comme des contrats qui lient les objets qui les implémentent et les objets qui appellent ces méthodes.

En Python, le module `abc` permet de simuler ce type d'approche. Le nom de ce module est la contraction de *abstract base classes*. Ce module fournit une méta-classe appelée `ABCMeta` qui permet de transformer une classe Python en classe abstraite. Ce module fournit également le décorateur `@abstractmethod` qui permet de déclarer comme abstraite une méthode, une méthode statique, une méthode de classe ou une propriété. Cela signifie qu'il n'est pas possible de créer une instance d'une classe

qui hérite d'une classe abstraite tant que toutes les méthodes abstraites ne sont pas implémentées.

Si nous reprenons notre exemple de la classe `Animal`. Cette classe déclare une méthode `crier()` pour laquelle il n'est pas vraiment possible de fournir une implémentation correcte pour un animal. On peut donc considérer que la classe `Animal` est abstraite en déclarant que sa méta-classe est `ABCMeta` et déclarer la méthode `crier()` comme une méthode abstraite.

```
from abc import ABCMeta, abstractmethod


class Animal(metaclass=ABCMeta):

    @abstractmethod
    def crier(self):
        pass
```

Toute tentative de créer un objet de type `Animal` échouera à l'exécution :

```
>>> a = Animal()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Animal with abstract methods crier
```

La classe `Animal` est en quelque sorte devenu un contrat qui indique que toute classe qui en hérite doit fournir une implémentation pour la méthode `crier()`.

```
class Chien(Animal):

    def crier(self):
        print("whouaf whouaf !")
```

```
>>> c = Chien()
>>> c.crier()
whouaf whouaf !
```

CHAPITRE 16

Les exceptions et la gestion des erreurs

Toutes les erreurs qui se produisent lors de l'exécution d'un programme Python sont représentées par une **exception**. Une exception est un objet qui contient des informations sur le contexte de l'erreur. Lorsqu'une exception survient et qu'elle n'est pas traitée alors elle produit une interruption du programme et elle affiche sur la sortie standard un message ainsi que la pile des appels (*stacktrace*). La pile des appels présente dans l'ordre la liste des fonctions et des méthodes qui étaient en cours d'appel au moment où exception est survenue.

Si nous prenons le programme suivant :

Code source 1 – Le fichier `programme.py`

```
1 def do_something_wrong():
2     1 / 0
3
4 def do_something():
5     do_something_wrong()
6
7 do_something()
```

L'exécution de ce programme affiche sur la sortie d'erreur

```
$ python3 programme.py

Traceback (most recent call last):
  File "test.py", line 7, in <module>
    do_something()
  File "test.py", line 5, in do_something
    do_something_wrong()
  File "test.py", line 2, in do_something_wrong
    1 / 0
ZeroDivisionError: division by zero
```

Nous voyons que le programme a échoué à cause d'une exception de type `ZeroDivisionError` et nous avons la pile d'erreur qui nous indique que le programme s'est interrompu à la ligne 2 suite à l'instruction `1 / 0` dans la fonction `do_something_wrong()` qui a été appelée par la fonction `do_something()` à la ligne 5, elle-même appelée par le programme à la ligne 7.

16.1 Traiter une exception

Parfois un programme est capable de traiter le problème à l'origine de l'exception. Par exemple si le programme demande à l'utilisateur de saisir un nombre et que l'utilisateur saisit une valeur erronée, le programme peut simplement demander à l'utilisateur de saisir une autre valeur. Plutôt que de faire échouer le programme, il est possible d'essayer de réaliser un traitement et, s'il échoue de proposer un traitement adapté :

```
1 nombre = input("Entrez un nombre : ")
2 try:
3     nombre = int(nombre)
4 except ValueError:
5     print("Désolé la valeur saisie n'est pas un nombre.")
```

À la ligne 2, on démarre un bloc `try` pour indiquer le traitement que l'on désire réaliser. Si ce traitement est interrompu par une exception, alors l'interpréteur recherche un bloc `except` correspondant au type (ou à un type parent) de l'exception. Dans notre exemple, si la méthode `int` ne peut pas créer un entier à partir du paramètre, elle produit une exception de type `ValueError`. Donc après le bloc `try`, on ajoute une instruction `except` pour le type `ValueError` (lignes 4 et 5). Ce bloc n'est exécuté que si la conversion en entier n'est pas possible.

Lorsqu'une exception survient dans un bloc `try`, elle interrompt immédiatement l'exécution du bloc et l'interpréteur recherche un bloc `except` pouvant traiter l'exception. Il est possible d'ajouter plusieurs blocs `except` à la suite d'un bloc `try`. Chacun d'entre-eux permet de coder un traitement particulier pour chaque type d'erreur :

```
1 try:
2     numerateur = int(input("Entrez un numérateur : "))
3     denominateur = int(input("Entrez un dénominateur : "))
4     resultat = numerateur / denominateur
5     print("Le resultat de la division est", resultat)
6 except ValueError:
7     print("Désolé, les valeurs saisies ne sont pas des nombres.")
8 except ZeroDivisionError:
9     print("Désolé, la division par zéro n'est pas permise.")
```

L'intérêt de la structure du `try except` est qu'elle permet de dissocier dans le code la partie du traitement normal de la partie du traitement des erreurs.

Note : Si le même traitement est applicable pour des exceptions de types différents,

il est possible de fournir un seul bloc `except` avec le tuple des exceptions concernées :

```
try:
    numerateur = int(input("Entrez un numérateur : "))
    denominateur = int(input("Entrez un dénominateur : "))
    resultat = numerateur / denominateur
    print("Le resultat de la division est", resultat)
except (ValueError, ZeroDivisionError):
    print("Désolé, quelque chose ne s'est pas bien passé.")
```

16.1.1 Récupérer les données d'une exception

Une exception est un objet, donc elle peut offrir des attributs et des méthodes. Les exceptions en Python héritent de la classe `BaseException` et possèdent une représentation sous forme de chaîne de caractères pour fournir un message. Pour avoir accès à l'exception, on utilise la syntaxe suivante :

```
1 nombre = input("Entrez nombre : ")
2 try:
3     nombre = int(nombre)
4 except ValueError as e:
5     print(e)
```

À la ligne 4, on précise que l'exception de type `ValueError` est accessible dans le bloc `except` sous le nom `e` ce qui permet de demander l'affichage de l'exception.

16.2 Clause *else*

Il est possible d'ajouter une clause `else` après les blocs `try except`. Le bloc `else` est exécuté uniquement si le bloc `try` se termine normalement, c'est-à-dire sans qu'une exception ne survienne.

```
1 try:
2     numerateur = int(input("Entrez un numérateur : "))
3     denominateur = int(input("Entrez un dénominateur : "))
4     resultat = numerateur / denominateur
5 except (ValueError, ZeroDivisionError):
6     print("Désolé, quelque chose ne s'est pas bien passé.")
7 else:
8     print("Le resultat de la division est", resultat)
```

Note : Le bloc `else` permet de distinguer la partie du code qui est susceptible de produire une exception de celle qui fait partie du comportement nominal du code mais qui ne produit pas d'exception.

16.3 Post-traitement

Dans certain cas, on souhaite réaliser un traitement après le bloc `try` que ce dernier se termine correctement ou bien qu'une exception soit survenue. Dans cas, on place le code dans un bloc `finally`.

```
1  try:
2      numerateur = int(input("Entrez un numérateur : "))
3      denominateur = int(input("Entrez un dénominateur : "))
4      resultat = numerateur / denominateur
5      print("Le resultat de la division est", resultat)
6  except (ValueError, ZeroDivisionError):
7      print("Désolé, quelque chose ne s'est pas bien passé.")
8  finally:
9      print("afficher ceci quel que soit le résultat")
```

Un bloc `finally` est systématique appelé même si le bloc `try` est interrompu par une instruction `return`.

16.4 Lever une exception

Il est possible de signaler une exception grâce au mot-clé `raise`.

```
if x < 0:
    raise ValueError
```

Pour la plupart des exceptions, il est possible de passer en paramètre un message pour décrire le cas exceptionnel :

```
if x < 0:
    raise ValueError("La valeur ne doit pas être négative")
```

Le mot-clé `raise` est également utilisé pour relancer une exception dans un bloc mot-clé `except`.

```
1  try:
2      numerateur = int(input("Entrez un numérateur : "))
3      denominateur = int(input("Entrez un dénominateur : "))
4      resultat = numerateur / denominateur
5      print("Le resultat de la division est", resultat)
6  except (ValueError, ZeroDivisionError):
7      print("Désolé, quelque chose ne s'est pas bien passé.")
8      raise
```

Dans l'exemple ci-dessus, l'exception traitée dans le bloc `except` est relancée à la ligne 8.

Note : Il est également possible de créer une nouvelle exception à partir d'une

exception existante. Dans ce cas, la nouvelle exception aura comme cause l'exception d'origine

```
raise MyException from cause_exception
```

16.5 Créer son exception

Il existe beaucoup de classes pour représenter des exceptions en Python. Vous pouvez consulter la documentation pour connaître la hiérarchie des exceptions. Néanmoins, il est très simple de créer ses propres exceptions. Il est recommandé de créer des exceptions en héritant de `Exception` ou d'une classe héritant de `Exception`. `Exception` est une classe qui hérite de `BaseException`. Pour simplifier l'implémentation, `BaseException` définit l'attribut `args` qui contient tous les paramètres passés au constructeur.

```
class MonException(Exception):
    pass

try:
    raise MonException("Mon message")
except MonException as e:
    print(e.args)
    # Affiche ('Mon message',)
    print(e)
    # Affiche Mon message
```

CHAPITRE 17

Les méthodes spéciales (*dunders*)

Une classe peut déclarer des méthodes spéciales. Ces méthodes sont destinées à permettre à un objet d'avoir des comportements particuliers. Par exemple, comment faire pour que les instances de notre classe `Vecteur` puissent être utilisées avec des opérateurs arithmétiques ? Il suffit que la classe fournisse une implémentation pour une ou plusieurs méthodes spéciales. Ces méthodes sont facilement reconnaissables car leur nom est encadré par deux caractères soulignés (*double underscores*) comme par exemple `__add__()`. Cela a valu à ces méthodes leur surnom (non officiel) de *dunder*. La documentation Python y fait parfois référence sous le nom de *magic methods*.

Important : Si une méthode spéciale ne peut pas réaliser correctement son traitement à cause de la valeur ou du type d'un des paramètres qui lui sont transmis, elle doit retourner le mot-clé spécial `NotImplemented` ou produire une exception.

17.1 Méthodes spéciales élémentaires

Nous avons déjà traité d'une méthode spéciale : la méthode `__init__()` qui désigne le constructeur. Pour des usages plus avancés, on peut définir les méthodes `__new__()` et `__del__()` pour réaliser les traitements de création et de suppression des objets.

On peut également déclarer la méthode `__repr__()` qui doit retourner la chaîne de caractères correspondant à la représentation de l'objet. Cette méthode est appelée directement par la fonction `repr()`. C'est également cette méthode qui est utilisée par la console Python pour afficher un objet. Si nous reprenons comme exemple notre classe `Vecteur` du chapitre sur la programmation objet, nous pouvons proposer une implémentation pour la méthode `__repr__()`.

```
import math

class Vecteur:

    __slots__ = ('x', 'y')

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    @property
    def norme(self):
        return math.sqrt(self.x**2 + self.y**2)

    def calculer_produit_scalaire(self, v):
        return self.x * v.x + self.y * v.y

    def normaliser(self):
        norme = self.norme
        self.x /= norme
        self.y /= norme

    def __repr__(self):
        return f"Vecteur({self.x},{self.y})"
```

```
>>> v = Vecteur(5, 15)
>>> v
Vecteur(5,15)
```

La méthode `__hash__()` permet de produire un nombre de hachage. Ce nombre est nécessaire si votre objet doit être ajouté dans un ensemble ou comme clé d'un dictionnaire. Deux objets égaux doivent avoir le même nombre de hachage. Par contre l'inverse n'est pas nécessairement vrai. Le calcul d'un nombre de hachage n'est pas trivial car ce nombre est utilisé en interne par les ensembles et les dictionnaires pour garantir l'unicité des objets mais aussi pour assurer une bonne performance dans l'accès aux données. La fonction `hash()` peut vous aider à le calculer à partir des attributs d'un objet si ces derniers peuvent eux-mêmes produire une valeur de hachage. Il peut être intéressant de produire une valeur de hachage pour les instances de notre classe `Vecteur`.

```
import math

class Vecteur:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    @property
    def norme(self):
        return math.sqrt(self.x**2 + self.y**2)

    def calculer_produit_scalaire(self, v):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        return self.x * v.x + self.y * v.y

    def normaliser(self):
        norme = self.norme
        self.x /= norme
        self.y /= norme

    def __repr__(self):
        return f"Vecteur({self.x},{self.y})"

    def __hash__(self):
        return hash((self.x, self.y))
```

De cette manière, une instance de `Vecteur` peut être ajoutée dans un ensemble.

```
>>> v1 = Vecteur(1, 2)
>>> v2 = Vecteur(5, 4)
>>> v3 = Vecteur(1, 2)
>>> ensemble_vecteurs = {v1, v2, v3}
>>> ensemble_vecteurs
{Vecteur(1,2), Vecteur(5,4)}
```

Prudence : La valeur de hachage est généralement dépendante de l'état interne de l'objet. Pour notre classe `Vecteur`, cette valeur est par exemple calculée à partir des valeurs des attributs `x` et `y`. Si ces valeurs changent, la valeur de hachage changera également. Hors, pour que les ensembles et les clés de dictionnaire fonctionnent correctement, la valeur de hachage ne doit plus changer à partir du moment où l'objet est ajouté dans un ensemble ou qu'il est utilisé comme clé dans un dictionnaire. Soit le développeur qui utilise les objets doit s'assurer que l'état de l'objet ne changera pas, soit le développeur de la classe doit s'assurer qu'une fois l'état de l'objet positionné, il n'est plus possible de le modifier. C'est cette dernière stratégie qui est la plus sûr et qui est utilisée dans l'API standard de Python. On parle alors **d'immutabilité** des objets ou de classe **immutable**. C'est pour cette raison que les classes `int`, `float`, `bool`, `str` et `tuple` sont immutables. Il n'est pas possible d'altérer l'état interne des objets une fois créés. Une opération de modification produit en fait un nouvel objet avec le nouvel état.

Pour notre exemple de la classe `Vecteur`, il faudrait réfléchir à deux fois à notre implémentation car la méthode `normaliser()` modifie l'état interne d'un objet de ce type. Soit il faut revoir l'implémentation de cette méthode afin qu'elle produise un nouvel objet, soit il faudra peut-être abandonner l'idée qu'un objet de type `Vecteur` produise une valeur de hachage.

17.2 Méthodes spéciales pour les conversions

Il est possible de réaliser des conversions lorsque l'objet est passé en paramètre de certaines fonctions. La conversion en valeur booléenne est également utilisée

lorsqu'un objet doit être évalué comme expression booléenne dans une structure `if` ou `while`.

Méthode spéciale	fonction de conversion
`__str__(self)`	`str`
`__bytes__(self)`	`bytes`
`__bool__(self)`	`bool` ou expression booléenne
`__int__(self)`	`int`
`__float__(self)`	`float`
`__complex__(self)`	`complex`
`__dict__(self)`	`dict`

Pour notre classe `Vector`, en nous inspirant du fonctionnement des nombres en Python, nous pourrions considérer qu'un vecteur est évalué à `True` si ses coordonnées sont différentes de zéro. La conversion en chaîne de caractères est également utile.

```
class Vecteur:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __bool__(self):
        return self.x != 0 and self.y != 0

    def __str__(self):
        return f"({self.x},{self.y})"

    # reste du code omis
```

```
v = Vecteur(0, 0)
if not v:
    print("évalué à False")
# Affiche évalué à False

v = Vecteur(5, 4)
if v:
    print("évalué à True")
# Affiche évalué à True

print(v)
# Affiche (5,4)
```

17.3 Méthodes spéciales pour les opérateurs unaires

Si les objets doivent pouvoir être utilisés avec les opérateurs unaires `+`, `-` ou s'ils peuvent être passés en paramètre de la fonction `abs()`, vous devez fournir respectivement une implémentation des méthodes `__pos__(self)`, `__neg__(self)`, `__abs__(self)`.

```
class Vecteur:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __neg__(self):
        return Vecteur(-self.x, -self.y)

    # reste du code omis
```

```
>>> v = Vecteur(2, 3)
>>> -v
Vecteur(-2,-3)
```

17.4 Méthodes spéciales pour les opérations arithmétiques

Si les objets doivent pouvoir être utilisés dans des opérations arithmétiques, alors vous pouvez fournir une implémentation pour les méthodes suivantes :

Méthode spéciale	opérateur ou fonction
`__add__(self, o)`	+
`__sub__(self, o)`	-
`__mul__(self, o)`	*
`__matmul__(self, o)`	@
`__truediv__(self, o)`	/
`__floordiv__(self, o)`	//
`__mod__(self, o)`	%
`__divmod__(self, o)`	`divmod()`
`__pow__(self, o, modulo)`	** ou `pow()`

Ces méthodes prennent toutes en paramètres `self` et le deuxième opérande de l'opérateur arithmétique. `__pow__(self, o, modulo)` accepte un troisième paramètre optionnel correspondant à la valeur du modulo passée en paramètre de la fonction `pow()`. Ces méthodes doivent retourner le résultat de l'opération sauf `__divmod__(self, o)` qui doit retourner le résultat de la division et le reste pour être conforme à `divmod()`. Normalement, ces méthodes ne doivent pas modifier l'état interne de l'objet mais, au plus, produire un nouvel objet (comme par exemple pour le résultat de l'addition).

Pour notre classe `Vecteur`, nous pouvons, par exemple, autoriser l'addition de deux vecteurs ou d'un vecteur avec un scalaire.

```
class Vecteur:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __add__(self, v):
        if isinstance(v, (int, float)):
            return Vecteur(self.x + v, self.y + v)
        if isinstance(v, Vecteur):
            return Vecteur(self.x + v.x, self.y + v.y)
        return NotImplemented

    # reste du code omis
```

```
>>> v1 = Vecteur(2, 3)
>>> v1
Vecteur(2,3)
>>> v2 = v1 + 2
>>> v2
Vecteur(4,5)
>>> v3 = v1 + v2
>>> v3
Vecteur(6,8)
```

Note : Remarquez l'utilisation du mot-clé `NotImplemented` pour le cas où l'addition ne concerne ni un scalaire, ni un vecteur. Le fait de retourner ce mot-clé produira une erreur de type `TypeError` lorsque l'addition ne sera pas possible.

```
>>> v = Vecteur()
>>> v + "ceci n'est pas un nombre ni un Vecteur"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Vecteur' and 'str'
```

La classe peut fournir une implémentation pour les méthodes dont le nom commence par un `r` (pour *right*). Ces méthodes seront appelées si l'objet est présent à droite d'un opérateur binaire et l'objet à gauche ne supporte pas l'opération.

Méthode	opérateur
`__radd__(self, o)`	`+`
`__rsub__(self, o)`	`-`
`__rmul__(self, o)`	`*`
`__rmatmul__(self, o)`	`@`
`__rtruediv__(self, o)`	`/`
`__rfloordiv__(self, o)`	`//`
`__rmod__(self, o)`	`%`
`__rpow__(self, o, modulo)`	`**`

Par exemple pour additionner un scalaire à un vecteur, il faut fournir une implémentation pour la méthode `__radd__(self, o)`.

```
class Vecteur:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __add__(self, v):
        if isinstance(v, (int, float)):
            return Vecteur(self.x + v, self.y + v)
        elif isinstance(v, Vecteur):
            return Vecteur(self.x + v.x, self.y + v.y)
        else:
            return NotImplemented

    def __radd__(self, v):
        return self.__add__(v)

    # reste du code omis
```

```
>>> v = Vecteur(2, 3)
>>> 10 + v
Vecteur(12,13)
```

Pour notre exemple, l'implémentation de la méthode `__radd__(self, o)` se contente de retourner le résultat de l'appel à `__add__(self, o)` puisque l'addition est commutative.

Enfin, la classe peut fournir une implémentation pour les opérateurs *in-place*. Il s'agit des opérateurs qui réalisent une opération et une affectation dans le même temps.

Méthode	opérateur
`__iadd__(self, o)`	`+=`
`__isub__(self, o)`	`-=`
`__imul__(self, o)`	`*=`
`__imatmul__(self, o)`	`@=`
`__itruediv__(self, o)`	`/=`
`__ifloordiv__(self, o)`	`//=`
`__imod__(self, o)`	`%=`
`__ipow__(self, o, modulo)`	`**=`

Généralement, l'implémentation de ces méthodes permet simplement une optimisation. Si ces méthodes ne sont pas présentes alors l'interpréteur appelle la méthode arithmétique équivalente et affecte le résultat à la variable. Ainsi, si l'appel à la méthode `__iadd__(self, o)` produit un résultat `NotImplemented`, l'instruction

```
a += b
```

sera réalisée par l'interpréteur sous la forme :

```
a = a.__add__(b)
```

Comme on s'attend généralement à ce que les opérateurs *in-place* modifient directement l'objet, fournir une implémentation pour ces opérateurs évite de créer un objet intermédiaire.

```
class Vecteur:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __iadd__(self, v):
        if isinstance(v, (int, float)):
            self.x += v
            self.y += v
        elif isinstance(v, Vecteur):
            self.x += v.x
            self.y += v.y
        else:
            return NotImplemented
        return self

    # reste du code omis
```

```
>>> v1 = Vecteur(2, 3)
>>> v2 = Vecteur(10, 50)
>>> v1 += v2
>>> v1
Vecteur(12,53)
```

17.5 Méthodes spéciales pour la comparaison

Par défaut, l'opérateur d'égalité == permet de comparer l'unicité en mémoire des objets. Ainsi les deux vecteurs ci-dessous ne sont pas égaux :

```
>>> v1 = Vecteur(1, 1)
>>> v2 = Vecteur(1, 1)
>>> v1 == v2
False
```

En effet, nous créons deux objets distincts que nous affectons respectivement à la variable v1 et à la variable v2.

Mais il serait plus intéressant de considérer que deux vecteurs sont égaux s'ils ont les mêmes valeurs pour x et pour y. Nous pouvons modifier ce comportement par défaut en fournissant notre propre méthode d'égalité :

```
class Vecteur:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __eq__(self, v):
        if isinstance(v, Vecteur):
            return self.x == v.x and self.y == v.y
        return False

    # reste du code omis
```

```
>>> v1 = Vecteur(1, 1)
>>> v2 = Vecteur(1, 1)
>>> v1 == v2
True
```

Important : En Python, si vous désirez vérifier s'il s'agit ou non du même objet en mémoire, il faut utiliser le mot clé `is`. Ainsi :

```
>>> v1 = Vecteur(1, 1)
>>> v2 = Vecteur(1, 1)
>>> v1 is v1
True
>>> v1 is v2
False
```

Ci-dessous, la liste des méthodes spéciales pour les opérateurs de comparaison :

Méthode spéciale	Opérateur de comparaison
`__eq__(self, v)`	o == v
`__ne__(self, v)`	o != v
`__lt__(self, v)`	o < v
`__le__(self, v)`	o <= v
`__gt__(self, v)`	o > v
`__ge__(self, v)`	o >= v

Note : S'il n'existe pas d'implémentation pour la méthode `__ne__(self, v)`, alors l'interpréteur qui tente de résoudre l'instruction :

```
o != v
```

appellera à la place :

```
not o.__eq__(v)
```

Astuce : Si vous souhaitez que les séquences de vos objets soient triables grâce à la fonction `sorted()`, vous devez au minimum fournir une implémentation pour les méthodes `__eq__(self, v)` et `__lt__(self, v)`.

17.6 Méthodes spéciales pour les conteneurs

Si vos objets doivent se comporter comme un conteneur (c'est-à-dire comme une liste ou un dictionnaire), vous pouvez fournir l'implémentation de méthodes spéciales telles que :

Méthode spéciale	Cas d'utilisation
`__len__(self)`	utilisation de la méthode `len()`
`__getitem__(self, key)`	o[key]
`__setitem__(self, key, value)`	o[key] = value
`__delitem__(self, key)`	del o[key]
`__contains__(self, key)`	key in o

```
class Vecteur:

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __len__(self):
        return 2

    def __getitem__(self, k):
        if k == 'x' or k == 0:
            return self.x
        if k == 'y' or k == 1:
            return self.y
        raise KeyError(k)

    def __setitem__(self, k, v):
        if not isinstance(v, (int, float)):
            raise TypeError
        if k == 'x' or k == 0:
            self.x = v
        elif k == 'y' or k == 1:
            self.y = v
        else:
            raise KeyError(k)

    # reste du code omis
```

```
>>> v = Vecteur(2, 5)
>>> len(v)
2
>>> v['x']
2
>>> v[0]
2
>>> v['y']
5
>>> v[1]
5
>>> v[0] = -2
>>> v[1] = -5
>>> v
Vecteur(-2,-5)
```

17.7 Tout est objet !

En découvrant les méthodes spéciales en Python, on se rend compte que les opérateurs arithmétiques ne sont que des raccourcis d'écriture pour les méthodes telles que `__add__(self, o)` ou `__sub__(self, o)`. Il en va de même pour les opérateurs `[]` ou même les opérations booléennes.

En fait en Python, **tout est un objet** ! Les nombres sont des objets et il est donc possible d'écrire :

```
>>> a = 1
>>> a.__add__(2)
3
```

Les objets représentant les entiers ont même des méthodes propres comme `bit_length()` pour connaître la taille en bits nécessaire pour stocker la valeur.

```
>>> a = 1
>>> a.bit_length()
1
>>> a = 1000000
>>> a.bit_length()
20
```

`int`, `float`, `complex`, `bool`, `str`, `tuple`, `list`, `set`, `dict` ne sont pas des fonctions mais les classes de ces différents types. Il est donc possible de créer des classes en héritant de ces types. Les classes `int`, `float`, `complex` et `bool` héritent même d'une classe commune : la classe `numbers.Number`.

```
>>> from numbers import Number
>>> isinstance(1, Number)
True
>>> isinstance(2.5, Number)
```

(suite sur la page suivante)

(suite de la page précédente)

```
True
>>> isinstance(1j, Number)
True
>>> isinstance(True, Number)
True
```

17.8 Les fonctions comme objet

En Python, une fonction (et une méthode) est aussi un objet de type `function`. Il existe une seule instance d'une fonction et on peut utiliser son nom pour accéder à ses attributs.

```
def compter():
    compter._cpt = getattr(compter, "_cpt", 0) + 1
    return compter._cpt
```

Dans l'exemple ci-dessus, la fonction `compter()` mémorise une valeur dans son attribut privé `_cpt`.

```
>>> print(compter())
1
>>> print(compter())
2
>>> print(compter())
3
>>> print(compter())
4
```

De même un objet peut se comporter comme une fonction. Pour cela, il suffit d'implémenter la méthode spéciale `__call__(self, args)`.

```
from numbers import Number


class Step:
    """Permet de réaliser un incrément ou un décrément d'une valeur"""

    @staticmethod
    def inc(step=1):
        return Step(step)

    @staticmethod
    def dec(step=1):
        return Step(-step)

    def __init__(self, step):
        self._step = step

    def __call__(self, valeur):
```

(suite sur la page suivante)

(suite de la page précédente)

```
        if not isinstance(valeur, Number):
            raise TypeError
        valeur += self._step
        return valeur
```

```
>>> inc = Step.inc()
>>> inc(1)
2
>>> dec = Step.dec()
>>> dec(3)
2
```

La fonction `callable()` permet de découvrir sur une variable ou un paramètre représente un type appelable, c'est-à-dire une fonction ou un objet disposant de la méthode `__call__(self, args)`.

```
>>> callable(print)
True
```

Note : Comme un objet peut se comporter comme une fonction, alors il est possible d'utiliser une classe également comme un décorateur.

```
1  class Traceur:
2
3      def __init__(self, func, msg_debut="Appel", msg_fin="Fin appel"):
4          self.func = func
5          self.msg_debut = msg_debut
6          self.msg_fin = msg_fin
7
8      def __call__(self, *args, **kwargs):
9          print(f"{self.msg_debut} {self.func.__name__} args={args} kwargs={kwargs}")
10         try:
11             return self.func(*args, **kwargs)
12         finally:
13             print(f"{self.msg_fin} {self.func.__name__}")
14
15
16 @Traceur
17 def say_hello(nom):
18     print(f"Hello {nom}")
```

```
>>> say_hello("David")
Appel say_hello args=('David',) kwargs={}
Hello David
Fin appel say_hello
>>> say_hello(nom="David")
Appel say_hello args=() kwargs={'nom': 'David'}
Hello David
Fin appel say_hello
```

17.9 Les classes comme objet

Les classes sont elles-mêmes des objets qui héritent de `type`. Par exemple, une classe possède un attribut `__name__` qui contient le nom de la classe. Ses attributs sont également composés des méthodes et des attributs de classe. Il est donc possible de traiter une classe comme n'importe quel objet dans un programme. On dit que Python est un **langage réflexif** car les éléments constitutifs du langage sont manipulables comme n'importe quel objet.

Si on crée la classe `Greeting` :

```
class Greeting:

    def say_hello(self, name):
        print(f"Hello {name}")
```

On peut bien sûr écrire :

```
>>> g = Greeting()
>>> g.say_hello("David")
Hello David
```

Mais on peut également traiter la classe comme un objet :

```
>>> g = Greeting()
>>> cls = type(g)
>>> cls.__name__
'Greeting'
>>> f = getattr(cls, "say_hello")
>>> f(g, "David")
Hello David
```

Les bibliothèques et les *frameworks* les plus avancés en Python peuvent ainsi découvrir dynamiquement la structure des objets et des classes au moment de l'exécution du programme et peuvent donc manipuler des types de données qui n'étaient pas connus au moment de l'écriture de ces bibliothèques et de ces *frameworks*.

17.10 Et bien d'autres méthodes spéciales…

Il existe encore d'autres méthodes spéciales pour la réflexivité ou encore pour permettre de réaliser des arrondis ou des opérations binaires. Pour la liste complète, vous pouvez consulter la documentation officielle du `Data model`. Le chapitre suivant présente les méthodes spéciales pour les itérateurs.

Ci-dessous, vous trouverez l'implémentation complète de la classe `Vecteur` :

```
1 import math
2
```

(suite sur la page suivante)

(suite de la page précédente)

```
3
4   class Vecteur:
5       """Un vecteur à deux dimensions"""
6
7       __slots__ = ('x', 'y')
8
9       def __init__(self, x=0, y=0):
10          self.x = x
11          self.y = y
12
13      @property
14      def norme(self):
15          return math.sqrt(self.x**2 + self.y**2)
16
17      def calculer_produit_scalaire(self, v):
18          return self.x * v.x + self.y * v.y
19
20      def normaliser(self):
21          norme = self.norme
22          self.x /= norme
23          self.y /= norme
24
25      def __repr__(self):
26          return f"Vecteur({self.x},{self.y})"
27
28      def __bool__(self):
29          return self.x != 0 and self.y != 0
30
31      def __str__(self):
32          return f"({self.x},{self.y})"
33
34      def __neg__(self):
35          return Vecteur(-self.x, -self.y)
36
37      def __add__(self, v):
38          if isinstance(v, (int, float)):
39              return Vecteur(self.x + v, self.y + v)
40          if isinstance(v, Vecteur):
41              return Vecteur(self.x + v.x, self.y + v.y)
42          return NotImplemented
43
44      def __radd__(self, v):
45          return self.__add__(v)
46
47      def __iadd__(self, v):
48          if isinstance(v, (int, float)):
49              self.x += v
50              self.y += v
51          elif isinstance(v, Vecteur):
52              self.x += v.x
53              self.y += v.y
54          else:
55              return NotImplemented
56          return self
57
58      def __eq__(self, v):
59          if isinstance(v, Vecteur):
```

(suite sur la page suivante)

(suite de la page précédente)

```
60              return self.x == v.x and self.y == v.y
61          return False
62
63      def __len__(self):
64          return 2
65
66      def __getitem__(self, k):
67          if k == 'x' or k == 0:
68              return self.x
69          if k == 'y' or k == 1:
70              return self.y
71          raise KeyError(k)
72
73      def __setitem__(self, k, v):
74          if not isinstance(v, (int, float)):
75              raise TypeError
76          if k == 'x' or k == 0:
77              self.x = v
78          elif k == 'y' or k == 1:
79              self.y = v
80          else:
81              raise KeyError(k)
```

17.11 Méthodes spéciales et classes abstraites

Beaucoup de méthodes abstraites n'ont de sens que lorsqu'elles sont implémentées ensemble par la même classe. Par exemple les méthodes `__len__(self)` `__getitem__(self, key)` permettent de définir une séquence puisqu'il est possible de connaître la taille et l'élément associé à une clé.

Le module `container.abc` fournit des classes abstraites qui définissent différents contrats. Il existe par exemple la classe abstraite `Sequence` qui déclare les deux méthodes de manière abstraite.

```
>>> from collections.abc import Sequence
>>> s = Sequence()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Sequence with abstract methods __getitem__
↪, __len__
```

Toutes les classes du module `colletions.abc` sont des classes abstraites qui sont là pour guider le développeur qui voudrait créer sa propre classe et qui souhaiterait que les objets de cette classe se comporte suivant un contrat. En hérite d'une des classes du module `colletions.abc`, cela permet au développeur de vérifier que son implémentation est conforme au contrat.

CHAPITRE 18

Itérateurs et Générateurs

18.1 Les itérateurs

Un itérateur permet de parcourir une séquence d'éléments. Les collections en Python comme les listes, les ensembles, les tuples, les dictionnaires et même les chaînes de caractères peuvent se comporter comme des itérateurs et être utilisés par exemple dans une expression `for`.

```
ma_liste = ["Pomme", "Poire", "Orange"]
for e in ma_liste:
    print(e)
```

Il est possible pour n'importe quel objet de se comporter comment un itérateur. Pour cela, il suffit qu'il implémente les méthodes spéciales `__iter__()` et `__next__()`.

`__iter__()` Retourne un objet qui sert d'itérateur. Un itérateur doit lui-même avoir une méthode `__iter__()` qui peut se limiter à retourner l'itérateur lui-même.

`__next__()` Retourne l'élément suivant. S'il n'y a plus d'élément, alors cette méthode doit lever une exception de type `StopIteration`.

Ci-dessous un exemple d'itérateur qui permet de compter jusqu'à 10 :

```
class Compteur:

    def __init__(self):
        self.nombre = 0

    def __iter__(self):
        return self
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def __next__(self):
        self.nombre += 1
        if self.nombre > 10:
            raise StopIteration
        return self.nombre
```

Il est possible d'utiliser la classe `Compteur` dans une expression `for` :

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

# Affiche les nombres de 1 à 10.
```

18.2 Les fonctions iter() et next()

La fonction `iter()` permet d'obtenir un itérateur à partir d'un objet. Concrètement, cette fonction appelle la méthode `__iter__()` de l'objet passé en paramètre.

La fonction `next()` attend un itérateur en paramètre et retourne l'élément suivant. Si l'itérateur est déjà positionné sur le dernier élément, cette fonction lève une exception de type `StopIteration`. Concrètement, cette fonction appelle la méthode `__next__()` de l'itérateur passé en paramètre.

```
it = iter(range(3))

print(next(it))
# affiche 0
print(next(it))
# affiche 1
print(next(it))
# affiche 2
print(next(it))
# provoque une exception StopIteration
```

Les méthodes `iter()` et `next()` permettent d'interagir directement avec un itérateur. Cependant on utilise la plupart du temps un itérateur dans une expression `for` ou avec le mot-clé `in`.

Si vous voulez rendre un objet itérable, vous pouvez simplement implémenter la méthode `__iter__()` dans votre classe de manière à ce qu'elle retourne le résultat d'un appel à la fonction `iter()` :

```
class Chemin:
    def __init__(self):
        self.direction = []

    def gauche(self):
        self.direction.append("gauche")
```

(suite sur la page suivante)

(suite de la page précédente)

```
    def droite(self):
        self.direction.append("droite")

    def __iter__(self):
        return iter(self.direction)


chemin = Chemin()
chemin.droite()
chemin.gauche()
chemin.gauche()
chemin.droite()

for direction in chemin:
    print(direction)

# Affiche
#   droite
#   gauche
#   gauche
#   droite
```

18.3 Les générateurs

Les générateurs sont une catégorie particulière d'itérateurs. Un générateur crée à la demande l'élément suivant de la séquence. Pour cela, le générateur peut utiliser une formule mathématique pour calculer une suite ou bien il peut utiliser une système externe comme une base de données pour extraire l'élément suivant. L'intérêt d'un générateur est qu'il n'est pas nécessaire de construire en mémoire la liste complète des éléments de la séquence. Les générateurs ont donc une empreinte mémoire très faible ce qui permet d'écrire des programmes optimisés.

Le générateur le plus couramment utilisé en Python est créé *via* la classe `range` :

```
for i in range(50000):
    print(i)
```

Dans l'exemple ci-dessus, la classe `range` ne crée pas un tableau de 50 000 éléments. Elle crée un itérateur qui se contente de reproduire un suite mathématique en ajoutant 1 à la valeur précédente.

Note : `range` est une amélioration notable de Python 3. En Python 2.x, son implémentation créée effectivement une séquence en mémoire de toutes les valeurs, ce qui est beaucoup moins performant.

Nous avons déjà présenté un exemple de générateur plus haut avec l'exemple de la classe `Compteur` qui est en fait une implémentation très simplifiée de `range` :

```
class Compteur:

    def __init__(self):
        self.nombre = 0

    def __iter__(self):
        return self

    def __next__(self):
        self.nombre += 1
        if self.nombre > 10:
            raise StopIteration
        return self.nombre
```

La classe `Compteur` ne conserve en mémoire que l'attribut `nombre`, c'est-à-dire la valeur courante. Cela lui permet de déduire la valeur suivante et de mettre à jour cet attribut à chaque appel de `__next(self)__`.

Il est donc possible de créer des générateurs en utilisant le principe d'implémentation des itérateurs. Cependant, Python fournit deux autres manières de créer des générateurs qui sont beaucoup plus simples et donc beaucoup plus utiles dans les programmes.

18.3.1 Les fonctions génératrices avec *yield*

Python dispose du mot-clé `yield`. Il permet de transformer une fonction en générateur. `yield` retourne l'élément suivant du générateur. Tout se passe comme si une instruction à `yield` suspendait l'exécution de la fonction qui se continuera au passage à l'élément suivant du générateur.

```
def ma_fonction():
    yield "un"
    yield "deux"
    yield "trois"


for x in ma_fonction():
    print(x)

# Affiche
#   un
#   deux
#   trois
```

Ainsi il est très facile d'implémenter la fonctionnalité identique à notre classe `Compteur` mais cette fois-ci sous la forme d'une fonction génératrice :

```
def compteur():
    cpt = 1
    while cpt <= 10:
        yield cpt
```

(suite sur la page suivante)

(suite de la page précédente)

```
        cpt += 1


for x in compteur():
    print(x)

# Affiche les nombres de 1 à 10
```

Une fonction génératrice est très souvent beaucoup plus simple à implémenter et à comprendre qu'un itérateur tout en permettant d'arriver au même résultat.

Il est possible d'utiliser la syntaxe `yield from` pour signaler que l'on souhaite créer une fonction génératrice à partir d'un générateur. Ainsi notre fonction génératrice `compteur()` peut simplement être implémentée à partir de `range` :

```
def compteur():
    yield from range(1, 11)
```

18.3.2 Les générateurs en compréhension

Comme pour les listes en compréhension, il est possible de définir un générateur en compréhension en utilisant des parenthèses plutôt que les crochets.

```
for i in (x**2 for x in range(5)):
    print(i)

# Affiche: 0 1 4 9 16
```

Même si la syntaxe est très proche, le mécanisme sous-jacent est très différent de la liste en compréhension. Si vous prenez les exemples ci-dessous :

```
[x**2 for x in range(1,1001)]


(x**2 for x in range(1,1001))
```

Le premier est une liste en compréhension qui crée donc une liste de 1000 éléments en mémoire. Le second est un générateur en compréhension. Il s'agit donc d'une fonction qui peut fournir à la demande la valeur de l'élément suivant de la séquence. Il n'y a donc aucune liste en mémoire qui est créée.

Note : Il n'est pas nécessaire d'écrire les parenthèses quand on passe le générateur comme paramètre d'une fonction :

```
sum(x**2 for x in range(10))
```

18.4 Les fonctions *enumerate*, *map*, *zip*, *filter*

Parmi les fonctions de base en Python (appelées `builtins functions`), il existe des fonctions qui produisent des itérateurs. Nous connaissons déjà `range()` (qui est en fait une classe en Python 3) : elle crée un itérateur sur une suite de nombres. Mais il en existe quatre autres fonctions extrêmement utiles.

`enumerate()` produit un itérateur qui retourne un tuple contenant un compteur de l'itération courante et la valeur obtenue à partir de l'itérateur passé en paramètre. Le paramètre nommé `start` permet d'indiquer la valeur de départ du compteur (par défaut 0) :

Code source 1 – Affichage des jours de la semaine avec leur numéro

```
la_semaine = ["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi",
↪"dimanche"]
for cpt, v in enumerate(la_semaine, start=1):
    print(cpt, v)

# Affiche
#   1 lundi
#   2 mardi
#   3 mercredi
#   4 jeudi
#   5 vendredi
#   6 samedi
#   7 dimanche
```

`map()` produit un itérateur qui accepte une fonction pour produire une nouvelle valeur à partir de la valeur obtenue par l'itérateur passé en second paramètre :

Code source 2 – Affichage des jours de la semaine en majuscules

```
la_semaine = ["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi",
↪"dimanche"]
for v in map(str.upper, la_semaine):
    print(v)

# Affiche
#   LUNDI
#   MARDI
#   MERCREDI
#   JEUDI
#   VENDREDI
#   SAMEDI
#   DIMANCHE
```

Cette fonction permet également de combiner les valeurs produites par plusieurs itérateurs :

Code source 3 – Concaténation deux à deux des lettres de deux mots

```
for v in map(lambda x, y: x + y, "hello", "world"):
    print(v)

# Affiche
#   hw
#   eo
#   lr
#   ll
#   od
```

zip() produit un itérateur qui produit un tuple regroupant les valeurs de chacun des itérateurs passés en paramètre. L'itération s'arrête lorsque l'un des itérateurs se termine.

filter() produit un itérateur qui retourne la valeur de l'itérateur passé en second paramètre que si la fonction passée en premier paramètre retourne `True` pour cette valeur.

Code source 4 – Affichage des jours de la semaine qui commence par un m

```
la_semaine = ["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi",
↪"dimanche"]
for v in filter(lambda x: x.startswith("m"), la_semaine):
    print(v)

# Affiche
#   mardi
#   mercredi
```

18.5 Le module *itertools*

Le module `itertools` fournit des fonctions très utiles pour créer des générateurs. Par exemple, il est possible de créer un générateur infini (du moins allant jusqu'à la plus grande valeur possible d'un entier) grâce à `count()` :

```
import itertools as it

for i in it.count():
    # Attention l'itération ne s'arrête pas avant longtemps
    print(i)
```

Il est également possible de réaliser un produit cartésien entre plusieurs générateurs grâce à la fonction `product()` :

```python
import itertools as it

for x, y in it.product(range(5), range(100,102)):
    print(x, y)

# Affiche
#   0 100
#   0 101
#   1 100
#   1 101
#   2 100
#   2 101
#   3 100
#   3 101
#   4 100
#   4 101
```

CHAPITRE 19

Gestion des fichiers

Python permet très facilement de manipuler des fichiers tout en garantissant une très bonne portabilité du code quel que soit le système d'exploitation sur lequel s'exécute le programme (pour peu que le développeur soit attentif à ce type de problématique).

Les opérations de manipulation de fichier : écriture, lecture, création, suppression sont toutes susceptibles d'échouer. Dans ce cas, les fonctions ou les méthodes produiront une erreur de type `OSError` ou d'un type héritant de cette exception.

19.1 Ouvrir et fermer un fichier

La fonction `open()` permet de récupérer un descripteur de fichier en passant en paramètre le chemin de ce fichier. Le descripteur de fichier va nous permettre de réaliser les opérations de lecture et/ou d'écriture dans le fichier.

Un descripteur de fichier est une ressource système. Cela signifie qu'il est géré par le système d'exploitation. Lorsque les opérations sur le fichier sont terminées, il faut fermer le descripteur de fichier à l'aide de la méthode `close()`. Ne pas fermer un descripteur de fichier conduit à une fuite de ressources (*resource leak*). Si un programme ouvre trop de fichiers sans jamais fermer les descripteurs de fichiers, le système d'exploitation peut finir par refuser d'ouvrir des fichiers supplémentaires.

```
f = open("monfichier.txt)
# faire des opérations sur le contenu du fichier
f.close()
```

Comme des erreurs peuvent survenir lors de la lecture ou de l'écriture dans un fichier ou comme il est très facile d'oublier d'appeler la méthode `close()`, Python fournit une syntaxe spéciale : le `with`.

```
with open("monfichier.txt") as f:
    # faire des opérations sur le contenu du fichier
    pass
```

La syntaxe `with` **appelle automatiquement** la méthode `close()` du descripteur de fichier à la sortie du bloc (même si la sortie du bloc est due à une erreur).

Note : La syntaxe `with` fonctionne en Python pour divers types de ressources du système : les fichiers, les accès réseau, les processus, les objets de synchronisation entre processus et threads.

Vous pouvez gérer plusieurs ressources avec un `with` :

```
with open("monfichier.txt") as f, open("autrefichier.txt") as f2:
    # faire des opérations sur le contenu des fichiers
    pass
```

Si vous souhaitez développer une classe qui fonctionne comme un gestionnaire de ressources et dont les instances peuvent être initialisées dans une structure `with` comme les descripteurs de fichiers, alors votre classe doit fournir une implémentation pour les méthodes spéciales `__enter__()` et `__exit__()`.

19.2 Lire le contenu d'un fichier

Par défaut, la fonction `open()` permet de lire le contenu d'un fichier texte.

Supposons que nous ayons un fichier nommé `dialogues.txt` dans le répertoire courant.

Code source 1 – Le fichier dialogues.txt

```
- Halt! Who goes there?

- It is I, Arthur, son of Uther Pendragon, from the castle
  of Camelot. King of the Britons, defeator of the Saxons,
  sovereign of all England!
```

Pour obtenir le contenu du fichier dans une chaîne de caractères :

```
with open("dialogues.txt") as f:
    contenu = f.read()
print(contenu)
```

Pour obtenir le contenu du fichier sous la forme d'un tableau de chaînes de caractères (un élément par ligne) :

```
with open("dialogues.txt") as f:
    lignes = f.readlines()
print(len(lignes))
# affiche 5
```

Un descripteur de fichier agit également comme une séquence sur les lignes d'un fichier.

```
with open("dialogues.txt") as f:
    for ligne in f:
        print(ligne)
```

Note : Lorsqu'on lit un fichier texte, chaque ligne inclue le caractère de retour à la ligne présent dans le fichier. Pour le supprimer, on peut utiliser la méthode `str.rstrip()`

```
with open("dialogues.txt") as f:
    for ligne in f:
        print(ligne.rstrip('\n'))
```

19.3 Les modes de fichier

Lorsqu'on ouvre un fichier, il faut préciser le mode d'ouverture qui dépend du type du fichier et des opérations que l'on souhaite réaliser. Le mode est représenté par une chaîne de caractères qui est passée à la fonction `open()` avec le paramètre `mode`.

Mode	Description
r	ouverture en lecture (mode par défaut)
w	ouverture en écriture (efface le contenu précédent)
x	ouverture uniquement pour création (l'ouverture échoue si le fichier existe déjà)
+	ouverture en lecture et écriture
a	ouverture en écriture pour ajout en fin de fichier
b	fichier binaire
t	fichier texte (mode par défaut)

```
# ouverture en écriture
with open("dialogues.txt", mode="w") as f:
    pass
```

```
# ouverture d'un fichier binaire en création
with open("fichier.bin", mode="xb") as f:
    pass
```

19.3.1 Spécificité du mode texte

Ouvrir un fichier en mode texte (mode par défaut ou `t`) entraîne un travail de conversion par Python. Convertir les données d'un fichier en chaîne de caractères exige d'utiliser une **famille d'encodage**. Il est possible de préciser la famille d'encodage d'un fichier grâce au paramètre `encoding` :

```
# ouverture en écriture
with open("dialogues.txt", encoding="utf-8") as f:
    pass
```

Si le paramètre `encoding` n'est pas spécifié alors Python utilise un encodage qui est **dépendant du système** qui exécute le code (ce qui peut nuire à la portabilité des fichiers produits). Pour connaître l'encodage utilisé par défaut par l'interpréteur, il faut utiliser les méthodes du module `locale` :

```
>>> import locale
>>> locale.getpreferredencoding()
'UTF-8'
```

Le mode texte entraîne également une conversion des caractères de fin de ligne puisque tous les systèmes d'exploitation n'utilisent pas la même convention. Python garantit une représentation universelle du caractère de fin de ligne en utilisant `\n`.

Le mode binaire (`b`) est un mode qui permet d'accéder directement au contenu du fichier sans conversion de la part de Python. Dans ce cas, la lecture du fichier retourne des `bytes` et non pas des chaînes de caractères.

19.4 Écrire dans un fichier

Pour écrire d'un bloc dans un fichier, on peut utiliser la méthode `write()` et pour écrire une liste de lignes, il faut utiliser la méthode `writelines()`. Attention pour les fichiers textes, ces méthodes n'ajoutent pas de caractères de fin de ligne, il faut donc les écrire explicitement.

```
lignes = ["- Pull the other one!\n",
          "- I am. And this my trusty servant Patsy.\n"]

# ouverture d'un fichier texte en ajout
with open("dialogues.txt", mode="a") as f:
    f.writelines(lignes)
```

Prudence : Il faut se rappeler que le mode d'ouverture en écriture (`w`) **remplace intégralement** le contenu du fichier. Pour ajouter à la fin du fichier, il faut ouvrir le fichier en mode ajout (`a`) sous peine de perdre tout le contenu précédemment sauvé.

19.5 Chemin de fichier

Les fichiers sont organisés selon une structure arborescente dans laquelle un nœud peut être soit un fichier soit un répertoire. Même si tous les systèmes d'exploitation suivent le même principe, il existe des différences majeures d'organisation. Le système MS-Windows utilise un système de fichiers multi-têtes (c :, d :, e :...) tandis que les systèmes *nix et MacOS utilisent un système mono-tête dont la racine est `/`. Dans la représentation des chemins de fichiers, MS-Windows utilise le caractère `\` pour séparer les composants d'un chemin :

```
C:\\Users\david\Documents\monfichier.txt
```

tandis que les systèmes *nix et MacOs utilisent le caractère `/`

```
/home/david/Documents/monfichier.txt
```

Enfin, il faut se souvenir que le système de fichiers de MS-Windows n'est pas sensible à la casse (*case insensitive*), c'est-à-dire que les mots peuvent être écrits en lettres majuscules ou en lettres minuscules. Au contraire, les systèmes *nix et MacOS sont sensibles à la casse (*case sensitive*), c'est-à-dire qu'un mot écrit en lettres majuscules est différent d'un mot écrit en lettres minuscules.

Toutes ces nuances peuvent rendre difficiles l'écriture d'un programme portable d'un système à l'autre. Heureusement, la bibliothèque standard Python fournit plusieurs solutions pour aider les développeurs.

19.5.1 Le module *os.path*

Le module `os.path` fournit des fonctions élémentaires pour nous aider à gérer les chemins de fichiers.

`join()` Cette fonction permet de créer un chemin en utilisant le séparateur approprié pour le système.

```
import os.path as path

chemin = path.join("fichiers", "monfichier.txt")
print(chemin)
# Sous Windows affiche fichiers\monfichier.txt
# Sous *nix ou MacOS, affiche fichiers/monfichier.txt
```

abspath() Cette fonction retourne le chemin absolu.

```
import os.path as path

chemin = path.abspath("monfichier.txt")
print(chemin)
# Si le répertoire de travail est /home/david/Documents
# affiche /home/david/Documents/monfichier.txt
```

19.5.2 Le module *pathlib*

Le module `pathlib` est un module de haut-niveau qui permet à la fois de manipuler un chemin mais également d'interagir avec le fichier ou le répertoire désigné par ce chemin. C'est un module tout-en-un qui facilite grandement le travail sur les fichiers. L'élément central du module est la classe `Path`.

```
>>> from pathlib import Path
>>> path = Path("/", "home", "david", "Documents", "monfichier.txt")
>>> str(path)
'/home/david/Documents/monfichier.txt'
>>> path.parts
('/', 'home', 'david', 'Documents', 'monfichier.txt')
>>> path.root
'/'
>>> path.drive
''
>>> path.name
'monfichier.txt'
>>> parent = path.parent
>>> parent.parts
('/', 'home', 'david', 'Documents')
>>> path.is_file()
True
>>> path.is_dir()
False
>>> path.parent.is_file()
False
>>> path.parent.is_dir()
True
```

La classe `Path` possède la méthode `open()` qui accepte les mêmes paramètres que la fonction `open()` sauf le chemin qui est déjà représenté par l'objet lui-même :

```
from pathlib import Path


chemin = Path("dialogues.txt")

with chemin.open() as f:
    for ligne in f:
        print(ligne)
```

On peut même faire l'économie de ce code en appelant la méthode `read_text()` qui

ouvre le fichier en mode texte, lit l'intégralité du fichier et referme le fichier :

```
from pathlib import Path


chemin = Path("dialogues.txt")
contenu = chemin.read_text()
```

Pour construire un chemin à partir d'un autre chemin, il suffit d'utiliser l'opérateur / qui est utilisé, non pas comme opérateur de la division, mais comme le séparateur universel de chemin de fichiers :

```
>>> chemin = Path("mondossier")
>>> chemin_fichier = chemin / "mon fichier.txt"
>>> chemin_fichier.parts
('mondossier', 'mon fichier.txt')
```

```
>>> chemin_fichier = Path.home() / "Documents" / "monfichier.txt"
>>> str(chemin_fichier)
'/home/david/Documents/monfichier.txt'
```

19.6 Actions sur les fichiers et les répertoires

Il est possible de réaliser des opérations élémentaires sur les fichiers et les répertoires soit avec les modules `os` et `shutil` soit avec le module `pathlib`. Les modules `os` et `shutil` sont historiquement les premiers modules qui ont été introduits en Python. Ils proposent surtout des fonctions alors que le module `pathlib` est orienté objet avec notamment la classe `Path`.

19.6.1 Connaître le répertoire de travail

Le répertoire de travail correspond au répertoire courant au moment du lancement de l'interpréteur Python.

Code source 2 – Avec le module `os`

```
>>> import os
>>> os.getcwd()
```

Code source 3 – Avec le module `pathlib`

```
>>> import pathlib
>>> pathlib.Path.cwd()
```

19.6.2 Connaître le répertoire de l'utilisateur

Code source 4 – Avec le module `os`

```
>>> import os
>>> os.environ['HOME']
```

Code source 5 – Avec le module `pathlib`

```
>>> import pathlib
>>> pathlib.Path.home()
```

19.6.3 Copier un fichier

Code source 6 – Avec le module `shutil`

```
>>> import shutil
>>> shutil.copy("monfichier.txt", "macopie.txt")
```

19.6.4 Supprimer un fichier

Code source 7 – Avec le module `os`

```
>>> import os
>>> os.remove("monfichier.txt")
```

Code source 8 – Avec le module `pathlib`

```
>>> import pathlib
>>> p = pathlib.Path("monfichier.txt")
>>> p.unlink()
```

19.6.5 Créer un répertoire

Code source 9 – Avec le module `os`

```
>>> import os
>>> os.mkdir("monrepertoire")
```

Code source 10 – Avec le module `pathlib`

```
>>> import pathlib
>>> p = pathlib.Path("monrepertoire")
>>> p.mkdir()
```

19.6.6 Supprimer un répertoire

Pour supprimer un répertoire, ce dernier doit être vide.

Code source 11 – Avec le module `os`

```
>>> import os
>>> os.rmdir("monrepertoire")
```

Code source 12 – Avec le module `pathlib`

```
>>> import pathlib
>>> p = pathlib.Path("monrepertoire")
>>> p.rmdir()
```

19.6.7 Vérifier qu'un fichier existe

Code source 13 – Avec le module `os.path`

```
>>> import os.path
>>> os.path.exists("monfichier.txt")
```

Code source 14 – Avec le module `pathlib`

```
>>> import pathlib
>>> p = pathlib.Path("monfichier.txt")
>>> p.exists()
```

19.6.8 Lister le contenu d'un répertoire

Code source 15 – Avec le module `os`

```
>>> import os
>>> liste_fichiers = os.listdir("monrepertoire")
```

Code source 16 – Avec le module `pathlib`

```
>>> import pathlib
>>> p = pathlib.Path("monrepertoire")
>>> liste_fichiers = list(p.iterdir())
```

La méthode `iterdir()` retourne un itérateur sur des objets de type `Path` plutôt qu'un tableau de chaînes de caractères comme `os.listdir()`.

19.6.9 Rechercher des fichiers

Le module `glob` permet d'effectuer une recherche dans l'arborescence de fichiers. On peut utiliser le caractère ? pour représenter n'importe quel caractère et * pour représenter n'importe quelle suite de caractères.

Code source 17 – Avec le module `glob`

```
>>> import glob
>>> liste_fichiers = glob.glob("*.py")
```

Il est possible d'effectuer une recherche récursive (c'est-à-dire en incluant les sous répertoires) en positionnant la paramètre `recursive` à `True` et en utilisant la séquence ** pour indiquer un ou plusieurs sous répertoires.

Code source 18 – Avec le module `glob`

```
>>> import glob
>>> liste_fichiers = glob.glob("**/*.py", recursive=True)
```

La classe `Path` possède également la méthode `glob()`.

Code source 19 – Avec le module `pathlib`

```
>>> import pathlib
>>> liste_fichiers = list(pathlib.Path.cwd().glob("**/*.py"))
```

La méthode `glob()` retourne un itérateur sur des objets de type `Path` plutôt qu'un tableau de chaînes de caractères comme `glob.glob()`.

19.7 Lecture de fichiers CSV

Le fichier CSV (*comma separated values*) est un format texte très simple pour stocker des tables de données. Le module `csv` offre des méthodes pour lire et écrire.

Si on dispose du fichier suivant :

Code source 20 – Le fichier *filmographie.csv*

```
1971,And Now for Something Completely Different
1975,Holy Grail
1979,Life of Brian
1983,The Meaning of Life
1996,The Wind in the Willows
```

```
import csv

with open("filmographie.txt") as f:
    lecteur = csv.reader(f)
    for ligne in lecteur:
        print(ligne)
```

Chaque ligne lue est un tableau de chaînes de caractères contenant chaque valeur en colonne :

```
['1971', 'And Now for Something Completely Different']
['1975', 'Holy Grail']
['1979', 'Life of Brian']
['1983', 'The Meaning of Life']
['1996', 'The Wind in the Willows']
```

CHAPITRE 20

Les tests unitaires automatisés avec unittest

Un test automatisé est un programme qui se découpe en trois étapes dites AAA pour *Arrange*, *Act*, *Assert*.

Arrange La mise en place de l'environnement : création et initialisation des objets nécessaires à l'exécution du test.

Act Le test proprement dit.

Assert La vérification des résultats obtenus par le test.

Le sous-système (l'ensemble des objets) éprouvé par le test est parfois appelé **SUT** (System Under Test).

On distingue différentes catégories de tests :

— Tests unitaires : testent une partie (une unité) d'un système afin de s'assurer qu'il fonctionne correctement (*build the system right*)
— Tests d'acceptation : testent le système afin de s'assurer qu'il est conforme aux besoins (*build the right system*)
— Tests d'intégration : testent le système sur une plate-forme proche de la plate-forme cible
— Tests de sécurité : testent que l'application ne contient pas de failles de sécurité connues (injection de code, attaque XSS, …)
— Tests de robustesse : testent le comportement de l'application au limite des ressources disponibles (mémoire, CPU, …) sur la plate-forme

Pour réaliser des tests unitaires, `unittest` est le framework de test intégré dans la bibliothèque standard Python.

20.1 Structure d'une classe de test

Les tests sont regroupés dans des classes de test. Généralement, on groupe dans une classe les tests ayant la même classe ou le même module comme point d'entrée.

La classe de test doit impérativement héritée de `unittest.TestCase`. Les tests sont représentés par des méthodes dont le nom commence par `test`.

Code source 1 – Exemple d'une classe de test

```
import unittest


class UneClasseDeTest(unittest.TestCase):

    def test_simple(self):
        self.assertTrue(True)
```

Pour exécuter une classe de test, il faut utiliser la fonction `unittest.main()`. Pour permettre de rendre le module de test exécutable, on ajoute donc dans le fichier :

```
if __name__ == '__main__':
    unittest.main()
```

Il est possible d'exécuter des instructions avant et après chaque test pour allouer et désallouer des ressources nécessaires à l'exécution des tests. On redéfinit pour cela les méthodes `setUp()` et `tearDown()`.

```
import unittest


class UneClasseDeTest(unittest.TestCase):

    def setUp(self):
        print("Avant le test")

    def tearDown(self):
        print("Après le test")

    def test_simple(self):
        self.assertTrue(True)


if __name__ == '__main__':
    unittest.main()
```

Pour réaliser les assertions, une classe de test hérite de `unittest.TestCase` des méthodes d'assertion.

Méthode	vérifie
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

Toutes ces méthodes acceptent le paramètre optionnel `msg` pour passer un message d'erreur à afficher si l'assertion échoue.

20.1.1 Exécution des tests

En rendant chaque module de test exécutable, il suffit de lancer le module lui-même pour produire un rapport de test.

Code source 2 – le fichier `test_str.py`

```
import unittest


class ChaineDeCaractereTest(unittest.TestCase):

    def test_reversed(self):
        resultat = reversed("abcd")
        self.assertEqual("dcba", "".join(resultat))

    def test_sorted(self):
        resultat = sorted("dbca")
        self.assertEqual(['a', 'b', 'c', 'd'], resultat)

    def test_upper(self):
        resultat = "hello".upper()
        self.assertEqual("HELLO", resultat)


if __name__ == '__main__':
    unittest.main()
```

```
$ python3 test_str.py

...
----------------------------------------------------------------------
Ran 3 tests in 0.000s
```

(suite sur la page suivante)

(suite de la page précédente)

```
OK
```

Il est également possible de lancer directement le module `unittest` grâce à l'option `-m` en passant comme paramètre les modules qui définissent les tests (le nom des fichiers sans l'extension `.py`). On peut ainsi lancer plusieurs modules de test comme une suite :

```
$ python3 -m unittest test_str

...
----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
```

Exercice - Tests unitaires de abs()

Écrire les tests unitaires pour la fonction `abs()`.

Note : L'exercice précédent propose de tester une méthode sans effet de bord (ce que l'on appelle également une fonction pure). Les tests sur ce type de méthodes sont faciles à écrire. Ils restent cependant l'exception lorsqu'on utilise la programmation orientée objet. En effet, l'appel d'une méthode sur un objet modifie le plus souvent son état et provoque généralement des effets de bord en sollicitant d'autres objets avec lesquels l'objet entretient des dépendances.

20.2 Tester des exceptions

Les tests unitaires automatisés sont utiles pour vérifier qu'une fonction (ou une méthode) produit le résultat attendu si l'appel est correct mais ils sont également très utiles pour vérifier qu'elle produit l'erreur attendue dans certains cas.

Pour tester qu'une exception survient, on peut utiliser la méthode `TestCase.assertRaises` conjointement avec une structure `with`.

```
import unittest


class AbsTest(unittest.TestCase):

    def test_abs_n_accepte_pas_une_chaine_de_caracteres(self):
        with self.assertRaises(TypeError):
            abs("a")
```

(suite sur la page suivante)

(suite de la page précédente)

```
if __name__ == '__main__':
    unittest.main()
```

```
$ python3 test_abs.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

20.3 Utilisation de doublure

Parfois, il est utile de contrôler l'environnement de test d'un objet ou d'une collaboration d'objets. Pour cela, on peut faire appel à des doublures qui vont se substituer lors des tests aux objets réellement utilisés lors de l'exécution de l'application dans un environnement de production.

Simulateur Un simulateur fournit une implémentation alternative d'un sous-système. Un simulateur remplace un sous-système qui n'est pas disponible pour l'environnement de test. Par exemple, on peut remplacer un système de base de données par une implémentation simplifiée en mémoire.

Fake object Un *fake object* permet de remplacer un sous-système dont il est difficile de garantir le comportement. Le comportement du *fake object* est défini par le test et est donc déterministe. Par exemple, si un objet dépend des informations retournées par un service Web, il est souhaitable de remplacer pour les tests l'implémentation du client par une implémentation qui retournera une réponse déterminée par le test lui-même.

Mock object Un objet *mock* est proche d'un *fake object* sauf qu'un objet *mock* est également capable de faire des assertions sur les méthodes qui sont appelées et les paramètres qui sont transmis à ces méthodes.

20.3.1 Utilisation d'un objet *mock*

Le module `unittest.mock` permet de créer facilement des objets *mock*. Ce module fournit les classes `Mock` et `MagicMock` (cette dernière est une extension `Mock` pour le support automatique des méthodes spéciales ou *dunders*). On peut programmer le comportement des méthodes d'un objet *mock* et vérifier ensuite qu'il a bien été utilisé comme attendu au cours du test.

Supposons que nous voulions tester la fonction suivante :

```
from pathlib import Path


def is_sourcefile(path):
    """Retourne True si le fichier est un fichier source Python"""
    if not path.is_file():
        raise Exception("Fichier indisponible")
    return path.suffix == ".py"
```

Nous pouvons utiliser un *mock* à la place du paramètre `path` pour contrôler l'appel de la méthode `is_file()` :

```
1  import unittest
2  from unittest.mock import Mock
3
4
5  class FonctionTest(unittest.TestCase):
6
7      def test_is_sourcefile_when_sourcefile(self):
8          path = Mock()
9          path.is_file.return_value = True
10         path.suffix = ".py"
11
12         resultat = is_sourcefile(path)
13
14         self.assertTrue(resultat)
15         path.is_file.assert_called()
16
17     def test_is_sourcefile_when_file_does_not_exist(self):
18         path = Mock()
19         path.is_file.return_value = False
20
21         with self.assertRaises(Exception):
22             is_sourcefile(path)
23
24         path.is_file.assert_called()
25
26     def test_is_sourcefile_when_not_expected_suffix(self):
27         path = Mock()
28         path.is_file.return_value = True
29         path.suffix = ".txt"
30
31         resultat = is_sourcefile(path)
32
33         self.assertFalse(resultat)
34         path.is_file.assert_called()
35
36
37 if __name__ == '__main__':
38     unittest.main()
```

À la ligne 8, nous créons un *mock* et nous pouvons prédéfinir la valeur qui sera retournée par la méthode `is_file()` ainsi que la valeur de l'attribut `suffix`.

Puis à la ligne 15, nous contrôlons que la méthode `is_file()` a bien était appelée pendant l'exécution du test.