

# DEV1 - DEV1T

## Développement I

### (partie théorique)

# Python — une introduction à la programmation

**2023-2024**

R. Absil  
P. Bishop  
D. Boigelot  
A. Hallal  
C. Leignel  
S. Rexhep  
N. Richard



Haute École Bruxelles-Brabant  
École Supérieure d'Informatique  
Bachelier en Informatique  
rue Royale 67, 1000 Bruxelles  
+32 (0)2 219 15 46  
esi@he2b.be



# Introduction

Ce chapitre présente les objectifs et l'organisation du cours, illustre plusieurs points importants de prise en main du langage Python, comme l'utilisation du terminal, ainsi que diverses notions de terminologie en termes d'algorithmes, de programmes et d'erreurs de programmation.

## Objectifs du cours

L'objectif de ce cours est de donner des notions de programmations en langage Python aux étudiants. On y présente la syntaxe de base de ce langage, afin de permettre à l'étudiant de développer de premiers programmes simples. En particulier, l'objectif de ce cours n'est pas

- ▷ d'obtenir une connaissance pointue de l'écosystème Python afin de développer des programmes complexes ;
- ▷ de développer de bonnes connaissances en résolution de problèmes informatiques (algorithmique).

En particulier, on se contentera de détailler

1. la notion de variable, type, instruction, expression, opération, argument, etc. ;
2. la découpe d'un programme en fonctions et modules ;
3. les instructions conditionnelles et les boucles ;
4. le mécanisme d'itération sur des structures de données simples.

## Environnement de travail

Tous les systèmes d'exploitation<sup>1</sup> possèdent au moins une *console*, parfois appelé *terminal* ou *shell* permettant d'exécuter des commandes sans passer par l'interface graphique. Ces commandes peuvent être utilisées pour rendre un service spécifique à un utilisateur, comme compiler un programme.

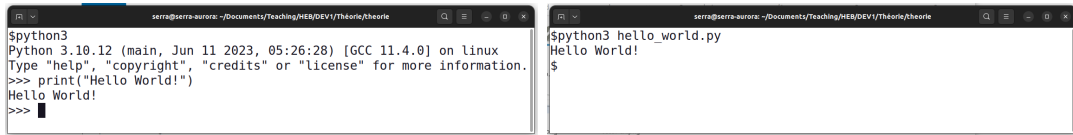
Dans le cadre de ce cours, on se servira du terminal principalement pour exécuter des programmes que l'on aura écrits nous-mêmes dans le langage Python. À ce titre, on suppose dans le cadre de ce cours que vous avez accès à un terminal, et que Python est installé et fonctionnel. Dans le cas contraire, vous pouvez vous référer au tutoriel de RealPython<sup>2</sup>.

---

1. Que l'on appelle parfois « OS », pour *Operating System*, en anglais.

2. <https://realpython.com/installing-python/> - Dernier accès le 31/07/2023.

On procédera soit en mode interactif, ou directement à partir d'un fichier, dans les deux cas à l'aide de la commande `python3`, tel qu'illustré à la figure 1. Le contenu du fichier `hello_world.py` est illustré dans le code 1. On remarque que le contenu de ce fichier est sensiblement identique à ce qui a été tapé en mode interactif, après le symbole `>>>`.



(a) Mode interactif

(b) Exécution à partir d'un fichier

FIGURE 1 – La commande `python3`

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

code/chap0/hello\_world.py

Code 1: Un programme « Hello world » en Python

Finalement, on sera souvent amené à suivre l'exécution de nos programmes pas à pas. C'est assez fastidieux de suivre cette évolutions « à la main ». Heureusement, il existe divers outils permettant de faire cela plus facilement.

Dans le cadre de ce cours, nous utiliserons l'outil en ligne<sup>3</sup> PythonTutor<sup>4</sup>. Un exemple d'utilisation de PythonTutor avec le code 2 est illustré à la figure 2. On peut utiliser cet outil pour simuler pas à pas l'exécution du programme, en l'exécutant ligne par ligne et en voyant directement quelles variables existent au sein du programme, ainsi que leurs valeurs à tout moment.

```
1 >>> x = 1
2 >>> y = 2
3 >>> print("x =", x, "and y =", y)
4 x = 1 and y = 2
5 >>> x = 3
6 >>> print("x =", x, "and y =", y)
7 x = 3 and y = 2
```

Code 2: Exemple de code à suivre

## Algorithmes et programmes

L'objectif de tout programme est de rendre service à un utilisateur ou un programmeur. Souvent, cela consiste en la résolution d'un problème particulier, comme répondre à la question « comment savoir si un nombre est pair ? ». Pour résoudre un problème, on est amené à écrire un *algorithme*, que l'on traduira

3. Vous ne devez donc rien installer, mais uniquement utiliser votre navigateur pour aller sur la page web.

4. <https://pythontutor.com/python-debugger.html#mode=edit> - Dernier accès le 17 mai 2022.

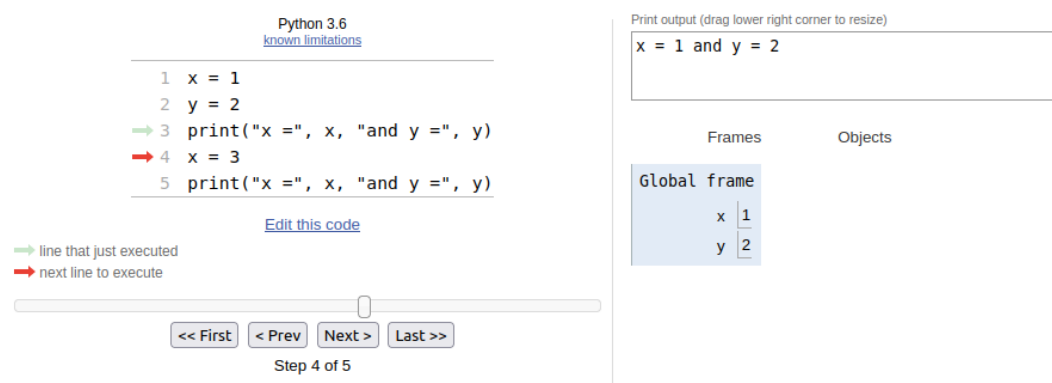


FIGURE 2 – Évolution des variables avec PythonTutor

dans un *langage de programmation* afin d'écrire un *programme*.

### Définition

Un *algorithme* est une séquence d'étapes précises et non ambiguës pouvant être exécutées pour résoudre un problème.

La première étape dans l'élaboration d'un algorithme est la caractérisation

1. des *entrées* (qu'a-t-on besoin pour résoudre le problème ?) ;
2. des *sorties* (que doit-on fournir comme solution ?).

Dans le cas du problème « comment savoir si un nombre est pair ? », on peut procéder de la manière suivante :

- ▷ entrée :  $x$ , un nombre *entier*,
- ▷ sortie : vrai ou faux.

On peut ainsi écrire un algorithme très simple, illustré à la figure 3. Pour tester la parité d'un nombre, on examine le reste de la division entière par 2 avec l'opération modulo : s'il est nul, alors le nombre est divisible 2, et est donc pair. Notez que l'on a dû utiliser une certaine forme de logique mathématique (sémantique) pour concevoir cet algorithme, et que, parfois, cette logique peut être particulièrement complexe. Ce cours n'étant pas dédié à de l'algorithmique, on ne se focalisera pas sur ce point. On remarque que la syntaxe de cet exemple est très proche du langage courant.

---

**Entrée(s) :**  $x$ , un nombre entier

**Sortie(s) :** vrai si  $x$  est pair, faux sinon

- 1: **Si**  $x \bmod 2 = 0$  **alors**
  - 2:     **Retourner** vrai
  - 3: **Sinon**
  - 4:     **Retourner** faux
- 

FIGURE 3 – Un algorithme permettant de savoir si un nombre est pair

Une fois un tel algorithme conçu, on peut l'implémenter en un *programme*.

### Définition

Un *programme* est une séquence d'instructions dans un langage de programmation donné qui spécifie comment réaliser un calcul ou une tâche.

On peut implémenter l'algorithme de test de parité d'un nombre dans le langage de programmation Python tel qu'illustré au code 3.

```
code/chap0/even.py
1 def is_even(x):
2     if x % 2 == 0:
3         return True
4     else:
5         return False
6
7 print(is_even(1))
8 print(is_even(2))
```

Code 3: Test de parité

On remarque plusieurs spécificités liées à l'utilisation d'un langage de programmation : la syntaxe est *très* stricte. Par exemple, on remarque que

- ▷ certaines lignes doivent se terminer par « : »,
- ▷ l'opération modulo est notée « % » et le test d'égalité par « == »,
- ▷ les littéraux vrai et faux sont notés en anglais, et commencent par une majuscule.

On ne détaillera pas ce genre de spécificités dans cette section : elles seront présentées dans la suite de ce document, l'objectif de ce cours étant d'introduire ces divers éléments de syntaxe afin d'avoir une première expérience de programmation élémentaire,

On remarque également sur ce code qu'on a ajouté deux lignes permettant de savoir si les nombres 1 et 2 sont pairs. Ces lignes permettent de *tester l'exactitude* du programme, une pratique courante en programmation, qui permet généralement de détecter de potentielles erreurs de logique ou de programmation.

## Erreurs de programmation

Les développeurs commettent souvent des erreurs de programmation. Il existe trois types d'erreurs :

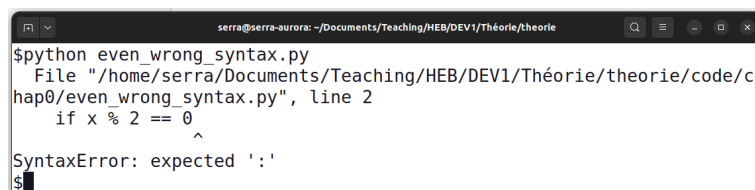
1. les erreurs *syntactiques*,
2. les erreurs d'*exécution*,
3. les erreurs *sémantiques*.

Les erreurs syntaxiques sont des erreurs où la grammaire du langage n'a pas été respectée, c'est-à-dire où l'on n'a pas suivi les règles qui régissent l'écriture d'un programme. L'une de ces règles est, par exemple, « toute instruction commençant par **if** doit se terminer par **:** ». Ces erreurs sont systématiquement signalées par l'interpréteur lors de l'analyse qu'il effectue pour exécuter le code.

Le code 4 est une variante du code 3 et y introduit une erreur de syntaxe. On remarque qu'il manque un caractère ':' à la ligne 2. Lorsque l'on lance l'interpréteur sur ce code, le terminal signale une erreur plutôt que d'exécuter le code, tel qu'illustré à la figure 4. Dans le cas présent, le compilateur signale clairement quelle est l'erreur, et où elle se trouve, ce qui permet de la corriger facilement. Parfois, ça ne sera pas le cas, dépendant du type d'erreur et de la capacité de l'interpréteur à en détecter l'origine. Il faut alors investiguer pour la corriger.

```
code/chap0/even_wrong_syntax.py
1 def is_even(x):
2     if x % 2 == 0
3         return True
4     else:
5         return False
6
7 print(is_even(1))
8 print(is_even(2))
```

Code 4: Une erreur de syntaxe



```
serra@serra-aurora: ~/Documents/Teaching/HEB/DEV1/Théorie/theorie
$python even_wrong_syntax.py
File "/home/serra/Documents/Teaching/HEB/DEV1/Théorie/theorie/code/c
hap0/even_wrong_syntax.py", line 2
    if x % 2 == 0
               ^
SyntaxError: expected ':'
$
```

FIGURE 4 – L'interpréteur détecte l'erreur de syntaxe

Les erreurs d'exécution sont des erreurs de programmation qui vont mettre fin à l'exécution de manière brutale et provoquer un crash. Par exemple, un programme pourrait

- ▷ tenter d'ouvrir un fichier sur lequel l'utilisateur n'a pas les droits d'accès ;
- ▷ perdre la connexion réseau lors d'un transfert de données ;
- ▷ tenter de lire des informations en mémoire dans une zone à laquelle il n'a pas accès.

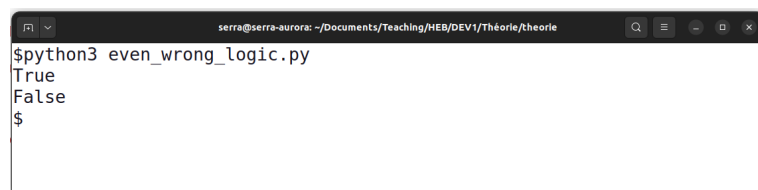
Les erreurs de sémantique (ou de logique) sont des erreurs de raisonnement, où le programme ne produit pas le résultat attendu. Elles apparaissent donc quand le programme est syntaxiquement correct, mais quand la logique utilisée dans l'algorithme que l'on a implémenté est incorrecte.

Par exemple, dans le code 3, on a testé qu'un nombre était pair en examinant le reste de la division par 2 : s'il est nul, le nombre est pair. On aurait pu être inattentif et écrire le contraire : s'il *n'est pas* nul, alors le nombre est pair. C'est évidemment une erreur, incluse dans le code 5.

Dans ce cas, on remarque que l'interpréteur ne signale pas l'erreur : c'est au programmeur de la détecter (la sortie du programme illustrée à la figure 5 affirme que 1 est pair et que 2 ne l'est pas) et de trouver son origine pour pouvoir la corriger.

```
1 def is_even(x):
2     if x % 2 != 0:
3         return True
4     else:
5         return False
6
7 print(is_even(1))
8 print(is_even(2))
```

Code 5: Une erreur de sémantique



```
serra@serra-aurora: ~/Documents/Teaching/HEB/DEV1/Théorie/theorie
$python3 even_wrong_logic.py
True
False
$
```

FIGURE 5 – L’interpréteur ne détecte pas l’erreur sémantique



Ce chapitre a pour objectif de donner une idée générale des possibilités offertes par Python, avant de s'y intéresser en détail.

## 1.1 Variables et types

Tout programme a besoin de mémoire, classiquement pour effectuer des calculs ou pour stocker leur résultat. Habituellement, on procède par le biais de *variables*. Une variable est un alias pour une valeur (c'est une autre façon d'appeler un objet). On la crée avec une *affectation*, et on peut ensuite manipuler cet objet par l'intermédiaire de cette variable, via son nom, tel qu'illustré à la figure 6.

```
>>> x = 2
>>> y = x + 1
>>> print(x, y)
2
3
>>> x = 4
>>> print(x, y)
4
3
```

Code 6: Exemple de variable

Sur ce code, on remarque que quand on affecte à nouveau `x` à `4`,

1. l'ancienne valeur de `x` est *écrasée* : `x` vaut maintenant `4` et plus `3` ;
2. la valeur de `y` n'a pas changé (sa valeur a été affectée avant la réaffectation de `x`).

Notons qu'en Python, on n'a pas besoin de préciser le *type* de cette variable (si c'est un entier, une chaîne de caractères, etc.). Il existe plusieurs types de variables, par exemple :

- ▷ les entiers, comme `2`,
- ▷ les flottants, comme `2.0`,
- ▷ les booléens, comme `True` et `False`,
- ▷ les chaînes de caractères, comme `"2"`,

ainsi que des types plus complexes que les tuples, listes, etc., qui seront vus ultérieurement.

## 1.2 Expressions

En pratique, on va faire plus de travail avec les variables que les affecter et en imprimer le contenu. Souvent, on va les combiner à l'aide d'*opérateurs* au sein d'*expressions* pour effectuer des calculs plus complexes. On a déjà vu l'expression `y = x + 1` : dans cette expressions, `+` est un opérateur, et on affecte `y` à `x + 1`.

Souvent, on va également utiliser des *fonctions* déjà programmées pour nous : ce sont de petits bouts de codes réutilisables, comme `sqrt` pour calculer la racine carrée. Pour utiliser une fonction, il faut classiquement l'*importer*.

Les expressions et l'importation de fonctions est illustrée au code 7.

```
1 >>> import math
2 >>> x1, y1 = 1, 2
3 >>> x2, y2 = 2, 3
4 >>> dist = math.sqrt((x1 - x2)**2 + (y1 - y2)**2)
5 >>> print(dist)
6 1.4142135623730951
```

Code 7: Exemple d'importation et d'expression

L'exécution de ce code se produit comme suit :

**ligne 1** on importe le module `math` afin de pouvoir utiliser les fonctions qui y sont définies, comme `sqrt` ;

**ligne 2** on définit deux variables `x1` et `y1`, et on affecte `x1` à 1 et `y1` à 2 ;

**ligne 3** on définit deux variables `x2` et `y2`, et on affecte `x2` à 2 et `y2` à 3 ;

**ligne 4** on définit une variable `dist` pour calculer la distance entre le point (`x1`, `y1`) et (`x2`, `y2`)

- ▷ on veut utiliser la formule  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$  : il faut la traduire en Python ;
- ▷ l'opérateur `**` signifie puissance, donc `**2` signifie « mettre au carré » ;
- ▷ on utilise la fonction `math.sqrt` pour invoquer la racine carrée ;
- ▷ on stocke le résultat du calcul dans `dist` ;

**ligne 5** on imprime la valeur de `dist`.

En pratique, il existe des opérateurs dédiés pour les types usuels :

- ▷ entiers : `+`, `-`, `*`, `**` (puissance), `//` (division entière), `%` (modulo) ;
- ▷ flottants : `+`, `-`, `*`, `/`, `**` ;
- ▷ chaînes de caractères : `+` (concaténation), `*` (concaténation multiples) ;
- ▷ booléens : `not`, `and`, `or` ;
- ▷ comparaison : `>`, `>=`, etc.

Quelques uns de ces opérateurs sont illustrés au code 8.

```

>>> print(2 + 3)
5
>>> print(7 % 2)
1
>>> print(7 / 2)
3.5
>>> print(7 // 2)
3
>>> print("Hello " + "World!")
Hello World!
>>> print("Ha" * 3)
HaHaHa
>>> print(True and False)
False
>>> print(2 <= 3)
True

```

Code 8: Exemple de variable

### 1.2.1 Conversions

Parfois, il est nécessaire d'effectuer des *conversions* d'un type vers un autre, par exemple lors de la lecture d'une entrée au clavier. Ces conversions doivent être demandées explicitement, grâce à une fonction de conversion, tel qu'illustré au code 9.

```

1 >>> s = input()
2 2
3 >>> print(s + 3)
4 Traceback (most recent call last):
5   File "<stdin>", line 1, in <module>
6   TypeError: can only concatenate str (not "int") to str
7 >>> print(s + "3")
8 23
9 >>> i = int(s)
10 >>> print(i + 3)
11 5
12 >>> f = float(s)
13 >>> print(f + 3.0)
14 5.0
15 >>> print(f + 3)
16 5.0

```

Code 9: Exemple de conversions

L'exécution de ce code se produit comme suit :

**ligne 1** on demande une entrée à l'utilisateur :

- ▷ il la tape au clavier (il tape « 2 ») et termine par « Enter » ;
- ▷ le résultat est stocké dans `s` sous la forme d'une chaîne de caractères ;

**ligne 3** on essaye d'imprimer le résultat de `s + 3`

- ▷ `s` est de type « chaîne de caractères » et `3` de type entier ;

- ▷ il n'existe pas d'opérateur `+` entre les chaînes de caractères et les entiers ;
- ▷ Python affiche une erreur ;

**ligne 7** on concatène `s` à la chaîne de caractères `"3"`, et on affiche le résultat ;

**ligne 9** on convertit `s` en entier à l'aide de la fonction `int`, et on stocke le résultat dans `i` ;

**ligne 10** on additionne `i` à l'entier `3`, et on affiche le résultat ;

**ligne 12** on convertit `s` en flottant à l'aide de la fonction `float`, et on stocke le résultat dans `f` ;

**ligne 13** on additionne `f` au flottant `3.0`, et on affiche le résultat ;

**ligne 15** on additionne `f` à l'entier `3`, et on affiche le résultat (flottant).

Notons que ce dernier cas où l'on applique l'opérateur `+` entre un entier et un flottant est le seul du genre on l'on peut l'appliquer entre deux opérandes de types différents.

## 1.3 Structures de données

Tout langage offre au programmeur des *structures de données*, qui lui permettent d'y stocker des objets en suivant une certaine logique. En Python, citons les tuples, les listes, les ensembles et les dictionnaires.

### Tuples et chaînes de caractères

Les tuples sont une séquence ordonnée de taille fixe. On peut y accéder en lecture<sup>1</sup> grâce à l'opérateur `[]`. Le code 10 illustre un exemple de tuple.

```
>>> t = (4, 5, 6)
>>> print(t[0])
4
>>> t[0] = 8
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> print(len(t))
3
```

Code 10: Utilisation des tuples

Sur ce code, on remarque que

- ▷ l'indice du premier élément (la position où il se trouve dans le tuple) est `0` ;
- ▷ l'indice du dernier élément est `len(t) - 1` ;
- ▷ tenter de modifier le tuple provoque une erreur.

---

1. Les tuples sont immuables : une fois créés, ils ne peuvent pas être modifiés.

Les chaînes de caractères se comportent de la même manière que les tuples (accès, longueur, lecture seule), tel qu'illustré au code 11. Pour les tuples comme les chaînes de caractères, on peut tester la présence d'un objet grâce à l'opérateur `in`.

```
>>> s = "tis but a scratch!"
>>> print(s[len(s) - 1])
!
>>> print("a" in s)
True
```

Code 11: Utilisation des chaînes de caractères

### 1.3.1 Listes

Les listes diffèrent des tuples sur un point essentiel : on peut y ajouter et supprimer des éléments<sup>2</sup>. Le code 12 illustre un exemple d'utilisation de listes.

```
>>> need = ["clothes", "boots", "motorcycle"]
>>> need.append("black sun glasses")
>>> print(need)
['clothes', 'boots', 'motorcycle', 'black sun glasses']
>>> need[1] = "leather boots"
>>> need.remove("clothes")
>>> del need[2]
>>> print(need)
['leather boots', 'motorcycle']
```

Code 12: Utilisation des listes

Sur ce code, on remarque que

- ▷ on peut ajouter des éléments en fin de liste avec `append` ;
- ▷ on peut supprimer la première occurrence d'un élément avec `remove` ;
- ▷ on peut supprimer un élément<sup>3</sup> à un indice donné avec `del`.

### 1.3.2 Ensembles

Les ensembles se comportent comme les ensembles mathématiques : ce sont des collections d'objets distincts, dont l'ordre n'importe pas pour le programmeur. Le code 13 illustre l'utilisation d'ensembles.

Sur ce code, on remarque que

- ▷ les doublons sont automatiquement supprimés à la construction d'un ensemble ;
- ▷ on peut utiliser les opérateurs ensemblistes mathématiques pour calculer la différence (`-`), l'union (`+`), l'intersection (`&`) et la différence symétrique d'ensembles (`^`).

2. ... et ceci a un coût en performances : favoriser systématiquement les listes plutôt que les tuples est donc une mauvaise pratique.

3. Ou une séquence complète de la liste.

```
>>> s = {1, 2, 3, 3, 4}
>>> print(s)
{1, 2, 3, 4}
>>> print(5 in s)
False
>>> a = set("abracadabra")
>>> b = set("alacazam")
>>> print(a)
{'a', 'r', 'b', 'c', 'd'}
>>> print(a - b)
{'r', 'd', 'b'}
>>> print(a | b)
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> print(a & b)
{'a', 'c'}
>>> print(a ^ b)
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Code 13: Utilisation des ensembles

### 1.3.3 Dictionnaires

Les dictionnaires sont des structures de données associatives, c'est-à-dire des ensembles de paires « clé / valeur » (on associe une valeur à une clé). Le code 14 illustre un exemple de dictionnaire.

```
>>> dct = {"alpha" : 42, "romeo" : 23 }
>>> print("romeo" in dct)
True
>>> print(dct["romeo"])
23
>>> dct["zulu"] = 12
>>> print(dct)
{'alpha': 42, 'romeo': 23, 'zulu': 12}
```

Code 14: Utilisation des dictionnaires

Sur ce code, on remarque que

- ▷ les valeurs sont accédées via leur clé, avec `[]` ;
- ▷ on peut tester la présence d'une clé avec l'opérateur `in` ;
- ▷ on peut ajouter des paires de clés / valeurs avec `[]`.

## 1.4 Définition de fonctions

L'une des caractéristiques de l'informatique est de décomposer les problèmes complexes en problèmes plus simples, dans une stratégie « divide and conquer ». Chaque sous-problème est ensuite résolu, et l'ensemble des solutions est utilisée pour résoudre le problème original.

L'une des illustrations de cette stratégie est l'utilisation de fonctions : plutôt que d'écrire un programme d'un seul bloc, on va le diviser en composants plus

ou moins indépendants. L'un de ces composants est la *fonction* : c'est une petite routine réutilisable que l'on peut appeler, comme « calcule-moi l'aire d'un cercle de rayon 2 ».

Par exemple, si l'on a besoin au sein d'un programme de calculer plusieurs fois une telle aire, plutôt que d'écrire plusieurs fois l'instruction

```
area = 3.141592 * radius * radius
```

on va l'écrire une fois et lui donner un nom pour pouvoir la réutiliser plus tard, autant de fois que l'on en a besoin.

Le code 15 illustre la définition et l'utilisation de fonctions.

```
>>> import math
>>> def circle_area(r):
...     area = math.pi * r * r #compute the area
...     return area
...
>>> print(circle_area(2))
12.566370614359172
>>> print(circle_area(3))
28.274333882308138
>>> print(circle_area(4))
50.26548245743669
```

Code 15: Utilisation de fonctions

Sur ce code, on remarque

- ▷ qu'on a *importé* le *module* `math`, dont on a besoin pour récupérer la valeur de  $\pi$  ;
- ▷ qu'on a défini une fonction à l'aide du mot-clé `def` ;
- ▷ que les instructions propres à la fonction sont *indentées* (décalées vers la droite) ;
- ▷ que l'on peut *commenter* du code avec `#` pour le rendre plus lisible ;
- ▷ qu'une fonction peut prendre des paramètres (ici, `r`) ;
- ▷ qu'une fonction peut retourner le résultat de son calcul (ici, elle retourne `area`) ;
- ▷ que pour appeler une fonction, il suffit de l'appeler en utilisant son nom et en fournissant ses paramètres.

## 1.5 Alternatives

Souvent, lorsqu'un programme se complexifie, on est amené à devoir effectuer des traitements alternatifs, c'est à dire demander au programme « si une propriété est vérifiée, alors fais ceci, sinon fais autre chose ».

En Python, ces traitements différés sont effectués à l'aide de blocs `if`, `elif` et `else`, tel qu'illustré au code 16.

Sur ce code, on remarque

```

>>> def print_sign(x):
...     if x > 0:
...         print("Positive")
...     elif x < 0:
...         print("Negative")
...     else:
...         print("Zero")
...
>>> print_sign(1)
Positive
>>> print_sign(-2)
Negative
>>> print_sign(0)
Zero

```

Code 16: Utilisation de if

- ▷ qu’au même titre que les fonctions, les instructions d’un `if` sont indentées,
- ▷ que la condition d’un `if` est toujours de type booléen.

## 1.6 Boucles

Souvent, dans un programme, on est amené à répéter l’exécution d’instructions un nombre arbitraire de fois. Pour cela, on utilise des *boucles*. Il existe deux types de boucles en Python :

- ▷ la boucle `for`, pour répéter des instructions un certain nombre de fois ;
- ▷ la boucle `while`, pour répéter des instructions tant qu’une condition est satisfaite.

Les codes 17 et 18 illustrent ces deux types de boucles :

- ▷ la fonction `print_ds` du code 17 parcourt une structure accessible avec `[]` et en imprime le contenu pas à pas ;
- ▷ la fonction `count_factors` du code 18 cherche à calculer le nombre de facteurs<sup>4</sup> égaux à `f` de `n`, et pour cela, elle divise `n` par `f` autant de fois que cela est possible et compte le nombre de fois où elle a effectué ces instructions.

Sur ces codes, on remarque

- ▷ qu’au même titre que les fonctions et les `if`, les instructions des boucles sont indentées,
- ▷ qu’on peut obtenir un intervalle de nombres grâce à `range` (dans ce cas, de zéro inclusivement jusqu’à `len(ds)` exclusivement).

---

4. Par exemple,  $40 = 2 \times 2 \times 2 \times 5$ , dont il y a 3 fois le facteur 2 dans 40.



```
>>> def print_ds(ds):
...     for i in range(len(ds)):
...         print(ds[i])
...
>>> t = (1, 2)
>>> print_ds(t)
1
2
>>> l = [1, 2, 3]
>>> print_ds(l)
1
2
3
```

Code 17: Utilisation de `for`

```
>>> def count_factors(n, f):
...     count = 0
...     while n % f == 0:
...         n = n / f
...         count = count + 1
...     return count
...
>>> print(count_factors(16, 2))
4
>>> print(count_factors(40, 2))
3
```

Code 18: Utilisation de `while`



# Concepts de base

Ce chapitre a pour objectif d'introduire plusieurs notions de base en programmation, telles que les variables, les valeurs, les expressions, etc.

Intuitivement, une variable est un alias pour une valeur d'un certain type, que l'on a construite à l'aide d'une affectation, tel qu'illustré à la figure 2.1.

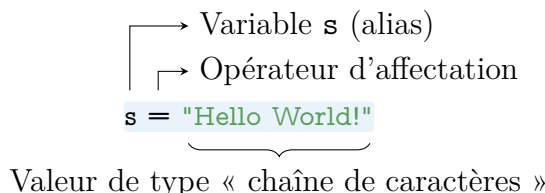


FIGURE 2.1 – Variables, valeur et affectation

Étant donné le caractère élémentaire de ce chapitre, l'intégralité du code présenté le sera en mode interactif, sans l'utilisation de fichiers tel que présenté au précédent chapitre.

## 2.1 Types et valeurs

Une *valeur* est un élément de base de tout programme. Elles permettent d'y introduire des éléments concrets, comme 42 ou "Hello World!".

En Python, ainsi que dans la grande majorité des langages, tout symbole que l'on manipule a un *type*, en particulier les valeurs.

Par exemple,

- ▷ 1 et 2 sont des entiers : Python dénote le type comme `int` ;
- ▷ 1.0 et 2.0 sont des nombres à virgule flottante<sup>1</sup> : Python dénote le type comme `float` ;
- ▷ "Hello World!" est une chaîne de caractères : Python dénote le type comme `str` ;
- ▷ `True` et `False` sont des booléens : Python dénote le type comme `bool`.

1. Parfois, on dit plus simplement que ce sont des *flottants*.

Les types sont utilisés dans les langages de programmation pour définir la taille en mémoire nécessaire pour une allocation mémoire<sup>2</sup>, pour le codage en binaire d'une valeur<sup>3</sup>, ou pour simplement dénoter le contexte dans lequel le symbole peut être utilisé.

On peut connaître le type d'un symbole grâce à la fonction<sup>4</sup> `type`, tel qu'illustré au code 19.

```
>>> type(2)
<class 'int'>
>>> type("Hello World")
<class 'str'>
>>> type(3.141592654)
<class 'float'>
>>> type(True)
<class 'bool'>
>>> type(print)
<class 'builtin_function_or_method'>
>>> type(int)
<class 'type'>
>>>
```

Code 19: Utilisation de la fonction `type`

Sur ce code, on remarque déjà des particularités syntaxiques :

- ▷ les nombres à virgule flottante sont écrits en utilisant un point « . » comme séparateur décimal plutôt qu'une virgule « , ». On écrit donc en Python 3.1415 plutôt que 3,1415.
- ▷ les chaînes de caractères doivent être entourées des guillemets anglais « " » (ou de l'apostrophe).

Ces différences vont permettre à l'interpréteur de *déduire* le type du symbole considéré, comme illustré au code 20.

## 2.2 Variables

### Définition

Une *variable* est un nom symbolique permettant de faire référence à une valeur et de la stocker en mémoire grâce à une *affectation*.

Ainsi, en quelque sorte, les variables sont des alias vers des espaces mémoires.

---

2. Lorsque l'on veut stocker « quelque chose » en mémoire, il faut réserver de la place. On appelle cela une *allocation mémoire*.

3. En informatique, toute information est éventuellement encodée en binaire sous la forme de suite de bits « 0 » ou « 1 ».

4. `type` est une fonction, ce qui signifie que l'on peut l'appeler (avec des paramètres potentiels) et récupérer le résultat de son exécution. Cette fonction retourne simplement le type d'un symbole. On a vu une autre fonction : `print`, qui permet d'effectuer des affichages. On verra plus de détails quant à l'utilisation des fonctions au chapitre 3.

```
>>> type(2)
<class 'int'>
>>> type(2.)
<class 'float'>
>>> type(2.0)
<class 'float'>
>>> type("2")
<class 'str'>
>>>
```

Code 20: Différentes écritures changent le type

Comme les valeurs, les variables ont un type, que l'on peut également connaître à l'aide de la fonction `type`.

La syntaxe d'une affectation est naturelle : `nom = valeur`. On peut par exemple écrire `i = 42`.

Une fois assignée, on peut retrouver la valeur d'une variable grâce à son nom. De là, on peut par exemple l'afficher à l'aide la fonction `print`, tel qu'illustré au code 21. Notons que dans ce code, on a également utilisé `print` pour afficher plusieurs variables en une seule fois.

```
>>> i = 42
>>> print(i)
42
>>> msg = "The answer to life, the universe and everything is"
>>> print(msg, i)
The answer to life, the universe and everything is 42
>>>
```

Code 21: Affichage de variables en console

Si l'on essaie d'utiliser une variable que l'on n'a pas préalablement définie (c'est-à-dire qui n'existe pas), l'interpréteur détecte une erreur syntaxique tel qu'illustré au code 22. Bien que le début du message d'erreur soit un peu mystérieux, la dernière ligne est explicite : le nom « `j` » n'est pas défini.

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

Code 22: Affichage de variables en console

En pratique, les valeurs des variables au sein d'un programme évoluent en fonction des besoins du programmeur. Pour cela, on a besoin de mettre à jour ces valeurs, et l'on procède également avec l'affectation. L'ancienne valeur est alors écrasée : le programme a « écrit par dessus », elle n'est plus disponible. Ce comportement est présenté au code 23, et illustré à la figure 2.2.

```
>>> i = 2
>>> print(i)
2
>>> i = 5
>>> print(i)
5
>>>
```

Code 23: Mise à jour des valeurs des variables



FIGURE 2.2 – Illustration simplifiée de la mémoire après affectations successives

En pratique, une affectation modifie la variable située à gauche du signe d'égalité, et lui donne la valeur de l'expression à sa droite. Le code 24, par exemple, effectue les opérations suivantes, dans cet ordre :

1. on définit la variable `a` comme étant égale à 2,
2. on définit la variable `b` comme étant égale à 5,
3. on définit la variable `b` comme étant égale à `a`,
4. on met à jour la variable `a` en lui donnant la valeur 3.

```
1 >>> a = 2
2 >>> b = 5
3 >>> print("a =", a, "and b =", b)
4 a = 2 and b = 5
5 >>> b = a
6 >>> print("a =", a, "and b =", b)
7 a = 2 and b = 2
8 >>> a = 3
9 >>> print("a =", a, "and b =", b)
10 a = 3 and b = 2
11 >>>
```

Code 24: L'affectation a lieu « une seule fois »

On pourrait s'attendre naturellement à avoir `a = 3` et `b = 3`, toutefois, l'interpréteur affirme que `b = 2`. C'est le comportement du langage<sup>5</sup> : la valeur de `b` a été assignée une unique fois, à la 2<sup>e</sup> ligne. Elle n'est donc pas mise à jour par la ligne 3. Ce comportement est illustré à la figure 2.3.

### 2.2.1 Variables temporaires

Parfois, on a besoin de créer des variables qu'à but temporaire, pour stocker le résultat d'un calcul ou de se souvenir d'une valeur que l'on est sur le point

5. Parfois, on dit que l'affectation fonctionne *par valeur* : elle effectue une copie, ce qui explique l'absence de mises à jours ultérieures. On verra plus de détails à ce sujet à la section 3.3 du chapitre 3, dédié aux fonctions et au passage d'arguments.

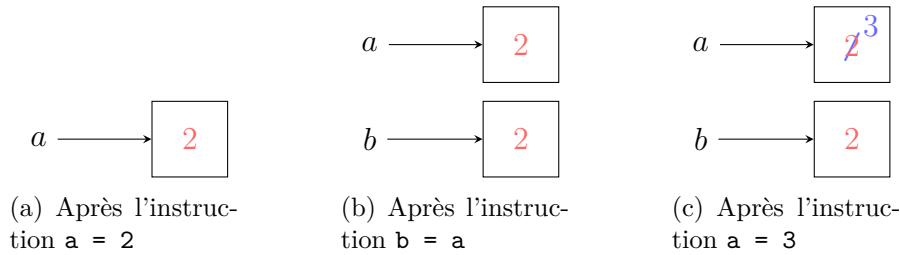


FIGURE 2.3 – Illustration simplifiée de la mémoire après affectations successives

d'écraser. On qualifie ces variables de *temporaires*.

Par exemple, comment peut-on échanger les valeurs de deux variables ? Si  $x = 1$  et  $y = 2$ , après l'échange, on souhaite que  $x = 2$  et  $y = 1$ . À l'évidence, le code 25 ne fonctionne pas :

1. la ligne 1 affecte la valeur 1 à `x` : `x` vaut donc 1,
2. la ligne 2 affecte la valeur 2 à `y` : `x` vaut donc 1, et `y` vaut 2,
3. la ligne 3 affecte la valeur `y` à `x` : `x` vaut donc 2, et `y` vaut 2 (la valeur de `x` a été écrasée),
4. la ligne 4 affecte la valeur `x` à `y` : `x` vaut donc 2, et `y` vaut 2.

Cet échange ne fonctionne donc pas : il s'agit d'une erreur sémantique. Ce comportement est illustré à la figure 2.4.

```

1 >>> x = 1
2 >>> y = 2
3 >>> x = y
4 >>> y = x
5 >>> print("x =", x, "and y =", y)
6 x = 2 and y = 2
7 >>>

```

Code 25: Mauvais échange de valeurs

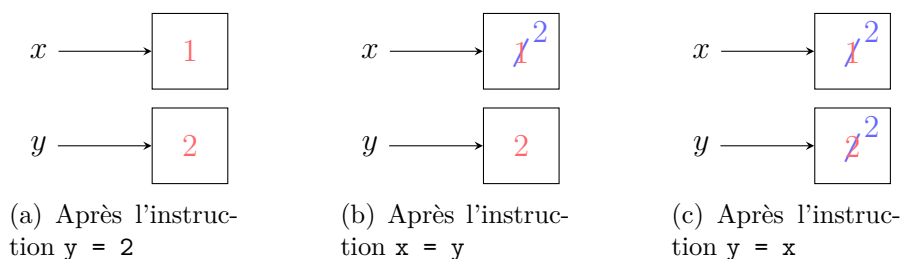


FIGURE 2.4 – Illustration simplifiée de la mémoire après un mauvais échange

Pour résoudre le problème, on a besoin d'une troisième variable, que l'on peut par exemple nommer `tmp`, tel qu'illustré au code 26 :

1. la ligne 1 affecte la valeur 1 à `x` : `x` vaut donc 1,
2. la ligne 2 affecte la valeur 2 à `y` : `x` vaut donc 1, et `y` vaut 2,
3. la ligne 3 affecte la valeur `x` à `tmp` : `x` vaut donc 1, `y` vaut 2, et `tmp` vaut 1,

4. la ligne 4 affecte la valeur `y` à `x` : `x` vaut donc 2, `y` vaut 2, et `tmp` vaut 1,
5. la ligne 5 affecte la valeur `tmp` à `y` : `x` vaut donc 2, `y` vaut 1, et `tmp` vaut 1,

ce qui est bien le résultat attendu de l'échange. Ce comportement est illustré à la figure 2.5.

```

1 >>> x = 1
2 >>> y = 2
3 >>> tmp = x
4 >>> x = y
5 >>> y = tmp
6 >>> print("x =", x, "and y =", y)
7 x = 2 and y = 1
8 >>>

```

Code 26: Bon échange de valeurs

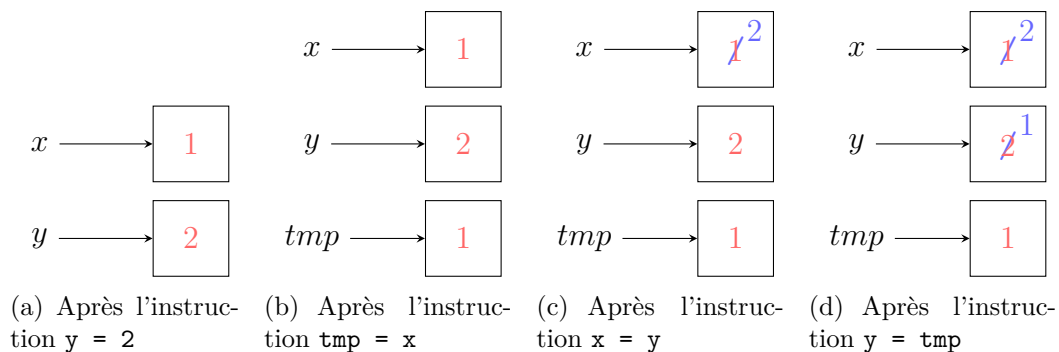


FIGURE 2.5 – Illustration simplifiée de la mémoire après un bon échange

L'évolution des variables avec PythonTutor est illustrée à la figure 2.6.

### 2.2.2 Contraintes de nommage et conventions

On a vu la syntaxe de base d'une affectation (`nom = valeur`) ainsi que son fonctionnement. Toutefois, on ne peut pas donner n'importe quel nom à une variable. Par exemple, aucune des affectations ci-dessous ne fonctionne :

- ▷ `2i = 3`,
- ▷ `class = 3`,
- ▷ `/i = 3`.

Toutes induisent une erreur de syntaxe clairement indiquée par l'interpréteur.

#### Contrainte

En pratique, le nom d'une variable *doit*

- ▷ commencer par une lettre ou le caractère « `_` » (underscore),
- ▷ ne contenir que des lettres (majuscules ou minuscules), des chiffres et le caractère « `_` ».



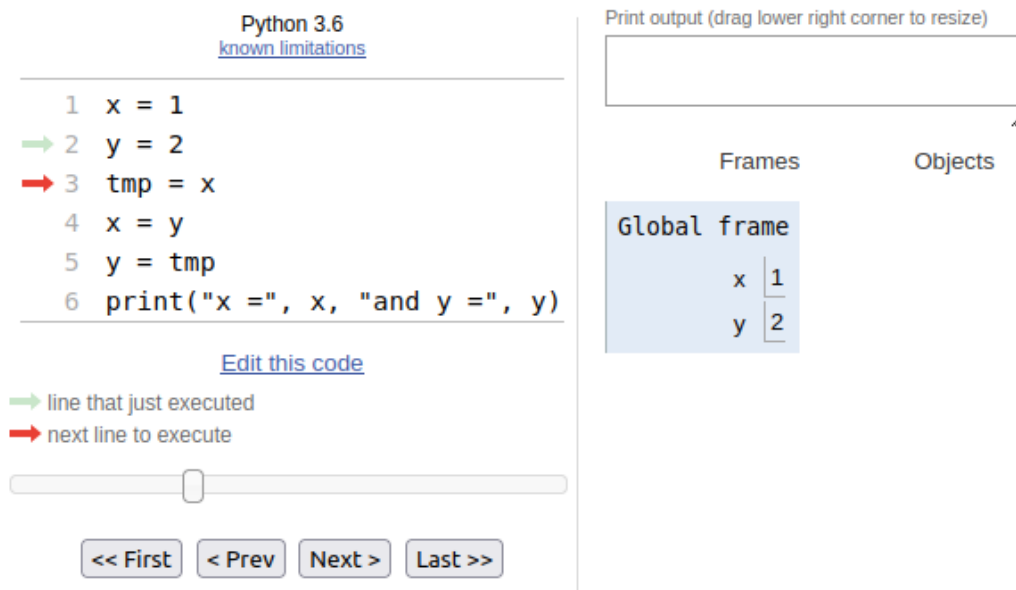


FIGURE 2.6 – Évolution des variables avec PythonTutor

De plus, le nom des variables ne peut être l'un des mots-clés du langage Python : ils lui sont réservés. La grammaire du langage définit 33 mots-clés, énumérés à la table 2.1.

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

TABLE 2.1 – Liste des mots-clés du langage Python

Notons également que Python est un langage sensible à la casse : les majuscules ne sont pas considérées de la même manière que des minuscules. Le code 27 illustre par exemple une erreur de ce type : on essaie d'afficher une variable `Pi` alors que l'on a défini une variable `pi`.

```
>>> pi = 3.141492654
>>> print(Pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Pi' is not defined. Did you mean: 'pi'?
```

Code 27: Erreur due à la casse

Finalement, il existe des *conventions de nommage*, c'est-à-dire une forme de sagesse à utiliser lorsque l'on nomme ses variables. Dans tout langage de programmation, on souhaite qu'une variable possède un nom qui indique clairement sa fonction. Ainsi, cela a du sens de nommer une variable comme `pi = 3.141592654`,

mais pas comme `truc = 3.141592654`. Dans le code 26, on avait introduit une variable `tmp`, nommée comme tel pour signifier que c'est une variable temporaire dont on ne va rapidement plus avoir besoin. En pratique, on requerra également que le nom des variables commence par une lettre minuscule.

## 2.3 Opérateurs et expressions

Pour l'instant, les instructions que l'on a fournies à l'interpréteur étaient très simples. En pratique, les algorithmes que l'on implémentera requerront une plus grande complexité.

Pour cette raison, Python fournit divers *opérateurs*, comme l'addition, la soustraction, etc., représentés par des symboles spéciaux, comme `+`, `-`, ... Leurs *opérandes* (les valeurs à droite et à gauche de l'opérateur) sont limités à des valeurs de certains types : par exemple, cela a du sens d'écrire `8 / 2`, mais pas `"Hello" / "World"`. Une combinaison d'opérateurs et leurs opérandes est appelée une *expression*, comme `(2 + 3) * 4`. Le code 28 illustre divers exemples d'expressions. Les opérateurs qui y sont utilisés seront détaillés dans les sections suivantes.

```
>>> pi = 3.141592654
>>> r = 3
>>> area = pi * r * r
>>> circ = 2 * pi * r
>>> print("A circle of radius", r, "has area", area, "and circumference", circ)
A circle of radius 3 has area 28.274333886 and circumference 18.849555924
>>> x = 1
>>> y = 2
>>> import math
>>> inside = math.sqrt((x * x) + (y * y)) < r
>>> print("Point (", x, ",", y, ") inside circle ?", inside)
Point ( 1 , 2 ) inside circle ? True
```

Code 28: Exemples d'expressions

Sur ce code, on a défini diverses expressions relatives aux cercles, comme le calcul de la surface, de la circonférence, etc. On remarque également que certaines opérations, comme le calcul de la racine carrée avec `sqrt`, ne sont pas « disponibles par défaut » : il faut les *importer*. On verra plus de détails à ce sujet au chapitre 4. Finalement, on remarque que les expressions, comme les variables et valeurs, peuvent avoir différents types :

- ▷ l'expressions `area = pi * r * r` est de type flottant ;
- ▷ l'expression `inside = math.sqrt((x * x) + (y * y)) < r` est de type booléen.

Dans le cadre de ce cours, on se limitera aux opérateurs arithmétiques (les opérateurs usuels mathématiques), aux opérateurs booléens (pour effectuer des comparaisons) et aux opérateurs de chaînes de caractères.

### 2.3.1 Opérateurs arithmétiques

La table 2.2 fournit une liste exhaustive des opérateurs arithmétiques et illustre leur utilisation.

Opérateur	Fonction	Exemple	Résultat
+	Addition	5 + 2	7
-	Soustraction	5 - 2	3
*	Multiplication	5 * 2	10
/	Division	5 / 2	2.5
//	Division entière	5 // 2	2
**	Exposant	5 ** 2	25
%	Modulo	5 % 2	1

TABLE 2.2 – Les opérateurs arithmétiques

On remarque que les opérateurs +, -, \*, / et \*\* se comportent comme attendu. Les opérateurs // et % sont quant à des des opérateurs classiquement associés aux entiers<sup>6</sup> et sont associés à la *division euclidienne* :

- ▷ en école primaire, on a appris à faire du calcul écrit permettant d'effectuer des divisions, par exemple « 61 divisé par 3 » ;
- ▷ on trouve « 20 », et il « reste 1 » que l'on ne peut plus diviser ;
- ▷ on peut donc écrire  $61 = 3 \times 20 + 1$  ;
- ▷ on appelle 20 le *quotient* et 1 le *reste* ;
- ▷ l'opérateur // permet de trouver ce 20, et % le 1 : et c'est pour cela qu'on les nomme « division entière » et « modulo »<sup>7</sup>.

Un exemple de ce division euclidienne est illustrée au code 29.

```
>>> n = 61
>>> d = 3
>>> q = n // d
>>> r = n % d
>>> print("n =", n, "=", d, "*", q, "+", r)
n = 61 = 3 * 20 + 1
```

Code 29: Division euclidienne

### Erreurs numériques

Quelle que soit la norme utilisée<sup>8</sup> pour représenter les nombres à virgule flottante, il existe des nombres que l'on ne saura pas représenter. En effet, les nombres *réels* ont potentiellement une quantité infinie de décimales qui ne se répètent pas. Si on voulait les représenter avec exactitude, on aurait besoin d'une quantité infinie de mémoire, ce que l'on n'a à l'évidence pas.

6. Même si, en pratique, on peut également les utiliser sur des flottants.

7. Bon, d'accord, on aurait pu appeler l'opérateur % « reste », mais pourquoi faire simple quand on peut terroriser les étudiants ☹ ?

8. Classiquement, les langages de programmation utilisent la norme hardware qui est souvent IEEE 754 (signe, exposant, mantisse).

Pour cette raison, le résultat fourni par certains opérateurs peut être entaché d’erreurs, que l’on qualifie d’*erreurs numériques*. Par exemple, le nombre 0,1 n’est pas représentable avec la norme IEEE 754, ce qui peut amener à des résultats « étranges », tel qu’illustré au code 30.

```
>>> print(0.1)
0.1
>>> print(2.5 - 2.4)
0.10000000000000009
>>> print(1 - 0.9)
0.09999999999999998
```

Code 30: Exemples d’erreurs numériques

On aurait pu s’attendre à ce que les trois expressions impriment la même valeur. Ce n’est pas le cas : l’arithmétique flottante est *très* complexe. Pour ces raisons, il faut parfois prendre des précautions particulières quand on effectue des opérations en virgule flottante, tel qu’un test d’égalité, comme illustré au code 31.

```
>>> i = 2.5 - 2.4
>>> j = 1 - 0.9
>>> print(i == j)
False
>>> print(abs(i - j) < 0.0000000001)
True
```

Code 31: Exemples d’erreurs numériques

Comme l’on pouvait s’en douter, tester si  $i = j$  avec l’expression `i == j` est erroné : les codages en mémoire de `i` et `j` *ne sont pas* identiques. Toutefois, on peut sémantiquement affirmer « deux nombres à virgule sont égaux s’ils sont suffisamment proches ». On peut définir « suffisamment proches » comme « à moins de 0,0000000001 » l’un de l’autre, c’est à dire tester si  $|i - j| < 0,0000000001$ . C’est ce qu’on a fait avec l’expression `print(abs(i - j) < 0.0000000001)`, qui fournit un résultat sémantiquement correct<sup>9</sup>.

Les précautions à prendre pour que la prise en charge des erreurs numériques soit cohérente au sein d’un programme vont dépendre des applications, on ne peut donc pas définir une stratégie universelle. Cette gestion des erreurs numériques dans des calculs arbitrairement complexes est une discipline à part entière, et n’est pas l’objet de ce cours : on s’en tiendra donc à cette courte introduction.

### 2.3.2 Opérateurs logiques et de comparaison

On a déjà vu dans les sections précédentes certains opérateurs permettant de comparer des valeurs, tels que `>` et `==`. Ces opérateurs retournent une valeur booléenne, ce qui signifie par exemple que l’expression `1 > 2` est de type booléen.

Une expression booléenne peut avoir deux valeurs

---

9. L’appel `abs(i - j)` calcule la valeur absolue de `i - j`.

- ▷ `True` pour les valeurs vraies ;
- ▷ `False` pour les valeurs fausses.

Pour rappel, la casse est importante, les valeurs `true` et `True` sont donc syntaxiquement incorrectes.

On utilise les expressions booléennes classiquement de deux façons :

- ▷ soit en manipulant des expressions booléennes directement avec les opérateurs logiques « et », « ou », etc. ;
- ▷ soit en effectuant des comparaisons arithmétiques.

### Opérateurs logiques

Il existe trois opérateurs logiques en Python :

- ▷ `and`, pour la conjonction (le « et ») ;
- ▷ `or`, pour la disjonction (le « ou ») ;
- ▷ `not`, pour la négation (le « non »).

Ces opérateurs se comportent de la même manière que ceux que vous avez vu dans votre cours de mathématiques discrètes. Leurs tables de vérité respectives sont rappelées en fin de chapitre, à la figure 2.7 (page 27). Le code 32 illustre le fonctionnement de ces opérateurs.

```
>>> p = True
>>> q = not p
>>> print(q)
False
>>> print(p and q)
False
>>> print(p or q)
True
```

Code 32: Illustration des opérateurs logiques

### Opérateurs de comparaison

Il est également possible d'effectuer des comparaisons entre les nombres en Python à l'aide d'opérateurs. Il en existe six :

- ▷ `x == y` pour tester si  $x$  et  $y$  sont égaux ;
- ▷ `x != y` pour tester si  $x \neq y$  ;
- ▷ `x < y` pour tester si  $x < y$  ;
- ▷ `x <= y` pour tester si  $x \leq y$  ;
- ▷ `x > y` pour tester si  $x > y$  ;
- ▷ `x >= y` pour tester si  $x \geq y$ .

Remarquez que l'expression `x = y` n'est pas la même que l'expression `x == y` :

- ▷ `x = y` est une affectation : elle affecte la valeur de  $y$  à  $x$  ;

▷ `x == y` est une comparaison : elle teste si  $x = y$ .

Le code 33 illustre un exemple d'utilisation du fonctionnement de ces opérateurs logiques.

```

1 >>> n = 42
2 >>> print(n == 42)
3 True
4 >>> print(n != 42)
5 False
6 >>> print(n > 42)
7 False
8 >>> print(n >= 42)
9 True
10 >>> print(n > 40 and n < 50)
11 True
12 >>> print(n < 40 or n > 50)
13 False
14 >>> print(n < 40 or n < 50)
15 True
16 >>> print(n < 40 and n > 50)
17 False

```

Code 33: Illustration des opérateurs logiques

Sur ce code, on remarque que certaines expressions ont peu de sens, et ceci *indépendamment* de la valeur de `n` :

- ▷ à la ligne 4, l'expression `n != 42` a un résultat prévisible : comme on sait à la ligne 2 que l'expression `n == 42` retourne vrai, sa négation `n != 42` retourne forcément faux ;
- ▷ à la ligne 14, on peut simplifier l'expression `n < 40 or n < 50` en `n < 50`, car si  $n < 40$ , alors  $n < 50$  ;
- ▷ à la ligne 16, l'expression est forcément fausse : aucun nombre n'est à la fois inférieur à 40 et supérieur à 50.

### 2.3.3 Opérateurs de chaînes de caractères

Il existe deux opérateurs pour manipuler les chaînes de caractères : le `+` et le `*` :

- ▷ `s + t` concatène la chaîne `t` à la chaîne `s` (elle suffixe `t` à `s`) ;
- ▷ `s * k` répète `k` fois la chaîne `s`.

Le code 34 illustre le fonctionnement de ces opérateurs.

On peut également utiliser les opérateurs de comparaison pour effectuer des comparaisons *lexicographiques* entre chaînes de caractères. Ces comparaisons sont effectuées caractère par caractère, en suivant un certain charset<sup>10</sup>.

Le code 35 illustre un exemple d'utilisation de ces opérateurs sur des chaînes de caractères.

10. Un charset est un système d'encodage de caractères en binaire. ASCII, UTF-8, etc. sont des exemples de charsets couramment utilisés. Par défaut, Python encode les chaînes de caractères en UTF-8.

```
>>> s = "Hello "
>>> t = "World!"
>>> u = s + t
>>> print(u)
Hello World!
>>> print("Ha" * 4)
HaHaHaHa
```

Code 34: Illustration des opérateurs logiques

```
>>> s = "Apple"
>>> t = "Linux"
>>> print(s == t)
False
>>> print(s < t)
True
```

Code 35: Illustration des opérateurs logiques

À l'évidence, `s == t` retourne `False` car les chaînes de caractères `s` et `t` ne sont pas les mêmes. Par ailleurs, l'expression `s < t` retourne `False` car la comparaison s'effectue caractère par caractère : l'interpréteur teste d'abord si  $A < L$ , et c'est le cas, car le charset encode<sup>11</sup> *A* comme 65, *L* comme 76 et  $65 < 76$ .

### 2.3.4 Opérateurs d'assignation

En plus des opérateurs habituels, Python offre des opérateurs permettant d'effectuer l'opération désirée sur une variable et de réassigner directement le résultat. Par exemple, `x += 3` est équivalent à `x = x + 3`.

Ces opérateurs sont associés à tous les opérateurs habituels, et sont listés à la table 2.3 pour les opérateurs arithmétiques et à la table 2.4 pour les opérateurs logiques. Il existe également les opérateurs `+=` et `*=` pour les chaînes de caractères.

Opérateur	Fonction	Exemple	Résultat si initialement, <code>x = 5</code>
<code>+=</code>	Addition	<code>x += 2</code>	7
<code>-=</code>	Soustraction	<code>x -= 2</code>	3
<code>*=</code>	Multiplication	<code>x *= 2</code>	10
<code>/=</code>	Division	<code>x /= 2</code>	2.5
<code>//=</code>	Division entière	<code>x //= 2</code>	2
<code>**=</code>	Exposant	<code>x **= 2</code>	25
<code>%=</code>	Modulo	<code>x %= 2</code>	1

TABLE 2.3 – Les opérateurs arithmétiques d'assignation

11. On peut obtenir le code correspondant à un caractère avec la fonction `ord`, par exemple en écrivant `ord("A")`.

Opérateur	Fonction	Exemple	Résultat si initialement, x = True
<code>&amp;=</code>	Addition	<code>x &amp;= False</code>	False
<code> =</code>	Soustraction	<code>x  = False</code>	True

TABLE 2.4 – Les opérateurs logiques d’assignation

### 2.3.5 Ordre d’évaluation et précedence des opérateurs

Comme en mathématiques, les opérateurs ont une priorité en Python : on évalue par exemple un `*` avant un `+`. Ainsi, l’expression `2 + 3 * 4` est évaluée comme `2 + (3 * 4)` et a la valeur `14`.

La table 2.5 donne la priorité de chacun des opérateurs que l’on a vu : une priorité haute signifie que l’opérateur est évalué avant les autres, de priorité moindre.

Priorité	Catégorie	Opérateurs
7	Exposant	<code>**</code>
6	Multiplication	<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>
5	Addition	<code>+</code> , <code>-</code>
4	Comparaison	<code>==</code> , <code>!=</code> , <code>&gt;</code> , <code>&gt;=</code> , <code>&lt;</code> , <code>&lt;=</code>
3	Logique	<code>not</code>
2	Logique	<code>and</code>
1	Logique	<code>or</code>

TABLE 2.5 – Priorité des opérateurs

Sur cette table, on remarque que, par exemple, l’expression

```
x + 2 * 5 >= 10 and y - 6 <= 20
```

est évaluée comme

```
((x + (2 * 5)) >= 10) and ((y - 6) <= 20).
```

On remarque que la première ligne, bien que syntaxiquement correcte, est assez cryptique. Il est donc souvent recommandé d’utiliser des parenthèses pour simplifier la lecture d’expressions complexes.

Les expressions comportant des opérateurs de même priorité sont quant à elles évaluées de gauche à droite. Par exemple, l’expression `2 + 3 + 4` est évaluée comme `(2 + 3) + 4`. Outre le fait d’être capable d’évaluer une expression comportant plusieurs opérateurs binaires, cela permet au compilateur d’effectuer une optimisation appelée *évaluation paresseuse* dans le cadre des expressions booléennes.

Considérons l’expression `False or True or f()`, qui implique l’exécution d’une fonction `f` très coûteuse en temps de calcul. Cette expression ne comprend que des opérateurs `or`, elle est donc évaluée de gauche à droite comme `(False or True) or f()`. Or, on sait que la première partie `(True or False)` a la valeur `True`. De plus, si l’on consulte la table de vérité du « ou », on remarque que `True` ou « n’importe quoi »



a la valeur `True`, ce qui signifie que le compilateur peut complètement ignorer l'appel de la fonction `f` et immédiatement retourner `True` après l'évaluation de `(True or False)`.

On a le même comportement avec une expression `True and False and f()` :

1. l'expression est évaluée comme `(True and False) and f()` ;
2. `(True and False)` a la valeur `False` ;
3. `False` et « n'importe quoi » a la valeur `False` ;
4. le compilateur ignore donc l'appel à la fonction `f`.

## 2.4 Conversions de types

En Python, il est possible de convertir des objets d'un type en des objets d'un autre type, par exemple de convertir la chaîne de caractères `"42"` en l'entier `42`.

Seules les conversions explicites<sup>12</sup> existent en Python : le programmeur *doit* les demander pour qu'elles se produisent.

Par exemple, le code 36 comporte une erreur de syntaxe : il n'est pas possible d'utiliser l'opérateur `+` entre un `int` et un `str`, et aucune conversion n'est demandée.

```
i = 2
s = "42"
print(s + i)
```

Code 36: Erreur de conversion

Cela a du sens qu'il faille demander une telle conversion : Python ne sait pas si l'on veut additionner les entiers `42` et `2` pour afficher `44`, ou concaténer les chaînes de caractères `"42"` et `"2"` pour afficher `422`.

Pour effectuer une conversion vers un type `T`, il faut en général utiliser la fonction `T()`. On peut donc corriger le code 36 soit en écrivant à la ligne 3

- ▷ `print(int(s) + i)` pour convertir `s` en `int` et additionner `42` et `2` pour afficher `44` ;
- ▷ `print(s + str(i))` pour convertir `i` en `str` et concaténer `"42"` et `"2"` pour afficher `422`.

Outre ces deux fonctions, il existe entre autre également la fonction `float()` pour convertir en nombre flottant.

Ces fonctions sont classiquement utilisées lors de la lecture au clavier avec la fonction `input` : quelle que soit l'entrée que l'utilisateur fournit, elle sera de type `str`, et il faut donc classiquement la convertir. Le code 37 illustre cette pratique.

Sur ce code,

12. Dans d'autres langages, comme le Java et le C++, il existe également des conversions *implicites*, qui interviennent sans que l'utilisateur le demande.

```

1 >>> s = input()
2 23
3 >>> print(s + "2")
4 232
5 >>> i = int(s)
6 >>> print(i + 2)
7 25
8 >>> d = float(s)
9 >>> print(d + 2.0)
10 25.0

```

Code 37: Entrée au clavier et conversion

- ▷ la ligne 1 appelle la fonction `input`, qui demande à l'utilisateur une entrée au clavier :
  - ▷ cet appel est *bloquant* : le code ne poursuit pas son exécution tant que l'utilisateur n'a pas entré une chaîne de caractères et tapé la touche « Enter » ;
  - ▷ l'utilisateur entre la chaîne 23 à la ligne 2, et cette chaîne est stockée dans la variable `s` ;
- ▷ aucune conversion n'est nécessaire à la ligne 3 : les deux opérandes sont de type `str`, et la concaténation se passe comme prévu ;
- ▷ la ligne 5 convertit `s` en `int`, et stocke le résultat dans `i`
- ▷ aucune conversion n'est nécessaire à la ligne 6 : les deux opérandes sont de type `int`, et l'addition se passe comme prévu ;
- ▷ la ligne 8 convertit `s` en `float`, et stocke le résultat dans `d`
- ▷ aucune conversion n'est nécessaire à la ligne 9 : les deux opérandes sont de type `float`, et l'addition se passe comme prévu.

Notons que bien qu'il n'existe pas d'opérateur `+`, `-`, `*` et `/` entre deux opérandes de différents types en général, il est néanmoins possible de les appeler entre des `int` et des `float`. Dans ce cas, l'opération arithmétique sera effectuée comme on s'y attend, et le résultat sera de type `float`. Ce comportement est illustré au code 38.

```

>>> i = 2
>>> d = 3.0
>>> print(i + d)
5.0

```

Code 38: Addition entre un `int` et un `float`

Ce code doit effectuer une addition entre l'entier 2 et le flottant 3.0, ce qu'il fait naturellement et affiche le flottant 5.0.

## Annexe : tables de vérités

La figure 2.7 rappelle les tables de vérité des opérateurs logiques utilisables en Python.

$p$	$\neg p$	$p$	$q$	$p \text{ and } q$	$p$	$q$	$p \text{ or } q$
True	False	True	True	True	True	True	True
True	False	True	False	False	True	False	True
False	True	False	True	False	False	True	True
		False	False	False	False	False	False

FIGURE 2.7 – Rappel des tables de vérités



L'objectif de ce chapitre est d'introduire la notion de *fonction* : une sorte de petite routine – comme faire du café, se brosser les dents – mais dans le cadre de la programmation : c'est quelque chose que l'on fait souvent. Quand on a besoin de cette routine, on l'*appelle*. Ce mécanisme est présenté dans ce chapitre.

## 3.1 Notion de fonction

### Définition

Une *fonction* est une séquence d'instructions à laquelle on donne un nom. Elle peut prendre en entrée une liste de *paramètres* (potentiellement vide) et fournit en sortie *une valeur de retour*.

Parfois, les paramètres sont appelés *arguments*.

En quelque sorte, une fonction peut être vue comme une « boîte noire » qui effectue un certain travail :

- ▷ les paramètres sont ce dont elle a besoin en entrée pour effectuer ce travail ;
- ▷ la valeur de retour est le résultat de ce travail.

Par exemple, on peut schématiser le travail d'une fonction qui convertit une distance fournie en kilomètres en une distance fournie en mètres comme illustré à la figure 3.1 : on convertit 23km en 23 000m. À l'évidence, cette fonction se contente de multiplier son entrée par 1 000 et de retourner le résultat de ce calcul.



FIGURE 3.1 – Illustration schématique d'une fonction

On a déjà vu plusieurs exemples de fonctions dans le chapitre 2, comme

- ▷ `print` qui prend un nombre arbitraire de paramètres et les affiche en console,
- ▷ `type` qui prend en paramètre une expression et retourne son type,
- ▷ `ord` qui prend en paramètre un caractère et retourne son encodage UTF-8,

### 3.2. DÉFINITION DE FONCTIONS

- ▷ `sqrt` qui prend en paramètre un nombre et retourne sa racine carrée,
- ▷ `abs` qui prend en paramètre un nombre et retourne sa valeur absolue.

Le code 39 illustre le fonctionnement d'appel de fonction de manière concrète avec la fonction `print` et la fonction `abs`.

```
1 >>> print("Hello World!")
2 Hello World!
3 >>> print(abs(-5))
4 5
5 >>> print(abs("Hello World!"))
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   TypeError: bad operand type for abs(): 'str'
9 >>>
```

Code 39: Exemples d'appel de la fonction `abs`

Ce code s'exécute de la manière suivante :

**ligne 1** La fonction `print` est appelée en fournissant en entrée la chaîne de caractères « Hello World! » :

- ▷ elle affiche en console la chaîne de caractères « Hello World! » ;

**ligne 3** La fonction `print` est appelée en fournissant en entrée la fonction `abs` en fournissant en entrée le nombre  $-5$  :

1. on appelle la fonction `abs` en lui passant en entrée  $-5$  elle retourne  $5$  ;
2. on appelle la fonction `print` en lui fournissant le retour  $5$  de la fonction `abs`, elle affiche  $5$  à l'écran ;

**ligne 5** La fonction `print` est appelée en fournissant en entrée la fonction `abs` en fournissant en entrée la chaîne de caractères « Hello World! » :

1. on appelle la fonction `abs` en lui passant en entrée « Hello World! » ;
2. l'objectif de cette fonction est de calculer la valeur absolue d'un nombre : lui passer une chaîne de caractères n'a pas de sens ;
3. la fonction `abs` renvoie une erreur et interrompt brutalement l'exécution du programme.

Notez que le message d'erreur de l'interpréteur sur ce dernier exemple est explicite : l'opérande de la fonction `abs` (la chaîne de caractères « Hello World ») n'est pas du bon type.

## 3.2 Définition de fonctions

Au vu de la notion de fonction de la section 3.1, on peut affirmer que ce sont de petits bouts de code *réutilisables*, grâce à leur nom. À ce titre, les programmeurs définissent souvent leurs propres fonctions, afin de leur rendre service plus tard dans leur programme.

Rappelons par exemple le code 28, page 28 : ce code avait entre autres pour objectif de calculer l'aire et la circonférence d'un cercle de rayon 3. Si l'on veut

calculer l'aire et la circonférence d'un cercle de rayon 4, il va falloir recopier les instructions

```
area = pi * r * r
circ = 2 * pi * r
```

Ce n'est pas très pratique...

D'autre part, si l'on copie ces instructions ailleurs dans notre programme, et qu'il s'avère que l'on a fait une erreur de sémantique<sup>1</sup>, il va falloir corriger cette erreur à *deux* endroits : dans le code original, et dans la copie. Encore une fois, ce n'est pas très pratique...

Finalement, on est parfois amené à écrire des programmes très complexes, et ça ne serait de nouveau pas pratique d'écrire de tels programmes « en un seul bloc ». On préférerait diviser ce programme en morceaux aux fonctionnalités similaires. Cette approche, parfois appelée « divide and conquer », est utile pour visualiser facilement le fonctionnement global d'un programme, ainsi que faciliter sa conception.

Ainsi, on préfère définir soi-même des fonctions `area` et `circumference` qui, à chaque fois qu'elles sont appelées, calculent l'aire et la circonférence d'un cercle de rayon donné. On procède tel qu'illustré au code 40, et on appelle ces fonctions comme dans le code 41.

```
code/chap3/circle.py
1 import math
2
3 def area(r):
4     a = math.pi * r * r
5     return a
6
7 def circumference(r):
8     return 2 * math.pi * r * r
9
10 def print_area_formula():
11     print("Area computed as 'pi * r * r'")
12
13 def print_circle_info(r, a, circ):
14     print("Circle of radius", r, "has area", a, "and circumference", circ)
```

Code 40: Définition de fonctions relatives au cercle

Sur le code 40, on remarque plusieurs caractéristiques syntaxiques :

- ▷ on définit systématiquement une fonction en commençant par le mot-clé `def` ;
- ▷ toute fonction a un *nom*, qui suit le mot-clé `def` ;
- ▷ certaines fonctions ont des *paramètres*, qui suivent le nom de la fonction, et sont entourés de parenthèses ;
- ▷ la liste des paramètres est suivie du caractère « : » ;

1. Pour rappel, une erreur de sémantique est une erreur de logique où le programme ne fournit pas le résultat attendu.

```

1 import circle
2
3 r = 2
4 a = circle.area(2)
5 circ = circle.circumference(2)
6 print("A circle of radius", r, "has area", a)
7 print("A circle of radius", r, "has circumference", circ)
8 circle.print_area_formula()
9 circle.print_circle_info(r, a, circ)

```

Code 41: Appel de fonctions relatives au cercle

- ▷ le code d'une fonction, que l'on appelle le *corps*, est « décallé vers la droite » (on parle d'*indentation*);
- ▷ certaines fonctions fournissent une *valeur de retour*, précédée du mot-clé `return`.

Cette terminologie est illustrée à la figure 3.2 pour la fonction `area`.

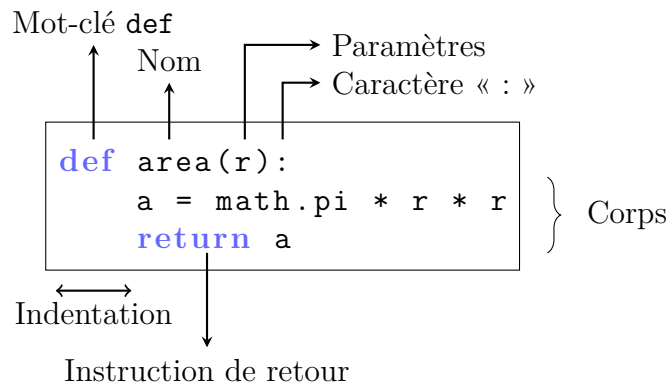


FIGURE 3.2 – Illustration syntaxique d'une fonction

Remarquons également, sur les codes 40 et 41, que

- ▷ l'on a enregistré le code 40 dans un fichier `circle.py` (on parle de *module*<sup>2</sup>), et que l'on importe ce module dans le code 41 que l'on a enregistré dans un fichier `circle-test.py`;
- ▷ certaines fonctions n'ont pas de paramètres (comme `print_area_formula`), et dans ce cas, il faut quand même ajouter une paire de parenthèses vides;
- ▷ certaines fonctions ont plusieurs paramètres (comme `print_circle_info`), et dans ce cas, il faut les séparer par des virgules;
- ▷ les fonctions utilisent leurs paramètres comme des variables;
- ▷ certaines fonctions (comme `print_area_formula` et `print_circle_info`) n'ont pas de valeur de retour : elles se contentent d'effectuer un travail sans rien retourner au programmeur;
- ▷ on peut retourner une valeur stockée dans une variable, comme la fonction `area`, ou directement une valeur calculée, comme `circumference`.

2. Le fonctionnement des modules sera détaillé au chapitre 4



Finalement, notons qu'au même titre que les variables, on ne peut pas donner n'importe quel nom à une fonction. En pratique, les contraintes liées aux noms de fonctions sont les mêmes que celles liées aux noms de variable (pas de mot réservé, commence par une lettre ou le caractère « `_` », etc.).

### 3.2.1 Valeur de retour

On a vu que certaines fonctions *retournent* une valeur, ce qui signifie qu'elles effectuent une forme de calcul et qu'elles renvoient son résultat. Le retour d'une valeur est effectué à l'aide de l'instruction `return`, tel qu'illustré au code 40, page 31.

On peut retourner cette valeur

- ▷ soit via une variable, comme dans la fonction `area` ;
- ▷ soit directement via une expression, comme dans la fonction `circumference`.

Notons que l'on a dû récupérer cette valeur pour l'imprimer,

- ▷ soit via une variable que l'on passe à la fonction `print` ;
- ▷ soit directement en la passant à la fonction `print`.

On peut ainsi réécrire les lignes 4, 5 et 9 du code 41, page 32 simplement en

```
circle.print_circle_info(r, circle.area(r), circle.circumference(r))
```

### Retours multiples

Il est possible en Python de retourner plusieurs valeurs en une fois, en les séparant par des virgules. On peut récupérer ces valeurs de la même manière dans des variables, ou via la syntaxe des *tuples*<sup>3</sup>, tel qu'illustré aux codes 42 et 43.

```
1 def double_square(x):
2     return 2 * x, x**2
```

code/chap3/multiple\_returns.py

Code 42: Une fonction retournant deux valeurs

```
1 import multiple_returns
2
3 a, b = multiple_returns.double_square(3)
4 print("Twice", 3, "is", a)
5 print("Squared", 3, "is", b)
6
7 tup = multiple_returns.double_square(3)
8 print("Twice", 3, "is", tup[0])
9 print("Squared", 3, "is", tup[1])
10 print(tup)
```

code/chap3/multiple\_returns\_test.py

Code 43: Récupération de valeurs multiples de retour

3. Un tuple est simplement un ensemble de valeurs placées dans un ordre fixé.

### 3.2. DÉFINITION DE FONCTIONS

La fonction `double_square` calcule simplement le double et le carré du paramètre de la fonction, et retourne ce calcul sous la forme d'un *tuple* de deux valeurs. On peut récupérer ces deux valeurs

- ▷ en une fois avec une affectation multiple de deux variables séparées par des virgules, comme à la ligne 3 ;
- ▷ grâce à la syntaxe dédiée aux tuples, comme aux lignes 8 et 9 : la première valeur est accédée grâce à l'expression `tup[0]` et la deuxième grâce à `tup[1]`. Ici, les nombres entre les crochets `[]` dénotent la place de la valeur dans le tuple : 0 signifie « première place » et 1 signifie « seconde place ».

Si l'on affiche directement le tuple comme à la ligne 10, il s'affiche comme le couple mathématique (6, 9).

Notons que la syntaxe d'affectation multiple permet également d'échanger deux variables de manière plus succincte que ce qu'on avait fait au code 26, page 16 (à l'aide d'une variable intermédiaire), tel qu'illustré au code 44.

```
1 >>> x = 1
2 >>> y = 2
3 >>> x, y = y, x
4 >>> print("x =", x, "and y =", y)
5 x = 2 and y = 1
6 >>>
```

Code 44: Échange en une seule instruction

#### Pas de valeur de retour

Certaines fonctions ne retournent pas de valeur : celles où l'on n'écrit pas d'instruction `return`, comme la fonction `print_area_formula` du code 40, page 31.

En réalité, en Python, toute fonction retourne une valeur. En l'absence d'une instruction `return`, la valeur `None` (de type `NonType`) est retournée implicitement<sup>4</sup>.

Ce comportement est illustré aux codes 45 et 46. Exécuter ce dernier code affiche

- ▷ « Hello there! » à la ligne 3,
- ▷ « None » à la ligne 4,
- ▷ « Hello there! » suivi de « None » à la ligne 6.

```
1 def say_hello():
2     print("Hello there!")
```

code/chap3/no\_return.py

Code 45: Une fonction ne retournant rien

---

4. C'est-à-dire sans demande explicite de la part du programmeur

```

1 import no_return
2
3 x = no_return.say_hello()
4 print(x)
5
6 print(no_return.say_hello())

```

Code 46: Impression du retour d'une fonction ne retournant rien

### Instruction `return` versus `print`

Pour l'instant, on a fait des affichages

- ▷ soit en mode interactif,
- ▷ soit avec la fonction `print`.

On a également créé des fonctions qui retournent des valeurs que l'on a affichées.

Il y a une différence entre *retourner* une valeur et l'*afficher*.

Afficher signifie « montrer quelque chose à l'écran », alors que retourner signifie « donner la valeur au programmeur qui l'a demandée ». Cette différence de comportement est illustrée aux codes 47 et 48.

```

1 def square_no_print(x):
2     return x**2
3
4 def square_print(x):
5     print(x**2)

```

Code 47: Deux fonctions, l'une avec `return`, l'autre avec `print`

```

1 import print_vs_return
2
3 x = print_vs_return.square_no_print(3)
4 print(x)
5 print()
6
7 x = print_vs_return.square_print(3)
8 print(x)

```

Code 48: Utilisation de fonctions avec `return` et avec `print`

Le code 48 s'exécute de la manière suivante :

**ligne 3** on appelle la fonction `square_no_print` :

1. le carré du paramètre 3 est calculé;
2. on retourne la valeur 9;
3. la valeur 9 est affectée à `x`;

**ligne 4** on affiche la valeur 9;

**ligne 7** on appelle la fonction `square_print` :

1. le carré du paramètre 3 est calculé ;
2. on affiche 9 ;
3. on retourne la valeur `None`, car la fonction `square_print` n'a pas d'instruction `return` ;
4. la valeur `None` est affectée à `x` ;

**ligne 8** on affiche la valeur `None`.

Dans un programme réel, on ne veut probablement pas affecter la valeur `None` à `x`...

Ce genre de problème est encore plus significatif dans le cas où l'on devait utiliser cette valeur `None`, tel qu'illustré au code 49.

```

1 import print_vs_return
2
3 x = print_vs_return.square_no_print(3)
4 x = x + 2
5 print(x)
6 print()
7
8 x = print_vs_return.square_print(3)
9 x = x + 2
10 print(x)

```

Code 49: Utilisation de fonctions avec `return` et avec `print`

Ce code s'exécute comme suit :

**ligne 3** on appelle la fonction `square_no_print` :

1. le carré du paramètre 3 est calculé ;
2. on retourne la valeur 9 ;
3. la valeur 9 est affectée à `x` ;

**ligne 4** la valeur `x + 2` est affectée à `x` : on y met la valeur 11 ;

**ligne 5** on affiche la valeur 11 ;

**ligne 8** on appelle la fonction `square_print` :

1. le carré du paramètre 3 est calculé ;
2. on affiche 9 ;
3. on retourne la valeur `None`, car la fonction `square_print` n'a pas d'instruction `return` ;
4. la valeur `None` est affectée à `x` ;

**ligne 9** la valeur `x + 2` est affectée à `x`

1. on tente de faire « +2 » sur une valeur `None`, ce n'est pas possible
2. cette opération provoque une erreur
3. un message d'erreur est affiché.

Le message du terminal affirme (entre autres) que la ligne 9 « `x = x + 2` » est problématique, en l'occurrence affiche

```
... line 9, in <module>
x = x + 2
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

Ce message est explicite : on n'a pas le droit de faire '+' entre un `None` et un entier car l'opération n'est pas définie.

En pratique, dans un code de qualité, on évitera à tout prix d'effectuer des impressions avec `print`, car on a besoin des valeurs calculées par les fonctions pour les utiliser dans d'autres calculs. Les affichages sont effectués soit dans des programmes de test (comme dans ce document), soit dans une interface utilisateur.

Les seules exceptions notoires à cette règle sont les fonctions dont l'objectif même est d'effectuer une impression, comme les fonctions `print_area_formula` et `print_circle_info` du code 40, page 31.

### Code mort

Comme l'objectif d'une instruction `return` est de retourner à l'appellant, cette instruction sera toujours la dernière exécutée dans le corps d'une fonction. Les instructions qui suivraient ce `return` ne seront donc *jamaïs* exécutées : on dit que c'est un code *mort*.

Le code 50 illustre un exemple de code mort. Sur ce code, bien que la fonction `f` soit appelée, le `print` qui suit l'instruction `return` n'est pas exécuté et n'affiche rien en console.

```
1 def f():
2     return 42
3     print("Snakes...why'd it have to be snakes?")
4
5 print(f())
```

code/chap3/dead\_code.py

Code 50: Exemple de code mort

### 3.2.2 Typage de paramètres

Le langage Python utilise un système *dynamique* de types, ce qui signifie

- ▷ qu'on ne doit pas spécifier explicitement le type des variables et paramètres de fonctions ;
- ▷ que l'interpréteur détermine automatiquement ce type à l'exécution afin de permettre au programme de fonctionner correctement.

Ainsi, quand on écrit la fonction `sum` du code 51, on peut l'utiliser à la fois avec des entiers, des flottants, ou des chaînes de caractères, car l'expression `x + y` est valide à la fois pour des entiers, des flottants et des chaînes de caractères. Ce comportement est illustré au code 52.

### 3.2. DÉFINITION DE FONCTIONS

```
1 def sum(x, y):  
2     return x + y
```

code/chap3/dyn\_typing.py

Code 51: Définition d'une simple fonction

```
1 import dyn_typing  
2  
3 print(dyn_typing.sum(1, 2))  
4 print(dyn_typing.sum(1.0, 2.0))  
5 print(dyn_typing.sum(1.0, 2))  
6 print(dyn_typing.sum("Hello", "World!"))
```

code/chap3/dyn\_typing\_test.py

Code 52: Typage dynamique des paramètres de la fonction `sum`

L'exécution de ce code se produit comme suit :

**ligne 3** on appelle la fonction `sum` :

1. l'interpréteur détermine que le type de `x` et `y` est `int` ;
2. l'interpréteur utilise l'algorithme d'addition des entiers ;
3. l'entier 3 retourné, et ensuite affiché ;

**ligne 4** on appelle la fonction `sum` :

1. l'interpréteur détermine que le type de `x` et `y` est `float` ;
2. l'interpréteur utilise l'algorithme d'addition des flottants ;
3. le flottant 3 retourné, et ensuite affiché ;

**ligne 5** on appelle la fonction `sum` :

1. l'interpréteur détermine que le type de `x` est `float` et celui de `y` est `int` ;
2. il n'existe pas d'algorithme pour additionner un `float` et un `int`, mais il existe une conversion implicite de `int` vers `float` ;
3. l'entier 2 est converti en `float` ;
4. l'interpréteur utilise l'algorithme d'addition des flottants ;
5. le flottant 3 retourné, et ensuite affiché ;

**ligne 6** on appelle la fonction `sum` :

1. l'interpréteur détermine que le type de `x` et `y` est `str` ;
2. l'interpréteur utilise l'algorithme d'addition des chaînes de caractères : la concaténation ;
3. la chaîne de caractères `Hello World!` retournée, et ensuite affichée.

Notons que, parfois, l'interpréteur ne sera pas capable d'effectuer les opérations qu'on lui a demandées, parce qu'il n'est pas capable d'exécuter le code avec les types qui ont été fournis. Par exemple, si l'on appelle la fonction `sum` avec un `int` et un `str`, tel qu'illustré au code 53, on provoque une erreur :

1. l'interpréteur détermine que le type de `x` est `int` et celui de `y` est `str` ;
2. il n'existe pas d'algorithme pour additionner un `int` et un `str`, et aucune conversion implicite de `int` vers `str` ou de `str` vers `int` ;

```

1 import dyn_typing
2
3 print(dyn_typing.sum(1, "2")) #error
4 print(dyn_typing.sum(str(1), "2")) #prints "12"
5 print(dyn_typing.sum(1, int("2"))) #prints 3

```

Code 53: Erreur lors de l'appel de `sum`

3. l'exécution renvoie une erreur, et s'interrompt brutalement.

Concrètement, l'interpréteur affiche le message d'erreur

```

... line 2, in sum
return x + y
TypeError: unsupported operand type(s) for +: 'int' and 'str'

```

Si l'on avait effectivement voulu que ce code fonctionne, il aurait fallu utiliser une conversion explicite :

- ▷ soit de `int` vers `str` (ligne 4), pour calculer `+` entre deux `str` (on concatène) et afficher 12,
- ▷ soit de `str` vers `int` (ligne 5), pour calculer `+` entre deux `int` (on additionne) et afficher 3,

tel qu'illustré dans la suite du code 53.

#### Information

En Python, on peut néanmoins restreindre le type des paramètres et des retours de fonctions à certains types seulement pour éviter les conversions implicites et erreurs illustrées ci-dessus, notamment grâce au module `typing.py`. Cette pratique et l'utilisation de ce module sortent du cadre de ce cours, et ne seront donc pas abordés.

### 3.2.3 Surcharge de fonctions

Dans un langage de programmation, on parle de *surcharge*<sup>5</sup> de fonction quand deux fonctions existent avec le même nom, mais avec des paramètres en nombre ou en types différents. Le compilateur ou l'interpréteur décide alors quelle fonction appeler parmi les surcharges disponibles en fonction du contexte.

En Python, on ne peut pas surcharger de fonctions. Le code 54, par exemple, provoque une erreur de syntaxe, car la fonction `f` existe « en deux exemplaires ».

### 3.2.4 Ordre des paramètres

L'ordre dans lequel on passe les paramètres à une fonction a de l'importance, tel qu'illustré par les codes 55 et 56.

Un appel à `ratio` avec

1. 3 comme premier paramètres et 2 comme second calcule  $\frac{3}{2}$ , c'est-à-dire 1,5 ;

5. Parfois, on parle d'*overload* de fonction.

```

1 def f():
2     print("f()")
3
4 def f(x):
5     print("f(", x, ")")
6
7 print(f())
8 print(f(1))

```

Code 54: Surcharge de fonction

```

1 def ratio(x,y):
2     return x / y

```

Code 55: L'ordre des paramètres a de l'importance

```

1 import order_matters
2
3 print(order_matters.ratio(3, 2))
4 print(order_matters.ratio(2, 3))

```

Code 56: L'ordre des paramètres a de l'importance

2. 2 comme premier paramètres et 3 comme second calcule  $\frac{2}{3}$ , c'est-à-dire 0,666...

### 3.2.5 Paramètres avec valeurs par défaut

Même si la surcharge de fonction n'est pas disponible, il est toutefois possible de définir des fonctions qui acceptent des paramètres *a priori* différents en nombres.

Pour cela, on définit des paramètres ayant des *valeurs par défaut*, c'est à dire à qui la fonction donne une valeur si on ne l'a pas précisée. Pour cela, on utilise l'opérateur `=` dans la liste des paramètres, comme illustré par les codes 57 et 58.

```

1 def hello(name="World!"):
2     print("Hello", name)

```

Code 57: Paramètres avec valeurs par défaut

```

1 import param_def
2
3 param_def.hello()
4 param_def.hello("there!")

```

Code 58: Paramètres avec valeurs par défaut

La fonction `hello` du code 57 accepte un paramètre `name`, et si ce paramètre n'est pas fourni à l'appel, alors il prend la valeur `"World!"`. C'est ce qui se passe lors de l'exécution du code 58 :

- ▷ la ligne 3 appelle `hello` sans paramètre, le paramètre `name` prend alors la valeur `"World!"`, et affiche `"Hello World!"` ;



- ▷ la ligne 4 appelle `hello` avec le paramètre `name` ayant la valeur `"there!"`, et affiche `"Hello there!"`.

Il existe néanmoins une contrainte lors de la spécification des paramètres avec valeurs par défaut : ceux-ci doivent être les *derniers* de la liste de paramètres. En effet, si l'on définit une fonction `f(x=0,y,z=0)`, le compilateur ne peut par exemple pas déterminer si un appel `f(1,2)`

- ▷ appelle `f(0,1,2)`, en décidant de mettre une valeur par défaut à `x` ;
- ▷ appelle `f(1,2,0)`, en décidant de mettre une valeur par défaut à `z`.

Une telle définition provoque une erreur de syntaxe, tel qu'illustré par le code 59. L'interpréteur affiche le message d'erreur « `SyntaxError: non-default argument follows default argument` ».

```

1 def f(x = 0, y, z = 0):
2     print("f(", x, ",", y, ",", z, ")")
3
4 print(f(1,2))

```

code/chap3/param\_def\_last.py

Code 59: Les paramètres avec valeurs par défaut doivent être les derniers

## 3.3 Passage de paramètres

On a vu qu'une étape importante dans l'appel d'une fonction est le *passage de paramètres*, c'est à dire l'étape où l'on fournit les paramètres à une fonction afin qu'elle puisse faire son travail. Cette étape n'est pas immédiate, est parfois complexe, et tous les langages ont leurs spécificités.

Cette section détaille la manière dont cela fonctionne en Python. On détaille d'abord à la section 3.3.1 deux concepts importants liés aux objets Python : l'*immuabilité* et l'*identité*. Comprendre ces concepts est fondamental pour le passage d'argument, qui sera différencié pour les objets immuables et ceux qui ne ne sont pas, tel que détaillé aux sections 3.3.3 et 3.3.4. Finalement, la section 3.3.5 définit comment les paramètres d'une fonction sont évalués.

### 3.3.1 Immuabilité et identité

On a vu qu'une caractéristique fondamentale d'un objet est son type, que l'on peut connaître grâce à la fonction `type`. Le type d'un objet permet entre autres de connaître la liste des opérations que l'on peut effectuer avec cet objet.

Deux autres caractéristiques importantes d'un objet sont l'*immuabilité* et l'*identité*.

#### Définition

Un objet est *immuable* si on ne peut pas effectuer d'opération qui change son état interne.

### 3.3. PASSAGE DE PARAMÈTRES

Jusqu'à présent, tous les objets que l'on a vus (les entiers, les flottants, les chaînes de caractères, les tuples) sont immuables. Par exemple, le code 60 provoque plusieurs erreurs d'exécution.

```
1 t = (1,2) #creates a tuple
2 print(t)
3 #t[0] = 42 #error
4 print(t)
5
6 s = "Hello World!"
7 print(s)
8 s[0] = "t" #error
9 print(s)
```

Code 60: Erreur avec des objets immuables

**ligne 1** On crée le tuple (1,2), et on l'affecte à la variable `t` ;

**ligne 2** On imprime `t`, ce qui affiche (1, 2) ;

**ligne 3** On tente de changer le premier élément du tuple ;

1. `t[0]` signifie que l'on utilise l'opérateur `[]` avec l'indice `0` sur `t` pour obtenir une référence vers le premier objet du tuple ;
2. On essaie de réaffecter ce premier objet avec `=` ;
3. `t` est un tuple, donc immuable, et l'affectation échoue ;
4. Un message d'erreur est retourné, et l'exécution s'interrompt brutalement ;

**ligne 6** On crée la chaîne de caractères « Hello World! », et on l'affecte à la variable `s` ;

**ligne 7** On imprime `s`, ce qui affiche Hello World! ;

**ligne 8** On tente de changer le premier élément de la chaîne de caractères ;

1. `s[0]` signifie que l'on utilise l'opérateur `[]` avec l'indice `0` sur `s` pour obtenir une référence vers le premier objet de la chaîne de caractères ;
2. On essaie de réaffecter ce premier objet avec `=` ;
3. `s` est une chaîne de caractères, donc immuable, et l'affectation échoue ;
4. Un message d'erreur est retourné, et l'exécution s'interrompt brutalement.

Les tuples étant immuables, on ne peut pas non plus « ajouter un élément » dans un tuple.

Il existe à l'évidence des objets qui sont muables, par exemples, les *listes*. Intuitivement, les listes sont, comme les tuples, des collections ordonnées d'objets. Le code 61 illustre la muabilité des listes, qui seront étudiées en détail au chapitre 6.

L'exécution de ce code se produit comme suit :

**ligne 1** On crée la liste [1,2,3,4,5], et on l'affecte à la variable `l` ;

**ligne 2** On imprime `l`, ce qui affiche [1, 2, 3, 4, 5] ;

```

1 l = [1, 2, 3, 4, 5]
2 print(l)
3 l[0] = 42
4 print(l)

```

Code 61: Illustration de la muabilité d'une liste

**ligne 3** On change le premier élément<sup>6</sup> de `l` en 42 ;

▷ contrairement aux tuples qui sont immuables, les listes ne le sont pas, et donc cette instruction fonctionne ;

**ligne 2** On imprime `l`, ce qui affiche `[42, 2, 3, 4, 5]`.

Ainsi, on remarque que la seule façon de « changer » un objet immuable comme un tuple est de l'*affecter*. Ceci implique toutefois quelques subtilités en termes d'identité<sup>7</sup>.

#### Définition

L'*identité* d'un objet est un entier constant unique associé à un objet, et existe pour toute la durée de vie de cet objet.

On peut connaître l'identité d'un objet en utilisant la fonction `id`, tel qu'illustré au code 62.

```

1 x = 2
2 print("id of x", id(x))
3 print()
4
5 y = 3
6 print("id of y", id(y))
7 print()
8
9 z = x
10 print("id of x", id(x))
11 print("id of z", id(z))
12 print()
13
14 s = "Hello World!"
15 print("id of s", id(s))

```

Code 62: Récupération de l'identité d'objets

En exécutant ce code, on produit par exemple<sup>8</sup> l'affichage

6. La syntaxe pour accéder à un élément d'une liste est la même que celle pour accéder à un élément d'un tuple.

7. Ce mécanisme d'identité est propre au Python. En Java, C++, PHP, etc., les objets sont habituellement identifiés par leurs adresses, et le comportement de ces adresses est significativement différent de celui des identités en Python.

8. Les identifiants des variables changent classiquement à chaque exécution : il est donc pratiquement impossible que vous obteniez exactement ces valeurs lorsque vous exécutez ce code.

### 3.3. PASSAGE DE PARAMÈTRES

```
id of x 140461005078800
id of y 140461005078832
id of x 140461005078800
id of z 140461005078800
id of z 140461004332016
```

et on remarque que

- ▷ **x** et **z** ont la même identité : c'est normal, on a défini **z** comme **z = x** : ce sont *les mêmes* objets ;
- ▷ **y** n'a pas la même identité que **x**, **z** et **s** : ce sont des objets *différents* ;
- ▷ **y** n'a pas la même identité que **x**, **z** et **s** : ce sont des objets *différents*.

Cette situation est illustrée à la figure 3.3.

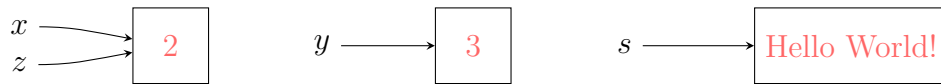


FIGURE 3.3 – Illustration de l'identité des variables

Il ne faut pas confondre *objet* et *variable*.

Une variable est un *alias* : un nom pour désigner un objet. Un objet est une entité qui contient des données. Une identité est associée à un objet, pas à une variable. Réassigner une variable ne change pas l'objet qu'elle désignait précédemment. Ce comportement est illustré au code 63, produisant par exemple l'affichage

```
id of x 140308839923984
id of y 140308839923984
id of z 140308839923984

id of x 140308839924048
id of y 140308839923984
id of z 140308839923984
```

Ce code s'exécute comme suit :

**ligne 1** on veut créer un objet avec la valeur 2 ;

1. cet objet n'existe pas ;
2. on en crée un nouveau, et on affecte la variable **x** à cet objet ;

**ligne 2** on affecte la variable **y** à **x** : aucun nouvel objet n'est créé

**ligne 3** on veut créer un objet avec la valeur 2 ;

1. cet objet existe déjà ;
2. on affecte la variable **y** à cet objet : aucun nouvel objet n'est créé

**ligne 10** on veut créer un objet avec la valeur 4 ;

1. cet objet n'existe pas ;
2. on en crée un nouveau, et on affecte la variable **x** à cet objet.

Ces quatre étapes sont illustrées à la figure 3.4.

```

1 x = 2
2 y = x
3 z = 2
4
5 print("id of x", id(x))
6 print("id of y", id(y))
7 print("id of z", id(z))
8 print()
9
10 x = 4
11 print("id of x", id(x))
12 print("id of y", id(y))
13 print("id of z", id(z))

```

Code 63: Identité et affectation

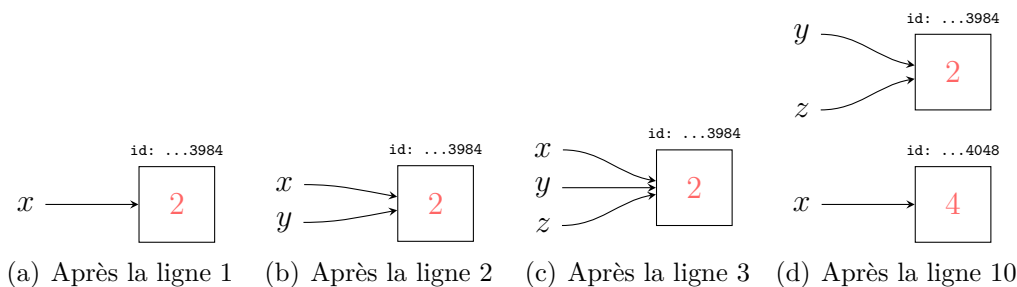


FIGURE 3.4 – Illustration de l'identité après affectation

### 3.3.2 Notion d'effet de bord

Une notion importante dans la définition de fonction est celle d'effet de bord.

#### Définition

Une fonction possède un *effet de bord* si celle-ci produit un effet qui est visible en dehors de la fonction.

Retourner une valeur n'est pas considéré comme un effet de bord, mais afficher une valeur à l'écran, écrire dans un fichier et modifier un paramètre sont des effets de bord.

La plupart du temps, c'est une bonne pratique d'écrire des fonctions sans effet de bord, sauf si le but de la fonction est explicitement d'en produire un, comme dans le cas de la fonction `print` ou de `print_circle_info` du code 40, page 31.

En effet, l'utilisateur d'une fonction sans effet de bord

- ▷ peut contrôler ce qui est affiché ou pas dans son programme ;
- ▷ ne court pas le risque de voir des valeurs de son programme modifiées sans un appel explicite de sa part.

Cette deuxième caractéristique permet d'écrire (et de déboguer) des programmes plus facilement : on s'imagine mal devoir utiliser la fonction `abs` si elle devait effectuer des affichages intempestifs ou modifier des valeurs dans notre programme !

Souvenez-vous également que *retourner* une valeur n'est pas la même chose que de l'*afficher*, et que l'on préfère en général écrire des fonctions qui retournent des valeurs que des fonctions qui les affichent. En plus des arguments évoqués précédemment, l'absence d'effet de bord est un argument supplémentaire en faveur de cette pratique.

#### 3.3.3 Passage d'objets immuables comme paramètres

Si l'on passe un objet immuable (comme un entier) à une fonction, par définition, on ne peut pas le modifier. Impossible donc de créer un effet de bord par le biais de ce paramètre.

Ce comportement est illustré au codes 64 et 65.

```
code/chap3/pass_immu.py
1 def pass_int(x):
2     x = 42
3     return x
4
5 def pass_tuple(t):
6     t = (42, 23)
7     return t
8
9 def pass_tuple2(t):
10    t[0] = 42
11    return t
```

Code 64: Passage d'objet immuable

```
code/chap3/pass_immu_test.py
1 import pass_immu
2
3 x = 2
4 pass_immu.pass_int(x)
5 print(x)
6 print()
7
8 t = (1,2)
9 pass_immu.pass_tuple(t)
10 print(t)
11 print()
12
13 pass_immu.pass_tuple2(t)
14 print(t)
15 print()
```

Code 65: Appel de fonction avec des objets immuables

Quand on exécute le code 65, on constate que

- ▷ ni la valeur de `x`, ni la valeur de `t` ne sont modifiées : les fonctions `pass_int` et `pass_tuple` n'ont donc pas d'effets de bord ;
- ▷ appeler la fonction `pass_tuple2` provoque une erreur, car on essaye de modifier le paramètre `t`, or celui-ci est immuable.

La seule manière de « changer » un objet immuable est de le réassigner<sup>9</sup>, comme dans les fonctions `pass_int` et `pass_tuple`. Toutefois, celles-ci n'ont pas d'effet de bord, pour les raisons liées aux identités détaillées dans la section 3.3.1.

En pratique, on dit que les *variables* sont passées *par valeur* : à chaque fois qu'on les fournit à une fonction, on copie l'*alias* (*pas* l'objet).

Pour détailler ce mécanisme, considérons les codes 66 et 67, qui ne diffèrent des codes précédents que par l'ajout de `print` afin de tracer les changements d'identités des objets référencés par les variables.

```

1 def pass_int(x):
2     print("Id of x inside fct before = :", id(x))
3     x = 42
4     print("Id of x inside fct after = :", id(x))
5     return x
6
7 def pass_tuple(t):
8     print("Id of t inside fct before = :", id(t))
9     t = (42, 23)
10    print("Id of t inside fct after = :", id(t))
11    return t

```

code/chap3/pass\_immu\_print.py

Code 66: Passage d'objet immuable

```

1 import pass_immu_print
2
3 x = 2
4 print("Id of x outside fct before calling :", id(x))
5 pass_immu_print.pass_int(x)
6 print("Id of x outside fct after calling :", id(x))
7 print(x)
8 print()
9
10 t = (1,2)
11 print("Id of t outside fct before calling :", id(t))
12 pass_immu_print.pass_tuple(t)
13 print("Id of t outside fct after calling :", id(t))
14 print(t)

```

code/chap3/pass\_immu\_print\_test.py

Code 67: Appel de fonction avec des objets immuables

Exécuter le code 67 produit par exemple l'affichage suivant :

9. En pratique, on a vu que l'on ne changeait pas l'objet, mais la variable qui le référençait.

### 3.3. PASSAGE DE PARAMÈTRES

```
Id of x outside fct before calling : 139845688099088
Id of x inside fct before = : 139845688099088
Id of x inside fct after = : 139845688100368
Id of x outside fct after calling : 139845688099088
2

Id of t outside fct before calling : 139845687348672
Id of t inside fct before = : 139845687348672
Id of t inside fct after = : 139845686080128
Id of t outside fct after calling : 139845687348672
(1, 2)
```

Concrètement, l'exécution se produit comme suit :

**ligne 3** : on crée un nouvel objet 2 avec son identité ...088 et on l'affecte à **x** ;

**ligne 5** : on appelle la fonction `pass_int` ;

1. on entre dans la fonction `pass_int` : l'identité de **x** n'a pas changé ;
2. on affecte 42 à **x** : on crée un nouvel objet  $x'$  avec l'identité ...368 ;
3. cet objet est dénoté dans la fonction par la variable **x** : l'objet **x** original de l'extérieur de la fonction est donc caché ;
4. on affecte 42 à cet objet, l'identité de **x** a donc changé *dans la fonction* ;
5. on retourne l'objet ...368 sans l'affecter : l'objet ...088 est toujours référencé par **x** en dehors de la fonction.

Ce comportement est illustré à la figure 3.5, et est identique pour **t** dans la fonction `pass_tuple`.

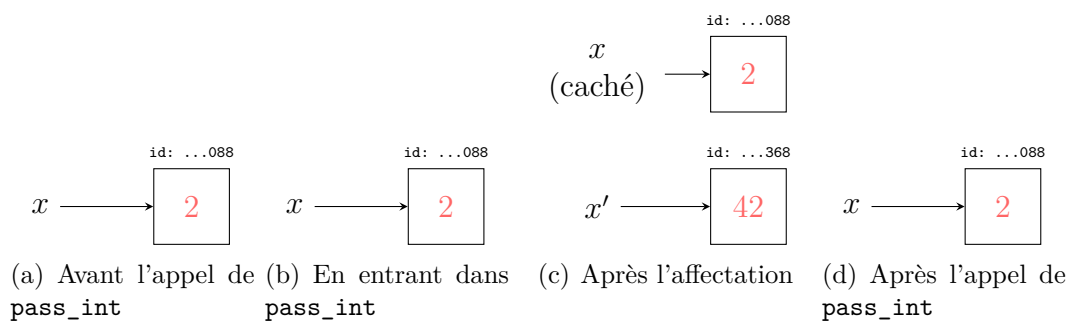


FIGURE 3.5 – Illustration du passage d'une variable à une fonction

#### 3.3.4 Passage d'objets muables comme paramètres

Le passage d'objets muables se passe sensiblement de la même manière que des objets qui ne le sont pas, à ceci près que ces objets peuvent être modifiés, et donc que de telles fonctions peuvent produire des effets de bord.

Les codes 68 et 69 illustrent ce comportement, et exécuter le code 69 produit par exemple l'affichage suivant :



```

Id of l1 outside fct : 140591919931712
Id of l2 outside fct : 140591920661888

Id of l inside fct before set : 140591919931712
Id of l inside fct after set : 140591919931712
[42, 2, 3]
Id of l1 outside fct : 140591919931712

Id of l1 inside fct before = : 140591919931712
Id of l2 inside fct after = : 140591920661888
[42, 2, 3]
Id of l1 outside fct : 140591919931712

```

```

1 def change_first_elem(l, i):
2     print("Id of l inside fct before set :", id(l))
3     l[0] = i
4     print("Id of l inside fct after set :", id(l))
5
6 def reaffect(l1, l2):
7     print("Id of l1 inside fct before =", id(l1))
8     l1 = l2
9     print("Id of l1 inside fct after =", id(l1))

```

code/chap3/pass\_mu\_print.py

Code 68: Passage d'objet muable

```

1 import pass_mu_print
2
3 l1 = [1, 2, 3]
4 l2 = [23, 42, 404]
5 print("Id of l1 outside fct :", id(l1))
6 print("Id of l2 outside fct :", id(l2))
7 print()
8
9 pass_mu_print.change_first_elem(l1, 42)
10 print(l1)
11 print("Id of l1 outside fct :", id(l1))
12 print()
13
14 pass_mu_print.reaffect(l1, l2)
15 print(l1)
16 print("Id of l1 outside fct :", id(l1))
17 print()

```

code/chap3/pass\_mu\_print\_test.py

Code 69: Passage d'objet muable

Concrètement, l'exécution se produit comme suit :

**lignes 5-6** : les listes l1 et l2 sont deux objets distincts, elles ont donc deux identités différentes ;

**ligne 9** : appel de la fonction `change_first_elem` :

1. on entre dans la fonction `change_first_elem` : l'identité de l1 n'a pas changé ;

### 3.3. PASSAGE DE PARAMÈTRES

2. on change le premier élément de `l1` en `42` ;
3. on n'a pas créé de nouvel objet (contrairement à une affectation) : l'identité de `l1` n'a pas changé dans la fonction ;

**ligne 10** : on imprime `l1`, ce qui affiche `[42, 2, 3]` : c'est un effet de bord attendu, car comme l'identité de `l1` n'a pas changé, les modifications sont visibles depuis l'extérieur de la fonction `change_first_elem` ;

**ligne 11** : l'identité de `l1` n'a pas changé après l'appel ;

**ligne 14** : appel de la fonction `reaffect` :

1. on entre dans la fonction `reaffect` : l'identité de `l1` n'a pas changé ;
2. on affecte `l1` en `l2` : on crée un nouvel objet `l1'` avec l'identité de `l2`
3. cet objet est dénoté dans la fonction par la variable `l1` : l'objet `l1` original de l'extérieur de la fonction est donc caché ;
4. on affecte `l2` à cet objet, l'identité de `l1` a donc changé *dans la fonction* ;

**ligne 15** : on imprime `l1`, ce qui affiche `[42, 2, 3]` : il n'y a pas d'effet de bord, car comme l'identité de `l1` a changé, les modifications ne sont pas visibles depuis l'extérieur de la fonction `reaffect`.

#### 3.3.5 Ordre d'évaluation

À l'évidence, il est nécessaire d'*évaluer* les paramètres d'une fonction avant de l'appeler, sinon, elle serait incapable d'effectuer son travail : comment pourrait-on calculer `abs(x)` si l'on ne connaît pas la valeur de `x` ?

Dans le cas où une fonction a plusieurs paramètres, ils sont évalués *de gauche à droite*. Ainsi, si l'on effectue un appel `f(x,y,z)`,

1. on évalue `x` ;
2. on évalue `y` ;
3. on évalue `z` ;
4. on appelle `f` avec les valeurs de `x`, `y` et `z`.

Ce comportement est illustré par les codes 70 et 71.

L'appel à la fonction `f` avec les paramètres `g1()`, `g2()` et `g3()`

1. évalue l'argument `g1()`, et donc appelle la fonction `g1`, ce qui affiche `"Calling g1"` et retourne `1` ;
2. évalue l'argument `g2()`, et donc appelle la fonction `g2`, ce qui affiche `"Calling g2"` et retourne `2` ;
3. évalue l'argument `g3()`, et donc appelle la fonction `g3`, ce qui affiche `"Calling g3"` et retourne `3` ;
4. appelle `f` avec les paramètres `1`, `2` et `3`, ce qui affiche `"Calling f"` et retourne `0`.

Cette valeur `0` est ensuite affichée en console par `print`.

```

1 def f(x, y, z):
2     print("Calling f")
3     return 0
4
5 def g1():
6     print("Calling g1")
7     return 1
8
9 def g2():
10    print("Calling g2")
11    return 2
12
13 def g3():
14    print("Calling g3")
15    return 3

```

Code 70: Ordre d'évaluation

```

1 import order_eval
2
3 print(order_eval.f(order_eval.g1(), order_eval.g2(), order_eval.g3()))

```

Code 71: Ordre d'évaluation

## 3.4 Portée de variables

Une variable ne peut pas être utilisée partout dans le code : elle a une certaine *portée*. La portée d'une variable est la portion de code où elle peut être utilisée via son nom.

En Python, il existe deux *portées*<sup>10</sup> de variables : *fonction*, et *globale*.

### 3.4.1 Portée fonction

Une variable **x** créée au sein d'une fonction est *locale* à cette fonction : le nom existe partout dans le corps de la fonction à partir de l'affectation.

Ce comportement est illustré par les codes 72, 73 et 74.

```

1 def f():
2     x = 10
3     print(x)
4
5 f()
6 print(x)

```

Code 72: Portée fonction

Dans le code 72, lors de l'appel à la fonction **f** à la ligne 5,

- ▷ on essaye d'imprimer **x** à la ligne 3 : ça fonctionne, car **x** est une variable déclarée dans une fonction, on peut donc l'utiliser partout dans cette fonction ;

10. Parfois, la portée est appelée *scope*, comme en anglais.

### 3.4. PORTÉE DE VARIABLES

```
code/chap3/function_scope2.py
1 def f():
2     print(x)
3     x = 10
4     print(x)
5
6 f()
```

Code 73: Portée fonction

```
code/chap3/function_scope3.py
1 def f():
2     x = 10
3     print(x)
4
5 f()
```

Code 74: Portée fonction

- ▷ on essaye d'imprimer `x` à la ligne 6 : cela provoque une erreur de syntaxe, car le nom `x` n'existe pas en dehors du corps de la fonction `f`.

Dans ce cas, l'interpréteur affiche un message d'erreur explicite : « `NameError: name 'x' is not defined` ».

Dans le code 73, lors de l'appel à la fonction `f` à la ligne 6, on essaye d'imprimer `x` à la ligne 2 : cela provoque une erreur de syntaxe, car même si `x` peut être utilisée dans le corps de la fonction, elle n'a pas encore été affectée à une valeur. Dans ce cas, l'interpréteur affiche un message d'erreur explicite : « `UnboundLocalError: local variable 'x' referenced before assignment` ».

Le code 74 fonctionne quant à lui très bien : l'appel à la fonction `f` de la ligne 5 imprime la valeur de `x` à la ligne 3, car `x` est locale.

Finalement, contrairement à d'autres langages, le fait qu'une variable soit locale n'est pas lié à un *bloc* : uniquement au fait qu'elle soit déclarée dans une fonction ou pas. Par exemple, dans de nombreux langages, le code 75 provoquerait une erreur. Dans ces langages, la variable `x` est déclarée au sein d'un bloc `if`, et a donc une portée locale à ce bloc. Pas en Python : la variable `x` est déclarée au sein d'une fonction `f`, et peut donc être utilisée partout dans ce bloc depuis son affectation. Ce code s'exécute donc parfaitement, et affiche **Even**.

```
code/chap3/function_scope4.py
1 def print_parity(x):
2     if x % 2 == 0:
3         s = "Even"
4     else:
5         s = "Odd"
6     print(s)
7
8 print_parity(10)
```

Code 75: Portée locale

### 3.4.2 Portée globale

À défaut de créer des variables dans le corps d'une fonction, il est possible d'en créer en dehors de toute fonction. Ces variables ont une portée *globale*, c'est-à-dire qu'on peut les utiliser partout depuis leur déclaration.

Ce comportement est illustré par les codes 76 et 77.

```

1 x = 2
2
3 def f():
4     print(x)
code/chap3/global_scope.py

```

Code 76: Portée globale

```

1 import global_scope
2
3 global_scope.f()
4 print(global_scope.x)
code/chap3/global_scope_test.py

```

Code 77: Portée globale

Lors de l'exécution du code 77,

- ▷ à la ligne 3, on appelle la fonction `f`, qui imprime la variable `x`, ce qui fonctionne car cette variable est globale ;
- ▷ à la ligne 4, on imprime directement la variable `x`, ce qui fonctionne car cette variable est globale.

Notons que si l'on essaie d'affecter une variable globale au sein d'une fonction, aucun effet de bord n'apparaît (la valeur reste inchangée), à cause des changements d'identité détaillés aux sections 3.3.1 et 3.3.3. Ce comportement est illustré par le code 78.

```

1 x = 2
2
3 def f():
4     x = 3
5     print(x)
6
7 f()
8 print(x)
code/chap3/global_scope2.py

```

Code 78: Aucun changement pour la variable globale

Lors de l'exécution de ce code, la ligne 7 appelle la fonction `f` et,

- ▷ la ligne 4 crée un nouvel objet, de valeur 3 et à l'identité distincte de la variable globale `x`, et l'affecte à une variable `x` cachant la globale ;
- ▷ la ligne 5 imprime la valeur de ce nouvel objet, et affiche 3.

Finalement, la ligne 8 imprime la valeur de la variable globale `x`, c'est-à-dire 2.

Si l'on souhaite véritablement changer la valeur de la variable globale `x`, il faut préciser explicitement au compilateur que c'est de cette variable que l'on parle, grâce au mot-clé `global`, tel qu'illustré au code 79.

```

1 x = 2
2
3 def f():
4     global x
5     x = 3
6     print(x)
7
8 f()
9 print(x)

```

code/chap3/global\_scope3.py

Code 79: Affecter une variable globale dans une fonction

C'est une mauvaise pratique de déclarer des variables globales.

En effet, ces variables peuvent potentiellement être modifiées partout dans un programme, ce qui peut rendre son débogage particulièrement difficile. À part dans de très rares cas, par exemple pour déclarer une variable globale `pi = 3.1416`, on évitera donc d'en déclarer.

#### Information

En Python, il est également possible de créer des variables globales directement au sein de fonctions. Comme c'est une mauvaise pratique en général et que cela dépasse le cadre de ce cours, ce ne sera pas abordé.

## 3.5 Flot d'exécution et pile d'exécution

Au sein d'un fichier, l'ordre dans lequel les instructions sont exécutées est appelé un *flot d'exécution*. L'exécution commence toujours à la première instruction, et les instructions suivantes sont exécutées ensuite, de haut en bas, l'une après l'autre.

À l'évidence, une fonction doit être définie avant d'être appelée, au même titre qu'une variable doit être définie avant d'être utilisée. Notez toutefois que la définition de fonction n'altère pas le flot d'exécution : les fonctions en question sont « disponibles » pour un appel, mais leur corps n'est exécuté que si elles sont appelées explicitement. Ainsi, un appel de fonction peut être vu comme un « détour » dans un programme.

Notons que même si une fonction doit être définie avant d'être appelée, on parle ici d'appel au sens « exécution » : en Python, l'ordre dans lequel les fonctions sont définies *n'a pas d'importance*, tel qu'illustré par les codes 80 et 81.

Exécuter ces deux codes produit le même résultat, et ne provoque pas d'erreur, et ceci même si la fonction `f`, appelée par la fonction `g`, est définie après `g` (code

```

1 def f():
2     print("f")
3
4 def g():
5     f()
6     print("g")
7
8 def h():
9     print("h")
10
11 g()

```

code/chap3/def\_order1.py

Code 80: L'ordre de définition des fonctions est sans importance

```

1 def g():
2     f()
3     print("g")
4
5 def f():
6     print("f")
7
8 def h():
9     print("h")
10
11 g()

```

code/chap3/def\_order2.py

Code 81: L'ordre de définition des fonctions est sans importance

81). Dans les deux cas, on remarque aussi que la fonction `h` n'est *jamais* appelée, son corps n'est donc jamais exécuté.

Concrètement, ce flot d'exécution est illustré à la figure 3.6 :

1. le *point d'entrée* du programme est la ligne 11 : on commence par là et on appelle `g` (figure 3.6(a)) ;
2. la première instruction de `g` appelle ensuite `f` (figure 3.6(b)) ;
3. on exécute le corps de la fonction `f` exécuté : on affiche `f` (figure 3.6(c)) ;
4. on *retourne à l'appelant*, c'est-à-dire on revient dans la fonction `g` (figure 3.6(d)), à l'instruction qui suit l'appel à `f` ;
5. on exécute la suite du corps de `g` et on affiche donc `"g"` (figure 3.6(e)) ;
6. finalement, une fois le corps de `g` exécuté, on retourne à l'appelant, et on revient « après » la ligne 11 : il n'y a plus d'instructions, l'exécution est donc terminée (figure 3.6(f)).

Sur cette figure, on remarque que les appels de fonctions « s'empilent » : on parle de *pile d'exécution*.

Si une erreur se produit lors de l'exécution d'un programme, Python affiche la pile d'exécution courante, tel qu'illustré au code 82.

Lors de l'exécution de ce code,

1. la ligne 7 appelle `f`, et exécute la ligne 2 ;

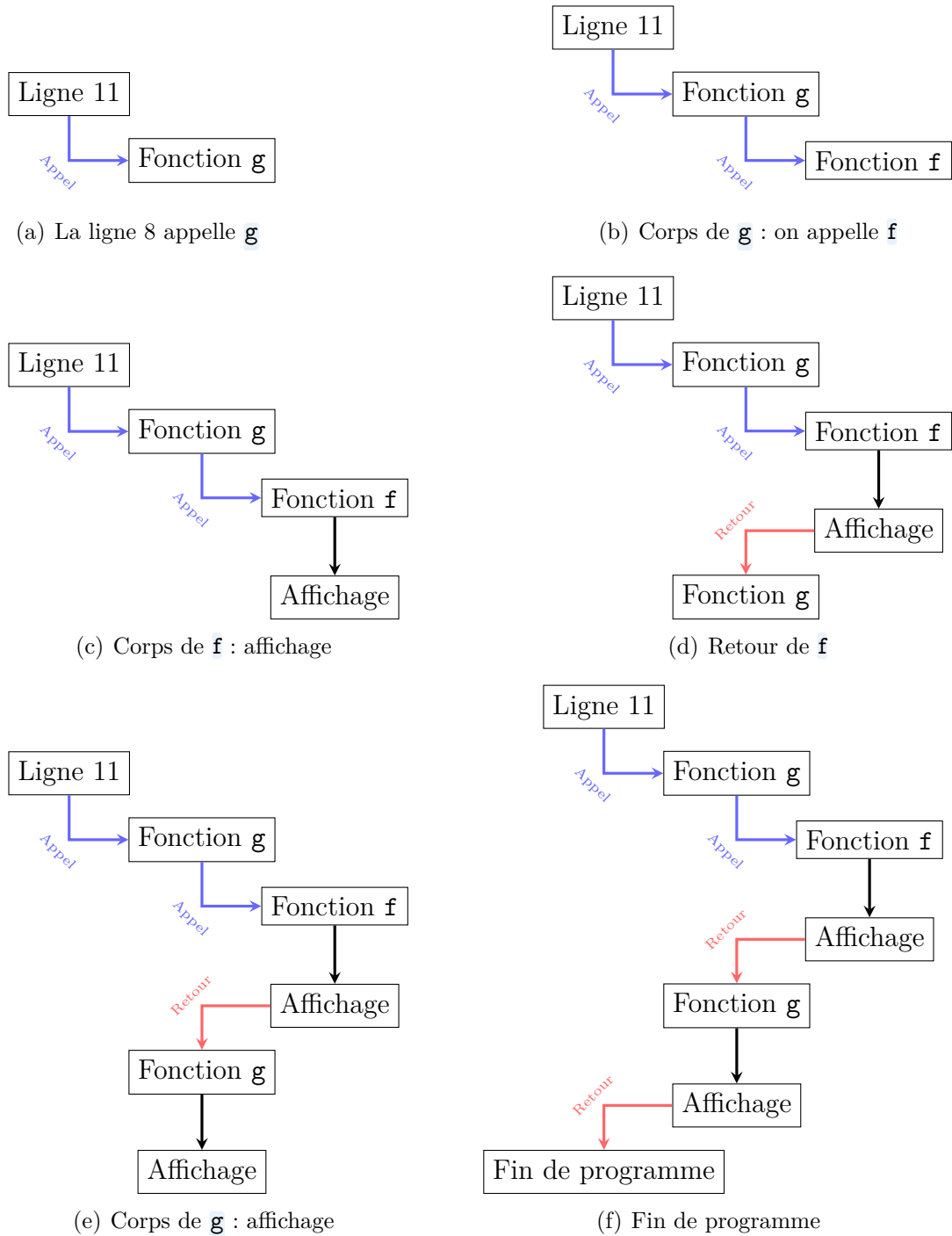


FIGURE 3.6 – Flot d'exécution des codes 80 et 81

```

1 def f(x):
2     return x + g(x)
3
4 def g(y):
5     return x * y
6
7 print(f(2))

```

code/chap3/fct\_stack\_error.py

Code 82: Affichage de la pile d'exécution courante



2. la ligne 2 appelle `g`, et exécute la ligne 5;
3. la ligne 5 provoque une erreur car le symbole `x` n'est pas défini;
4. l'interpréteur affiche la pile d'exécution courante.

Concrètement, exécuter ce code affiche la pile suivante :

```
Traceback (most recent call last):
  File ".../code/chap3/fct_stack_error.py", line 7, in <module>
    print(f(2))
  File ".../code/chap3/fct_stack_error.py", line 2, in f
    return x + g(x)
  File ".../code/chap3/fct_stack_error.py", line 5, in g
    return x * y
NameError: name 'x' is not defined
```



# Modules, packages et bonnes pratiques

Souvent, un programmeur n'écrit pas l'intégralité de son code dans un unique fichier mais l'organise en différents modules. Diviser un programme plus ou moins complexe en modules et packages offre divers avantages.

1. **Divide and conquer** : plutôt que de s'attaquer à l'intégralité de la résolution d'un problème, ou à la conception d'un système complexe, un module est typiquement responsable d'une partie de ce problème ou de ce système. Ainsi, le programmeur peut se consacrer entièrement à des éléments plus restreints, ce qui facilite le développement et est moins sujet aux erreurs.
2. **Maintenabilité** : la stratégie diviser et conquérir partitionne un programme en modules, qui sont désignés de telle sorte à ce que chaque module soit responsable d'une partie logique du programme. En plus de cela, les modules sont typiquement implémentés de manière à limiter leurs inter-dépendances<sup>1</sup>. Procéder de cette manière limite les chances qu'une modification dans un module ait un impact dans le reste du programme (comme modifier un autre module), ce qui facilite la maintenance du programme quand celui-ci grandit.
3. **Réutilisation** : les fonctions définies au sein d'un module sont facilement réutilisables, ce qui limite la duplication<sup>2</sup> de code. On évite en général de dupliquer du code pour des raisons de maintenabilité. Par exemple, si on doit retravailler un extrait de code dupliqué parce qu'il y a une erreur, il faut classiquement retravailler toutes les copies.
4. **Espaces de noms** : les entités définies dans un module le sont par le biais du nom du module, ce qui évite les collisions de symboles lors de l'importation. Sans module, il est par exemple impossible de définir deux fonctions `f` : elles ont le même nom, l'interpréteur ne peut donc pas savoir laquelle appeler (cf. section 3.2.3). Toutefois, si on définit une fonction `f` dans un module `m1`, et une fonction `f` dans un module `m2`, on peut appeler chacune des fonctions.

Vous verrez dans vos cours d'analyse diverses techniques afin de savoir comment découper un système en sous-systèmes afin qu'il ait les avantages ci-dessus. On se consacrera dans ce chapitre uniquement à la syntaxe relative aux modules et packages.

---

1. On dit qu'un module *x* *dépend* d'un autre module *y* si *x* a besoin de *y* pour fonctionner.

2. Les « copier / coller ».

Par ailleurs, lorsque l'on programme, il convient de respecter une certaine hygiène en termes de qualité de code, de nommage, etc. Ces bonnes pratiques sont également présentées dans ce chapitre.

### 4.1 Modules

En Python, on peut utiliser un module de trois façons :

1. importer un module de la distribution Python installée sur la machine,
2. l'écrire soi-même en Python et l'importer,
3. l'écrire soi-même dans un autre langage, comme le C++ ou le Rust, et l'importer.

On se concentrera sur les deux premiers cas.

On remarque qu'il est dans tous les cas nécessaire d'*importer* un module *avant* de l'utiliser.

#### 4.1.1 Un résumé simple

Créer des modules en Python est très facile. Considérons par exemple le module du code 83, enregistré dans un fichier `mod_example.py`.

```
code/chap4/mod_example.py
1 s = "Follow the white rabbit"
2 origin = (0,0)
3
4 def hello_world():
5     print("Hello World!")
```

Code 83: Exemple de module

Ce module définit trois objets :

- ▷ `s` : une chaîne de caractères,
- ▷ `origin` : un tuple,
- ▷ `hello_world` : une fonction.

On remarque qu'en soi, ce module ne produit aucun affichage, et n'effectue aucun calcul : il ne fait que définir des objets que l'on peut appeler. C'est un critère de qualité important.

Si l'on veut utiliser ce module dans un script, il suffit de l'importer à l'aide de l'instruction `import`, tel qu'illustré au code 84.

```
code/chap4/mod_example_test.py
1 import mod_example
2
3 print(mod_example.s)
4 print(mod_example.origin)
5 mod_example.hello_world()
```

Code 84: Exemple d'utilisation de module

### 4.1.2 Recherche de modules

Quand l'interpréteur exécute une instruction `import mod`, il recherche un fichier `mod.py` dans les emplacements suivants :

- ▷ en mode script, dans le répertoire dans lequel se trouve le fichier qui utilise l'instruction `import`,
- ▷ en mode interactif, dans le répertoire courant,
- ▷ une liste de répertoires contenue dans la variable d'environnement <sup>3</sup> `PYTHONPATH`, si cette variable existe,
- ▷ une liste de répertoires définie par la distribution Python lors de son installation.

Si le module n'est pas trouvé par la procédure ci-dessus, l'interpréteur renvoie une erreur. Le chemin de recherche complet peut être obtenu via la variable `sys.path`, tel qu'illustré au code 85. Notons que le contenu de cette variable est dépendant de votre installation, il est donc probable que vous n'obteniez pas le même affichage.

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

Code 85: Chemin de recharge des modules

Ainsi, si vous souhaitez que l'interpréteur trouve votre module, vous pouvez

- ▷ le placer dans le répertoire courant (en mode interactif), ou dans le répertoire du script appelant (en mode script),
- ▷ le placer dans un répertoire reconnu par votre distribution,
- ▷ modifier la variable `PYTHONPATH`.

On se concentrera sur la première façon et ses variantes.

### 4.1.3 L'instruction `import`

On a déjà vu que l'instruction `import <module_name>` importe un module nommé `<module_name>`, sous réserve de la disponibilité de ce module (cf. section 4.1.2).

Notons que le seul effet de cette instruction est de placer `<module_name>` dans la table des symboles du script qui utilise cette instruction : quand l'interpréteur rencontre le symbole `<module_name>`, il sait que c'est un symbole qui existe. Les noms des objets contenus dans ce module ne sont pas disponibles directement

3. Une variable d'environnement est une variable définie au sein du système d'exploitation et qui est classiquement utilisée par les processus en cours d'exécution afin de définir leur environnement, tel que les endroits où ils peuvent trouver les ressources dont ils ont besoin.

depuis le script appelant, on doit utiliser la notation « . », pour par exemple appeler `<module_name>.foo` pour obtenir l'objet `foo` du module `<module_name>`.

Sur les codes 83 et 84 (page 60), on remarque donc, lors de l'exécution du code 84,

**ligne 1** `import mod_example` place `mod_example` dans la table des symboles du code 84,

- ▷ on ne peut pas obtenir directement les objets `s`, `origin` et `hello_world` du code 83,

- ▷ écrire `print(s)` provoque une erreur ;

**ligne 3** l'interpréteur rencontre le symbole `mod_example` : c'est un symbole connu,

- ▷ on récupère l'objet `s` du module `mod_example`,

- ▷ on affiche cet objet `s` avec `print(mod_example.s)`.

Le code poursuit son exécution de manière similaire aux lignes 4 et 5. Si l'on avait directement tenté d'utiliser par exemple le symbole `s`, l'interpréteur aurait provoqué une erreur, tel qu'illustré au code 86.

```
>>> import mod_example
>>> print(s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 's' is not defined
>>>
```

Code 86: Erreur lors de l'utilisation de modules

### L'instruction `from <module> import <name>`

Il est également possible d'importer un objet d'un module directement dans la table des symboles du script appelant, grâce à l'instruction

```
from <module_name> import <name>
```

Cette technique permet d'utiliser directement l'objet `<name>` dans le script appelant, sans utiliser la notation « . », tel qu'illustré par les codes 83 (page 60) et 87.

```
1 from mod_example import s
2 from mod_example import origin
3 from mod_example import hello_world
4
5 print(s)
6 print(origin)
7 hello_world()
```

Code 87: Exemple d'importation d'objets

Notons que les trois premières lignes du code 87 peuvent être remplacée par une unique instruction `from mod_example import s, origin, hello_world`.

## Conflits de noms

Bien qu'elle semble pratique, cette approche offre un inconvénient : comme les symboles sont placés directement dans la table des symboles du script appelant, on pourrait créer un conflit en important deux objets de même nom (de deux modules différents), comme illustré aux codes 88, 89, et 90, ou en important un objet du même nom qu'un autre objet du script appelant, tel qu'illustré aux codes 88 et 91.

```
1 s = "Is this just real life?"
```

code/chap4/mod\_conflict1.py

Code 88: Un module avec un objet `s`

```
1 s = "Is this just fantasy?"
```

code/chap4/mod\_conflict2.py

Code 89: Un module avec un autre objet `s`

```
1 from mod_conflict1 import s
2 from mod_conflict2 import s
3
4 print(s)
```

code/chap4/mod\_conflict\_test1.py

Code 90: L'objet `s` est écrasé

```
1 from mod_conflict1 import s
2
3 s = "Mamaaaaaa"
4
5 print(s)
```

code/chap4/mod\_conflict\_test2.py

Code 91: L'objet `s` est écrasé

Dans les deux cas de figure, c'est le dernier objet défini qui est utilisé. Ainsi, le code 90 affiche « `Is this just fantasy?` » et le code 91 affiche « `Mamaaaaaa` ».

Ce type de conflit de nom peut être résolu avec l'instruction

```
from <module_name> import <name> as <label>
```

ce qui a comme effet d'importer l'objet `<name>` du module `<module_name>` dans la table des symboles du script appelant sous le nom `<label>`. Ainsi, si par exemple on retravaille le code 90 légèrement pour obtenir le code 92, on peut utiliser les deux chaînes `s` sans problème.

```
1 from mod_conflict1 import s as s1
2 from mod_conflict2 import s as s2
3
4 print(s1)
5 print(s2)
```

code/chap4/mod\_conflict\_test3.py

Code 92: Le conflit de nom a disparu

À titre indicatif, notez qu'on peut également importer un module entier sous un autre nom, avec une instruction

```
import <module_name> as <label>
```

tel qu'illustré au code 93. Cette instruction est principalement utilisée pour raccourcir les noms de modules que le programmeur trouverait trop longs.

```
1 import math as m
2
3 print(m.pi)
```

code/chap4/mod\_label.py

Code 93: Importer un module sous un autre nom

## 4.2 Packages

Quand un projet grandit, on peut être amené à créer divers modules, ce qui peut être fastidieux à maintenir si on les stocke tous au même endroit, en particulier s'ils ont des noms ou des fonctionnalités similaires. Ainsi, dans une continuité de la stratégie Divide and Conquer, Python permet de grouper des modules en *packages*.

Les packages permettent une organisation hiérarchique des modules, et on peut les importer de manière similaire aux modules, à l'aide d'instructions `import` et de la notation « . ».

Créer un package est très facile : Python se sert de la hiérarchie du système de fichiers du système d'exploitation. En résumé,

- ▷ pour créer un module, on crée habituellement un fichier ;
- ▷ pour créer un package, on crée habituellement un dossier dans lequel on place les modules.

Considérons par exemple la hiérarchie suivante d'un projet `myproject` :

```
myproject/
├── package1/
│   ├── mod1.py
│   └── mod2.py
├── package2/
│   ├── mod3.py
│   └── mod4.py
└── main.py
```

On peut utiliser entre autres les fonctions du module `mod1` dans `main.py` grâce à une instruction `import package1.mod1`.

Le code complet de cet exemple est fourni à titre indicatif aux codes 94, 95, 96, 97 et 98.

Comme précédemment, on remarque qu'effectuer une telle importation ne place que `package1.mod1` dans la table des symboles de `main.py`, pas les objets qui y sont déclarés. Pour les utiliser, on peut de nouveau



```
1 def f():  
2     print("Hello")
```

code/chap4/myproject/package1/mod1.py

Code 94: Le module `mod1` du package `package1`

```
1 def g():  
2     print("There")
```

code/chap4/myproject/package1/mod2.py

Code 95: Le module `mod2` du package `package1`

```
1 def h():  
2     print("General")
```

code/chap4/myproject/package2/mod3.py

Code 96: Le module `mod3` du package `package2`

```
1 def i():  
2     print("Kenobi")
```

code/chap4/myproject/package2/mod4.py

Code 97: Le module `mod4` du package `package2`

```
1 import package1.mod1  
2 import package1.mod2  
3 import package2.mod3  
4 import package2.mod4  
5  
6 package1.mod1.f()  
7 package1.mod2.g()  
8 package2.mod3.h()  
9 package2.mod4.i()  
10 print("You are a bold one!")
```

code/chap4/myproject/main.py

Code 98: Le script `main.py`

- ▷ soit les référencer avec la notation « . » (ce qui est fait sur l'exemple ci-dessus) ;
- ▷ soit les importer avec l'instruction `from <stuff> import <name>`.

Sur cet exemple, on peut par exemple dans `main.py`

- ▷ remplacer la ligne 1 par l'instruction `from package1.mod1 import f` ;
- ▷ remplacer la ligne 6 directement par l'instruction `f()`.

Sur l'exemple ci-dessus, notons que tous les modules sont stockés dans des fichiers de noms différents, et y déclarent des fonctions de noms différents. En pratique, ce n'est pas obligatoire :

- ▷ des modules de packages différents peuvent avoir des noms identiques ;
- ▷ des fonctions de modules différents peuvent avoir des noms identiques.

En effet, la notation « . » permet systématiquement d'éviter les conflits de noms. Si on en crée néanmoins à l'aide de l'instruction `from <stuff> import <name>`, on peut les régler comme précédemment en « renommant » les objets importés grâce à l'instruction `from <stuff> import <name> as <label>`.

### 4.2.1 Initialisation

On remarque au vu des remarques précédentes qu'importer un package directement (et pas un module qui s'y trouve) est ainsi globalement inutile : ainsi, écrire une instruction `import package1` place `package1` dans la table des symboles du script appelant, mais pas les modules de ce package. Parfois, on voudrait importer un package dans son intégralité, sans devoir écrire une ligne par module.

À ce titre, si un package contient un fichier `__init__.py`, les instructions qui s'y trouvent sont exécutées quand le package ou l'un de ses modules est importé. On utilise habituellement ce fichier pour effectuer des importations implicites de modules et y définir des variables globales.

Un exemple de projet complet est illustré aux codes 99, 100, 101 et 102. Ce projet a la structure suivante :

```
mysecondproject/
├── super_package/
│   ├── __init__.py
│   ├── stuff.py
│   └── blah.py
└── main.py
```

```
1 s = "I'm afraid, Dave."
2
3 from super_package.stuff import f as hal
4 import super_package.blah
```

code/chap4/mysecondproject/super\_package/\_\_init\_\_.py

Code 99: Le fichier `__init__.py` du package `super_package`

Notons que les instructions qui sont écrites dans un fichier `__init__.py` sont exécutées *une unique fois*, indépendamment du nombre d'importations qui sont effectuées par les scripts appelants. Ainsi, les codes 103 et 104 n'impriment qu'une

```

1 def f():
2     print("Dave, my mind is going.")
3
4 def g():
5     print("Open the pod bay doors!")

```

code/chap4/mysecondproject/super\_package/stuff.py

Code 100: Le module `stuff`

```

1 def f():
2     print("I can feel it.")

```

code/chap4/mysecondproject/super\_package/blah.py

Code 101: Le module `blah`

```

1 import super_package
2
3 print(super_package.s)
4 super_package.hal()
5 super_package.blah.f()

```

code/chap4/mysecondproject/main.py

Code 102: Le script principal

fois la chaîne « Only try to realize the truth », bien que le package soit importé deux fois<sup>4</sup> dans `main.py`. La structure du projet associé est la suivante :

```

mythirdproject/
├── pack/
│   ├── __init__.py
│   └── main.py

```

```

1 print("Only try to realize the truth")

```

code/chap4/mythirdproject/pack/\_\_init\_\_.py

Code 103: Le fichier `__init__.py` du package `pack`

```

1 import pack
2 import pack
3
4 print("There is no spoon")

```

code/chap4/mythirdproject/main.py

Code 104: Le script principal

## 4.3 Commentaires et documentation

Le code est plus souvent *lu* qu'il n'est *écrit*. Pour cette raison, il doit être facile de comprendre rapidement le fonctionnement d'un extrait de code en le lisant. Une manière de faciliter cette compréhension est de *documenter* le code.

Pour documenter du code, une pratique courante dans tous les langages est d'y inclure des *commentaires*. En Python, les lignes précédées du caractère `#` sont

4. À l'évidence, les deux importations effectuées sont artificielles, car au sein d'un même fichier. En pratique, ce cas de figure se produit quand deux modules importent le même package. Pour des raisons de simplicité, cet exemple a été réduit au minimum.

considérées comme des commentaires, et ne sont pas exécutées. Le code 105 illustre un exemple de code commenté.

```
1 #start at step 2
2 i = 2
3
4 i = i + 0. #make sure the result is float
5
6 print(i)
```

code/chap4/comment.py

Code 105: Exemple de commentaires

L'exemple ci-dessus est académique. En pratique, on utilise des commentaires pour

- ▷ planifier : quand on écrit un bout de code complexe, c'est parfois utile de décrire les étapes de l'implémentation à vertu de planification (ces commentaires sont habituellement supprimés une fois le code écrit) ;
- ▷ décrire du code : expliquer l'objectif de l'extrait courant ;
- ▷ décrire un algorithme : lors que l'on utilise un algorithme (en particulier un algorithme complexe), décrire clairement son fonctionnement est utile, ainsi que la raison pour laquelle on a choisi cet algorithme<sup>5</sup> ;
- ▷ taguer : parfois, il reste un peu de travail à faire sur un extrait de code, et on marque ce code pour s'en souvenir, par exemple avec des tags « TODO », « FIXME », etc.

#### 4.3.1 Documenter ses fonctions

Parfois, il est nécessaire de documenter un extrait de code avec plus que quelques commentaires au sein même du code. Par exemple, lorsque l'on écrit une fonction, il est classique de

- ▷ décrire son fonctionnement général,
- ▷ décrire les paramètres,
- ▷ décrire les retours.

Ainsi, un développeur lisant ce typer de documentation sait exactement à quoi elle sert et comment l'utiliser.

En Python, on documente les fonctions à l'aide de *docstrings*. L'idée de base des docstrings est que ce sont une forme de commentaire qui respecte une certaine structure. En plus de cela, Python fournit la fonction `help` qui renvoie au programmeur le docstring associé à un objet.

Un exemple de l'utilisation de cette fonction est fournie au code 106.

Sur cet exemple, on voit que

---

5. Souvent, les raisons d'un tel choix sont liées aux performances de la tâche à réaliser : parfois, une approche naïve pour résoudre un problème est trop coûteuse en temps ou en mémoire, et il faut donc écrire un algorithme plus performant, et l'implémenter. L'écriture de tels algorithmes n'est pas l'objet de ce cours et ne sera donc pas abordée.

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
>>>
```

Code 106: Utilisation de la fonction help

- ▷ `abs` est une fonction qui prend un seul argument,
- ▷ `abs` est une fonction qui retourne la valeur absolue de l'argument.

Dans certains cas, cette documentation est plus verbeuse, pour les fonctions plus complexes, tel qu'illustré au code 107. Sur cet exemple, la description fournit également une idée du fonctionnement de l'algorithme sous-jacent, précise le rôle de chacun des paramètres (il y en a deux), et dit dans quels cas elle produit une erreur.

```
>>> import math
>>> help(math.comb)
Help on built-in function comb in module math:

comb(n, k, /)
    Number of ways to choose k items from n items without repetition and
    ↪ without order.

    Evaluates to  $n! / (k! * (n - k)!)$  when  $k \leq n$  and evaluates
    to zero when  $k > n$ .

    Also called the binomial coefficient because it is equivalent
    to the coefficient of k-th term in polynomial expansion of the
    expression  $(1 + x)^n$ .

    Raises TypeError if either of the arguments are not integers.
    Raises ValueError if either of the arguments are negative.
```

Code 107: Utilisation de la fonction help

Si l'on veut écrire soi-même la documentation d'une fonction, il faut donc écrire un docstring qui sera reconnu par la fonction `help`, en respectant un certain formatage. Dans ce cours, le standard choisi est *re-structured text*<sup>6</sup> (reST). Les contraintes de ce standard sont les suivantes :

- ▷ le docstring doit être contenu entre trois paires de double quotes (par exemple `"""Bonjour"""`);
- ▷ le docstring commence par une ligne fournissant une description brève de la fonction;
- ▷ le docstring est suivi par une description facultative plus longue;
- ▷ le docstring contient une section **Parameters** décrivant les paramètres éventuels;

6. <https://docutils.sourceforge.io/rst.html> - Dernier accès le 17 septembre 2023.

- ▷ le docstring contient une section **Returns** décrivant les retours éventuels ;
- ▷ le docstring contient une section **Raises** décrivant les erreurs éventuelles qu'elle provoque sur certaines valeurs de paramètres.

À l'évidence, si une fonction ne prend pas de paramètres, la section **Parameters** est facultative (même remarque pour **Returns** et **Raises**).

Le code 108 illustre une fonction documentée. On peut vérifier que

```
help(doc_fct.double_and_square)
```

fournit bien la description de la fonction (après avoir importé le module `doc_fct`).

```
1 def double_and_square(x):
2     """
3     Returns the squared double of the argument.
4
5     More precisely, computes and returns 2x * 2x = 4x**2
6
7     Parameters
8     -----
9     x: the argument to double and square
10
11     Returns
12     -----
13     x doubled and squared, that is, 4x**2
14
15     """
16     return 4*x**2
```

Code 108: Une fonction documentée

## 4.4 Conventions et bonnes pratiques

Souvent, écrire un code « qui fonctionne » ne suffit pas : au delà de critères pragmatiques tels que la performance, il faut également l'écrire en suivant des conventions et des bonnes pratiques, afin d'en augmenter la qualité, la lisibilité et la maintenabilité. Cette section détaille ces critères.

### 4.4.1 Commentaires

Il faut absolument éviter les commentaires qui contredisent le code (ils sont contre-productifs), et il faut les maintenir avec l'évolution du code.

Les commentaires doivent être des phrases complètes, et être typographiés comme tels, mais doivent rester brefs. Les commentaires *doivent* être écrits en *anglais*.

Les commentaires doivent se trouver aussi près que possible du code qu'ils commentent, et ne doivent pas contenir du formatage complexe (tel que des figures ASCII), qui a tendance à distraire le lecteur. Il faut également éviter la redondance.

La meilleure manière de commenter son code est de le designer pour qu'il se commente lui-même : en utilisant des noms de fonctions et de variables parlants, qui rendent la compréhension du code immédiate.

En conclusion, il faut se souvenir que les commentaires sont là pour le *lecteur* (ce qui inclut le concepteur), pour l'aider à comprendre l'objectif et le design du programme.

#### 4.4.2 Modules

Il faut éviter d'utiliser des noms de modules contenant le caractère « - » (tiret), car lors d'une importation, ce caractère est confondu avec l'opérateur « moins ». Pour séparer les parties logiques d'un nom de module, on favorise les abréviations, ou le caractère « \_ » (underscore).

Lors de la définition d'un module, il faut éviter d'imprimer en console ou de définir des variables en effectuant des calculs lourds. Dans le cas contraire, ces impressions et calculs lourds seront effectués lors de la première importation, ce qui est souvent non désiré.

#### 4.4.3 Conventions de style

Cette section est principalement tirée du PEP8<sup>7</sup>.

##### Indentation

Quand il faut indenter du code (lors de la définition de fonctions, de `if`, etc.), la convention dicte d'utiliser 4 espaces plutôt que des tabulations (ou un autre nombre d'espaces).

Le PEP8 fournit également d'autres recommandations en termes d'indentation de lignes de codes trop longues.

##### Longueur maximum d'une ligne

Une ligne de code doit être limitée à 79 caractères. Si on doit « casser » une ligne sur un opérateur binaire, il faut le faire *avant* cet opérateur. Ainsi, on privilège

```
income = (gross_wages
          + taxable_interest)
```

plutôt que

```
income = (gross_wages +
          taxable_interest)
```

##### Lignes vides

Il faut entourer les définitions de fonctions par deux lignes vides. Dans le corps d'une fonction, il convient également de séparer ses sections logiques par des lignes vides.

7. <https://peps.python.org/pep-0008/> - Dernier accès le 17 septembre 2023.

##### **Encodage**

Le code Python doit être encodé en UTF-8.

##### **Imports**

Les imports doivent habituellement être sur des lignes séparées.

##### **Espaces**

Il faut éviter de terminer une ligne par une espace, et de mettre une espace après une parenthèse ouvrante, ou avant une parenthèse fermante.



# Structures de contrôle

Un programme qui s'exécute toujours de la même façon du début à la fin n'a pas beaucoup d'intérêt. Souvent, on veut se doter d'outils permettant à nos programmes de se comporter de façon différente d'une exécution à l'autre ; en d'autres termes, on souhaite créer différents *flux* d'exécution et contrôler ceux-ci via des *structures de contrôle*.

## 5.1 Expression booléenne

Le type booléen a déjà été abordés au cours de chapitres précédents : il s'agit d'un type de donnée qui ne peut prendre que deux valeurs, à savoir *vrai* ou *faux*. Une expression booléenne est donc une expression qui retourne soit vrai, soit faux. Une telle expression peut consister de :

- ▷ Un *symbole* booléen

```
>>> True
True
```

- ▷ Le résultat d'un opérateur de *comparaison*. Cf. 2.3.2

```
>>> 5 < 2
False
>>> 2 <= 5 < 9
True
```

- ▷ Une variable contenant une valeur booléenne

```
>>> x = 5 < 2
>>> x
False
```

- ▷ Un appel à une fonction qui retourne un résultat booléen

```
>>> def est_paire(x):
...     return x % 2 == 0
...
>>> est_paire(3)
False
>>> est_paire(6)
True
```

- ▷ Le résultat d'un *opérateur logique*, qui combine deux expressions booléennes et retourne une nouvelle valeur booléenne. Cf. 2.3.2

```
>>> x = False
>>> 1 < 3 and x
False
>>> est_paire(5) or 5 > 2
True
>>> not x
True
```

Nous utiliserons les expressions booléennes afin d'exprimer les conditions qui nous permettront de contrôler nos flux d'exécution.

## 5.2 Conditions et alternatives

### 5.2.1 Instructions conditionnelles

Les *instructions conditionnelles* ou conditions permettent d'exécuter des instructions uniquement *si* une certaine condition est vérifiée.

```
1 x = int(input("Entrez un nombre entier x : "))
2 if x >= 0 :
3     print("x est positif")
4 print("Merci d'avoir participé !")
```

code/chap5/if\_alone.py

Code 109: Exemple de condition

Le code 109 récupère une valeur entrée par l'utilisateur au clavier, la convertit en nombre entier, et affiche **x est positif** si ce nombre est (non strictement) positif.

Remarquons immédiatement la syntaxe :

- ▷ le mot-clé **if** (« si » en anglais) démarre l'alternative ;
- ▷ suivi d'une expression à valeur booléenne, qui peut être aussi complexe qu'on le souhaite ;
- ▷ l'expression se termine par le caractère « : » ;
- ▷ la ou les lignes de code à exécuter si la condition est vérifiée est séparée par son *indentation*, de façon analogue au corps d'une fonction. Ce bloc de code doit contenir *au moins* une ligne de code ;
- ▷ la ligne de code 4, puisqu'elle n'est pas indentée, est exécutée quelle que soit la valeur entrée par l'utilisateur. Elle ne fait pas partie de la condition.

### 5.2.2 Alternatives

Souvent, il n'est pas suffisant d'exécuter du code si une condition est vérifiée ; on souhaite aussi exécuter un *autre* code quand la condition n'est *pas* vérifiée. On écrira alors qu'un bloc de code s'exécute *si* une condition est vérifiée, et qu'un autre bloc de code s'exécute *sinon*. On parle alors d'*alternative* :

Le code 110 est une version modifiée du code précédent : cette fois-ci, si le nombre entré par l'utilisateur est négatif, **x est négatif** s'affiche.

```

1 x = int(input("Entrez un nombre entier x : "))
2 if x >= 0 :
3     print("x est positif")
4 else :
5     print("x est négatif")

```

Code 110: Exemple d'alternative

Parfois, il n'est pas suffisant d'écrire deux cas de figure mutuellement exclusifs. À la place, on souhaitera, si la première condition n'est pas vérifiée, tester une *deuxième* condition et créer deux autres alternatives (ou plus!) :

```

1 x = int(input("Entrez un nombre entier x : "))
2 if x > 0 :
3     print("x est positif")
4 elif x < 0 :
5     print("x est négatif")
6 else :
7     print("x est nul")

```

Code 111: Exemple d'alternative multiple

Le code 111 est une dernière modification : à présent, les nombres strictement positifs sont séparés de la valeur 0, qui affiche `x est nul`.

Au niveau de la syntaxe :

- ▷ le mot-clé `else` (« sinon » en anglais) ajoute une alternative qui s'exécute si aucune des conditions précédentes n'ont été vérifiées ;
- ▷ il n'est *pas* suivi d'une expression booléenne, uniquement de « : » ;
- ▷ le mot-clé `elif` (abrégié pour « else if », « sinon si » en anglais) permet d'ajouter une alternative avec une nouvelle condition ;
- ▷ à l'instar de `if`, il est suivi d'une expression booléenne puis de « : » ;
- ▷ le bloc de code associé est indenté dans les deux cas, de façon identique à `if`.

## Ordre des conditions

L'ordre dans lequel les conditions sont écrites est significatif ! Lorsqu'un programme Python rencontre une alternative, les conditions sont testées *une par une*, et c'est la *première* condition vérifiée qui est exécutée. Toutes les conditions `elif` ou `else` suivant celle-ci sont alors ignorées.

Le code 112, une fois exécuté, affichera `Ce nombre est pair` et non `Ce nombre est un multiple de 10` comme souhaité, car la condition à la ligne 2 (qui vérifie si le nombre est pair) est vérifiée avant celle à la ligne 6 (qui vérifie si le nombre est un multiple de 10).

Le code 113 est une version corrigée de ce programme : cette fois-ci, la priorité des conditions est correctement renseignée. De manière générale, lors de l'écriture d'une alternative, il est essentiel de vérifier la ou les conditions *la plus contraignante* en premier lieu.

```

1 n = 10
2 if n % 2 == 0:
3     print("Ce nombre est pair")
4 elif n % 5 == 0:
5     print("Ce nombre est un multiple de 5")
6 elif n % 10 == 0:
7     print("Ce nombre est un multiple de 10")
8 else:
9     print("Ce nombre n'est pas un multiple de 2, 5 ou 10")

```

Code 112: Exemple de conditions mal enchaînées

```

1 n = 10
2 if n % 10 == 0:
3     print("Ce nombre est un multiple de 10")
4 elif n % 2 == 0:
5     print("Ce nombre est pair")
6 elif n % 5 == 0:
7     print("Ce nombre est un multiple de 5")
8 else:
9     print("Ce nombre n'est pas un multiple de 2, 5 ou 10")

```

Code 113: Correction du code 112

### Enchaînement de plusieurs conditions non-exclusives

Il ne faut pas tomber dans le piège de croire que si plusieurs conditions s'enchaînent, elles sont forcément *mutuellement exclusives*. On peut aussi vouloir tester *toutes* les conditions l'une après l'autre, *sans* s'arrêter dès qu'une des conditions est vérifiée.

Pour cela, il suffit d'écrire plusieurs conditions `if`, sans utiliser les mots-clés `elif` ou `else`. Par exemple :

```

1 n = 10
2 if n % 2 == 0:
3     print("Ce nombre est pair")
4 if n % 5 == 0:
5     print("Ce nombre est un multiple de 5")

```

Code 114: Conditions non mutuellement exclusives

Quand on exécute le code 114, il affiche :

```

Ce nombre est pair
Ce nombre est un multiple de 5

```

Attention toutefois : si plusieurs conditions sont placées à la suite l'une de l'autre comme ceci, mais qu'un mot-clé `else` ou `elif` est employé, il ne sera lié qu'au `if` qui le précède *directement* !

Ainsi, le code 115 affiche de manière incorrecte :

```

Ce nombre est pair
Ce nombre n'est pas un multiple de 2 ou de 5

```

```

1 n = 8
2 if n % 2 == 0:
3     print("Ce nombre est pair")
4 if n % 5 == 0:
5     print("Ce nombre est un multiple de 5")
6 else :
7     print("Ce nombre n'est pas un multiple de 2 ou de 5")

```

Code 115: Conditions non mutuellement exclusives

Ceci est dû au fait que le `else` à la ligne 6 se déclenche si et seulement si la condition à la ligne 4 n'est pas vérifiée, ce qui est ici le cas. Si on souhaite écrire une alternative qui répond à toutes les conditions précédentes, il faudra alors réorganiser son code (par exemple en remplaçant le `else` par une nouvelle condition plus spécifique, ou en remaniant entièrement les conditions).

### Alternative « selon que »

Dans certains cas, les différentes alternatives correspondent à des valeurs spécifiques d'une variable. Plutôt que d'écrire une série d'instructions `if`, `elif`, `else` où chaque condition est une comparaison, on préférera indiquer clairement que la condition porte sur certaines valeurs.

Depuis Python 3.10, une telle alternative se note via le mot-clé `match` (« fait correspondre »).

```

1 def jour_de_semaine(x):
2     match x:
3         case 1:
4             return "Lundi"
5         case 2:
6             return "Mardi"
7         case 3:
8             return "Mercredi"
9         case 4:
10            return "Jeudi"
11        case 5:
12            return "Vendredi"
13        case 6:
14            return "Samedi"
15        case 7:
16            return "Dimanche"
17        case _:
18            return None

```

Code 116: Exemple de selon-que

Le code 116 définit une fonction qui reçoit le numéro d'un jour de la semaine (entre 1 et 7) et renvoie le nom du jour correspondant en chaîne de caractère.

Notons la syntaxe :

- ▷ le mot-clé `match` suivi du nom d'une variable ;
- ▷ la ligne se termine par « : » ;

- ▷ les mots-clés `case` ("cas") sont *indentés* ;
- ▷ chacun est suivi d'une valeur qui définit ce cas, puis à nouveau de « : » ;
- ▷ le code à exécuter est *indenté* une deuxième fois ;
- ▷ la notation `case _` permet de définir un *cas par défaut*, qui se déclenche si aucun des cas précédents n'est rencontré. Il est utilisé dans ce cas-ci pour traiter les erreurs.

Le mot-clé `match` est également capable de définir des cas de façon plus souple que comme présenté ici. Toutefois, cette utilisation sort du cadre de ce cours et ne sera pas abordée en détail ici.

## 5.3 Structures itératives

Un autre cas de figure fréquent en développement est le besoin d'effectuer une même tâche *plusieurs fois*, avec de très légères variations entre chaque exécution. Dans certains cas, il serait possible de répéter les instructions qui effectuent cette tâche, même si cela va à l'encontre d'une des règles de bonne écriture en programmation (DRY : *don't repeat yourself* - ne vous répétez pas!). Dans d'autres cas, il est *impossible* de savoir à l'avance (c'est-à-dire *au moment où le code est écrit*) le nombre d'exécutions nécessaires de la tâche.

Pour pallier à ces problèmes, différentes structures itératives existent, permettant de répéter l'exécution de certaines instructions à plusieurs reprises.

### 5.3.1 Boucle `while`

Le mot-clé `while` (« tant que » en anglais) permet d'exécuter une tâche *tant qu'une certaine condition est vérifiée*. Elle fonctionne donc de la même façon qu'une condition, à ceci près qu'une fois que le bloc de code a terminé de s'exécuter, la condition est *à nouveau vérifiée* et le bloc de code est exécuté à nouveau si c'est toujours le cas.

```

1 start = 3
2 while start > 0 :
3     print(start)
4     start = start - 1
5 print("Partez !")

```

Code 117: Exemple de boucle `while`

Le code 117 effectue un compte à rebours à partir de 3. Une fois exécuté, il affiche :

```

3
2
1
Partez !

```

Notons la syntaxe similaire aux conditions :

- ▷ le mot-clé `while` ;

- ▷ suivi d'une expression booléenne ;
- ▷ la ligne se termine par « : » ;
- ▷ le bloc de code de la boucle est *indenté* ;
- ▷ la ligne de code 5, puisqu'elle n'est pas indentée, n'est exécutée qu'une fois que la condition à la ligne 2 (`start > 0`) n'est *pas* vraie, c'est-à-dire lorsque `start` est inférieure ou égale à 0.

De plus, on remarque que

- ▷ une boucle `while` est exécutée au minimum 0 fois,
- ▷ les variables utilisées dans la condition doivent être initialisées avant le `while`.

Dans une boucle `while`, on se dote généralement d'une variable de travail dont la valeur change à chaque tour de boucle. Cette variable peut être un compteur, mais ce n'est pas obligatoire. Une pratique fréquente est de créer une (voire plusieurs) variable *flag* (« drapeau » en anglais) de valeur booléenne, dont le rôle est de changer de valeur lorsqu'une certaine condition nécessaire à l'arrêt de la boucle est rencontrée.

```

1 active = True
2 while (active):
3     message = input("Dis-moi quoi répéter, ou entre 'quit' pour arrêter : ")
4     if message == "quit":
5         active = False
6     else :
7         print(message)

```

code/chap5/flag\_var.py

Code 118: Exemple de boucle *while* avec une variable drapeau

Le code 118 répète tout message entré par l'utilisateur, mais s'arrête si `quit` est entré dans la console. La variable `active` est utilisée pour gérer cet arrêt, mais n'est pas utilisée en tant que tel dans l'affichage. Il serait possible de remplacer `while active` par `while message != "quit"` ; cependant, cela signifie qu'on teste deux fois la même condition, ce qui est rarement une bonne idée quand notre condition devient plus difficile à vérifier.

### Boucles infinies

Un risque majeur lorsqu'on emploie des boucles est de créer une *boucle infinie* : une boucle qui ne s'arrête jamais. On peut en créer très simplement en écrivant une condition qui ne devient jamais fausse, comme par exemple le code 119 :

```

1 while True:
2     print("Je ne m'arrête jamais !")

```

code/chap5/while\_true.py

Code 119: Boucle infinie *while true*

Cette boucle est facile à identifier comme une boucle infinie, mais ce n'est pas toujours aussi simple ! Prenons un autre exemple dans le code 120, la variable `i`

n'est plus décrémentée lorsqu'elle atteint 0, mais la boucle continue de s'exécuter tant qu'elle est de valeur supérieure ou égale à 0. Par conséquent, la boucle ne cesse jamais de s'exécuter.

```

1 i = 10
2 while i >= 0 :
3     if i > 0:
4         print ("Encore " + i + " étapes !")
5         i = i-1
6     else :
7         print ("On est arrivés !")

```

code/chap5/infinite\_loop.py

Code 120: Une autre boucle infinie

Lorsqu'on écrit du code utilisant une boucle `while`, il est donc *essentiel* de s'assurer qu'une condition d'arrêt sera toujours déclenchée à un moment ou un autre !

### 5.3.2 Boucle for

Le mot-clé `for` (« pour » en anglais, ou « pour chaque ») permet d'exécuter une tâche pour chaque élément dans une collection d'éléments. On récupère alors chaque élément de cette collection, et les instructions dans la boucle sont exécutées *sur cet élément*. Tout type d'objet qu'il est possible de parcourir de la sorte est ce qu'on appellera un *itérable*.

Sans faire de supposition sur la nature de cet itérable pour l'instant, une boucle `for` s'écrit de la manière illustrée sur la figure 5.1.

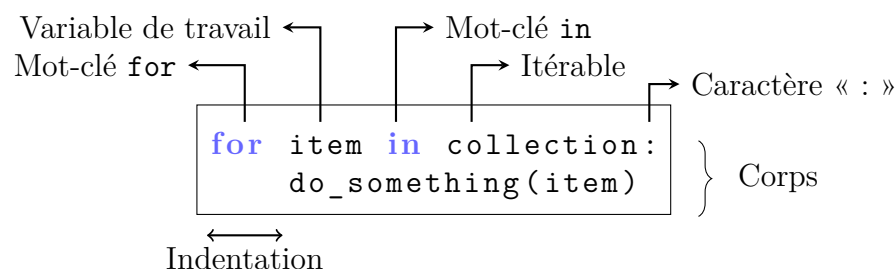


FIGURE 5.1 – Illustration syntaxique d'une fonction

Notons les éléments de syntaxe :

- ▷ le mot-clé `for` est suivi d'un *nom de variable* qui sera utilisée *au sein de la boucle* ;
- ▷ ensuite vient le mot-clé `in` suivi du nom de l'itérable contenant ces objets ;
- ▷ comme à l'accoutumée, la ligne se termine par « : » ;
- ▷ le bloc de code des instructions à répéter est indenté.

Notons qu'il est possible de modifier l'itérable *au sein même* de la boucle `for` : ce n'est en général pas une bonne pratique, dans la mesure où cela devient difficile de savoir quand la boucle va s'arrêter.

Nous verrons quelques exemples d'itérables au chapitre 6 : nous aborderons ici deux exemples d'itérables simples.



## Chaînes de caractères en tant qu'itérable

En Python, les chaînes de caractères peuvent se traiter comme une *séquence* de caractères, c'est-à-dire une collection ordonnée de caractères. Cela signifie que :

- ▷ une chaîne de caractères est associée à un ensemble d'objets (de type caractère) ;
- ▷ chacun de ces objets possède une position précise au sein de la chaîne.

Par conséquent, les chaînes de caractère peuvent être manipulées en tant qu'itérables. Le code 121 montre un tel exemple, où chaque caractère d'une chaîne "bonjour" est ajouté deux fois dans une nouvelle chaîne.

```

1 str = ""
2 for lettre in "bonjour":
3     str += lettre * 2
4 print(str)

```

code/chap5/for\_str.py

Code 121: Exemple de boucle `for` sur une chaîne de caractère

## Intervalles

`range` est un type d'objet Python représentant un intervalle. On peut créer un intervalle via la fonction éponyme, `range()`. Cette fonction peut être employée de plusieurs façons :

- ▷ `range(n)` crée un intervalle reprenant les nombres entre 0 (**inclus**) et  $n$  (**exclus**)
- ▷ `range(a, b)` crée un intervalle reprenant les nombres entre  $a$  (**inclus**) et  $b$  (**exclus**)
- ▷ `range(a, b, step)` crée un intervalle reprenant les nombres entre  $a$  (**inclus**) et  $b$  (**exclus**), par incréments donné par `step` ; par exemple, une valeur de `step` de 2

Les intervalles nous permettent d'écrire des boucles simplifiées lorsqu'on souhaite exécuter des boucles sur toutes sortes de compteurs, surtout s'il n'est pas nécessaire d'utiliser ce compteur en-dehors de la boucle.

```

1 for i in range(5):
2     print(i)

```

code/chap5/for\_counter.py

Code 122: Exemple de boucle `for` avec un intervalle

Le code 122 montre un exemple d'utilisation d'un intervalle simple. Une fois exécuté, il affichera :

```

0
1
2
3
4

```

Par exemple, il est possible de réécrire le compte à rebours via boucle `while` dans le code 117 (page 78) via une boucle `for` et un intervalle comme illustré au code 123. Le décompte commence à 3, se termine à 0 (non inclus), et descend d'une unité par tour de boucle, d'où le `step` de  $-1$ .

```
1 for n in range(3,0,-1):
2     print(n)
3 print("Partez !")
```

code/chap5/for\_countdown.py

Code 123: Compte à rebours avec une boucle `for`

Finalement, il est possible (par mégarde) de donner des itérables créant des boucles infinies, comme `range(3,0,1)`.

### 5.3.3 Choix du type de boucle

Comme montré ci-dessus, il est souvent possible d'écrire le même programme en utilisant soit une boucle `while`, soit une boucle `for`. Une question se pose dès lors : quelle boucle utiliser dans un problème donné ?

Nous vous renverrons au cours d'algorithmique pour plus d'informations sur le sujet, toutefois, la question peut se résumer de la façon suivante :

- ▷ La boucle `while` est idéale si, *au moment où la boucle démarre durant l'exécution*, il n'est pas encore possible de déterminer le nombre d'itérations qui auront lieu. Pour exemples : si une action de l'utilisateur détermine l'arrêt de la boucle, ou si un résultat spécifique d'une opération est attendu.
- ▷ La boucle `for` est idéale si, *au moment où la boucle démarre durant l'exécution*, il est possible de déterminer le nombre d'itérations qui auront lieu. Le développeur n'a pas besoin de connaître ce nombre d'itération, du moment qu'il est possible de le calculer durant l'exécution. Pour exemples : si la boucle fait intervenir un compteur dont les bornes sont déjà connues, ou si elle parcourt un itérable.

### 5.3.4 Boucles imbriquées

Le bloc de code à l'intérieur d'une boucle peut contenir n'importe quelle instruction, y compris des alternatives (comme déjà montré dans différents exemples ci-dessus) mais aussi d'autres boucles. On parlera alors de *boucles imbriquées* ; c'est une pratique très commune avec de nombreux cas d'utilisation.

Il est donc important de comprendre l'ordre des opérations lorsque deux boucles sont imbriquées. Prenons par exemple le script 124, qui affiche le début et la fin de chaque itération pour chacune des deux boucles imbriquées.

```

1 for i in range(2):
2     print("Première boucle itération " + str(i))
3     for j in range(2):
4         print("Deuxième boucle itération " + str(j))
5         #Code utile
6         print("Fin de l'itération " + str(j) + " de la deuxième boucle")
7     print("Fin de l'itération " + str(i) + " de la première boucle")

```

Code 124: Compte à rebours avec une boucle `for`

```

Première boucle itération 0
Deuxième boucle itération 0
Fin de l'itération 0 de la deuxième boucle
Deuxième boucle itération 1
Fin de l'itération 1 de la deuxième boucle
Fin de l'itération 0 de la première boucle
Première boucle itération 1
Deuxième boucle itération 0
Fin de l'itération 0 de la deuxième boucle
Deuxième boucle itération 1
Fin de l'itération 1 de la deuxième boucle
Fin de l'itération 1 de la première boucle

```

À chaque itération de la première boucle, la boucle qui est imbriquée à l'intérieur de celle-ci est donc exécutée *dans sa totalité*, et c'est seulement après cela que la première boucle continue son exécution puis passe à l'itération suivante.

### 5.3.5 Récursivité

Une autre façon d'obtenir un comportement répété en programmation est via l'utilisation de fonctions, puisqu'une fonction sert déjà de définir une action que l'on fait souvent sans en répéter le code. Pour cela, il suffit que la fonction s'appelle elle-même : on parle alors de fonction *récursive*.

Prenons par exemple une fonction qui calcule la factorielle d'un nombre en mathématiques. Pour rappel : la factorielle d'un nombre entier est le produit de tous les nombres entiers entre 1 et lui-même. Il serait possible d'écrire une fonction qui calcule la factorielle d'un nombre via une boucle, comme montré dans le code 125.

```

1 def factorielle_boucle(n):
2     resultat = 1
3     for i in range(1, n+1):
4         resultat *= i
5     return resultat

```

Code 125: Fonction factorielle implémentée par une boucle

Mais on peut aussi se servir de la définition par récursion de la factorielle :

$$n! = \begin{cases} 1 & \text{si } n = 0, \\ n \cdot (n - 1)! & \text{sinon.} \end{cases}$$

Cette définition se traduit naturellement en une fonction récursive comme montré dans le code 126.

```
code/chap5/facto_recur.py
1 def factorielle_recur(n):
2     if n == 0:
3         return 1
4     else:
5         return n * factorielle_recur(n-1)
```

Code 126: Fonction factorielle implémentée par une boucle

Prudence toutefois : à l'instar d'une boucle `while`, écrire une fonction récursive risque de créer des exécutions infinies ! Une fonction récursive comprend généralement une alternative qui arrête la récursion (un « cas de base ») et renvoie un résultat, mais il faut aussi vous assurer que l'argument de la fonction change à chaque appel, et finit par déclencher cette alternative.

# Structures de données

Ce chapitre aborde en détail la manipulation des structures de données les plus couramment utilisées en Python : les listes, tuples, et dictionnaires.

## 6.1 Listes

Une *liste* est une séquence de valeurs, c'est-à-dire une collection indexée : chaque valeur contenue dans la liste possède une position qui lui est propre, qu'on appelle *l'indice* de cette valeur. Elle possède les propriétés suivantes :

- ▷ les éléments d'une liste peuvent avoir n'importe quel type, et en Python, ils ne doivent pas tous être du même type ;
- ▷ la liste a une taille variable : on peut y ajouter ou en retirer des éléments ;
- ▷ la liste est mutable : on peut modifier ses éléments.

### 6.1.1 Création d'une liste

Créer une liste en Python se fait via la notation crochets []. Il suffit de placer les éléments contenus dans la liste à l'intérieur de ceux-ci, séparés par des virgules, comme montré dans le code 127.

```
>>> vide = [] # empty list
>>> compte = [1, 2, 3] # list of integers
>>> fruits = ["apple", "banana", "cherry", "kiwi", "mango"] # list of strings
>>> quelconque = ["texte", 5, 2.0, ["a", "b", "c"]] # list of any elements
```

Code 127: Instanciations de listes

Rien ne force à renseigner des valeurs littérales pour chaque élément d'une liste. Toute expression peut être placée dans une liste, auquel cas, c'est le résultat de cette expression qui est stocké dans la liste, comme illustré au code 128.

```
>>> liste = [2+2, 3 > 1, abs(-5)] # contains [4, True, 5]
```

Code 128: Instanciations d'une liste via des expressions

### 6.1.2 Opération sur les listes

#### Longueur de liste

Une première information qu'on peut obtenir sur une liste est de mesurer sa *longueur*, c'est-à-dire le nombre d'éléments présents dans la liste. Pour cela, il suffit d'employer la fonction `len`, tel qu'illustré au code 129

```
>>> liste = [1, 2, 3]
>>> len(liste)
3
```

Code 129: Longueur d'une liste

#### Opérateur d'accès

Une fois une liste créée, on veut pouvoir accéder aux éléments contenus dans celle-ci. Cela se fait via l'opérateur d'accès, qui se note lui aussi via des crochets `[]`, placés après la liste à laquelle on souhaite accéder. À l'intérieur des crochets, un ou plusieurs indices permettent de préciser à quel(s) élément(s) on souhaite accéder.

Attention : en programmation, la plupart des collections indexées commencent leur numérotation à 0, et c'est bien le cas des listes ! Cela signifie que pour accéder au *premier* élément d'une liste, on utilisera `[0]`. Le code 130 montre un exemple d'accès à un élément via son indice. Cela signifie aussi que la valeur maximale qu'un indice peut prendre est `len(liste)-1` : toute valeur supérieure à celle-ci est considérée comme *en-dehors des bornes de la liste* et déclenche une erreur !

```
>>> liste = [1, 2, 3]
>>> liste[0]
1
>>> liste[2]
3
>>> liste[3]
IndexError: list index out of range
```

Code 130: Accès à des éléments

L'opérateur d'accès permet aussi de sélectionner un objet à partir de la *fin* de la liste, ce qui nous permet de ne pas devoir calculer les dernières positions de façon explicite, comme montré dans le code 131. Attention là aussi à ne pas dépasser les bornes de la liste.

Le tableau 6.1 reprend quelques exemples de la correspondance des indices d'une liste.

L'opérateur d'accès permet aussi de *modifier* le contenu de la liste en le remplaçant par une nouvelle valeur. Le code 132 montre un exemple des assignations possibles. Attention : ce type d'assignation ne peut se faire qu'à des positions déjà existantes !

```

>>> liste = [1, 2, 3]
>>> liste[-1]
3
>>> liste[-2]
2
>>> liste[-4]
IndexError: list index out of range

```

Code 131: Accès à des éléments

Indice positif	Indice négatif	Position réelle
0	-(len(liste))	Première position
1	-(len(liste)-1)	Deuxième position
2	-(len(liste)-2)	Troisième position
...	...	...
len(liste)-2	-2	Avant-dernière position
len(liste)-1	-1	Dernière position
len(liste) ou +	N/A	Erreur : hors bornes
N/A	-len(liste) ou -	Erreur : hors bornes

TABLE 6.1 – Correspondance d’indices positifs et négatifs dans une liste `liste` quelconque

```

>>> liste = [1, 2, 3]
>>> liste[2] = 4
>>> liste
[1, 2, 4]
>>> liste[-2] = 5
>>> liste
[1, 5, 4]
>>> liste[4] = 6
IndexError: list assignment index out of range

```

Code 132: Accès à des éléments

## Découpage

L'opérateur d'accès possède une autre fonctionnalité : le *découpage* de liste (on parlera parfois aussi de tranchage, en anglais *slicing*, ou d'accès à une *sous-liste*). Cela crée une nouvelle liste, en récupérant certains éléments de la liste d'origine, référencés par leurs indices. Pour cela, entre les crochets de l'opérateur d'accès, il faut définir un intervalle, commençant par la position du premier élément qui sera *inclus* dans la sous-liste, et en terminant par la position du premier élément qui sera *exclus* de la sous-liste. Un caractère deux points « : » sépare ces deux positions.

```
>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste[1:4]
[2, 3, 4]
>>> liste[2:-2]
[3, 4, 5]
>>> liste[-4:5]
[4, 5]
>>> liste[5:5]
[]
>>> liste[5:4]
[]
>>> liste[7:10]
[]
>>> liste[5:10]
[6,7]
```

Code 133: Accès à des éléments

Le code 133 montre quelques exemples de découpage :

- ▷ Dans le premier découpage, la position de départ est 1 (la *deuxième* position ; attention, on compte toujours à partir de 0!), et la position de fin de sous-liste est 4 (la cinquième position). Les positions incluses dans le découpage sont donc les indices 1, 2, et 3.
- ▷ Les indices négatifs peuvent aussi être employés lors d'un découpage. Ainsi, dans le deuxième découpage, la position de départ est 2, et la position de fin est -2, correspondant à l'avant-dernière position dans la liste, donc la position 5. Ce sont donc les indices 2, 3, et 4 qui sont inclus.
- ▷ Même principe pour le troisième découpage, à ceci près que l'indice de début est un indice négatif. -4 correspond à la quatrième position en partant de la fin, donc ici à l'indice 3. La position de fin est 5 ; par conséquent, ce sont les positions 3 et 4 qui sont incluses.
- ▷ Les deux exemples suivants montrent ce qui se produit si la position de début est égale ou supérieure à celle de fin : le découpage produit alors une liste vide, puisqu'aucun élément ne peut être inclus
- ▷ Lors d'une découpe, il est même possible d'utiliser des indices en-dehors de la liste lorsqu'il est défini : les indices sont calculés de la façon habituelle, et seules les positions valides sont prises en compte.

Si une des deux positions n'est pas renseignée, le découpage ira jusqu'au début (pour la position de départ) ou la fin (pour la position d'arrivée) de la liste,



```
>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste[:4]
[1, 2, 3, 4]
>>> liste[4:]
[5, 6, 7]
>>> liste[:]
[1, 2, 3, 4, 5, 6, 7]
```

Code 134: Accès à des éléments

comme montré dans le code 134. En ne renseignant aucune des deux bornes, l'opérateur de découpage crée alors une *copie* de la liste !

À l'instar des intervalles, on peut également donner un incrément autre que 1 afin de "passer" certains éléments lors du découpage, séparé des positions de début et de fin par « : » également. Cet incrément peut même être négatif si on souhaite parcourir la liste dans le sens inverse.

```
>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste[1:6:2]
[2, 4, 6]
>>> liste[5:1:-3]
[6, 3]
>>> liste[3:-1]
[7, 6, 5]
>>> liste[::-1]
[7, 6, 5, 4, 3, 2, 1]
```

Code 135: Accès à des éléments

Le code 135 montre des exemples de ce découpage avec incréments :

- ▷ Le premier découpage inclut les indices 1, 1+2 (3), 3+2 (5) puis s'arrête puisque l'indice suivant (7) dépasse la position de fin demandée (6)
- ▷ Le deuxième découpage inclut les indices 5, 5-3 (2), puis s'arrête puisque l'indice suivant est au-delà de la position de fin demandée (1)
- ▷ Le troisième découpage montre que lorsqu'un incrément négatif est renseigné, les bornes sont inversées : si la position de départ est omise, c'est donc de la *fin* de la liste qu'on commence le découpage. De même, si la position de fin est omise, le découpage ira jusqu'au *début* de la liste d'origine
- ▷ Le quatrième découpage montre que par conséquent, `liste[: -1]` crée une copie de la liste d'origine, mais cette fois-ci dans l'ordre inverse !

De la même façon qu'un accès à une case unique, l'accès à un intervalle peut permettre de modifier une liste. Seul un objet *itérable* peut être assigné de la sorte, et ses objets remplacent les objets aux positions ciblées par l'intervalle.

Le code 136 montre des exemples d'utilisation :

- ▷ `liste[3:5] = [8]` sélectionne les positions 3 et 4, et efface les objets à ces positions (les numéros 4 et 5). Ensuite, *à la même position*, les objets contenus dans `[8]` sont ajoutés à la liste ; par conséquent, un numéro 8 apparaît entre les numéros 3 et 6.

```

>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste[3:5] = [8]
>>> liste
[1, 2, 3, 8, 6, 7]
>>> liste[2:4] = 9
TypeError: can only assign an iterable
>>> liste[2:4] = [9, 10, 11]
>>> liste
[1, 2, 9, 10, 11, 6, 7]
>>> liste[len(liste):] = [8]
>>> liste
[1, 2, 9, 10, 11, 6, 7, 8]

```

Code 136: Accès à des éléments

- ▷ `liste[2:4] = 9` montre que si un élément non itérable est assigné, une erreur se déclenche
- ▷ `liste[2:4] = [9, 10, 11]` est un deuxième exemple qui fonctionne. Les éléments aux positions 2 et 3 sont sélectionnés et effacés (les numéros 3 et le 8 précédemment ajoutés) ; ensuite, les éléments de `[9, 10, 11]` sont ajoutés à leur place, et les numéros 9, 10, et 11 se retrouvent donc entre les numéros 2 et 6 encore présents dans la liste.
- ▷ Puisqu'il est possible de définir des intervalles sur des indices en-dehors des bornes, `liste[len(liste):] = [8]` est une façon d'ajouter un élément à la fin de liste. Il sera généralement préférable d'utiliser les méthodes `append` ou `extend` (voir la section 6.1.2) par souci de lisibilité.

## Parcours de liste

Une liste est un itérable ; par conséquent, il est possible de parcourir une liste et de manipuler chacun de ses éléments de manière itérative via une boucle `for`. Le code 137 montre un exemple qui calcule la somme d'éléments dans une liste.

```

1 sum = 0
2 for element in [1, 2, 3]:
3     sum += element
4 print(sum) # 6

```

code/chap6/for\_list.py

Code 137: Somme d'éléments d'une liste via une boucle `for`

Cependant, notons que la notation `for ... in` permet uniquement d'accéder aux *valeurs* contenues dans la liste. Ces valeurs sont *copiées* dans la variable de boucle ; si on souhaite pouvoir *modifier* le contenu de la liste, il nous faudra avoir accès à l'*indice* correspondant au sein de la liste.

Une première option pour ce faire est d'itérer non pas sur la liste directement, mais sur un intervalle qui contient tous les indices de la liste en question. Puisque les indices d'une liste sont contenus entre 0 (inclus) et `len(liste)` (exclus), on peut utiliser la fonction `range(len(liste))` pour obtenir un tel intervalle, comme montré dans le code 138, ou l'on écrit une boucle qui double la valeur de chaque élément dans une liste.

```

1 liste = [1,2,3]
2 for i in range(len(liste)) :
3     liste[i] = liste[i] * 2
4 print(liste) # [2, 4, 6]

```

Code 138: Somme d'éléments d'une liste via une boucle `for`

Une autre option est d'utiliser la fonction `enumerate`, qui permet de récupérer une liste de tuples (voir la section 6.2) contenant l'indice d'un élément et sa valeur. On peut alors effectuer une assignation à un tuple de variable (voir la section 6.2.3) afin de récupérer directement ces deux informations dans des variables séparées. Cette manipulation est démontrée dans le code 139.

```

1 liste = [1,2,3]
2 for index, value in enumerate(liste) :
3     liste[index] = value * 2
4 print(liste) # [2, 4, 6]

```

Code 139: Somme d'éléments d'une liste via une boucle `for` avec `enumerate`

## Insertion et suppression d'éléments

Jusqu'ici, nous avons vu comment accéder aux éléments de liste et modifier les éléments existants. L'opérateur de découpe nous permet aussi de supprimer et remplacer des éléments de liste ; toutefois, on effectue souvent les quelques mêmes manipulations. Par conséquent, des fonctions ont été définies pour effectuer ces insertions et suppressions fréquentes de façon plus lisible qu'en utilisant l'opérateur de découpe.

Il s'agit en particulier de *méthodes* des listes. Ce concept sera approfondi dans le cours DEV2 ; pour le moment, contentez-vous de les considérer comme des fonctions qui sont appelées *sur* un objet précis. La différence principale vient de la syntaxe : pour appeler une méthode sur une liste, cela se notera en écrivant le nom de la liste, suivi d'un point `.`, puis du nom de la méthode et de ses arguments entre parenthèses comme pour tout appel de fonction.

- ▷ `append(x)` ajoute un élément `x` en fin de liste

```

>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste.append(8)
>>> liste
[1, 2, 3, 4, 5, 6, 7, 8]

```

- ▷ `insert(i, x)` ajoute un élément `x` à la position `i`. Tout comme avec l'opérateur d'accès, un indice négatif est acceptable. Un indice hors des bornes de la liste existante ajoute un élément en fin de liste (indice positif) ou au début de la liste (indice négatif).

```

>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste.insert(3,9)
>>> liste
[1, 2, 3, 9, 4, 5, 6, 7]

```

```
>>> liste.insert(-2,10)
>>> liste
[1, 2, 3, 9, 4, 5, 10, 6, 7]
>>> liste.insert(13,11)
>>> liste
[1, 2, 3, 9, 4, 5, 10, 6, 7, 11]
>>> liste.insert(-13,0)
>>> liste
[0, 1, 2, 3, 9, 4, 5, 10, 6, 7, 11]
```

- ▷ `remove(x)` cherche l'élément `x` dans la liste, et s'il est présent, supprime sa première instance au sein de la liste.

```
>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste.remove(5)
>>> liste
[1, 2, 3, 4, 6, 7]
>>> liste.remove(10)
ValueError: list.remove(x): x not in list
```

- ▷ `pop()` ou `pop(i)` supprime un élément, soit en dernière position, soit à la position `i`, et retourne sa valeur en tant que résultat. Un indice négatif est accepté, mais un indice hors borne déclenche une erreur.

```
>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste.pop(1)
2
>>> liste
[1, 3, 4, 5, 6, 7]
>>> liste.pop(-3)
5
>>> liste
[1, 3, 4, 6, 7]
>>> liste.pop(13)
IndexError: pop index out of range
```

- ▷ `extend(iterable)` parcourt l'itérable passé en argument, et ajoute ses éléments un par un à la fin de la liste.

```
>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste.extend([10, 11])
>>> liste
[1, 2, 3, 4, 5, 6, 7, 10, 11]
```

- ▷ `clear()` vide une liste : elle supprime tous ses éléments.

```
>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste.clear()
>>> liste
[]
```

En plus de ces méthodes, le mot-clé `del`, qui permet d'effacer un élément en mémoire, peut être employé sur une liste afin de supprimer un élément de celle-ci via l'opérateur d'accès, comme montré dans le code 140.

```
>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> del liste[3]
>>> liste
[1, 2, 3, 5, 6, 7]
```

Code 140: Utilisation de `del`

### Recherche au sein d'une liste

Parfois, plutôt que d'accéder aux éléments d'une liste, on souhaite effectuer une *recherche* au sein d'une liste, c'est-à-dire déterminer si une certaine valeur est présente dans cette liste. Plusieurs options s'offrent à nous, selon l'information dont nous avons besoin.

```
>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> 3 in liste
True
>>> 9 in liste
False
```

Code 141: Mot-clé `in`

Le mot-clé `in` démontré dans le code 141 est un opérateur similaire aux opérateurs de comparaison : il demande si une valeur est présente dans la liste, et renvoie une valeur booléenne (vrai ou faux) en réponse.

```
>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste.index(3)
2
>>> liste.index(9)
ValueError: 9 is not in list
>>> repeat = [3, 4, 3]
>>> repeat.index(3)
0
```

Code 142: Méthode `index`

La méthode `index` démontrée dans le code 142 récupère quant à elle la position de la *première* occurrence de la valeur demandée. Toutefois, elle déclenche une erreur si la valeur n'est pas présente ; il peut être avisé de la combiner avec `in` pour éviter de déclencher une erreur, comme montré dans le code 143.

La méthode `count` compte le nombre d'occurrences d'une valeur demandée au sein d'une liste, comme illustré dans le code 144.

### Réorganisation de liste

Deux méthodes permettent de modifier l'ordre des éléments d'une liste. La méthode `reverse` inverse l'ordre des éléments de la liste, comme montré dans le code 145.

Enfin, la méthode `sort` permet de *trier* une liste. Par défaut, le tri se fait via l'opérateur de comparaison `<` et trie donc les éléments dans l'ordre croissant.

```

>>> def affiche_position(liste, elem) :
...     if elem in liste :
...         print("L'élément " + str(elem) + " se trouve à la position " +
↪ str(liste.index(elem)) + ".")
...     else :
...         print("L'élément " + str(elem) + " ne se trouve pas dans la liste.")
...
>>> liste = [1, 2, 3, 4, 5]
>>> affiche_position(liste, 3)
L'élément 3 se trouve à la position 2.
>>> affiche_position(liste, 7)
L'élément 7 ne se trouve pas dans la liste.

```

Code 143: Méthode `index` combinée avec le mot-clé `in`

```

>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste.count(2)
1
>>> liste.count(9)
0

```

Code 144: Méthode `count`

```

>>> liste = [1, 2, 3, 4, 5, 6, 7]
>>> liste.reverse()
>>> liste
[7, 6, 5, 4, 3, 2, 1]

```

Code 145: Méthode `reverse`

Elle possède en outre deux arguments optionnels : **key** permet de sélectionner un autre critère de comparaison, mais nous n'en discuterons pas plus en détails dans le cadre du cours ; **reverse** permet d'inverser l'ordre de tri (avec le critère de comparaison par défaut, la liste sera donc triée dans l'ordre décroissant). Des exemples d'utilisations sont présents dans le code 146.

```
>>> liste = [1, 6, 5, 2, 3, 7, 4]
>>> liste.sort()
>>> liste
[1, 2, 3, 4, 5, 6, 7]
>>> liste.sort(reverse=True)
>>> liste
[7, 6, 5, 4, 3, 2, 1]
```

Code 146: Méthode `sort`

### Une méthode d'initialisation de liste plus puissante

Très souvent, lorsqu'on initialise une liste, on souhaite définir son contenu non pas de façon *explicite*, en écrivant chacun de ses éléments un par un, mais de façon *implicite*, en la décrivant comme une liste d'éléments remplissant une certaine condition. Cette définition est analogue à la définition d'ensemble en compréhension en mathématiques : pour décrire l'ensemble des dix premiers carrés parfaits, on peut écrire :

$$\text{squares} = \{x^2 | x \in [1, 10]\}$$

Une boucle nous permet de créer cette liste. Il faudrait cependant alors créer une liste vide, puis exécuter une boucle qui ajoute les éléments un par un dans celle-ci, comme montré au code 147.

```
1 squares = []
2 for i in range(1,11):
3     squares.append(i**2)
```

code/chap6/list\_init\_loop.py

Code 147: Initialisation d'une liste via une boucle `for`

Puisque cette manipulation est fréquente, le langage Python s'est doté d'une définition de *liste en compréhension* tout à fait analogue aux mathématiques : dès l'initialisation de la liste, on lui indique que ses éléments sont créés via une boucle. Le code 148 montre un programme qui crée la même liste, cette fois en compréhension.

```
1 squares = [i ** 2 for i in range(1,11)]
```

code/chap6/list\_init\_comprehension.py

Code 148: Initialisation d'une liste en compréhension

Il est même possible d'introduire des conditions ou d'imbriquer plusieurs boucles lors d'une définition en compréhension. Le code 149 filtre une liste de fruits, en ne gardant que ceux qui possèdent un "a" dans leur nom.

```

1 fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
2 newlist = [x for x in fruits if "a" in x]

```

Code 149: Initialisation d'une liste en compréhension avec une condition

Lorsque plusieurs boucles et/ou conditions sont ainsi utilisées, elles se lisent de gauche à droite, comme si elles étaient mutuellement imbriquées l'une dans l'autre. Le code 150 montre une version équivalente, sans compréhension.

```

1 fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
2 newlist = []
3 for x in fruits :
4     if "a" in x:
5         newlist.append(x)

```

Code 150: Code équivalent à 149, sans liste en compréhension

Comme toute notation plus compacte, la définition de liste en compréhension est à utiliser avec parcimonie : prenez soin de vous assurer que votre code reste *lisible*.

Faites aussi attention à ne pas créer une liste en compréhension à *la place* d'écrire une boucle si c'est ce dont vous avez réellement besoin. Le code 151 montre un exemple, où le comportement *voulu* est d'afficher les dix premiers carrés parfaits ; ceci peut être fait immédiatement dans une boucle `for`, sans passer par une liste. Une liste en compréhension n'est utile que s'il est nécessaire de *stocker* ces données, sans quoi vous utilisez de la mémoire sans raison.

## 6.2 Tuples

Un *tuple* est une séquence d'éléments, comme une liste ; cependant, il s'agit d'une séquence *immuable*, ce qui signifie qu'il est impossible de modifier son contenu une fois le tuple créé.

### 6.2.1 Création d'un tuple

Un tuple est défini par des parenthèses contenant des éléments, séparés par des virgules, comme montré dans le code 152. La syntaxe est proche des listes ; notons malgré tout qu'un tuple d'un seul élément doit malgré tout contenir une virgule, sans quoi les parenthèses seront interprétées comme celles d'une expression mathématique.

La fonction `tuple` permet également de créer un tuple à partir d'un autre objet *itérable*. Elle peut aussi être employée pour créer un tuple en compréhension. Le code 153 montre différentes utilisations de cette fonction.

### 6.2.2 Opérations sur les tuples

L'accès aux données d'un tuple se fait par le même opérateur d'accès que pour une liste, y compris l'utilisation des indices et la possibilité d'effectuer des dé-



```

>>> squares = [i ** 2 for i in range(1,11)]
>>> for square in squares :
...     print(square)
...
1
4
9
16
25
36
49
64
81
100
>>> for i in range(1,11) :
...     print(i ** 2)
...
1
4
9
16
25
36
49
64
81
100

```

Code 151: Comparaison d'une liste en compréhension vs. utilisation directe d'une boucle

```

>>> vide = () # empty tuple
>>> compte = (1, 2, 3) # tuple of integers
>>> quelconque = ("texte", 5, 2.0) # tuple of any elements
>>> unseul = (1,) # tuple with a single element

```

Code 152: Instantiations de tuples

```

>>> vide = tuple()
>>> vide
()
>>> letters = tuple("Hello !")
>>> letters
('H', 'e', 'l', 'l', 'o', ' ', '!')
>>> fruits = tuple(["apple", "banana", "cherry", "kiwi", "mango"])
>>> fruits
('apple', 'banana', 'cherry', 'kiwi', 'mango')
>>> squares = tuple(i ** 2 for i in range(1,11))
>>> squares
(1, 4, 9, 16, 25, 36, 49, 64, 81, 100)

```

Code 153: Instantiations via la fonction `tuple()`

coupes. Le code 154 montre quelques utilisations de cet opérateur : les notations sont identiques.

```
>>> fruits = ("apple", "banana", "cherry", "kiwi", "mango")
>>> fruits[0]
'apple'
>>> fruits[-1]
('apple', 'banana', 'cherry', 'kiwi')
>>> fruits[2::2]
('cherry', 'mango')
```

Code 154: Accès aux éléments d'un tuple

De même, toute opération qui *accède* aux données d'une liste peut aussi s'exécuter sur un tuple. Citons à titre d'exemple les boucles `for`, le mot-clé `in`, ou les méthodes `index` et `count`, comme montrées dans le code 155.

```
>>> fruits = ("apple", "banana", "cherry", "kiwi", "mango")
>>> "apple" in fruits
True
>>> fruits.index("apple")
0
>>> fruits.count("apple")
1
```

Code 155: Recherche de données dans un tuple

### 6.2.3 Assignment à un tuple de variables

Une utilisation unique des tuples par rapport aux listes est la possibilité d'effectuer une assignation directement à un tuple de variables. Cela nous permet aussi d'assigner une valeur à plusieurs variables en une seule ligne de code, en passant à droite de l'assignation un itérable qui compte le même nombre d'éléments.

```
>>> a, b = (7, 13)
>>> a
13
>>> b
7
>>> a, b = (7, 13, 3)
ValueError: too many values to unpack (expected 2)
```

Code 156: Assignment à un tuple de variables

Le code 156 montre un cas simple, où on assigne des valeurs numériques à deux variables. Le gain ici est minime : il aurait été possible de faire deux assignations consécutives `a = 7` puis `b=13`.

Une assignation à un tuple de variables est cependant très utile dans certains cas de figure. Un premier cas est si la nouvelle valeur des variables dépend de l'ancienne valeur de chacune des variables que l'on modifie. Imaginons par exemple que l'on souhaite permuter le contenu de deux variables : dès lors qu'on

modifie l'une d'entre elle, sa valeur ne nous est plus accessible pour l'assigner à l'autre variable, et il faut donc passer par une variable temporaire pour conserver cette valeur, comme montré au code 157.

```
>>> a = 7
>>> b = 13
>>> temp = b
>>> b = a
>>> a = temp
>>> a
13
>>> b
7
```

Code 157: Algorithme standard pour permuter deux variables

Lorsqu'on effectue une assignation à un tuple de variable, ceci n'est plus nécessaire, comme on le voit dans le code 158. En effet, la valeur du tuple `b, a` à droite du symbole d'assignation est calculée dans son entièreté *avant* d'assigner des nouvelles valeurs aux variables `a` et `b`, et aucune valeur ne sera par conséquent perdue.

```
>>> a = 7
>>> b = 13
>>> a, b = b, a
>>> a
13
>>> b
7
```

Code 158: Utilisation d'une assignation à un tuple pour permuter deux variables

Un autre cas de figure où ce type d'assignation est utile est pour récupérer les résultats d'une fonction. Puisqu'une fonction en Python ne peut retourner qu'une seule valeur de retour, lorsqu'une fonction génère plusieurs résultats, il est fréquent de placer ceux-ci à l'intérieur d'un tuple, qui est alors retourné comme résultat. Lors de l'appel d'une telle fonction, il est souvent plus pratique de placer chacun des éléments de ce tuple dans une variable différente ; l'assignation à un tuple de variable nous le permet. Dans le code 159, la fonction `min_max` renvoie les valeurs minimum et maximum dans un itérable ; il est possible de les récupérer en tant que tuple directement, ou d'assigner chacune de ces valeurs à une variable séparée via l'assignation à un tuple de variables.

## 6.3 Ensembles

Un *ensemble* (*set* en anglais) est une structure de données non ordonnée, collectionnant des objets distincts. Un même objet ne peut donc pas se trouver deux fois dans un même ensemble (alors qu'il est possible de placer plusieurs fois un seul objet dans une liste ou un tuple). Pour permettre d'assurer que chaque élément d'un ensemble est unique, tous les éléments d'un ensemble doivent être

```

1 def min_max(iterable):
2     min = max = iterable[0]
3     for n in iterable:
4         if n < min:
5             min = n
6         if n > max:
7             max = n
8     return min, max
9 print(min_max(range(7,13))) # (7, 8, 9, 10, 11, 12)
10 min, max = min_max(range(7,13))
11 print(min) # 7
12 print(max) # 13

```

Code 159: Utilisation d'un tuple comme valeur de retour d'une fonction

*hashables*, ce qui veut dire qu'ils doivent être convertibles en un nombre entier de façon déterministe<sup>1</sup> afin de pouvoir les comparer un par un.

### 6.3.1 Création d'un ensemble

Tout comme les listes se notent en Python par des crochets, les ensembles se notent via des accolades `{}`. Chaque élément de l'ensemble est placé à l'intérieur de celles-ci, et les éléments sont séparés par des virgules.

```

>>> s = {1, 2, 3, 4}
>>> s
{1, 2, 3, 4}
>>> multiples={1, 2, "trois"}
>>> multiples
{1, 2, 'trois'}
>>> duplicates = {1, 1, 2, 3, 4}
>>> duplicates
{1, 2, 3, 4}
>>> nb_pairs = {i for i in range(2,11,2)}
>>> nb_pairs
{2, 4, 6, 8, 10}

```

Code 160: Instanciation d'ensembles

Le code 160 montre quelques exemples de créations d'ensembles. Notons :

- ▷ la possibilité de placer des éléments de plusieurs types dans un même ensemble ;
- ▷ si un ensemble est défini avec des éléments en doublon, l'ensemble produit ne contient qu'un seul élément : les doublons sont éliminés automatiquement ;
- ▷ la possibilité de définir un ensemble en compréhension, de la même façon que pour une liste.

La fonction `set` permet aussi d'initialiser un ensemble à partir d'un autre itérable. Les éléments de cet itérable sont alors parcourus et ajoutés à l'ensemble créé. Si un élément est présent plusieurs fois dans l'itérable, il n'est bien sûr présent

1. Nous n'aborderons pas ce concept plus en détails dans le cadre de ce cours.

```
>>> spell = set("abracadabra")
>>> spell
{'a', 'r', 'd', 'b', 'c'}
```

Code 161: Ensemble créé à partir d'une chaîne de caractères

qu'une seule fois dans l'ensemble résultant. Le code 161 montre une initialisation d'un ensemble à partir d'une chaîne de caractère : chaque caractère est alors présent dans l'ensemble créé une seule fois, en ignorant les lettres en doublon.

### 6.3.2 Opération sur les ensembles

#### Taille d'un ensemble

À l'instar des listes, la fonction `len(set)` permet de compter le nombre d'éléments présents dans un ensemble, comme illustré au code 162.

```
>>> spell = set("abracadabra")
>>> spell
{'a', 'r', 'd', 'b', 'c'}
>>> len(spell)
5
```

Code 162: Longueur d'un ensemble

#### Consulter les éléments d'un ensemble

Puisqu'un ensemble n'a pas d'ordre, il n'est pas possible d'accéder à un élément via un indice ou autre clé. On peut, en revanche, interroger un ensemble pour savoir si un élément y est présent, de façon similaire à toute autre séquence, via le mot-clé `in`, comme illustré au code 163.

```
>>> spell = set("abracadabra")
>>> spell
{'a', 'r', 'd', 'b', 'c'}
>>> "a" in spell
True
>>> "k" in spell
False
```

Code 163: Mot-clé `in` sur un ensemble

#### Parcourir un ensemble

Il est aussi possible de parcourir un ensemble via une boucle `for`, comme pour toute structure de données. Le code 164 montre un exemple d'une telle boucle, qui calcule la somme des éléments d'un ensemble.

Attention toutefois : puisqu'un ensemble n'a pas de notion d'ordre, il ne faut pas compter sur le fait que les éléments de l'ensemble seront parcouru dans un certain ordre ! Le code 165 montre un exemple où ce problème se déclare : on

```
>>> nb_pairs = {i for i in range(2,11,2)}
>>> nb_pairs
{2, 4, 6, 8, 10}
>>> sum = 0
>>> for i in nb_pairs :
...     sum += i
...
>>> sum
30
```

Code 164: Boucle `for` sur un ensemble

décompose une phrase en ses mots composants, qui sont ajoutés à un ensemble `sentence`. Lorsqu'on recrée une nouvelle phrase dans laquelle on ajoute chaque mot de l'ensemble `sentence`, les mots se sont mélangés !

```
>>> words = "the quick brown fox jumps over the lazy dog".split(" ")
>>> sentence = set(words)
>>> new_sentence = ""
>>> for word in sentence :
...     new_sentence += word + " "
...
>>> new_sentence
'the quick lazy brown dog fox over jumps '
```

Code 165: Influence du manque d'ordre lors d'un parcours d'ensemble

### Modification d'un ensemble

À l'instar des listes, les ensembles possèdent des *méthodes* permettant de modifier leur contenu :

- ▷ `add(elem)` permet d'ajouter l'élément `elem` à l'ensemble ;

```
>>> s = {1, 2, 3, 4}
>>> s.add(5)
>>> s
{1, 2, 3, 4, 5}
```

- ▷ `remove(elem)` permet de supprimer l'élément `elem` de l'ensemble, et déclenche une erreur si l'élément n'y est pas présent ;

```
>>> s = {1, 2, 3, 4}
>>> s.remove(4)
>>> s
{1, 2, 3}
>>> s.remove(5)
KeyError: 5
```

- ▷ `discard(elem)` permet de supprimer l'élément `elem` de l'ensemble, mais ne déclenche *pas* une erreur si l'élément n'y est pas présent ;

```
>>> s = {1, 2, 3, 4}
>>> s.discard(4)
>>> s
```

```
{1, 2, 3}
>>> s.discard(5)
>>> s
{1, 2, 3}
```

- ▷ `pop()` supprime un élément de l'ensemble, choisi de façon arbitraire, et le renvoie comme résultat ;

```
>>> s = {1, 2, 3, 4}
>>> s.pop()
1
>>> s
{2, 3, 4}
```

- ▷ `clear()` vide l'ensemble.

```
>>> s = {1, 2, 3, 4}
>>> s.clear()
>>> s
set()
```

### Comparaison d'ensembles

En mathématiques, il est possible de comparer deux ensembles via les opérateurs d'inclusion :  $A \subseteq B$  si tous les éléments de  $A$  sont aussi des éléments de  $B$ , et inversement,  $A \supseteq B$  si tous les éléments de  $B$  sont aussi des éléments de  $A$ . Enfin, deux ensembles sont considérés comme égaux s'ils contiennent exactement les mêmes éléments.

Puisque les symboles correspondant à ces opérateurs ne font pas partie de tous les encodages et clavier standard, Python a redéfini les symboles de comparaison numériques, lorsqu'ils sont employés sur des ensembles, pour traduire ces opérateurs mathématiques. Le code 166 montre l'utilisation de ces différents opérateurs. Une inclusion *stricte* est représentée via `<` et `>`, une inclusion *non stricte* via `<=` et `>=`, et l'égalité et l'inégalité d'ensembles via `==` et `!=`.

### Opérations sur des ensembles

Les ensembles mathématiques possèdent un certains nombres d'opérations permettant de combiner deux ensembles. Les opérations de base sont traduits en Python via les opérateurs suivants :

- ▷ L'*union* de deux ensembles, `|`, est un ensemble qui contient tous les éléments présents dans l'un ou l'autre des opérandes.
- ▷ L'*intersection* de deux ensembles, `&`, est un ensemble qui contient tous les éléments présents dans *les deux* opérandes.
- ▷ La *différence* de deux ensembles, `-`, est un ensemble qui contient tous les éléments présents dans le premier opérande, mais *pas* dans le deuxième.
- ▷ La *différence symétrique* de deux ensembles, `^`, est un ensemble qui contient tous les éléments présents soit dans le premier opérande, soit dans le deuxième, mais *pas* dans les deux.

Le code 167 illustre l'emploi de ces différents opérateurs et leurs résultats.

```

>>> a = {1, 2, 3, 4}
>>> b = {i for i in range(10)}
>>> b
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> c = {1, 2, 3, 4}
>>> a < b
True
>>> a < c
False
>>> a <= b
True
>>> a <= c
True
>>> b > a
True
>>> a >= c
True
>>> a == c
True
>>> a == b
False
>>> a != b
True

```

Code 166: Comparaison d'ensembles

```

>>> a = set("abracadabra")
>>> b = set("alacazam")
>>> a
{'a', 'r', 'd', 'b', 'c'}
>>> b
{'a', 'm', 'l', 'c', 'z'}
>>> a | b
{'a', 'r', 'd', 'b', 'm', 'l', 'c', 'z'}
>>> a & b
{'a', 'c'}
>>> a - b
{'d', 'b', 'r'}
>>> a ^ b
{'l', 'r', 'd', 'z', 'b', 'm'}

```

Code 167: Opérations sur des ensembles



## 6.4 Dictionnaires

Un *dictionnaire* est une structure de données qui associe à chaque valeur une *clé*, qui ne doit pas nécessairement être numérique, contrairement aux indices dans une liste ou un tuple. Ces clés doivent par contre être *uniques* : deux éléments ne partagent pas la même clé. Par conséquent, de la même façon que les éléments d'un ensemble, les clés d'un dictionnaire doivent être *hachables* afin de pouvoir tester leur unicité ; en outre, le langage Python que les clés soit *immuables*, ce qui inclut les nombres (entiers ou décimaux) mais aussi les chaînes de caractères et tuples. Toujours de façon similaire aux ensembles, les clés d'un dictionnaire n'ont pas d'ordre particulier.

On peut par conséquent voir un dictionnaire de deux façon différentes :

- ▷ comme un *ensemble* de clés, chaque clé correspondant à une valeur unique,
- ▷ comme une *séquence* de paires clé - valeur.

Ces deux approches seront utilisées lors des définitions et manipulations de dictionnaires.

### 6.4.1 Création d'un dictionnaire

Puisqu'un dictionnaire peut être perçu comme un ensemble de clés, à chacune desquelles correspond une valeur, le langage Python note aussi les dictionnaires entre accolades `{}`. Cependant, à l'intérieur de celles-ci, on ne note pas une série d'éléments séparés par des virgules, mais une série de *paires* clé - valeur, en respectant la syntaxe `clé : valeur`.

```
>>> vide = {} # empty dictionary
>>> type(vide)
<class 'dict'>
>>> traduction = {"one" : "un", "two" : "deux", "three" : "trois"}
>>> multiples = {1: "objet 1", "deux" : "objet 2", "trois" : 3}
>>> emails = {("Turing", "Alan") : "aturing@npl.co.uk", ("Lovelace", "Ada") :
    ↪ "ada@lovelace.co.uk"}
>>> liste_cle = {["apple", "orange"] : 2}
TypeError: unhashable type: 'list'
```

Code 168: Instanciation de dictionnaires

Le code 168 montre quelques exemples de créations de dictionnaires. Notons :

- ▷ par défaut, la notation `{}` sans élément à l'intérieur crée un *dictionnaire* vide, et non un *ensemble* vide.
- ▷ qu'il est possible de mélanger plusieurs types de clé dans un même dictionnaire ;
- ▷ qu'il est aussi possible de placer des *valeurs* de types différents dans un même dictionnaire ;
- ▷ que le dictionnaire `emails` utilise des tuples en tant que clés ;
- ▷ qu'utiliser une liste en tant que clé, par contre, déclenche une erreur.

```
>>> squares = {i : i**2 for i in range(1,11)}
>>> squares
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100}
```

Code 169: Dictionnaire en compréhension

Il est également possible de définir un dictionnaire en compréhension, de la même façon que pour une liste. Le code 169 montre un dictionnaire qui utilise chaque nombre entier de 1 à 10 comme clé, et lui associe le carré de ce nombre comme valeur.

Un dictionnaire peut aussi être perçu comme un ensemble de paires clé - valeur. Par conséquent, il est aussi possible de créer un dictionnaire à partir d'une séquence de *tuples*, si chacun de ces tuples contient exactement deux éléments. Le premier élément servira alors que clé, et le deuxième élément de valeur.

```
>>> matricules_noms = ((72634, "Maurice Convert"), (82647, "Keeley
    ↪ Pattison"), (28374, "Keri Sadiq"))
>>> students = dict(matricules_noms)
>>> students
{72634: 'Maurice Convert', 82647: 'Keeley Pattison', 28374: 'Keri Sadiq'}
```

Code 170: Dictionnaire créé à partir d'une séquence de tuples

Le code 170 montre un exemple de création d'un dictionnaire contenant une liste de noms (fictifs) d'étudiants, chacun étant associé à son matricule étudiant en tant que clé. Ces informations sont données au dictionnaire via un tuple qui contient lui-même plusieurs tuples de deux éléments, chacun contenant un matricule et un nom.

## 6.4.2 Opérations sur les dictionnaires

### Taille du dictionnaire

À l'instar des listes, la fonction `len(dictionnaire)` permet de compter le nombre d'éléments présents dans un dictionnaire, comme illustré au code 171.

```
>>> traduction = {"one" : "un", "two" : "deux", "three" : "trois"}
>>> len(traduction)
3
```

Code 171: Longueur d'un dictionnaire

### Accès à un élément de dictionnaire

L'opérateur d'accès, commun aux listes et tuples, existe aussi sur les dictionnaires. On peut donc accéder à un élément d'un dictionnaire via des crochets [], en plaçant la *clé* de l'élément auquel on souhaite accéder entre les crochets.

Le code 172 montre des exemples d'accès aux données. En particulier, tenter d'accéder à une clé qui n'existe pas dans le dictionnaire déclenche une erreur. Ce

```
>>> traduction = {"one" : "un", "two" : "deux", "three" : "trois"}
>>> traduction["one"]
'un'
>>> traduction["three"]
'trois'
>>> traduction["five"]
KeyError: 'five'
```

Code 172: Opérateur d'accès sur un dictionnaire

comportement est parfois souhaitable afin de pouvoir traiter cette erreur lorsqu'elle se déclenche ; cependant, il est parfois souhaitable d'éviter de déclencher une erreur. Ceci est faisable via la méthode `get`, qui sert aussi à récupérer un élément d'un dictionnaire à partir de sa clé, mais avec la possibilité de renseigner une *valeur par défaut*, qui est renvoyée si la clé n'est pas présente.

```
>>> traduction = {"one" : "un", "two" : "deux", "three" : "trois"}
>>> traduction.get("one", "je ne sais pas")
'un'
>>> traduction.get("three", "je ne sais pas")
'trois'
>>> traduction.get("five", "je ne sais pas")
'je ne sais pas'
>>> traduction.get("one")
'un'
>>> traduction.get("five") # returns nothing
```

Code 173: Méthode `get` de dictionnaire

Le code 173 montre des exemples d'utilisation de cette méthode. Notons que la valeur par défaut est optionnelle ; si elle n'est pas renseignée et que la clé demandée n'est pas présente dans le dictionnaire, rien n'est renvoyé. Il faudra malgré tout alors s'assurer qu'un manque de valeur de retour est compatible avec le reste de votre code.

### Consulter les clés d'un dictionnaire

Deux fonctions permettent d'obtenir la liste des clés présentes dans un dictionnaire. La première est la fonction `list` (fonction de création de liste) ; si elle reçoit un dictionnaire en argument, elle retourne une liste des clés du dictionnaire. Ces clés ne sont toutefois pas ordonnées de façon prévisible : selon l'interpréteur Python, les clés peuvent apparaître dans un ordre différent (le plus fréquent est l'ordre d'*ajout au dictionnaire*, mais ce n'est pas garanti).

Si malgré tout il est nécessaire d'obtenir les clés triées, la fonction `sorted` retourne les clés, cette fois-ci triées (selon le type de données ; ordre croissant pour des clés numériques, ordre lexicographiques pour des chaînes de caractère). Attention toutefois : *toutes* les clés du dictionnaire doivent être comparables, sans quoi cette fonction déclenche une erreur.

Le code 174 montre des exemples de ces deux fonctions.

```
>>> traduction = {"one" : "un", "two" : "deux", "three" : "trois"}
>>> list(traduction)
['one', 'two', 'three']
>>> sorted(traduction)
['one', 'three', 'two']
>>> dico_mixte = {"c" : 1, "abc" : 2, 5 : 3, 3 : 2}
>>> list(dico_mixte)
['c', 'abc', 5, 3]
>>> sorted(dico_mixte)
TypeError: '<' not supported between instances of 'int' and 'str'
```

Code 174: Fonctions `list` et `sorted` sur un dictionnaire

Si on souhaite vérifier si une clé spécifique est présente dans un dictionnaire, sans récupérer la valeur associée, le mot-clé `in` remplit cette fonction, comme montré dans le code 175.

```
>>> traduction = {"one" : "un", "two" : "deux", "three" : "trois"}
>>> "one" in traduction
True
>>> "five" in traduction
False
```

Code 175: Mot-clé `in` avec un dictionnaire

### Ajout et suppression d'éléments

Puisque les éléments d'un dictionnaire ne sont pas ordonnés, l'ajout et la modification d'éléments dans un dictionnaire se fait via l'opérateur d'accès, et la suppression via le mot-clé `del`, comme illustré au code 176. Il n'y a pas d'équivalent aux méthodes `append`, `insert`, `remove`, `extend`. Par contre, la méthode `pop` permet toujours de supprimer un élément et de le récupérer, et la méthode `clear` est disponible pour vider un dictionnaire de toutes ses valeurs.

### Parcours d'un dictionnaire

Comme toute séquence, un dictionnaire peut être parcouru à l'aide d'une boucle `for`. Cette boucle itère alors sur les *clés* du dictionnaire, comme montré au code 177. Si on souhaite accéder aux valeurs, il faudra donc utiliser l'opérateur d'accès; la méthode `get` n'est pas particulièrement utile puisqu'on est sûr que les clés sont présentes.

Parfois, plutôt que d'utiliser l'opérateur d'accès, on souhaite accéder à la fois à la clé et la valeur correspondante dans des variables lors de notre parcours, sans devoir répéter l'opérateur d'accès. C'est possible via la méthode `items()` des dictionnaires, qui renvoie une liste de tuples, chaque tuple contenant la clé et sa valeur correspondante. Il est alors possible de parcourir cette liste via une assignation à un tuple de variable, comme démontré dans le code 178.

```

>>> traduction = {"one" : "un", "two" : "deux", "three" : "trois"}
>>> traduction["four"] = "quatre"
>>> traduction
{'one': 'un', 'two': 'deux', 'three': 'trois', 'four': 'quatre'}
>>> traduction["four"] = "pas quatre"
>>> traduction
{'one': 'un', 'two': 'deux', 'three': 'trois', 'four': 'pas quatre'}
>>> del traduction["four"]
>>> traduction
{'one': 'un', 'two': 'deux', 'three': 'trois'}
>>> del traduction["five"]
KeyError: 'five'
>>> traduction.pop("one")
'un'
>>> traduction
{'two': 'deux', 'three': 'trois'}
>>> traduction.clear()
>>> traduction
{}

```

Code 176: Insertion et suppression dans un dictionnaire

```

>>> traduction = {"one" : "un", "two" : "deux", "three" : "trois"}
>>> for k in traduction:
...     print(k)
...
one
two
three
>>> for k in traduction:
...     print(traduction[k])
...
un
deux
trois

```

Code 177: Boucles `for` sur un dictionnaire

```

>>> traduction = {"one" : "un", "two" : "deux", "three" : "trois"}
>>> traduction.items()
dict_items([('one', 'un'), ('two', 'deux'), ('three', 'trois')])
>>> for anglais, francais in traduction.items():
...     print("Le mot anglais " + anglais + " veut dire " + francais + " en
      ↪ francais")
...
Le mot anglais one veut dire un en francais
Le mot anglais two veut dire deux en francais
Le mot anglais three veut dire trois en francais

```

Code 178: Utilisation de la méthode `items` dans une boucle `for`

## 6.5 Bien choisir sa structure de données

Les listes, tuples et dictionnaires sont tous les trois des séquences d'éléments, et possèdent des points communs concernant leur utilisation. Par conséquent, lorsqu'on souhaite définir une structure de données, il peut être tentant de n'utiliser que l'une de ces structures, sans jamais avoir recours aux autres. C'est une mauvaise idée ! Ce sont justement les spécificités de chacune de ces structures qui font leur intérêt.

De façon générale, on choisira une structure selon les besoins spécifiques d'un problème :

- ▷ Une *liste* est adaptée lorsque nos données sont stockées de manière *séquentielle* (i.e. dans un ordre précis) et qu'on y accède (souvent) de manière séquentielle également, mais que le *contenu* des données peut changer (modification de valeurs présentes, mais aussi ajout ou suppression d'éléments)
- ▷ Un *tuple* est adapté lorsque les données sont stockées et parcourues de manière *séquentielle*, mais que les données n'ont *pas* besoin de changer. On peut d'ailleurs utiliser un tuple pour *s'assurer* que les données ne changeront pas.
- ▷ Un *ensemble* est adapté lorsqu'il est uniquement nécessaire d'interroger l'ensemble sur son contenu : possède-t-il une valeur précise, ou est-il inclus dans un autre ensemble ;
- ▷ Un *dictionnaire* est adapté lorsqu'on a besoin de consulter les données de façon non séquentielle, car l'accès à un élément d'un dictionnaire par sa clé est très efficace ; en d'autres termes, un dictionnaire est utile s'il est possible, lors d'un accès standard aux données, de formuler une clé qui déterminera la donnée qui nous intéresse à l'intérieur de la structure.