Richard Gomez

Le petit Python

Aide-mémoire pour Python 3

2e édition

ellipses

ellipses

Références sciences

Le petit Python

Aide-mémoire pour Python 3

2e édition

Richard Gomez

ellipses

Collection Références sciences

dirigée par Paul de Laboulaye
paul.delaboulaye@editions-ellipses.fr

Retrouvez tous les livres de la collection et des extraits sur www.editions-ellipses.fr

ISBN 9782340-112926
Dépôt légal : mai 2026

www.editions-ellipses.fr

À Pepe Belinchón Cantarero (mi bisabuelo), Martina Martínez Roldán (mi bisabuela), Pilar Belinchón Martínez (mi abuela), Pedro Sáez Martínez (mi abuelo), Brígida Sáez Belinchón (mi madre), Juliana Sáez Belinchón (mi tía), Manolo Sáez Belinchón (mi tío), Pedrito Sáez Belinchón (mi tío), Federico Gómez Rosell (mi abuelo), Antonio Gómez Minaya (mi padre), David Gomez (mon frère), Didier Calas, Farid Belkalem, Ivan Revel, Ludovic Jacquet, les Reeds, Jean-Jacques Chervin, Stéphane Jammes, Mahfoud Chebboub, Stéphane Baseilhac, Miguel Angel Muñoz Fuentes, Alexandre Trijoulet, Ysé Commenges, Yanis Riguet et Estelle Chaumont.

À la mémoire de Rufino Jimenez Rosell, Pepa Belinchón Martínez, Bautista Villanueva Sáez, Cédric Darolles et Nicolas Guilloux.

À la mémoire de Leonardo Torres Quevedo (1852-1936).

Prologue

Prologue de la présente édition

Suite au succès de la première édition [2], Paul de Laboulaye, des éditions Ellipses, m'a proposé de rééditer *Le petit Python*. J'ai accepté et profité de cette occasion pour intégrer les nouveautés du langage (comme `match-case`, par exemple) et étoffer certaines parties, notamment celles consacrées aux bibliothèques, à l'encodage du texte et aux modules `math`, `cmath`, `fractions`, `decimal`, `random`, `os`, `os.path` et `turtle`. Nous avons également ajouté un chapitre consacré au module `statistics`.

Prologue corrigé de la première édition

Python est un langage de programmation (*langage de script*) permettant de pratiquer la programmation *impérative* (écriture d'une séquence d'instructions), la programmation *fonctionnelle* (résolution de problèmes par la création de fonctions) et la programmation *orientée objet* (définition d'objets que l'on fait interagir entre eux).

Ce langage a été créé en 1989 par Guido van Rossum et publié en 1991. Python est distribué sous une licence libre. On peut le télécharger gratuitement sur le site officiel (`http://www.python.org`) et l'installer sur n'importe quel ordinateur (quel que soit le système, y compris les tablettes et les calculatrices).

Python est le deuxième langage le plus utilisé au monde (classement RedMonk 2025). Le big data et l'intelligence artificielle l'ont massivement adopté. Il est largement employé par les géants du web comme Google, YouTube, Dropbox ou Instagram, et il alimente des technologies telle que le framework Django. Il a également été adopté par le milieu universitaire, les étudiants, la plupart des scientifiques – en particulier les mathématiciens – et par de nombreux systèmes éducatifs. Aujourd'hui, les lycéens français écrivent, dès la classe de seconde, des programmes en Python pour résoudre des problèmes en mathématiques et en physique.

Les qualités de Python sont nombreuses et séduisantes :

- une syntaxe élégante, lisible et intuitive ;
- une facilité d'apprentissage ;
- une évolution constante ;
- une certaine puissance ;
- et des milliers de bibliothèques couvrant tous les usages.

Face à une telle richesse et à de si nombreuses possibilités, le présent ouvrage propose un aide-mémoire permettant de retrouver rapidement et efficacement n'importe quelle instruction, objet ou élément de syntaxe du langage. Ce petit volume permet de maîtriser Python de manière avancée, sans avoir à faire de longues recherches sur Internet.

Nous présentons clairement et rigoureusement les notions fondamentales : les objets, les types, l'aliasing, les espaces de noms, la mutation, la résolution d'un identifiant, les modules, l'encodage des nombres, les conditions logiques, les boucles, les exceptions et leur traitement, les gestionnaires de contexte, les fonctions, les décorateurs, les booléens, les chaînes de caractères (encodage, formatage, etc.), les octets, les listes, les ensembles, les dictionnaires, les fichiers, les itérateurs, les générateurs, ainsi que les normes ASCII et UNICODE. Nous abordons également, en passant, l'auto-test, les docstrings, les doctests et les fonctions lambda. Nous détaillons les modules `math`, `cmath`, `fractions`, `decimal`, `random`, `statistics`, `os`, `os.path` et `turtle`. Enfin, nous consacrons un chapitre à l'introspection, c'est-à-dire à l'exploration du langage lui-même.

L'intérêt de cet ouvrage réside dans sa clarté et la profusion d'exemples : chaque notion est illustrée par un ou plusieurs cas concrets.

Le lecteur n'est pas obligé de lire les chapitres dans l'ordre, il peut se rendre directement à l'endroit où se trouve l'information recherchée. La table des matières est détaillée et conçue dans ce but (711 entrées). À la fin de l'ouvrage, on trouve un index pratique et riche de 752 entrées. Lorsqu'une notion apparaît dans plusieurs chapitres, des renvois clairs indiquent la section où poursuivre son exploration.

Le présent ouvrage repose sur Python version 3.13.15. Le lecteur pourra consulter les nouveautés (à venir) sur le site officiel, à l'adresse `https://docs.python.org/3/whatsnew/index.html`.

Dans ce texte, nous employons quelques abus d'écriture. Par exemple, pour indiquer que `x` est un objet de type `bytes`, nous écrivons parfois : `x` est un `bytes`. Le *s* peut surprendre alors que nous sommes au singulier, mais nous considérons le mot `bytes` comme un nom propre, c'est-à-dire le nom d'une classe d'objets. D'ailleurs, `x` est probablement composé non pas d'un seul byte, mais de plusieurs. On retrouve le même usage avec des phrases comme : `x` et `y` sont des `str`, pour signifier que `x` et `y` sont des objets de type `str`. Ici, `str` reste au singulier, car il s'agit du nom propre d'une classe. Un autre choix d'écriture consiste à couper une ligne de code lorsque celle-ci s'avère trop longue pour le format de ce livre. Néanmoins, la césure est faite de manière à ce que la lecture reste facile et naturelle. Autre point important : lorsqu'une référence renvoie à la section 6.5, par exemple, cela signifie la section 5 du chapitre 6. La référence 18.16.1, quant à elle, désigne la sous-section 1 de la section 16 du chapitre 18.

Le présent document est le fruit d'une longue compilation d'informations récoltées et mûries au fil des années, des besoins, et de divers projets de programmation. Je tiens à remercier chaleureusement mon ami et collègue Jocelyn Etienne, avec lequel j'ai pu échanger de nombreuses idées sur la programmation et sur Python en particulier, par le biais de conversations, de messages électroniques (parfois de SMS) et de défis passionnants lancés mutuellement. Je le remercie également pour toutes les remarques

qu'il a bien voulu formuler en lisant ces pages ainsi que pour les multiples erreurs et coquilles qu'il a relevées. Je remercie aussi mon ami et collègue Stéphane Lévy, avec lequel j'ai pu également échanger sur la programmation. Stéphane a lui aussi participé activement à la recherche des erreurs et des coquilles. Je tiens à citer ChatGPT avec lequel j'ai testé certaines idées de programmation. Je remercie Charles-Elie Liebart (informaticien) et Philippe Joyez (expert en TeXmacs) pour m'avoir aidé à utiliser correctement l'éditeur de texte ayant servi à la première édition de ce document. Pour la deuxième édition, je remercie Joris van der Hoeven (créateur de TeXmacs, [21]) et Grégoire Lecerf (expert TeXmacs) pour le temps qu'ils m'ont consacré. Sans eux, ce document n'existerait probablement pas sous cette forme. Je remercie ma compagne Estelle Chaumont pour son soutien et la patience dont elle a fait preuve au cours des mois ayant précédé la publication de ce livre. Je remercie également mes parents, Antoine et Brigitte, pour leur soutien. Enfin, je voudrais remercier Paul de Laboulaye, Corinne Baud et les éditions Ellipses pour avoir eu la gentillesse de s'intéresser à ce document et pour avoir accepté de le publier.

Ce livre a été composé avec TeXmacs 2.1.4 – éditeur de documents scientifiques, logiciel libre (GNU).

Table des matières

Chapitre 1
Lancer Python et éditer un programme

1.1 Télécharger Python

On télécharge Python sur le site officiel de la Python Software Foundation, `www.python.org` (c'est gratuit). Le téléchargement et l'installation ne durent pas plus de 5 minutes.

1.2 IDLE

Nous conseillons aux débutants de travailler avec IDLE PYTHON (*Integrated DeveLopment Environment*). Il s'agit de l'IDE (*integrated development environment*) incluse dans le Python téléchargé sur `www.python.org`. Elle est simple et efficace.

1.3 Fenêtre `Shell`

Quand on lance IDLE PYTHON, la fenêtre `Python Shell` s'ouvre :

```
Python 3.13.5 (v3.13.5:6cb20a219a8, Jun 11 2025, 12:23:45) [Clang
16.0.0 (clang-1600.0.26.6)] on darwin
Enter "help" below or click "Help" above for more information.
>>> |
```

Derrière l'invite de commande `>>>` (prompt) se trouve le curseur `|` qui attend nos instructions : nous nous trouvons dans un environnement interactif. Dans les versions précédentes, le texte d'accueil nous invitait à taper `help`, `copyright`, `credits` ou `license()`. Le lecteur testera lui-même ces suggestions. Il testera également un petit calcul en ligne. Nous tapons « `salut` » et nous validons avec la touche ENTRER :

```
>>> salut
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
```

```
    salut
NameError: name 'salut' is not defined
```

Python répond par un message d'erreur : pour lui, `salut` ne veut rien dire.

1.4 Versions

Pour connaître la version de Python utilisée par notre machine, on peut consulter la variable `version` du module `sys` :

```
>>> from sys import version
>>> version
'3.13.5 (v3.13.5:6cb20a219a8, Jun 11 2025, 12:23:45) [Clang 16.0.0
(clang-1600.0.26.6)]'
```

1.5 Éditer un script

Pour écrire un programme (un script) avec IDLE PYTHON :

`Menu : File / New File ⌘N`

Une fenêtre d'édition de texte « `untitled` » s'ouvre dans laquelle on peut écrire un programme (un code, un script). On travaille dans cette fenêtre d'édition comme dans n'importe quel éditeur de texte (édition, sauvergarde, etc.)

Pour exécuter un programme depuis la fenêtre d'édition :

`Menu : Run / Run Module F5`

ou plus simplement la touche F5.

Les affichages et les saisies (entrées, *inputs*) se font dans la fenêtre `Python Shell`.

Pour ouvrir un ancien script depuis la fenêtre `Python Shell` :

`Menu : File / Open...  ⌘O`

ou :

`Menu : File / Recent Files >`

1.6 Recommandations

On recommande d'écrire des lignes de code de 79 caractères maximum (recommandation PEP 8 [1]). Si une ligne est trop longue, on peut la casser avec un \. Python comprend qu'il ne s'agit pas d'un saut de ligne :

```
print("Les chansons de Reeds sont accessibles sur \
https://www.youtube.com/@davidreeds-s2x")
```

Pour que cela fonctionne, il ne faut pas d'espace après le \. Cette méthode n'est pas recommandée (PEP 8 [1]), on lui préfère celle-ci :

```
print("Les chansons de Reeds sont accessibles sur "
      "https://www.youtube.com/@davidreeds-s2x")
```

(l'indentation ci-dessus n'est pas obligatoire mais recommandée).

On peut également casser une ligne entre deux items dans une séquence (liste, uplet, dictionnaire, etc.) :

```
conway = [1, 11, 21, 1211, 111221, 312211, 13112221,
          1113213211, 31131211131221, 13211311123113112211]
```

On peut écrire plusieurs instructions sur la même ligne en les séparant avec un point-virgule, mais c'est déconseillé (PEP 8) car cela nuit à la lisibilité du script :

```
a = input(); x = 2 * a; print(x)
```

1.7 Commentaires

Il est conseillé d'écrire des commentaires dans nos programmes. On utilise pour cela le symbole # :

```
âge = input("Ton âge : ")     # Saisie de l'âge
âge = float(âge)              # conversion en nombre
année = 2025 - âge           # Calcul de l'année de naissance
print(année)                 # affichage de l'année
```

1.8 Shebang et encodage

Typiquement, l'en-tête d'un programme ressemble à ceci :

```
#!/usr/bin/env python3
#-*- coding: utf-8 -*-
"""
Ici, je tape la docstring (documentation) de mon programme.
La documentation peut être aussi longue que l'on veut.
```

```
La docstring est recommandée.
"""
```

La première ligne s'appelle *shebang* ou *hashbang*. Elle n'est pas obligatoire. Elle indique à un système d'exploitation UNIX, LINUX ou MAC OS qu'il faut appeler Python 3 pour interpréter le script, ce qui permet l'appel direct du programme en ligne de commande (depuis un terminal) sans avoir à écrire `python` devant. Elle est inutile si c'est un système WINDOWS qui exécute le script.

La deuxième ligne indique que le script est encodé selon la norme UTF-8 (voir chapitre 30). Elle n'est pas obligatoire. Cette ligne est de moins en moins utile car Python suppose par défaut que le script est encodé avec UTF-8 et la plupart des éditeurs utilisent cette norme. Si un jour on utilisait un éditeur fonctionnant avec une autre norme, on l'indiquerait dans l'entête.

1.9 Blocs et indentations

Le langage Python ne délimite pas les blocs d'instructions avec des balises `début`/`fin`, `begin`/`end` ou `{`/`}`. Les blocs sont tout simplement définis grâce à l'indentation. Python a choisi l'indentation pour améliorer la lisibilité des scripts.

La ligne qui précède un bloc d'instructions se termine toujours par un deux-points (`:`) comme ci-dessous :

```
def f(x):
    if x == 0: return 0     #
    else: return 1 / x      #
n = 2
if n > 0:
    print("Ceci")
    print("Cela")
else:
    print("Blabla")
for x in ["Tony", "Brigi", "David", "Richard"]:
    n = n + 1
    print("Étape :", n)
while n < 12:
    n = n + 1
    print(f(n))
```

Lorsqu'un bloc ne contient qu'une ligne, on peut l'écrire juste après le deux-points (sans aller à la ligne). C'est ce qu'on voit aux lignes marquées par un `#` dans l'exemple ci-dessus.

Cette série d'exemples n'est pas exhaustive : on aurait pu ajouter des `try`, `class`, `with`, `match`, etc.

On utilise l'instruction `pass` pour créer un bloc vide (un bloc qui ne fait rien) :

```
def f(x):
    pass         # à compléter plus tard
if x > 0:
    print("x est supérieur à 0")
else:
    pass         # à compléter plus tard
```

Attention. Dans un programme en langage Python, l'indentation (tabulation) fait partie du langage. Le non respect des règles d'indentation produit des messages d'erreur.

L'indentation est gérée automatiquement par IDLE (ou n'importe quel IDE servant à écrire du Python). Chaque indentation mesure 4 caractères (cadratins). On peut les générer avec la touche TAB. Le lecteur essaiera d'écrire le code ci-dessous :

```
"""
Étude d'une suite géométrique
"""
q = 0.3    # raison
u = 545    # premier terme u(0)
e = 0.01   # précision
n = 0
while abs(u) >= e:
    u = u * q
    n = n + 1
print("u(n) <", e, "à partir de n =", n)
```

1.10 Affichage dans le `Shell`

Quand on travaille sur le `shell` (en direct avec l'interpréteur), on n'a pas besoin de la fonction `print`. Au lieu d'écrire :

```
>>> print(2 + 3)
5
```

on se contente de :

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

Dans le `shell`, l'identifiant `_` (underscore) pointe vers la dernière donnée retournée :

```
>>> 2 * 3
6
>>> _ + 1
7
```

1.11 Utiliser un terminal

On peut lancer Python depuis un terminal. Sous Windows, on fait comme ceci :

```
C:\> python
```

(on peut aussi utiliser la commande `py`.) Sous MAC OS, c'est plutôt :

```
richard$ python3
```

car `python` tout court lançait Python 2 qui était installé par défaut jusqu'à récemment.

On se retrouve alors avec le même affichage que celui de la fenêtre `Shell` montré à la section 3. On dit que l'on se trouve dans l'environnement interactif de Python. Pour sortir de cet environnement, on appelle `exit` :

```
>>> exit()
```

On peut demander la version de Python utilisée depuis le terminal :

```
richard$ python3 --version
Python 3.13.5
```

On peut, depuis le terminal, exécuter une ligne de code Python sans entrer dans son environnement. Il suffit pour cela d'utiliser l'option `-c` :

```
richard$ python3 -c "print('Hello, World') ; print('Reeds')"
```

On peut faire en sorte qu'un programme Python se comporte comme une fonction depuis le terminal. Par exemple le programme `addition.py` ci-dessous :

```
from sys import argv
```

```
def somme(x, y): return x + y
x = float(argv[1])
y = float(argv[2])
print(somme(x, y))
```

est appelé de cette manière depuis le terminal :

```
richard$ python3 addition.py 23.65 100.1
```

On notera que

- tout argument de la commande `python3` est reçu sous forme de chaîne ;
- `argv[0]` reçoit le premier argument, c'est-à-dire `"addition.py"`.

Si on souhaite rester dans l'environnement de Python après l'exécution d'un programme, on utilise l'option `-i` :

```
richard$ python3 -i test.py
```

Si on fait du Python depuis le terminal, on peut modifier l'invite et les espaces d'indentation de l'environnement interactif. Normalement, on a ceci :

```
>>> for k in [0, 1, 2]:
    ....print(k)
```

mais après modification :

```
>>> import sys
>>> sys.ps1, sys.ps2 = "Richard> "," "
```

on obtient ceci :

```
Richard> for k in [0, 1, 2]:
             print(k)
```

1.12 Fonction `help`

Un appel `help(<objet>)` affiche une documentation sur l'objet passé en argument ou le type de cet objet, selon les cas. Un appel `help()` sans argument ouvre un environnement d'aide permettant de taper n'importe quel objet ou mot réservé. On invite le lecteur à utiliser cette fonction le plus possible :

```
>>> help(sum)
(...)
```

Chapitre 2
Mots réservés et objets internes

2.1 Mots réservés

Python possède 35 mots réservés :

```
>>> from keyword import kwlist
>>> kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await',
'break', 'class', 'continue', 'def', 'del', 'elif', 'else',
'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in',
'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return',
'try', 'while', 'with', 'yield']
```

On ne peut pas nommer un objet (une donnée) avec un mot réservé :

```
>>> for = 2300
SyntaxError: invalid syntax
```

2.2 Fonctions internes

Python possède 71 fonctions internes (*built-in functions*) :

```
abs       bin         callable    delattr
aiter     bool        chr         dict
all       breakpoint  clasmethod  dir
anext     bytearray   compile     divmod
any       bytes       complex
ascii

enumerate filter      getattr     hasattr
eval      float       globals     hash
exec      format                  help
```

```
          frozenset               hex

id        len         map         next
input     list        max
int       locals      memoryview
isinstance            min
issubclass
iter

object    pow         range       set
oct       print       repr        setattr
open      property    reversed    slice
ord                   round       sorted
                                  staticmethod
                                  str
                                  sum
                                  super

tuple     vars        zip         __import__
type
```

Pour appeler une fonction, on utilise des parenthèses à l'intérieur desquelles on insère éventuellement des arguments. Appelons, par exemple, la fonction `help` sans argument :

```
>>> help()
Welcome to Python 3.13's help utility!
If this is your first time using Python, you should
definitely check out the tutorial on the Internet at
https://docs.python.org/3.13/tutorial/ (...)
help>
```

L'invite `help>` attend notre question. Le lecteur tapera `for` et ENTRER pour obtenir de l'aide sur les *boucles for*, par exemple. Pour quitter l'environnement *help*, on tape `quit` et ENTRER :

```
help> quit
>>>
```

Appelons, par exemple, la fonction `abs` avec un argument :

```
>>> abs(-56)
56
```

Notes. 1) La plupart des fonctions internes sont des objets de type *builtin function or method*, mais il y a des exceptions : certaines d'entre elles sont des objets de type `type`. C'est le cas, par exemple, de `bool`, `bytearray`, `bytes`, etc. Cela ne change rien au fait que dans la pratique, ces objets sont utilisés comme des fonctions. Notons que `help` est de type *Helper*.

2) Les objets `quit`, `exit`, `copyright`, `license` et `credits` ne sont pas catalogués comme fonctions internes du langage bien qu'ils soient utilisés comme des fonctions. On notera d'ailleurs, que ces objets ne sont même pas de type *function*. Le lecteur affichera `type(exit)`, par exemple. Ces objets sont importés automatiquement depuis le module `site` dès le lancement de Python. Ils ont vocation à être appelés depuis le `Shell`. Le lecteur les testera lui-même. Curieusement, on peut appeler `copyright` et `credits` sans utiliser de parenthèses.

2.3 Module `builtins`

Le programme ci-dessous importe le module `builtins` pour en extraire les noms des fonctions internes présentes dans Python :

```
import builtins
def appelable(chaîne):
    attribut = getattr(builtins, chaîne)
    return callable(attribut)
def sans_underscore(chaîne): return chaîne[0] != "_"
def sans_capitales(chaîne): return chaîne.islower()
NOMS = dir(builtins)
fonctions_internes = [k for k in NOMS if appelable(k) and
                      sans_underscore(k) and sans_capitales(k)]
```

Ce programme se base sur une astuce : grosso modo, les fonctions internes de Python sont les attributs de `builtins` dont le nom ne commence pas par un underscore et ne contient que des lettres minuscules.

On lance ce programme et on affiche la récolte :

```
>>> fonctions_internes
['abs', 'aiter' 'all', 'anext', 'any', 'ascii', 'bin', 'bool',
'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classme-
thod', 'compile', 'complex', 'copyright', 'credits', 'delattr',
'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
'filter', 'float', 'format', 'frozenset', 'getattr', 'globals',
'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isins-
tance', 'issubclass', 'iter', 'len', 'license', 'list', 'locals',
'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr',
'reversed', 'round', 'set', 'setattr', 'slice', 'sorted', 'static-
method', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

On retrouve toutes les fonctions internes (sauf `__import__`) ainsi que les objets `copyright`, `credits`, `exit`, `license` et `quit` ! Pour un programme plus élaboré, on consultera le chapitre 31 et [3].

Il aurait été difficile de trier les attributs de `builtins` à la main, sachant qu'il y en a 160 :

```
>>> len(dir(builtins))
160
```

Note. En vérité, il n'y a pas besoin d'importer `builtins`. Le nom `__builtins__` pointe déjà vers ce module. On recommande néanmoins d'utiliser `builtins` pour faire de l'introspection.

```
>>> builtins is __builtins__
True
```

2.4 Méthodes internes

Python est un *langage orienté objet* : toutes les données qu'il traite sont des *objets*. Chaque objet possède des *attributs*. Parmi les attributs, il y a des *méthodes*. On consultera éventuellement le chapitre 1 de [3] pour se familiariser avec ces notions.

Pour lister les attributs d'un objets `x`, on appelle `dir(x)`. Pour connaître les attributs d'une chaîne de caractères, par exemple, on peut regarder `dir("blabla")`, ou directement `dir(str)`, sachant que `str` est le type des chaînes de caractères.

```
>>> attributs = dir(str)
>>> len(attributs)
81
```

La plupart des attributs d'un objet sont des méthodes appelées *méthodes internes* car elles sont définies en interne par Python. Parmi elles, il y a des *méthodes spéciales* (leurs noms commencent et finissent par deux underscores) et des *méthodes ordinaires* (leurs noms sont écrits en lettres minuscules).

Quand on débute, on n'utilise que les méthodes internes ordinaires. Les méthodes spéciales sont utilisées par l'interpréteur en interne ou par nous-mêmes quand on fait de la programmation orientée objet (surcharge).

L'appel d'une méthode suit une syntaxe précise : pour appeler la méthode `maméthode` de l'objet `monobjet` en faisant passer des arguments `x`, `y` et `z`, par exemple, on écrit :

```
monobjet.maméthode(x, y, z)
```

Exemple :

```
"Alexandre Trijoulet".count("r")
2
```

Nous donnons ci-dessous le nombre de méthodes internes (spéciales ou ordinaires) pour chaque type usuel :

Type	Méthodes spéciales	Méthodes ordinaires
`bool`	61	7
`int`	61	7
`float`	50	5
`complex`	38	1
`str`	32	47
`range`	28	2
`bytes`	34	42
`bytearray`	37	50
`list`	34	11
`tuple`	31	2
`set`	37	17
`frozenset`	34	8
`dict`	32	11
function	24	0
fichier binaire	32	20
fichier texte	31	17

Toutes ces méthodes, à l'instar des fonctions internes, enrichissent considérablement le langage. Nous laissons au lecteur le soin d'écrire un programme lui permettant de recalculer ce tableau. Il notera que certaines méthodes apparaissent dans plusieurs types différents.

2.5 Méthodes spéciales

Tous les objets possèdent des méthodes spéciales. Le nom d'une méthode spéciale commence et termine toujours par deux underscores.

Derrière chaque opération se cache une méthode spéciale et c'est cette dernière qui est appelée en coulisse pour effectuer l'opération demandée.

Nous parlons ici de toutes sortes d'opérations : construction d'objet, initialisation d'objet, destruction, dérivation d'une classe, calcul de l'espace des noms, représentations formelle et informelle, formatage, comparaison, opération d'arité 1, opération d'arité 2, calcul de la valeur booléenne d'un objet, accès à la valeur associée à une clé ou un indice, test d'appartenance, calcul de longueur, inversion, itération, hachage, appel, gestion de contexte, accès à un attribut, accès à un descripteur, introspection, etc.

Les méthodes spéciales n'intéressent que les programmeurs désirant les surcharger dans des classes, elles ne concernent donc que la programmation orientée objet. Toutes ces notions sont étudiées en détail dans [3]. Nous nous contenterons ici de trois exemples. Quand l'interpréteur de Python tombe sur « `x + y` », il exécute en interne ceci :

```
x.__add__(y)
```

Quand l'interpréteur tombe sur « `x < y` », il exécute ceci :

```
x.__lt__(y)
```

Quand l'interpréteur tombe sur « `a = x[y]` », il exécute ceci :

```
a = x.__getitem__(y)
```

Notons que les méthodes spéciales n'ont rien à voir avec les méthodes fournies par le module `operator`. En effet, même si ce dernier fournit les méthodes spéciales sous formes de fonctions, on ne mélangera pas les usages. On utilise une fonction de `operator` dans une fonction de haut niveau, comme par exemple ci-dessous :

```
from operator import itemgetter
famille = [("Tony", 80), ("David", 52),("Brigida", 77),
           ("Richard", 54)]
famille_ordonnée = sorted(famille, key=itemgetter(1))
```

Quand on envoie un objet `x` à la fonction `itemgetter(1)`, celle-ci retourne `x[1]`. Le programme ci-dessus crée une liste `famille_ordonnée` en classant les items de la liste `famille` par âge croissant, c'est-à-dire en ordonnant selon la valeur de `item[1]` pour chaque `item`. Dans ce contexte, on utilise une fonction de `operator` et non pas une méthode spéciale. L'exemple ci-dessous montre comment calculer le produit des items d'une liste avec la fonction `mul` de `operator` :

```
from operator import mul
from functools import reduce
liste = [2, 10, 5, 3]
produit = reduce(mul, liste)      # 2 * 10 * 5 * 3
```

Il existe de nombreuses fonctions de haut niveau : `map`, `filter`, `sorted`, `reduce`, etc.

2.6 Autres objets internes

Il y a d'autres objets internes comme `Ellipsis`, `NotImplemented`, `None`, etc. qui sont décrits en détail dans [3].

2.7 Introspection

Prenons une fonction interne quelconque, regardons comment Python la représente dans un affichage et examinons son type précis :

```
>>> abs
<built-in function abs>
>>> type(abs)
<class 'builtin_function_or_method'>
```

On constate que `abs` est de type *builtin function or method*. Ce n'est pas le cas de la fonction interne `float` :

```
>>> float
<class 'float'>
>>> type(float)
<class 'type'>
```

Examinons de la même manière une méthode interne ordinaire :

```
>>> "https://www.youtube.com/@davidreeds-s2x".index
<built-in method index of str object at 0x7fa94aea4330>
>>> type("https://www.youtube.com/@davidreeds-s2x".index)
<class 'builtin_function_or_method'>
```

Nous ne sommes pas étonnés de voir qu'une méthode interne ordinaire est de type *builtin function or method*. Examinons une méthode spéciale :

```
>>> x = 6
>>> x.__add__
<method-wrapper '__add__' of int object at 0x101173b60>
>>> type(x.__add__)
<class 'method-wrapper'>
>>> int.__add__
<slot wrapper '__add__' of 'int' objects>
>>> type(int.__add__)
<class 'wrapper_descriptor'>
```

On constate que *add* est un *wrapper descriptor* de `int` alors que la méthode *add* liée à un entier est un *method wrapper*.

Chapitre 3
Types offerts par Python

3.1 Objets

Chez Python, tout est objet. Toutes les données manipulées sont des objets. Les nombres, les chaînes, les listes, les fonctions et les exceptions (erreurs) sont des objets. Même le type d'un objet est un objet !

Tout objet possède une valeur, un type, des attributs, une identité et zéro ou plusieurs noms. Voici un exemple :

```
>>> liste = [1, 2, 3]
>>> id(liste)
140309299644288
>>> u = (0, 1, liste)
```

Nous venons de créer l'objet `<140309299644288>` et :

- la valeur de `<140309299644288>` est `[1, 2, 3]` ;
- le type de `<140309299644288>` est `list` ;
- les attributs de `<140309299644288>` sont `append`, `clear`, `copy`, etc.
- les identificateurs (noms) de `<140309299644288>` sont `liste` et `u[2]`.

Tout objet, quel que soit son niveau d'abstraction, peut être affiché grâce à la fonction `print`.

On notera que `print(x)` affiche la chaîne de caractère retournée par `str(x)`, tandis que « `x` suivi de ENTRER » dans le `Shell`, affiche la chaîne retournée par `repr(x)`, voir sections 1.10, 19.6 et 19.10 :

```
>>> liste
[1, 2, 3]
```

3.2 Types usuels

1. Objet de type `NoneType` (le rien)

```
>>> None
>>> print(None)
None
>>> type(None)
<class 'NoneType'>
```

2. Objets de type `bool` (booléens)

```
>>> True
True
>>> 2 > 5
False
>>> type(True)
<class 'bool'>
```

3. Objets de type `int` (entiers)

```
>>> -56
-56
>>> type(-56)
<class 'int'>
```

4. Objets de type `float` (nombres décimaux)

```
>>> 5.63
5.63
>>> type(5.63)
<class 'float'>
```

5. Objets de type `complex` (nombres complexes)

```
>>> 5 + 3j
(5+3j)
>>> type(5 + 3j)
<class 'complex'>
```

6. Objets de type `str` (chaînes de caractères)

```
>>> "Je suis content"
'Je suis content'
>>> print("Je suis content")
Je suis content
>>> type("Je suis content")
<class 'str'>
```

7. Objets de type `bytes` (chaînes d'octets)

```
>>> b"AaAaAAaBB"
b'AaAaAAaBB'
>>> type(b"AaAaAAaBB")
<class 'bytes'>
```

8. Objets de type `bytearray` (tableau d'octets)

```
>>> bytearray(b"AaAaAAaBB")
bytearray(b'AaAaAAaBB')
>>> type(bytearray(b"AaAaAAaBB"))
<class 'bytearray'>
```

9. Objets de type `range` (plages d'entiers)

```
>>> range(5, 10, 2)
range(5, 10, 2)
>>> type(range(5, 10, 2))
<class 'range'>
```

10. Objets de type `tuple` (uplets)

```
>>> (4, 0, 10, "a")
(4, 0, 10, 'a')
>>> type((4, 0, 10, "a"))
<class 'tuple'>
```

10. Objets de type `list` (listes)

```
>>> [4, 0, 10, "a"]
[4, 0, 10, 'a']
>>> type([4, 0, 10, 'a'])
<class 'list'>
```

11. Objets de type `set` (ensembles)

```
>>> {4, 0, 10, "a"}
{0, 10, 4, 'a'}
>>> type({4, 0, 10, "a"})
<class 'set'>
```

12. Objets de type `frozenset` (ensembles)

```
>>> frozenset({"Pedro", "Pilar"})
frozenset({'Pedro', 'Pilar'})
>>> type(frozenset({"Pedro", "Pilar"}))
<class 'frozenset'>
```

13. Objets de type `dict` (dictionnaires)

```
>>> {"Ivan": 74, "David": 73, "Richard": 71}
{'Ivan': 74, 'David': 73, 'Richard': 71}
>>> type({"Ivan": 74, "David": 73, "Richard": 71})
<class 'dict'>
```

14. Objets de type *function* (fonctions)

```
>>> def f(x, y): return x**2 + y**2
>>> f
<function f at 0x7f9c4dfc94c0>
>>> type(f)
<class 'function'>
>>> lambda x, y: x**2 + y**2
<function <lambda> at 0x7f9c4df350d0>
>>> type(lambda x, y: x**2 + y**2)
<class 'function'>
```

15. Objets de type *fichier texte*

```
>>> fichier = open("riri", "w")
>>> fichier.close()
>>> fichier
<_io.TextIOWrapper name='riri' mode='w' encoding='UTF-8'>
>>> type(fichier)
<class '_io.TextIOWrapper'>
```

16. Objets de type *fichier binaire*

```
>>> fichier = open("riri", "rb")
>>> fichier.close()
>>> fichier
<_io.BufferedReader name='riri'>
>>> type(fichier)
<class '_io.BufferedReader'>       # fichier bin-lecture
>>> fichier = open("riri", "wb")
>>> fichier.close()
```