Langages informatiques - Python

Université Panthéon-Sorbonne – EMS

Enseignant : Matthieu Jacq - matthieu.jacq@univ-paris1.fr

🎯 Objectif de la séance

Maîtriser les types de données standards en Python avec les méthodes les plus courantes associées :

- 🔢 Nombres (int, float)
- 🔤 Chaînes de caractères (str)
- ✅ Booléens (bool)
- 📋 Listes (list)
- 📦 Tuples (tuple)
- 📚 Dictionnaires (dict)
- 🎨 Ensembles (set)

🔢 Nombres (int, float)

Les deux types de données numériques principaux en Python sont `int` (entiers) et `float` (nombres à virgule flottante).

Le séparateur décimal est le point (`.`).

```
# Entiers
a = 10
b = -5

# Flottants
c = 3.14
d = -0.001
```

1 2 3 4 Nombres (int, float)

Les opérations arithmétiques les plus courantes sont :

- `+` : addition
- `-` : soustraction
- `*` : multiplication
- `/` : division
- `//` : division entière (arrondi à l'entier inférieur, ex : `7 // 2` donne `3`)
- `%` : modulo (reste de la division entière, ex : `7 % 2` donne `1`)
- `**` : puissance

🔢 Nombres (int, float)

Peut-on faire des opérations entre `int` et `float` ? Oui.

```
a = 5        # int
b = 2.5      # float

c = a + b    # ➡️ c est un float
print(c)     # Affiche 7.5
```

Règles de **conversion automatique** :

- opération entre `float` et `float` → `float`
- opération entre `float` et `int` → `float`
- ⚠️ division entre `int` et `int` → `float`
- toute autre opération entre `int` et `int` → `int`

🔢 Nombres (int, float)

Conversion automatique avec les booléens (`bool`)

Lors d'une opération avec un booléen (`bool`), Python le convertit automatiquement en `int` (`True` → `1`, `False` → `0`).

```
a = 10          # int
b = True        # bool

c = a + b       # ➡️ c est un int
print(c)        # Affiche 11
```

> ⚠️ En revanche, vous ne pouvez pas faire d'opérations entre `int` / `float` et les autres types (ex : `str`, `list`, etc.) sans conversion explicite.

```
a = 10          # int
b = "5"         # str

c = a + int(b) # ➡️ c est un int
print(c)        # Affiche 15
```

🔢 Nombres (int, float)

Exemples d'opérations :

```
print(2 + 2) # 4
print(2 - 1.0) # 1.0
print(2 * 3) # 6
print(2 / 3) # 0.6666666666666666
print(4 / 2) # 2.0 (⚠️ division renvoie toujours en float)
print(13 // 3) # 4 (division entière: 13 = 3 * 4 + 1)
print(13 % 3) # 1 (reste de la division entière)
print(2 ** 3) # 8 (puissance)
print(2 ** 0.5) # 1.4142135623730951 (racine carrée)
print(2 ** -1) # 0.5 (inverse)
```

🔢 Nombres (int, float)

Conversion explicite entre `int` et `float` : on utilise les fonctions `int()` et `float()`.

```
a = 5        # int
b = float(a)  # conversion en float
print(b)     # Affiche 5.0
```

```
c = 3.14     # float
d = int(c)     # conversion en int (troncature)
print(d)     # Affiche 3
```

🔢 Nombres (int, float) 🎁

Les cas particuliers :

- `float('inf')` : infini positif
- `float('-inf')` : infini négatif
- `float('nan')` : "not a number" (résultat d'opérations indéfinies)
- `int` peut représenter des entiers de taille arbitraire (limité par la mémoire disponible) contrairement à d'autres langages où la taille est fixe (ex : 32 ou 64 bits).
- Pour des calculs scientifiques, on peut utiliser la bibliothèque `numpy` qui offre des types de données numériques supplémentaires et des opérations optimisées.

🔢 Nombres (int, float) 🎁

⚠️ Les erreurs d'arrondi avec les `float` :

```
a = 0.1
b = 0.2
c = a + b
print(c)  # Affiche 0.30000000000000004
```

- `float` a une précision limitée (environ 15-17 chiffres décimaux significatifs).
- Pour des calculs nécessitant une grande précision sur les décimaux, on peut utiliser le module `decimal` de la bibliothèque standard.

```
from decimal import Decimal
a = Decimal('0.1')
b = Decimal('0.2')
c = a + b
print(c)  # Affiche 0.3
```

🔢 Nombres (int, float) 🎁

⚠️ Les erreurs d'arrondi avec les `float` :

```
a = 0.1
b = 0.2
c = a + b
print(c == 0.3)  # Affiche False
```

Pour tester des égalités entre flottants, vous pouvez utiliser une tolérance :

```
import math
a = 0.1
b = 0.2
c = a + b
print(math.isclose(c, 0.3, rel_tol=1e-9))  # Affiche True
```

🔢 Nombres (int, float)

Quand une opération est invalide, Python **lève une exception**. **Le programme s'arrête** et affiche un message d'erreur.

Les erreurs les plus fréquenteslevées par Python avec les nombres :

- `ZeroDivisionError` : division par zéro.
- `TypeError` : opération entre types incompatibles (ex : addition d'un `int` et d'une `str`). Ex : `1 + '1'`

> 🎁 D'autres exceptions existent (ne pas apprendre):
>
> - `ValueError` : conversion invalide (ex : convertir une chaîne non numérique en nombre). Ex : `int('bonjour')`
> - `OverflowError` : résultat trop grand pour être représenté (rare avec les `int` en Python).
> - etc.

🔢 Nombres (int, float) : les fonctions mathématiques

Python propose de nombreuses fonctions mathématiques via le module `math` de la bibliothèque standard. Pour les utiliser, il faut d'abord importer le module :

```
import math
```

🔢 Nombres (int, float) : les fonctions mathématiques

Quelques fonctions courantes :

- `math.sqrt(x)` : racine carrée de `x`.
- `math.pow(x, y)` : `x` élevé à la puissance `y`.
- `math.sin(x)`, `math.cos(x)`, `math.tan(x)` : fonctions trigonométriques (x en radians).
- `math.log(x)` : logarithme naturel (népérien) de `x`.
  - 🎁 pour utiliser d'autres bases `math.log(x, 10)` : logarithme en base 10 de `x`.
- `math.exp(x)` : exponentielle de `x` (e^x).

Chaînes de caractères (str)

Les chaînes de caractères (`str`) sont des séquences de caractères utilisées pour représenter du texte.

```
# Exemple de chaîne de caractères
message = "Hello, world!"
print(message)  # Affiche: Hello, world!
```

Elles sont définies entre guillemets simples (`'...'`) ou doubles (`"..."`).

```
str1 = 'Bonjour'
print(str1)  # Affiche: Bonjour
```

Chaînes de caractères (str)

Nous avons déjà vu les f-strings pour formater des chaînes de caractères :

```
name = "Alice"
age = 30
greeting = f"Hello, my name is {name} and I am {age} years old."
print(greeting)  # Affiche: Hello, my name is Alice and I am 30 years old.
```

Chaînes de caractères (str)

Les chaînes sur plusieurs lignes

Pour créer une chaîne de caractères sur plusieurs lignes, on utilise des triples guillemets (`'''...'''` ou `"""..."""`).

```
multi_line_str = """Ceci est une chaîne
sur plusieurs lignes.
Elle conserve les sauts de ligne."""
print(multi_line_str)
```

Chaînes de caractères (str)

Les caractères spéciaux

Certains caractères spéciaux peuvent être insérés dans une chaîne à l'aide de séquences d'échappement :

- `\n` : saut de ligne
- `\t` : tabulation
- `\\` : barre oblique inversée
- `\'` : guillemet simple
- `\"` : guillemet double

Chaînes de caractères (str)

Les raw strings

Pour ignorer les séquences d'échappement, on peut utiliser des raw strings en préfixant la chaîne avec un `r` :

```
raw_str = r"C:\Users\Name\Documents"
print(raw_str)  # Affiche: C:\Users\Name\Documents
```

Chaînes de caractères (str)

Les opérateurs `+` et `*`

- Concaténation : `+`
- Répétition : `*`

```
greeting = "Hello, " + "world!"  # Concaténation
print(greeting)  # Affiche: Hello, world!

laugh = "Ha" * 3  # Répétition
print(laugh)  # Affiche: HaHaHa
```

Chaînes de caractères (str)

Accès aux caractères

- Indexation : `str[index]`
- Slicing (extraire une sous-chaîne) : `str[start:end]` (⚠️ end est exclus)

> - Les indices peuvent être négatifs (compte à partir de la fin).
> - Si `start` est omis, il est considéré comme `0`.
> - Si `end` est omis, il est considéré comme la longueur de la chaîne.

```
my_string = "Python"
print(my_string[0])    # 'P'
print(my_string[0:1])  # 'P'
print(my_string[0:2])  # 'Py'
print(my_string[:2])  # 'Py'
print(my_string[-1])   # 'n'
print(my_string[0:-1])  # 'Pytho'
print(my_string[-2:])  # 'on'
print(my_string[1:])  # 'ython'
```

Chaînes de caractères (str)

⚠️ Immutabilité

Les chaînes de caractères en Python sont immuables, ce qui signifie qu'une fois créées, elles **ne peuvent pas être modifiées**. Toute opération qui semble modifier une chaîne de caractères renvoie en réalité une **nouvelle chaîne**.

```
my_string = "Hello"
my_string[0] = "h"  # Erreur : TypeError
```

```
my_string = "Hello"
new_string = "h" + my_string[1:]  # Création d'une nouvelle chaîne
print(new_string)  # Affiche: hello
```

Chaînes de caractères (str)

Itération et longueur

- Itération : boucle `for char in my_string:`
- Longueur : `len(str)`

```
my_string = "Hello"
for char in my_string:
    print(char)
```

```
print(len(my_string))  # Affiche: 5
```

```
i = 0
while i < len(my_string):
    print(my_string[i])
    i += 1
```

Chaînes de caractères (str)

Appartenance

- Opérateur `in` pour vérifier si une sous-chaîne est présente dans une chaîne.

```
my_string = "Hello, world!"
print("world" in my_string)  # Affiche: True
print("Python" in my_string)  # Affiche: False
```

```
my_string = "Hello, world!"
if "world" in my_string:
    print("Found 'world'!")
```

Chaînes de caractères (str)

Comparaison

Les chaînes de caractères peuvent être comparées à l'aide des opérateurs de comparaison (`==` , `!=` , `<` , `>` , `<=` , `>=`).

```
str1 = "Hello"
str2 = "World"
print(str1 == str2)  # Affiche: False
print(str1 != str2)  # Affiche: True
print(str1 < str2)   # Affiche: True (comparaison lexicographique)
```

```
str1 = "abc"
str2 = "ABC"
print(str1 == str2)  # Affiche: False
print(str1.lower() == str2.lower())  # Affiche: True
```

Chaînes de caractères (str)

Conversion explicite entre `str`

```
num = 42
str_num = str(num)  # Conversion en chaîne de caractères
print(str_num)      # Affiche: '42'
print(str_num + " is the answer.")  # Affiche: '42 is the answer.'
```

Chaînes de caractères (str)

Méthodes courantes des chaînes de caractères

Les méthodes sont des fonctions un peu spéciales associées aux objets (ici, aux chaînes de caractères). Quelques-unes des méthodes les plus courantes pour les chaînes de caractères :

- `str.lower()` : convertit en minuscules
- `str.upper()` : convertit en majuscules
- `str.title()` : met en majuscules la première lettre de chaque mot
- `str.capitalize()` : met en majuscules la première lettre de la chaîne
- `str.strip()` : supprime les espaces en début et fin
- `str.replace(old, new)` : remplace les occurrences de `old` par `new`
- `str.split(sep)` : divise la chaîne en une liste selon le séparateur `sep`
- `str.join(iterable)` : joint les éléments d'un itérable avec la chaîne comme séparateur

✅ Booléens (bool)

Les booléens (`bool`) représentent deux valeurs : `True` (vrai) et `False` (faux).

```
is_raining = True
is_sunny = False
print(is_raining)  # Affiche: True
print(is_sunny)    # Affiche: False
```

Le résultat des comparaisons et des opérations logiques est toujours un booléen.

```
a = 5
b = 10
print(a < b)       # Affiche: True
print(a == b)      # Affiche: False
```

✅ Booléens (bool)

Opérateurs logiques

Les opérateurs logiques permettent de combiner des expressions booléennes :

- `and` : et logique
- `or` : ou logique
- `not` : négation logique
- 🎁 `^` : ou exclusif (xor)

```
print(True and True)  # Affiche: True
print(True and False)  # Affiche: False
print(True or True)   # Affiche: True
print(True or False)   # Affiche: True
print(not True)        # Affiche: False
print(True ^ False)    # Affiche: True
print(True ^ True)     # Affiche: False
```

✅ Booléens (bool)

Conversion explicite entre `bool` et autres types

- `bool(value)` : convertit `value` en booléen.
  - Les valeurs **nulles ou vides** sont considérées comme `False` : `0`, `0.0`, `''` (chaîne vide), `[]` (liste vide), `{}` (dictionnaire vide), `range(0)` (intervalle vide), `None`.
  - Toutes les autres valeurs sont considérées comme `True`.

```
print(bool(0))         # Affiche: False
print(bool(42))        # Affiche: True
print(bool(''))        # Affiche: False
print(bool('Hello'))   # Affiche: True
```

✅ Booléens (bool)

Conversion automatique dans les conditions

Dans les structures conditionnelles (`if` , `while`), Python convertit automatiquement les expressions en booléens.

```
value = 10
if value:
    print("Value is True")  # ✅ S'exécute car 10 est considéré comme True
```

```
value = ''
if value:
    print("Value is True")  # ❎ Ne s'exécute pas car '' est considéré comme False
```

📋 Listes (list)

Les listes (`list`) sont des collections ordonnées et modifiables d'éléments. Elles peuvent contenir des éléments de types différents.

```
# Exemple de liste
fruits = ["apple", "banana", "cherry"]
print(fruits)  # Affiche: ['apple', 'banana', 'cherry']
```

```
mixed_list = [1, "two", 3.0, True]
print(mixed_list)  # Affiche: [1, 'two', 3.0, True]
```

📋 Listes (list)

Création de listes

```
# Liste vide
empty_list = []
print(empty_list)  # Affiche: []

# Liste avec des éléments
numbers = [1, 2, 3, 4, 5]
print(numbers)  # Affiche: [1, 2, 3, 4, 5]
```

Listes (list)

Accès aux éléments

- Indexation : `list[index]`
- Slicing (extraire une sous-liste) : `list[start:end]` ou `list[start:end:step]`
- Itération : boucle `for item in my_list:`
- Longueur : `len(list)`

```
my_list = [10, 20, 30, 40, 50]
print(my_list[0])    # 10
print(my_list[1:4])  # [20, 30, 40]
print(my_list[::2])  # [10, 30, 50]

for item in my_list:
    print(item)

print(len(my_list))  # 5
```

Listes (list)

Itération simple (valeur seule)

```
my_list = ['a', 'b', 'c']
for value in my_list:
    print(value)
```

Itération avec index

```
my_list = ['a', 'b', 'c']
for i in range(len(my_list)):
    print(f"Index {i}: {my_list[i]}")
```

Itération avec valeur et index (énumération)

```
my_list = ['a', 'b', 'c']
for index, value in enumerate(my_list):
    print(f"Index {index}: {value}")
```

📋 Listes (list)

Mutabilité

Les listes en Python sont **mutables**, ce qui signifie que vous pouvez modifier leur contenu après leur création.

```
my_list = [1, 2, 3]
my_list[0] = 10  # Modification du premier élément
print(my_list)   # Affiche: [10, 2, 3]
```

Listes (list)

Modification des listes

- Ajout d'éléments : `list.append(item)` (ajoute à la fin), `list.insert(index, item)` (ajoute à l'index spécifié)
- Suppression d'éléments :
  - à l'aide de l'index : `del list[index]`
  - à l'aide de la valeur : `list.remove(item)` (retire la première occurrence de `item`)
- Modification d'éléments : `list[index] = new_value`

```
my_list = [1, 2, 3]
my_list.append(4)          # [1, 2, 3, 4]
my_list.insert(1, 1.5)     # [1, 1.5, 2, 3, 4]
my_list.remove(2)          # [1, 1.5, 3, 4]
del my_list[2]            # [1, 1.5, 4]
my_list[0] = 42            # [42, 1.5, 4]
```

Listes (list)

Concaténation et répétition

- Concaténation : `+`
- Répétition : `*`

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined = list1 + list2  # [1, 2, 3, 4, 5, 6]
print(combined)
```

```
my_list = [0] * 5  # [0, 0, 0, 0, 0]
print(my_list)
```

Listes (list)

Tri et recherche

- Tri : `list.sort()` (tri en place), `sorted(list)` (renvoie une nouvelle liste triée)
- Recherche : `item in list` (vérifie la présence de `item` dans la liste), `list.index(item)` (renvoie l'index de la première occurrence de `item`)

```
my_list = [3, 1, 4, 2]
my_list.sort()              # Tri en place
print(my_list)              # Affiche: [1, 2, 3, 4]
new_list = sorted([3, 1, 4, 2])  # Nouvelle liste triée
print(new_list)             # Affiche: [1, 2, 3, 4]
```

```
my_list = [10, 20, 30]
print(20 in my_list)        # Affiche: True
print(my_list.index(30))    # Affiche: 2
```

Listes (list)

Fonctions utiles

- `len(list)` : renvoie la longueur de la liste
- `min(list)` : renvoie le plus petit élément
- `max(list)` : renvoie le plus grand élément
- `sum(list)` : renvoie la somme des éléments (si la liste contient des nombres

Listes (list)

Conversion d'autres types en liste

- `list(iterable)` : convertit un itérable (ex : chaîne de caractères, tuple) en liste

```
str_value = "hello"
char_list = list(str_value)  # ['h', 'e', 'l', 'l', 'o']
print(char_list)

range_list = list(range(5))  # [0, 1, 2, 3, 4]
print(range_list)
```

📋 Listes (list)

Copie de listes : erreur courante

```
original_list = [1, 2, 3]

new_list = original_list  # ⚠️ Mauvaise façon de copier une liste
new_list[0] = 10

print(original_list)  # Affiche: [10, 2, 3] (original_list est modifiée !)
```

L'opération `new_list = original_list` ne crée pas une nouvelle liste, mais fait que `new_list` référence la même liste que `original_list`. Toute modification de `new_list` affecte également `original_list`.

Copie de listes (shallow copy)

Deux façons courantes de créer une copie **indépendante** (shallow copy) d'une liste :

- `my_list.copy()` : crée une nouvelle liste contenant les mêmes éléments que l'original
- `list(original)` : utilise la fonction `list()` pour créer une nouvelle liste

```
original_list = [1, 2, 3]
new_list = original_list.copy()  # Méthode copy()
# ou
new_list = list(original_list)    # Fonction list()
new_list[0] = 10
print(new_list)  # Affiche: [10, 2, 3]
print(original_list)  # Affiche: [1, 2, 3] (original_list n'est pas modifiée)
```

📋 Listes (list)

🎁 Copie de listes (deep copy)

Pour copier des listes imbriquées (listes contenant d'autres listes), une **deep copy** est nécessaire pour éviter que les sous-listes ne soient partagées entre l'original et la copie.

```
import copy

original_list = [1, 2, [3, 4]]
deep_copy = copy.deepcopy(original_list)
deep_copy[2][0] = 10

print(original_list)  # Affiche: [1, 2, [3, 4]]
print(deep_copy)      # Affiche: [1, 2, [10, 4]]
```