Nicolas Bourgeois

3e édition

Python

Du grand débutant à la programmation objet

Cours et exercices corrigés

ellipses

Chapitre 1

Découvrir le langage

1.1 Installer et exécuter Python

Nous supposons que vous souhaitez commencer à programmer au plus vite sans vous préoccuper trop avant des questions d'environnement, c'est pourquoi nous commençons par quelques instructions qui pourront vous sembler un peu directives. Lorsque la question de choisir l'environnement de travail qui vous convient le mieux vous paraîtra pertinente, vous trouverez facilement sur le web un comparatif des différentes possibilités.

Avec un IDE

Si vous travaillez sur une machine pour laquelle vous disposez de droits d'administrateur, par exemple votre ordinateur personnel, nous vous recommandons d'installer successivement un interpréteur Python et un environnement de travail intégré (IDE).

Le premier est le minimum absolu pour pouvoir exécuter un script Python. Vous pouvez le télécharger sur le site de la *Python Software Foundation*. Les exemples de ce livre ont été vérifiés avec la version 3.12, mais nous vous recommandons d'installer la version stable la plus récente – il n'y aura pas de problème de rétrocompatibilité pour les versions 3.x entre elles.

Il est théoriquement possible de travailler avec le seul interpréteur et une console. Toutefois pour des raisons de confort nous recommandons d'installer également un IDE compatible avec votre système d'exploitation tel que *Thonny*, *Pyzo*, *PyCharm* ou encore *IDLE* – ce dernier étant fourni avec la distribution Python, mais pas nécessairement le plus facile à prendre en main. Vous disposerez ainsi d'une interface unique dans laquelle écrire votre code et lire les résultats, ainsi que d'un grand nombre d'autres fonctionnalités que vous découvrirez au fur et à mesure de votre progression.

Alternativement au fait d'installer sur votre machine l'interpréteur et l'IDE, vous avez la possibilité d'utiliser une interface en ligne, comme `pythonanywhere.com`, qui vous offre les fonctionnalités d'un IDE à distance.

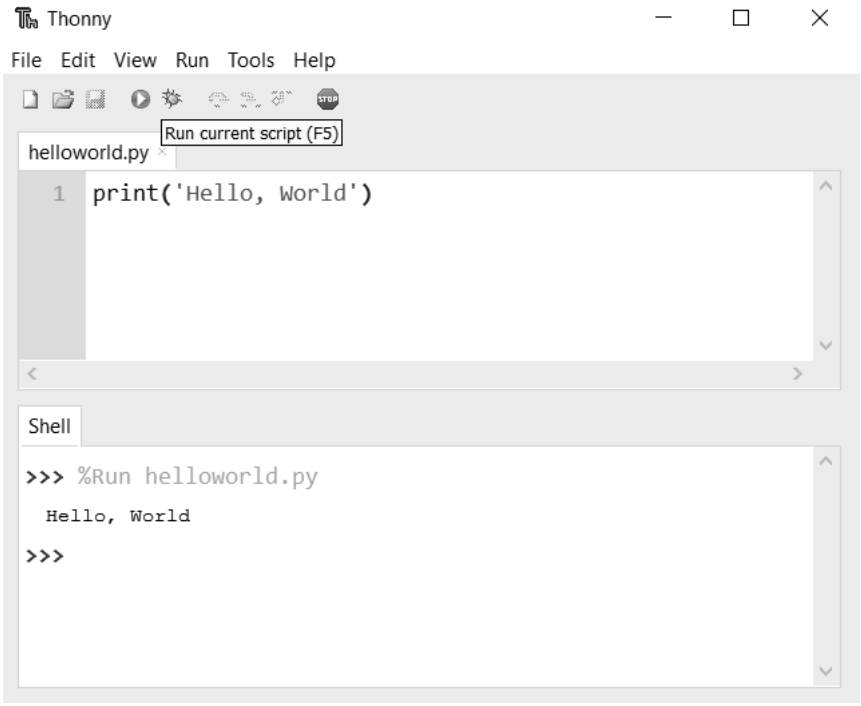

FIGURE 1.1 – Exemple d'interface (Thonny). Le code est écrit dans l'éditeur intégré et le résultat affiché dans la console (Shell) juste en-dessous.

Dans tout ce livre, nous ne ferons appel qu'à des librairies fournies dans la distribution de base, aussi nous n'aurons pas à nous préoccuper d'installer des modules complémentaires. Si vos centres d'intérêts vous amènent à avoir besoin de librairies externes, par exemple pour le calcul scientifique, vous pourrez vous référer à la documentation de *pip*.

Sans IDE

Si vous ne souhaitez pas ou ne pouvez pas installer d'IDE, par exemple parce que vous travaillez dans une salle informatique de votre université, c'est un tout petit peu plus compliqué, mais pas insurmontable. Il faut cependant au minimum qu'un interpréteur Python soit installé sur la machine [1].

Vous écrirez votre code dans un éditeur de texte quelconque, de préférence proposant une fonctionnalité de coloration syntaxique et ne générant pas de problème d'interprétation des caractères spéciaux. *Notepad++* sous Windows, *textwrangler*

1. Sachez en dernier recours que toute distribution Linux intègre un interpréteur Python, pas nécessairement la version la plus à jour cependant (typiquement, s'il s'agit d'une version de Python 2.x et non 3.x il vous faudra modifier un peu la syntaxe utilisée dans ce livre).

sous Mac OS ou encore *scribes*, *gedit* sous Linux sont de bons exemples. Vous enregistrerez ensuite votre fichier sous un nom de fichier portant l'extension .py. Ceci fait, vous ouvrirez un terminal et écrirez simplement `python xxx.py` où `xxx` est le nom de votre fichier, qui sera alors exécuté. Le retour sera affiché directement dans le terminal.

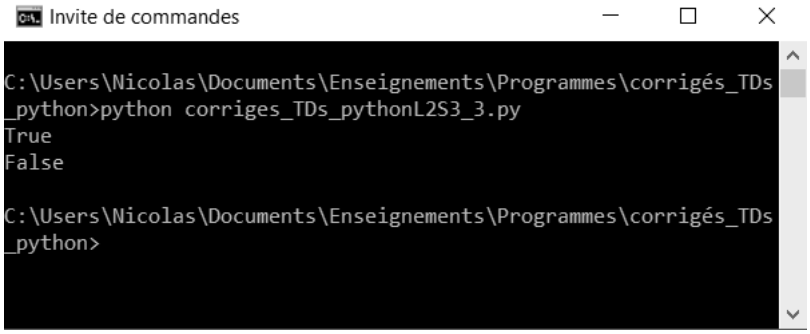

FIGURE 1.2 – Exemple de terminal (Windows).

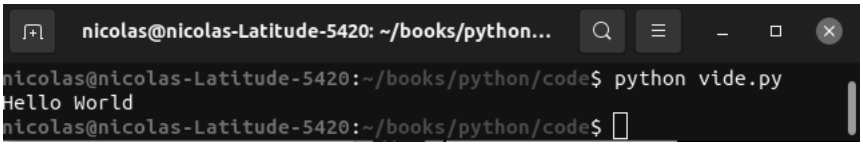

FIGURE 1.3 – Exemple de terminal (Ubuntu).

1.2 Utiliser Python comme calculatrice

Même dans l'hypothèse où vous n'avez aucune notion d'algorithmique, vous êtes certainement familier-ère avec le fonctionnement d'une calculatrice. Le moins qu'on puisse attendre de Python est qu'il vous permette d'en simuler une. Cela nous fournit l'occasion de notre premier script :

```
x = 3798
y = 48518
z = y * x
print(z)
```

Lequel produit dans la console le résultat suivant :

```
>>>
184271364
```

Les lignes une et deux affectent la valeur à deux variables, la troisième en effectue la multiplication, la quatrième utilise la fonction `print()` pour provoquer un affichage dans la console. Le même résultat aurait pu être obtenu en écrivant directement `print(3798 * 48518)`.

Si vous avez déjà programmé dans d'autres langages, vous remarquerez que la déclaration de variable est simplifiée à l'extrême : pas de mention de type ni d'allocation de mémoire, pas même de mot-clef indiquant la déclaration. Cela ne signifie pas pour autant que Python n'est pas un langage typé. Simplement, le type est résolu à partir du contexte.

```
x = 3798
y = 48518
z = y * x
print(type(z))
s = 'Hello World'
print(type(s))
```

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

Remarquez que Python attribue aux variables des types (ici `int` pour entier et `str` pour chaîne de caractères), lesquels peuvent être retrouvés à partir de la fonction `type()`. Ces types conditionnent le comportement des fonctions, opérateurs, etc. qui leur sont appliqués. Nous aurons amplement l'occasion de revenir sur cette notion au fur et à mesure de notre progression, mais voyons tout de suite un exemple très simple qui illustre combien il est important d'avoir cette idée de type en tête quand on programme.

```
x, y, z, t = 2, 1 / 3, '2', '1/3'
print(type(x), type(y), type(z), type(t))
print(x + y)
print(z + t)
```

```
>>>
<class 'int'> <class 'float'> <class 'str'> <class 'str'>
2.3333333333333335
21/3
```

À la ligne 3, nous effectuons une addition entre un entier et un décimal (`float`) et nous obtenons le résultat attendu – à part l'arrondi un peu bizarre. À la ligne 4, nous utilisons le même opérateur `+`, mais ici entre deux chaînes de caractères : le résultat obtenu est non pas une addition mais une concaténation. Si vous faites l'expérience, vous remarquerez enfin que tenter `x+z` provoque un affichage d'erreur :

```
>>>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Retenons donc de tout ceci une chose : un opérateur fonctionne différemment, voire pas du tout, selon le type des objets auxquels on l'applique. Il est donc important de distinguer le type des objets manipulés, par exemple l'entier `3` de la chaîne de caractères `'3'`.

Contrairement à certains autres langages comme PHP qui effectueraient une conversion implicite, Python explique qu'il ne sait pas comment interpréter cet opérateur entre deux objets pour lesquels le comportement prévu est radicalement différent : il préfère lancer une exception plutôt que de risquer la propagation d'un comportement non prévu. Toutefois lorsqu'un type est inclus dans un autre (`int` dans `float`), il effectue une conversion vers le type le plus général.

Pour le moment, nous n'avons utilisé que les opérations élémentaires, mais il est certain que Python peut également gérer des fonctions mathématiques plus complexes.

```
from math import *

print(sqrt(2), log(7.12), factorial(6))
```

```
>>>
1.4142135623730951 1.9629077254238845 720
```

Pas de surprise dans le comportement des fonctions racine carrée (`sqrt`), logarithme (`log`) et factorielle (`factorial`). Notez cependant la première ligne qui sert à importer ces fonctions depuis la librairie `math`, sans quoi le code renverrait un message d'erreur. Nous verrons un peu plus loin le fonctionnement des librairies, mais

l'idée ici est simple : seules quelques fonctions sont immédiatement disponibles pour Python, les autres doivent être explicitement importées depuis une librairie afin que l'interpréteur sache exactement ce qu'il doit faire.

Si la librairie `math` contient l'essentiel des fonctions mathématiques de base, elle ne saurait être exhaustive. Parfois vous aurez besoin de coder vous-même une fonction spécifique, parfois vous préfèrerez aller la chercher parmi le grand nombre de bibliothèques existantes sur internet. Si la seconde solution est souvent optimale en terme de temps de calcul et permet de limiter la taille de votre code, la première peut être plus pédagogique, plus modulable et vous faire économiser du temps de recherche. C'est un dilemme auquel vous serez souvent confronté-e.

Par exemple, si vous souhaitez calculer le nombre de combinaisons de p parmi n, il est plus économique de programmer vous-même :

```
from math import factorial

p, n = 7, 23
print(factorial(n) // (factorial(p) \
                       * factorial(n - p)))
```

```
>>>
245157
```

Exercice 1

Trouvez les solutions de $3x^2 - 7x = 23$.

Exercice 2

Sachant qu'il existe dans la librairie `math` une fonction `gcd(a,b)` qui renvoie le plus grand diviseur commun à `a` et `b`, trouvez la forme irréductible de la somme :

$$\frac{217}{440} + \frac{101}{256} + \frac{86}{71}$$

Pour clore cette première section, voici les opérateurs et fonctions mathématiques parmi les plus fréquemment utilisés.

Opérateurs sur les nombres

`+`	addition	`-`	soustraction	`*`	multiplication
`/`	division réelle	`//`	quotient de la division euclidienne	`%`	reste de la division euclidienne
`**`	puissance				

Fonctions mathématiques

`log`	logarithme	`exp`	exponentielle	`sqrt`	racine carrée
`floor`	partie entière	`gcd`	plus grand diviseur commun	`factorial`	factorielle
`cos`	cosinus			`abs`	valeur absolue

1.3 Structures de contrôle

Les conditions `if ... : else:`

Pour sortir de ce rôle de calculatrice et commencer à faire un peu d'algorithmique, nous allons maintenant introduire les conditions et les boucles. Afin de ne pas être trop théorique, commençons par un petit exemple :

```
s = input("Entrez un nombre pair : ")
n = int(s)
if n % 2 == 0:
    print(n // 2)
else:
    print("Le nombre n'est pas pair")
```

Si l'utilisateur entre un nombre pair, nous obtenons :

```
>>>
Entrez un nombre pair : 26
13
```

À l'inverse, s'il écrit un nombre impair :

```
>>>
Entrez un nombre pair : 33
Le nombre n'est pas pair
```

La première ligne du programme demande à l'utilisateur de rentrer un nombre ; le retour de l'utilisateur est toujours stocké *comme une chaîne de caractères* par Python, c'est pourquoi à la seconde ligne nous lui demandons explicitement de le convertir en entier. Si l'utilisateur avait rentré d'autres caractères, l'interpréteur aurait renvoyé un message d'erreur à ce moment.

L'instruction `if ...:` teste cette condition. Si elle est vraie, le bloc suivant est exécuté (ici la ligne 4), sinon il ne l'est pas et le bloc situé après le mot-clef `else:` est exécuté à la place (ici la ligne 6).

Ce qui détermine le bloc est l'indentation, c'est-à-dire l'espace situé au début de la ligne. Tous les éléments de même indentation font partie d'un même bloc, tandis qu'un retour à un niveau d'indentation inférieur indique que le bloc est terminé. Quant aux deux points (`:`) ils servent à indiquer la fin de la condition.

On peut mesurer le rôle de l'indentation avec l'exemple ci-dessous :

```
x = 3
if x > 5:
```

```
    x = x + 2
    x = x + 7
x = x + 1
print(x)
```

```
>>>
4
```

Comme la condition n'est pas vérifiée, les deux premières incrémentations, qui sont indentées et font donc partie du bloc conditionnel, sont ignorées. En revanche, la troisième, qui n'est pas indentée, ne fait pas partie de la condition et est donc exécutée indépendamment de celle-ci.

Là où l'indentation est dans beaucoup de langages une convention destinée à faciliter la lecture, elle est obligatoire en Python et se substitue aux accolades. Si cela peut sembler un peu contraignant, cela contribue à épurer la syntaxe, de même l'absence de points-virgules en fin d'instruction, puisque celles-ci sont simplement gérées par les retours à la ligne.

À noter que mot-clef `if` peut également être utilisé en ligne dans certains contextes, par exemple pour une affectation, comme dans l'exemple ci-dessous. C'est ce qu'on appelle parfois l'opérateur ternaire.

```
n = 67
suivant = n // 2 if n % 2 == 0 else 3 * n + 1
print(suivant)
```

```
>>>
202
```

La boucle `for ... in range()`

```
f = 1
for i in range(1, 11):
    f = f * i
print(f)
```