Séance 1: Données et opérations de base

Philipp Ahrendt

November 28, 2025

Contents

1 Types et opérations simples

En python, les données sont catégorisées en *types* (ou *classes*).

Voici les types basiques les plus importants:

Type python	Description
`int`	Nombres entiers
`bool`	Vrai ou faux
`float`	Nombres décimaux
`str`	Chaînes de caractères

Il y a aussi un objet "vide" `None`, qui a son type à part `NoneType` (c'est l'équivalent du `void` de C++).

Exemples:

```
n = 5
print("type de", n, ":", type(n),"\n")

pi = 3.14
print("type de", pi, ":", type(pi),"\n")

phrase = "Hello World!"
print("type de", phrase, ":", type(phrase),"\n")


f = (1 == 0)
print("type de", f, ":", type(f),"\n")


print("type de", None, ":", type(None),"\n")

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

Remarque: Contrairement à C++, on n'est pas obligé de donner le type d'une variable avant de l'initialiser, python déduit le type de la valeur attribuée.

1.1 Opérations sur `int` et `float`.

Avec `int` et `float`, on peut faire des calculs simples (addition, multiplication, etc...). Pour la plupart on peut mélanger sans problème des entiers et flottants. Python fait les conversions nécessaires.

Notation	Opération	Type du résultat
`i+j`	Addition	`int` si `i` et `j` sont de type `int`, sinon `float`
`i-j`	Soustraction	`int` si `i` et `j` sont de type `int`, sinon `float`
`i*j`	Multiplication	`int` si `i` et `j` sont de type `int`, sinon `float`
`i/j`	Division	`float`
`i**j`	Puissance	`float`

Remarque: On peut aussi additionner deux `str`

```
print("Hello " + "world!")
```

1.2 Opérations sur `int` uniquement

Notation	Opération
`i//j`	Division entière
`i%j`	Reste de la division entière

1.3 Opérations sur `bool`.

Notation	Opération
`a and b`	Conjonction
`a or b`	Disjonction
`not a`	Négation

1.4 Comparaisons (`int`, `float` et `str`, résultat de type `bool`)

Sur des nombres, les comparaison font ce à quoi on s'attend.

Notation	Opération
`i == j`	Test d'égalité
`i != j`	Test de différence
`i <= j`	Test d'infériorité
`i >= j`	Test de supériorité
`i < j`	Test d'infériorité stricte
`i > j`	Test de supériorité stricte

On peut aussi les utiliser sur des chaînes de caractères, ce qui peut être pratique.

```
print("Hello == Hello")
print("Hello" == "Hello", "\n")

print("Hell < Hello")
print("Hell" < "Hello", "\n")

print("Hallo < Hello")
print("Hallo" < "Hello", "\n")

print("ello < Hello")
print("ello" < "Hello")
```

Petit exemple de calcul (Attention aux réaffectations des variables)

```
rayon = 2
pi = 3.14

aire = pi*(rayon**2)

rayon += 1

print("rayon:", rayon)
print("aire mise à jour:", pi*(rayon**2))
print("aire ancienne:", aire)
```

2 Structures de contrôle

2.1 Clauses conditionnelles

Une clause conditionnelle est une partie du code qui n'est exécutée que si une condition est vérifiée. Voici un exemple:

```
import random

x = random.random()
a = 1

if x < 0.3:
    a += 1

elif x < 0.6:
    a += 2

else:
    a += 3

print(a)
```

Important: En python, l'intérieur d'une clause est marqué par un alinéa. EN général, cela replace les accolades `{ }` de C++. Normalement l'éditeur va mettre automatiquement les bons espaces au début. Le code ne fonctionne pas si les alinéas ne sont pas bons.

Dans des clauses imbriquées, on empile les alinéas.

```
x = random.random()
```

```
a = 0

if x < 0.5:
    x += 0.2
    a += 1
    if x < 0.5:
        a += 1
    else:
        a += 2
else:
    a += 4

print(a)
```

2.2 Boucles

Pour exécuter un même morceau de code plusieurs fois, on a deux méthodes: `for` et `while`.

Une boucle `for` exécute un code un nombre fixe de fois.

```
somme = 0

for i in range(1,8):
    somme += i

print(somme)
```

Une boucle `while` exécute un code tant qu'une condition est vraie

```
somme = 0
i = 1

while i <= 7:
    somme += i
    i += 1

print(somme)
```

Les boucles `while` sont utiles quand on ne connaît pas à l'avance le nombre d'itérations à faire. Dans ce cas, il peut ne pas être possible de

les remplacer par des boucles `for`. Comme on vient de le voir, on peut toujours remplacer une boucle `for` par une boucle `while`.

Il est aussi possible de sortir d'une boucle avant qu'elle se termine. Pour cela, on peut utiliser `break`. Cela nous sort immédiatement de la boucle, en sautant tout ce qui reste.

```
somme = 0

for i in range(1,8):
    somme += i
    if somme == 10:
        break

    print(somme)

print("\n",somme)
```

Si on met `break` dans une boucle imbriquée, on ne sort que de la boucle intérieure.

On peut aussi sauter des étapes à l'intérieur de la boucle sans en sortir, avec `continue`

```
somme = 0

for i in range(1,8):
    print(somme)

    if i == 5:
        continue

    somme += i

print("\n",somme)
```

3 Fonctions

Une fonction est qui dépend d'un nombre de paramètres d'entrée, et qui retourne un résultat.

Exemple: Une fonction qui prend en entrée x et n, et retourne $\sum_{0 \leq k \leq n} x^k$

```
def f(x,n):
    # toutes les variables utilisées ici sont locales. Leurs noms peuvent être réutilis

    somme = 0
    for k in range(n+1):
        somme += x**k
    return somme

n = 3
x = 2.0

print(f(x,n))
```

Ici encore, pas besoin de donner les types d'entrée, ni de sortie quand on déclare une fonction. Les types sont déterminés par le contenu de la fonction. `f` contient une itération de `0` à `n`, ce qui nécessite que `n` soit de type `int`. Mettre autre chose (par exemple un `float`) à la place de `n` donne une erreur de type.

Les fonctions permettent d'organiser le code en sous parties simples et réutilisables. Si on utilise plusieurs fois une même séquence d'instructions, on peut la mettre dans une fonction et appeler celle-ci quand on en a besoin. Cela a plusieurs avantages:

- **Efficacité**
  - Moins de lignes de code à écrire
- **Lisibilité**
  - Un code structuré avec des fonctions (avec des noms indicatifs) est bien plus facile à comprendre par un lecteur, y compris vous-même, qu'une longue liste d'instructions, clauses, boucles imbriquées...
- **Flexibilité**
  - Changer une méthode plusieurs fois est beaucoup plus rapide si elle est définie en un seul endroit!

Le code à l'intérieur d'une fonction, en plus de donner une sortie, peut aussi avoir des effets secondaires. Cela peut avoir des conséquences inattendues si on ne fait pas attention.

```
a = 0

def f_bis(x,n):
    somme = 0
    for k in range(n+1):
        somme += x**k

    global a
    a += 1

    return somme


print(a)

n = 0
x = 1.8

while f_bis(x,n) < 20:
    n += 1

print(a)
```

Une fonction termine immédiatement dès qu'il y a un `return`.

```
def somme(n,stop):
    somme = 0

    for k in range(1,n+1):
        print(somme)

        if k == stop:
            return somme

        somme += k

    return somme

somme(10,4)
```

Une fonction peut très bien ne pas avoir d'entrée ou de sortie. Il n'est alors pas nécessaire de mettre un `return`

```
def fun():
    for i in range(10):
        print("Hello")

fun()
```

Une fonction sans valeur de sortie peut contenir un `return`, c'est alors juste pour indiquer qu'elle doit terminer sans rien faire de plus.

```
def fun(n):
    for k in range(n):
        if k == 4:
            return
        else:
            print(k)

fun(8)
```

En python, contrairement à C++, les fonctions sont bien des objets ayant un type `function`. Cela veut dire, en particulier, qu'une fonction peut très bien prendre en entrée, ou donner en sortie, une fonction.

```
def decal(n):
    def f(m):
        return n+m
    return f

print(decal(2)(4))

def comp(f,g):
    def compose(x):
        return g(f(x))
    return compose

def f(x):
    return x+2

def g(x):
```

```
    return x**2

print(comp(f,g)(2))
```