

# Chapitre 1

## Types et opérateurs

### À la fin de ce chapitre, je sais :

- ✎ Utiliser et identifier les types simples (int, float, boolean, complex)
- ✎ Utiliser les opérateurs en lien avec les types simples numériques
- ✎ Utiliser une variable de type simple
- ✎ Reconnaître les principaux mots-clefs du langage Python

### A Types

L'informatique traite l'information. Dans ce but, il faut être capable de stocker l'information sous une forme accessible au traitement informatique. Selon la nature de l'information, un type de données différent est choisi pour la représenter.

■ **Définition 1 — Typage.** Le typage désigne l'action de choisir une représentation à une donnée selon ses caractéristiques. Cette représentation est nommée type. Le typage est effectué soit par le programmeur, soit par le compilateur soit par l'interpréteur.

■ **Définition 2 — Type simple.** Les types simples correspondent à des informations comme les nombres, des constantes ou les valeurs booléennes.

Les types simples en Python sont :

- `int` permet de représenter un sous-ensemble des entier relatif,
- `float` permet de représenter un sous-ensemble des décimaux,
- `complex` permet de représenter un sous-ensemble des nombres complexes,
- `bool` permet de représenter une valeur booléenne, vrai ou faux.

Les codes 1.1 donnent des exemples d'usage de ces types en Python. Il faut noter que , comme de nombreux langages modernes, Python est un langage multiparadigme. Il implémente notamment le paradigme objet, c'est pourquoi les types simples sont des classes d'objet.

#### Code 1.1 – Types simples

```
1 print(type(42))      # <class 'int'> an integer
2 print(type(2.37))   # <class 'float'> a floating point number
3 print(type(3-2j))   # <class 'complex'> a complex number
4 print(True, type(True)) # True <class 'bool'>
5 print(False, type(False)) # False <class 'bool'>
```

■ **Définition 3 — Inférence de type** . L'inférence de type est une action d'un compilateur ou d'un interpréteur qui permet de typer une données automatiquement d'après sa nature.


■ **Définition 4 — Typage explicite** . Un langage est dit à typage explicite s'il exige de toute donnée qu'elle soit déclarée selon un type. Le contraire est un typage **implicite**.

■ **Définition 5 — Typage dynamique** . Un langage est dit à typage dynamique si une variable peut changer de type au cours de l'exécution du programme. Inversement, on parle de typage **statique**.

Dans l'exemple ci-dessus, Python fait donc de l'inférence de type, puisque la donnée 42 est interprétée comme un `int`, 2.37 comme un `float`...

**P** Python est un langage à typage implicite et dynamique.

■ **Définition 6 — Transtypage**. Transtyper une variable c'est modifier son type en opérant une conversion de la donnée.

 **Vocabulary 1 — Type Casting** ↔ En anglais, le transtypage est désigné par les termes *type casting* ou *type conversion*.

**P** En Python, on peut transtyper une variable numérique en utilisant un constructeur numérique `int` ou `float`. Par exemple :

```
1 b = float(3)
2 print(b, type(b)) # 3.0 <class 'float'>
3 c = int(3.56)
4 print(c, type(c)) # 3 <class 'int'>
```

■ **Définition 7 — Types composés.** Un type composé est un type de données qui agrège des types simples dans un objet homogène (tableau, énumération) ou inhomogène (structure, union). Dans les types composés (in)homogènes, les données (ne) sont (pas) toutes du même type.

**P** En Python les types composés sont les chaînes des caractères, les tuples, les listes, les ensembles et les dictionnaires.

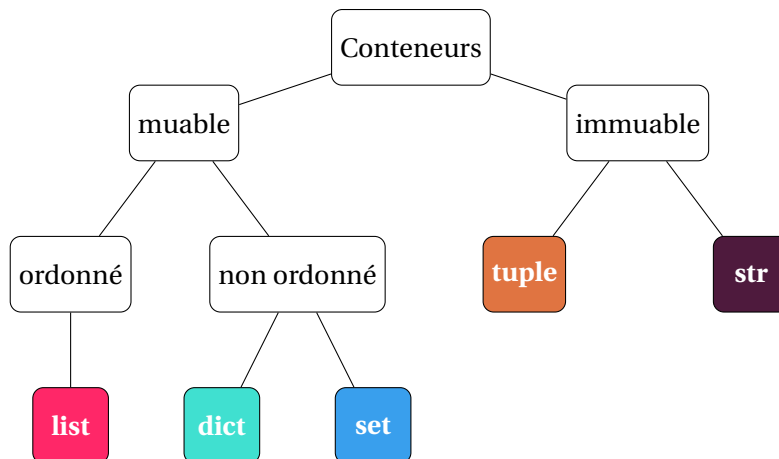


FIGURE 1.1 – Types de données composés du langage Python : les conteneurs


## B Opérateurs

Les opérateurs peuvent être classés en catégories selon la nature de leur action, le type d'opérandes ou le nombre d'opérandes dont ils ont besoin.

### a Opérateurs arithmétiques

Ce sont les transpositions informatiques des opérateurs mathématiques : ils agissent sur des représentations de nombre en machine, les types numériques.

- $a + b$  l'addition,
- $a - b$  la soustraction,
- $a * b$  la multiplication,
- $a // b$  la division entière,
- $a \% b$  l'opération modulo,
- $a / b$  la division,
- $a ** n$  l'élévation à la puissance  $n$  ou exponentiation.

 **Vocabulary 2 — Arithmetic operators**  $\rightsquigarrow$  En anglais, on les désigne par :

- Addition,
- Substraction,
- Multiplication,
- Floor division,
- Modulus,
- Division,
- Exponentiation.

Merci le latin et Guillaume le Conquérant.

**P** Les opérateurs `+`, `-` et `*` s'appliquent indifféremment aux types `int` et `float`. Néanmoins, le résultat d'une opération entre `int` sera un `int`. Le résultat est un `float` si l'une des deux opérandes au moins en est un. On parle alors de transtypage implicite, les `int` sont transtypés par l'interpréteur en `float` pour réaliser l'opération, de manière transparente.

L'opérateur de division `/` renvoie en un `float` alors que `//` renvoie un `int` : c'est en fait le quotient de la division euclidienne ou la partie entière du résultat de `/`, d'où le nom en anglais.

L'opérateur `**` s'applique aux `int` et aux `float` et renvoie un `float`.

L'opérateur `%` renvoie généralement le reste de la division euclidienne de `a` par `b`. Cependant, lorsque les deux opérandes sont négatives, le modulo résulte en nombre négatif afin d'avoir également  $a = (a//b) * b + a \% b$ .

Tous les langages n'adoptent pas forcément les mêmes conventions que Python, il faut donc rester vigilant.

La syntaxe infixe de ces opérateurs fait qu'il peut être ambigu d'écrire certaines expressions. Par exemple, comment interpréter `2 + 3 * 4`? Doit-on calculer d'abord l'addition `2+3` puis multiplier le résultat? Ou bien doit-on effectuer la multiplication d'abord? Le résultat ne sera pas le même. Quelle est la bonne opération? Quel opérateur doit-on appliquer en premier? Sans parenthèses, cette ambiguïté demeure à moins qu'on ne décide d'attribuer une priorité différente aux opérateurs. C'est ce qui est fait par la plupart des langages.

**P** La priorité des opérateurs en Python est définie comme indiqué sur le tableau 1.1. C'est pourquoi l'expression `2 + 3 * 4` est évaluée en  $14^a$ . De plus, lorsque le degré de priorité est identique, on associe en priorité de la gauche vers la droite. Ainsi, l'expression `15 / 5 * 2` est évaluée en  $6^b$ .

*a.* et non pas 20

*b.* et non pas 1.5

**R** Attention : certains opérateurs apparaissent identiques syntaxiquement mais se comporte différemment selon le type de donnée. Par exemple `*` ou `+` sont des opérateurs qui peuvent

Priorité	Opérateurs	Description
1	()	parenthèses
2	**	exponentiation
3	+x -x ~x	plus et moins unaire, négation bits à bits
4	* / // %	multiplication, division division entière, modulo
5	+ -	addition, soustraction
6	<< >> >>= <<=	décalage binaire
7	&	et bits à bits
8	^	ou exclusif bits à bits
9		ou bits à bits
10	==, != >, >=, <, <= is, is not, in, not in	identité comparaison appartenance
11	not	négation logique
12	and	et logique
13	or	ou logique

TABLE 1.1 – Priorités des opérateurs en Python : dans l'ordre d'apparition, du plus prioritaire (1) au moins prioritaire (13)

également agir sur les chaînes de caractères ou sur les listes. Dans ce cas, ils possèdent une autre signification.

## b Opérateurs logiques

■ **Définition 8 — Opérateur logique.** Les opérateurs logiques produisent une valeur booléenne à partir d'autres valeurs booléennes en les combinant. Ils prennent le plus souvent deux opérandes.

Les opérateurs logiques Python sont :

- `a and b` la conjonction, renvoie `True` si `a` et `b` sont tous les deux à vrais,
- `a or b` la disjonction, renvoie `True` si `a` ou `b` sont vrais,
- `not a` la négation, renvoie `True` si `a` est faux, `False` sinon.

**P** En Python les valeurs booléennes sont `True` et `False`. Ce sont deux objets qui peuvent être interprétés numériquement par 0 ou 1.

Il faut noter également que d'autres objets peuvent être interprétés comme `False` :

- `None` qui est un l'objet qui représente *rien*,
- les représentations de zéro pour les types numériques dont `int`, `float`,

- les séquences et collections vides : "", (), [], , set(), range()

**M** **Méthode 1 — Tester un booléen** Pour tester la valeur d'un booléen `a`, on n'écrit jamais `if a == True:`, mais simplement `if a:`. De même, pour tester le cas faux, on écrira `if not a:`.

## c Opérateurs d'affectation

■ **Définition 9 — Affectation.** L'affectation est l'opération qui consiste à assigner une valeur à une variable et donc de modifier son état en mémoire.

Le fait que l'affectation soit une instruction est caractéristique des langages impératifs. En Python, il existe plusieurs opérateurs d'affectation :


- `a = 3` affectation simple,
- `a += 3` affectation simple combinée avec une addition équivalent à `a = a + 3`,
- `a -= 3` affectation simple combinée avec une soustraction équivalent à `a = a - 3`.

L'affectation combinée évite à l'interpréteur de créer une variable intermédiaire pour le calcul. L'opération se fait in place, c'est à dire sur l'espace mémoire même associé à la variable.

## C Opérateurs sur les chaînes de caractères

Les chaînes de caractères sont des objets immuables en Python. Leurs valeurs sont initialisées en utilisant les guillemets `"`. Certains symboles sont utilisés pour désigner plusieurs opérateurs différents. C'est le cas par exemple de `+` et `*` qui peuvent désigner des opérateurs sur les chaînes de caractères. C'est ce qu'illustre l'exemple suivant

```
1 s = "Z comme "
2 print(s + " Zorglub")      # Z comme  Zorglub
3 print(s + "42")           # Z comme 42
4 print(s*3)                # Z comme Z comme Z comme
```

 **Vocabulary 3 — Double and simple quotes** ↔ Les guillemets `"` correspondent au mot anglais *double quote*. Il existe aussi le symbole apostrophe `'`, *simple quote* en anglais, qui permet également d'initialiser des chaînes de caractères.

## D Une variable Python est une référence

Tout est dans le titre de la section, mais je vais l'écrire comme une règle d'or :

**P** En Python, une variable est une référence vers l'adresse d'un objet en mémoire.

■ **Définition 10 — Type immuable.** Un type immuable est un type de donnée que l'on peut modifier. Inversement, un type muable est un type dont on peut modifier la valeur.

**P** En Python, les types immuables sont :

- les types simples numériques : `int`, `float`, `bool`,
- les `str`,
- les `tuple`.

Les types muables sont :

- les listes `list`,
- les dictionnaire `dict`,
- les ensembles `set`.

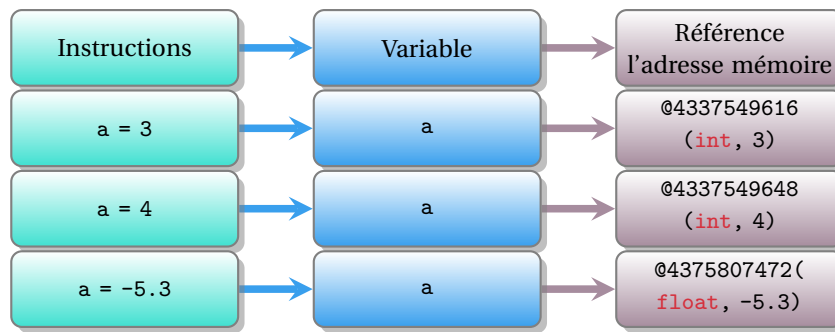
Ces types muables et immuables combinés à la règle d'or précédente permettent d'expliquer énormément de codes qui peuvent parfois sembler très peu clairs au débutant. C'est par exemple ce qui explique pourquoi l'affectation ne se comporte pas de la même manière avec toutes les variables, ce que l'on observera au cours des TP de l'année.

■ **Exemple 1 — Affectation d'immuables.** Un entier est une donnée immuable : on ne peut pas en modifier la valeur, 42 vaut 42. De même pour un nombre flottant ou un booléen, vrai reste vrai et faux, faux... En Python, une variable à qui on a affecté une donnée référence un objet d'un certain type représentant cette donnée en mémoire. Si on affecte une nouvelle donnée à une variable immuable, le type étant immuable, c'est donc à la référence de changer. L'interpréteur alloue donc un nouvel espace mémoire à cette donnée et la variable référence ce nouvel objet. C'est ce qu'illustrent les figures 1.2 et 1.3 avec un type `int` immuable et un type `list` muable.

L'ancienne valeur peut demeurer un moment en mémoire, jusqu'à ce que le ramasse-miette de l'interpréteur (Garbage Collector) désalloue cet espace mémoire pour libérer la mémoire, si cette valeur n'est plus référencée par aucune variable.

## E Mots-clefs du langage

Les mots-clefs d'un langage sont des mots réservés, c'est à dire qu'on ne peut et qu'on ne doit pas les utiliser pour nommer des variables pour des fonctions. Ils sont rassemblés sur le tableau 1.2.

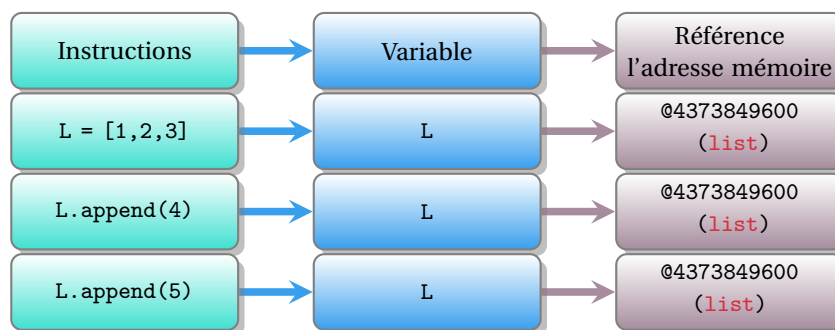
**Code 1.2 – Affectations de type immuable**

```

1 a = 3
2 print(a, id(a)) #3 4337549616
3 a = 4
4 print(a, id(a)) #4 4337549648
5 a = -5.3
6 print(a, id(a)) #-5.3 4375807472

```

FIGURE 1.2 – Variable de type immuable, affectation et référencement en mémoire (à gauche) et programme Python pour la visualisation des adresses en mémoire (à droite)

**Code 1.3 – Affectations de type muable**

```

1 L = [1, 2, 3]
2 print(L, id(L))
3 # [1, 2, 3] 4373849600
4 L.append(4)
5 print(L, id(L))
6 # [1, 2, 3, 4] 4373849600
7 L.append(5)
8 print(L, id(L))
9 # [1, 2, 3, 4, 5] 4373849600

```

FIGURE 1.3 – Variable de type muable, affectation et référencement en mémoire (à gauche) et programme Python pour la visualisation des adresses en mémoire (à droite)



Mot-clef	Usage
<code>if</code>	structure conditionnelle
<code>elif</code>	structure conditionnelle
<code>else</code>	structure conditionnelle
<code>for</code>	pour créer une boucle pour
<code>while</code>	pour créer une boucle tant que
<code>break</code>	sortir d'une boucle
<code>continue</code>	continuer la boucle
<code>pass</code>	pour ne rien faire
<code>and</code>	et logique
<code>not</code>	négation logique
<code>or</code>	ou logique
<code>False</code>	valeur booléenne faux
<code>True</code>	valeur booléenne vrai
<code>assert</code>	créer une assertion en programmation défensive
<code>None</code>	constante pour représenter le rien
<code>in</code>	pour vérifier si une valeur est présente dans une séquence
<code>is</code>	pour tester l'égalité de deux variables
<code>from</code>	pour importer tout ou partie d'un module
<code>import</code>	pour importer un module
<code>as</code>	pour créer un alias
<code>def</code>	pour définir une fonction
<code>return</code>	pour sortir d'une fonction et renvoyer une valeur
<code>lambda</code>	pour créer des fonctions anonymes
<code>raise</code>	pour lever une exception
<code>try</code>	pour gérer les exception
<code>except</code>	pour gérer les exceptions
<code>finally</code>	pour gérer les exceptions
<code>with</code>	pour gérer les exceptions
<code>class</code>	définir une classe
<code>del</code>	pour supprimer un objet
<code>global</code>	pour déclarer une variable globale et l'utiliser
<code>nonlocal</code>	pour déclarer une variable non locale
<code>yield</code>	pour créer une coroutine (hors programme)

TABLE 1.2 – Mots-clefs du langage Python