

# Introduction au langage de programmation Python

Rédigé par : Fabien Poirier  
Supervisé par : Geoffrey Groff

February 10, 2024

- 1 Introduction
- 2 Propriétés du langage
- 3 Initiation à la programmation Python
- 4 Mécanisme d'exécution
- 5 Sources

# Plan du Chapitre

- 1 Introduction
  - Histoire
  - Domaines d'application
  - Les versions
- 2 Propriétés du langage
- 3 Initiation à la programmation Python
- 4 Mécanisme d'exécution
- 5 Sources

# La naissance de Python



Figure: Guido van Rossum

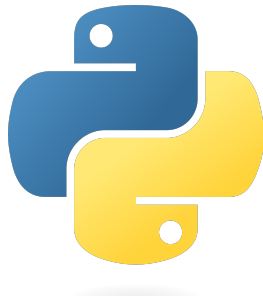


Figure: Python (1991)

# Source d'inspiration

## Influencé par :

- C (1972)
- ABC (1975)
- Modula-3 (1978)
- Eiffel (1986)
- Perl (1987)
- etc...

## A influencé :

- OCaml (1996)
- Ruby (1995)
- Go (2009)
- Julia (2012)
- Swift (2014)
- etc...

## Outils utilisant Python ?

Connaissez-vous des logiciels ou des outils utilisant Python ?

# Exemple d'outils utilisant Python



Instagram



Dropbox



Reddit



ChatGPT



Linux



Netflix



Youtube



Jupyter



Pycharm

**django**



Flask  
web development,  
one drop at a time

# Domaines d'application

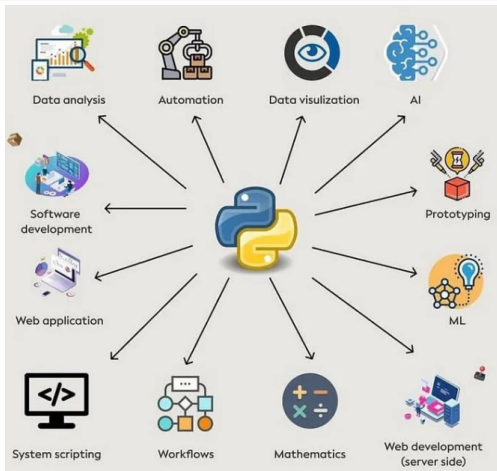


Figure: Domaines d'application de Python (Ashan Madusanka)



## Popularité du langage : top 3 !

| Année | Langage le plus populaire | Année | Langage le plus populaire |
|-------|---------------------------|-------|---------------------------|
| 2003  | C++                       | 2014  | Javascript                |
| 2004  | PHP                       | 2015  | Java                      |
| 2005  | Java                      | 2016  | Go                        |
| 2006  | Ruby                      | 2017  | C                         |
| 2007  | <b>Python</b>             | 2018  | <b>Python</b>             |
| 2008  | C                         | 2019  | C                         |
| 2009  | Go                        | 2020  | <b>Python</b>             |
| 2010  | <b>Python</b>             | 2021  | <b>Python</b>             |
| 2011  | Objective-C               | 2022  | C++                       |
| 2012  | Objective-C               | 2023  | C#                        |
| 2013  | transact-SQL              | 2024  |                           |

Table: Tiobe index (Source)

# Avantages / Inconvénients

## Avantages ✓

Syntaxe lisible et expressive.

Facilité d'apprentissage.

Grande communauté  
de développeurs.

Open Source

Large écosystème de  
bibliothèques et de  
frameworks.

## Inconvénients X

Plus lent que certain langage.

Tout les codes ne se valent pas.

Gestion de la concurrence  
peut être complexe.

Les erreurs

Dépendance entre les librairies

Les changements dû aux mises à jour

# Les diverses versions

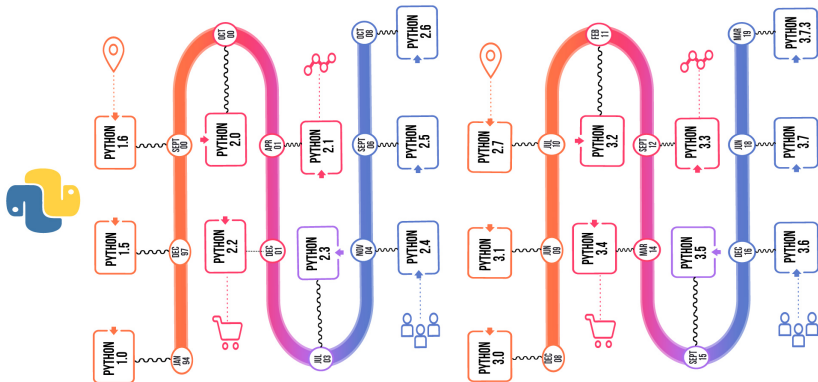


Figure: Les versions de Python [1994 - 2019] (Source)

# Python 2 vs Python 3

|                           | Python 2                     | Python 3                     |
|---------------------------|------------------------------|------------------------------|
| <b>Date de sortie</b>     | 2000                         | 2008                         |
| <b>Syntaxe</b>            | Complexe et difficile        | Lisible et facile            |
| <b>Performance</b>        | Lente                        | Rapide                       |
| <b>Impression</b>         | Instruction                  | Fonction                     |
| <b>Division</b>           | Résultat entier              | Résultat flottant            |
| <b>Unicode</b>            | ASCII par défaut             | Unicode par défaut           |
| <b>Intervalle</b>         | Plusieurs fonctions          | Une seule fonction           |
| <b>Rétrocompatibilité</b> | Python2 → Python3            | <del>Python2 ← Python3</del> |
| <b>Bibliothèques</b>      | <del>Compatible à 100%</del> | <del>Compatible à 100%</del> |

# Les saveurs de Python :9

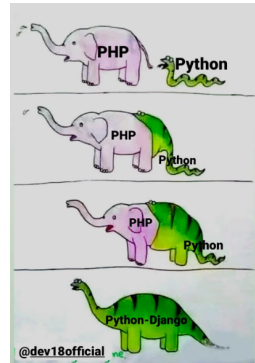


Figure: Les déclinaisons de Python (Source)

Figure: Python est partout !

# Plan du Chapitre

- 1 Introduction
- 2 Propriétés du langage
  - Les fondements de Python
  - Les concepts clés
  - La documentation et les PEPs
- 3 Initiation à la programmation Python
- 4 Mécanisme d'exécution
- 5 Sources

# Caractéristiques



- Langage de haut niveau
- Interprété
- Dynamiquement typé
- Typage fort
- Gestion automatique de la mémoire
- Orienté objet

# Caractéristiques ?

Mais que représentent chacun de ces concepts ?

Seriez-vous en mesure de les définir ?



# Langage bas niveau / langage haut niveau

## Bas Niveau

- Proche de la machine
- Contrôle direct sur le matériel
- Gestion de la mémoire manuelle
- Plus rapide
- Code Binaire, Assembleur, C...

## Haut niveau

- Proche du langage naturel
- Abstraction matériel (Focus sur le code)
- Gestion automatique de la mémoire
- Plus lent
- Python, Java, PHP, Javascript....

# Compilé / Interprété

## Compilé

- Traduit le code source dans son intégralité
- Le code compilé peut être exécuté (1 à  $\infty$  fois)
- Exécution plus rapide
- Détection des erreurs avant l'exécution
- Langages : C, Java, C++...

## Interprété

- Traduit et exécute le code ligne par ligne
- Exécution plus lente
- Détection des erreurs pendant l'exécution
- Langages : Python, PHP, Javascript, shell (Bash)...

# Typage Static / Typage Dynamique

## Typage Static

- Les types sont définis explicitement dans le code
- Les types sont vérifiés à la compilation
- Une fois le type attribué il ne peut pas changer
- Les erreurs de type sont détectés à la compilation
- Java, C++, C#...

## Typage Dynamique

- Le type d'une variable est déterminé à l'exécution
- Les variables peuvent changer de type
- Les erreurs de type sont détectés pendant l'exécution
- Langages : Python, PHP, Javascript...

# Typage fort / typage faible

## Typage Faible

javascript

 Copy code

```
let prix = 10;  
let monnaie = "€";  
let result = prix + monnaie;  
  
// Affichera "10€"
```

Plus permissif en ce qui concerne les conversions entre types différents. Les opérations entre types différents peuvent être effectuées (conversion implicite).

Langages : Javascript, PHP,  
Shell (Bash)

## Typage Fort

python

 Copy code

```
prix = 10  
monnaie = "€"  
result = prix + monnaie  
  
# Cela générera une erreur
```

Règles strictes concernant la manipulation des types de données. Les conversions automatiques entre types différents sont limitées.

Langages : Java, Python, C#,  
C++

# Java vs Python



- Langage de haut niveau
- Compilé
- Typage Statique
- Typage Fort
- Garbage collector
- Orienté Objet
- Syntaxe rigide et structuré



- Langage de haut niveau
- Interprété
- Typage Dynamique
- Typage Fort
- Garbage collector
- Orienté Objet
- Syntaxe simple et lisible

# Les mots clés propres au langage

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

# Les types associés au langage

| <u>Types numériques</u>   | <u>Types itérables</u>   | <u>Autre types</u>   |
|---|--|--|
| <ul style="list-style-type: none"><li>• int</li><li>• float</li><li>• complex</li></ul> | <ul style="list-style-type: none"><li>• tuple</li><li>• list</li><li>• set</li><li>• frozenset</li><li>• dict</li><li>• str</li><li>• bytearray</li><li>• file</li><li>• range</li></ul> | <ul style="list-style-type: none"><li>• None</li><li>• type</li><li>• object</li><li>• slice</li><li>• NotImplementedType</li><li>• bool</li><li>• exception</li><li>• function</li><li>• module</li></ul> |

# Un langage défini par les indentations

## L'indentation le ";" de Python !

"PYTHON INDENTATION"

### (CODE THAT WORKS)

```
n = [3, 5, 7]

def double_list(x):

    for i in range(0, len(x))
        x[i] = x[i] * 2
    return x

print double_list(n)
```

### (CODE THAT FAILS)


```
n = [3, 5, 7]

def double_list(x):


    for i in range(0, len(x))
        x[i] = x[i] * 2
    return x

print double_list(n)
```



 [HTTPS://TAPAS10/SERIES/GRUMPY-CODES](https://twitter.com/TAPAS10/SERIES/GRUMPY-CODES)



 [CARDBOARDVOICE](https://www.instagram.com/CARDBOARDVOICE)



# La documentation et les PEPs

Mais où sont définies les règles d'écriture du langage ?

- 1 Site Officiel : <https://www.python.org>
- 2 Les PEPs

Qu'est ce qu'une PEP ?

# Une PEP ?

## PEP : Python Enhancement Proposal

- La PEP, ou Python Enhancement Proposal (Proposition d'amélioration pour Python en français), est un mécanisme utilisé dans la communauté Python pour proposer et discuter des améliorations potentielles du langage, des bibliothèques ou des processus liés à Python.
- Les PEPs (Python Enhancement Proposal) sont des documents formels qui décrivent une nouvelle fonctionnalité, une modification de comportement, une norme ou un processus pour la communauté Python.

# Pourquoi avoir des PEP ?



- 1 Communication ouverte
- 2 Consensus
- 3 Archivage
- 4 Documentation

# Organisation des PEP

- Draft (Brouillon)
- Accepted (Acceptée)
- Rejected (Rejetée)
- Final (Finale)
- Withdrawn (Retirée)

# Où trouver les PEP ?

## PEP Abordées à travers ce cours :

- PEP 8 : Style Guide for Python Code
- PEP 257 : Docstring Conventions
- PEP 484 : Type Hints

Toutes les PEPs sont disponibles sur le site officiel de Python :  
<https://peps.python.org/pep-0000/>

# Plan du Chapitre

- 1 Introduction
- 2 Propriétés du langage
- 3 Initiation à la programmation Python
  - Convention et Opérations Fondamentales
  - Les structures conditionnelles
  - Les boucles
  - Les collections
- 4 Mécanisme d'exécution
- 5 Sources

## Les normes d'écriture

Pouvez-vous me citer les trois normes d'écriture couramment utilisés en informatique ?

Et pouvez-vous me dire laquelle est utilisé en Python ?

# Les conventions de nommage



## snake\_case

Pros: Concise when it consists of a few words.

Cons: Redundant as hell when it gets longer.

`push_something_to_first_queue, pop_what, get_whatever...`



## PascalCase

Pros: Seems neat.

`GetItem, SetItem, Convert, ...`

Cons: Barely used. (why?)



## camelCase

Pros: Widely used in the programmer community.

Cons: Looks ugly when a few methods are n-worded.

`push, reserve, beginBuilding, ...`

# PEP (8): Nom en Python = Snake\_case



# Les opérations arithmétiques et binaires

|              | Opération            | Symbole | Exemple          |
|--------------|----------------------|---------|------------------|
| Arithmétique | Addition             | +       | $1 + 1 = 2$      |
|              | Soustraction         | -       | $1 - 1 = 0$      |
|              | Multiplication       | *       | $1 * 2 = 2$      |
|              | Division             | /       | $1 / 2 = 0.5$    |
|              | Division entière     | //      | $1 // 2 = 0$     |
|              | Modulo               | %       | $1 \% 2 = 1$     |
|              | Exponentiation       | **      | $2 ** 2 = 4$     |
| Binaire      | Et Logique (AND)     | &       | $2 \& 4 = 0$     |
|              | Ou Logique (OR)      |         | $2   4 = 6$      |
|              | Ou exclusif (XOR)    | ^       | $2 \wedge 4 = 6$ |
|              | Complement à 1 (NOT) | ~       | $\sim 0 = -1$    |
|              | Décalage à gauche    | <<      | $4 << 2 = 16$    |
|              | Décalage à droite    | >>      | $4 >> 2 = 1$     |

# Incrémentation / Décrémentation

**La notation `variable++` n'existe pas en python !**

## Les opérateurs d'assignation

- incrémentation  $\rightarrow$  `variable += valeur`
- décrémentation  $\rightarrow$  `variable -= valeur`
- Ces opérateurs d'assignation combinée sont compatibles avec toutes les opérations arithmétiques et binaires disponibles.

La syntaxe reste la même : la variable, suivi du symbole de l'opération, puis le signe égal, et enfin la valeur souhaitée.

## Les opérations : syntaxe

PEP (8) : Entourez les opérateurs avec un espace  
de chaque côté

### Syntaxe valide (✓)

→ `a + b`

→ `a // b`

→ `a >> b`

### Syntaxe invalide (X)

→ `a+b`

→ `a// b`

→ `a >>b`

# Typage dynamique != Cast ?

## Python :

- ✓ Typage dynamique
- ✓ Typage fort

## Le Cast

- Le "cast" (ou conversion de type) est un processus permettant de changer le type d'une variable d'un type à un autre. En Python, le cast est souvent réalisé à l'aide de fonctions intégrées telles que **int()**, **float()**, **str()**, etc. Ces fonctions permettent de convertir une variable d'un type à un autre, lorsque cela est possible.

Exemple : **int(variable)** / **float(variable)** / **str(variable)**

→ Obtenir le type d'une variable : **type(variable)**

# Les commentaires

**PEP (8) : Toujours placer un espace entre le # et le commentaire**

## Commentaire sur une ligne

```
# Ceci est un commentaire
```

## Commentaire multi-lignes

```
'''  
Ceci est un commentaire  
sur plusieurs lignes.  
'''
```

```
"""  
Ceci est un autre  
commentaire sur plusieurs  
lignes.  
"""
```

## Saisie & Affichage (Input \ Output)

| Descriptif                               | Instruction                                      |
|--|--|
| Saisie utilisateur                       | <code>name = input("Entrez votre nom : ")</code> |
| Affichage : texte                        | <code>print("Hello !")</code>                    |
| Affichage : texte + variable             | <code>print("Enchanté:", name)</code>            |
| Affichage : concaténation                | <code>print("Enchanté: " + name)</code>          |
| Affichage avec f-string                  | <code>print(f"Enchanté: {name}")</code>          |
| Affichage sans saut<br>de ligne à la fin | <code>print("Aurevoir !", end="")</code>         |

**En Python : "MESSAGE" == 'MESSAGE'**

# Les caractères spéciaux

Liste des caractères spéciaux utilisables dans vos affichages :

- Nouvelle ligne (`\n`)
- Tabulation (`\t`)
- Retour chariot (`\r`)
- Caractère d'échappement (`\`)

# Python le HTML de la programmation ?

```
# Demander à l'utilisateur d'entrer un nombre  
nombre = int(input("Entrez un nombre : "))
```

— instruction

```
# Vérifiez si le nombre est pair ou impair
```

```
if nombre % 2 == 0:  
    print(f"{nombre} est un nombre pair.")
```

```
else:  
    print(f"{nombre} est un nombre impair.")
```

} bloc d'instructions

## On peut observer !

- Une instruction ne termine pas par ;
- Un bloc d'instruction commence toujours par : suivi d'un retour à la ligne et d'une indentation
- Code sous forme de script : pas de main



## Instruction / bloc d'instruction : syntaxe

- PEP (8) Une instruction  $< 80$  caractères
- PEP (8) Une indentation = 4 espaces !
- En python : Espace  $\neq$  Tabulation

# Les structures de contrôle conditionnelles : le if

Mot clé      Condition      2 points

```
if nombre % 2 == 0 :
```

```
    print(f"{nombre} est un nombre pair.")
```

Instruction à exécuter si la condition est vraie

Figure: Structure du if

- Il est possible d'avoir autant de **if** successif que voulu

## Les structures de contrôle conditionnelles : le else

Mot clé 2 points

else :

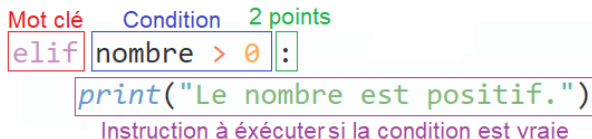
```
print(f"{nombre} est un nombre impair.")
```

Instruction à exécuter si la condition du if précédent est fausse

Figure: Structure du else

- Il ne peut y avoir qu'un **else** par bloc **if**, et celui-ci doit lui succéder.

## La combinaison du if et du else : le elif



```
elif nombre > 0 :  
    print("Le nombre est positif.")
```

Instruction à exécuter si la condition est vraie

Figure: Structure du elif

- Il peut y avoir autant de **elif** que voulu tant qu'ils sont précédés par un **if**.

## Les opérateurs de condition

| Comparaison            | Syntaxe | Exemple                |
|------------------------|---------|------------------------|
| Strictement supérieur  | >       | <code>a &gt; b</code>  |
| Strictement inférieur  | <       | <code>a &lt; b</code>  |
| Supérieur ou égale     | >=      | <code>a &gt;= b</code> |
| Inférieur ou égale     | <=      | <code>a &lt;= b</code> |
| Égalité                | ==      | <code>a == b</code>    |
| Inégalité              | !=      | <code>a != b</code>    |
| L'opérateur d'identité | is      | <code>a is b</code>    |

- Chaque comparaison s'écrit en suivant son sens de lecture.
- L'égalité possède un double égale car une égalité fonctionne des 2 côtés.
- L'opérateur **is** vérifie si 2 objets se trouve au même emplacement mémoire.

## Les connecteurs logiques

| Nom      | Syntaxe | Exemple  |
|----------|---------|--|
| Et       | and     | <code>if chiffre % 2 == 0 and chiffre &gt; 0 :</code>  |
| Ou       | or      | <code>if chiffre % 2 == 0 or chiffre % 4 == 0 :</code> |
| Négation | not     | <code>if not chiffre % 2 == 0 :</code>                 |

- Vous êtes libre d'utiliser autant de connecteurs différents que nécessaire.
- Toutefois, veillez à maintenir la lisibilité du code ; dans le cas contraire, privilégiez l'imbrication de plusieurs conditions.

# La condition ternaire

résultat si True      condition      résultat si False

```
resultat = "Pair" if x % 2 == 0 else "Impair"
```

## La condition ternaire

Les ternaires sont utilisées pour évaluer une condition et retourner une valeur en fonction de cette condition, le tout sur une seule ligne de code.

# Match case le switch case de python !

Variable  
à tester

mot clé

```
match status:
```

Valeur à vérifier

```
    case 400:
        print("Bad request")
    case 404:
        print("Not found")
    case 418:
        print("I'm a teapot")
    case _:
        print("Something's wrong with the internet")
```

## Cas par défaut

- La valeur `_` du dernier case est un joker représentant toutes les valeurs non précisées c'est l'équivalent d'un default case en Java.



## Match multi case

```
match status :  
    case 0 | 2 | 4 | 6 | 8 :  
        print("Chiffre pair !")  
  
    case 1 | 3 | 5 | 7 | 9 :  
        print("Chiffre impair !")
```

### Le multi case

- Pour obtenir un multi case, séparez vos valeurs par des pipes (|).

### Attention !

- Le match case n'est disponible qu'à partir de la version 3.10 !

# Les assertions

```
# Si l'assertion échoue,  
# un AssertionError sera levé avec le message spécifié  
assert valeur > 0, "La valeur doit être supérieure à zéro"
```

mot clé      condition      virgule      message d'erreur

## Les assertions

Les assertions sont utilisées pour vérifier des conditions qui ne devraient pas se produire dans des situations normales. Contrairement à une instruction **if**, si une assertion échoue, elle mettra fin au programme en affichant un message d'erreur.

# Début du TP 1 !

## TP 1 : Structure conditionnelle

Vous développez un système d'affranchissement de courrier.

L'utilisateur devra renseigner le poids en grammes de sa lettre à défaut d'avoir une balance.

- Tout courrier pesant moins de 10g devra être affranchi à hauteur de 2€
- Tout courrier pesant plus de 100g devra être affranchi à hauteur de 6€
- Les courriers entre deux devront être affranchis de 3,50€

# La boucle for

mot clé    itérateur    fonction d'intervall    2 points

```
for i in range(0, 10, 1):
```

`print(i)` — instruction

mot clé    élément actuel    collection

```
for element in liste:
```

`print(element)` — instruction

-2 points

- Sémantiquement, une boucle **for** est à utiliser lorsque l'on connaît précisément le nombre de tours à effectuer.
- Dans une boucle le mot clef **in** permet de spécifier la séquence sur laquelle itérer.
- En dehors d'une boucle le mot clef **in** permet de vérifier qu'une valeur est présente dans une collection.

# La méthode range

## range()

- la méthode range permet de créer une séquence d'entier itérable
- Par défaut il suffit de lui préciser uniquement la fin de cette séquence

Exemple : **range(3)** → 0, 1, 2

- Il est aussi possible de lui préciser le début de la séquence, sa fin ainsi que le pas représentant l'écart entre chaque nombre.

Exemple : **range(0, 10, 1)** → 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

→ À noter que la borne de fin est exclus de cet interval

# La boucle while

`i = 0` — itérateur

mot clé    condition d'arrêt

`while` `i < 10` `:` — 2 points

`print(i)`  
`i += 1` — instructions

Figure: While avec itérateur

`reponse = -1`

mot clé    condition d'arrêt    2 points

`while` `reponse < 0 or reponse > 10` `:` — instructions

`reponse = input("Entrez un nombre : ")`  
`print("Vous avez entré :", reponse)`

Figure: While sans itérateur

- Sémantiquement, une boucle **while** est à utiliser quand on ne connaît pas le nombre de tours à effectuer.

# La boucle infinie

```
mot clé condition toujours vraie  
while True : 2 points  
    print("Boucle infinie !")  
instruction
```

Figure: Structure d'une boucle infinie

## Attention !

Ce genre de boucle est à utiliser avec précaution !

# Break, Continue et Pass

## Break

L'instruction **break** est utilisée pour interrompre une boucle avant qu'elle ne se termine normalement.

## Continue

L'instruction **continue** est utilisée pour passer à l'itération suivante sans exécuter le reste du code à l'intérieur de la boucle pour l'itération actuelle.

## Pass

L'instruction **pass** est utilisée pour indiquer un espace réservé pour un morceau de code à implémenter ultérieurement afin que le reste de votre code soit syntaxiquement valide.



# Liste de compréhension ?

## Comprehension list

Une compréhension de liste (ou "list comprehension" en anglais) est une façon concise et expressive de créer des listes en Python. Elle permet de combiner une boucle for avec une expression qui génère les éléments de la liste. Cela permet d'écrire du code plus compact et lisible.

# L'élégance de la liste de compréhension !

→ **Calculer le carré de chaque nombre d'une liste !**

- Code utilisant une boucle

```
numbers = [1, 2, 3, 4, 5]
squared = []

for x in numbers :
    squared.append(x ** 2)
```

- Code utilisant une liste de compréhension

```
numbers = [1, 2, 3, 4, 5]
squared = [x**2 for x in numbers]
```

# La compréhension de liste : explication (1)

liste de résultat      élément  
retourné      élément actuel      collection

```
squared = [x**2 for x in numbers]
```

mot clé

## Attention !

- N'oubliez pas la paire de [ ] autour de votre boucle / expression pour lui faire comprendre que le resultat attendu est une liste.

## La compréhension de liste : explication (2)

```
# Liste de compréhension avec if
even_numbers = [x for x in numbers if x % 2 == 0]
                expression / boucle      condition

# Liste de compréhension avec if / else
squared_cubed = [x**2 if x % 2 == 0 else x**3 for x in numbers]
                 resultat condition (if) condition resultat si      boucle
                 si if vrai      (else) else vrai
```

- Le **elif** n'existe pas dans les compréhensions de liste.
- Pour des conditions imbriquées placez vos autres **if / else** entre votre **else** et votre boucle.
- Attention pour **gagner en lisibilité** privilégiez plusieurs expressions plutôt qu'une énorme et complexe.

## La compréhension de liste : explication (3)

```
# Liste de compréhension utilisant 2 boucles  
flattened = [num for row in matrix for num in row]
```

retour      Boucle 1      Boucle 2

```
# Liste de compréhension utilisant 2 boucles et une condition  
flattened_pair = [num for row in matrix for num in row if num % 2 == 0]
```

retour      Boucle 1      Boucle 2      Condition

- Si vous souhaitez utiliser des listes de compréhension avec plusieurs boucles, placez votre première boucle juste après la valeur à retourner.
- Les instructions **if** seront toujours à placer après vos boucles, contrairement au bloc **if / else**, qui lui sera à placer avant celles-ci.

## Les compréhensions : tuple, set et dict

```
# Compréhension de tuple pour créer un tuple de carrés  
# de nombres pairs  
numbers = [1, 2, 3, 4, 5]  
squared_tuple = tuple(x**2 for x in numbers if x % 2 == 0)  
print(squared_tuple) # Output: (4, 16)
```

```
# Compréhension de set pour créer un ensemble de carrés  
# de nombres pairs  
squared_set = {x**2 for x in numbers if x % 2 == 0}  
print(squared_set) # Output: {16, 4}
```

```
# Compréhension de dict pour créer un dictionnaire avec  
# des lettres en clé et leur ASCII en valeur  
letters = {'a', 'b', 'c'}  
ascii_dict = {letter: ord(letter) for letter in letters}  
print(ascii_dict) # Output: {'a': 97, 'b': 98, 'c': 99}
```

# Les collections

- String → structure de données contenant une séquence de caractères Unicode, utilisé pour représenter du texte.
- List → structure de données contenant une séquence d'éléments, utilisé pour regrouper des éléments.
- Set → structure de données contenant des éléments uniques, utilisé pour effectuer des opérations ensemblistes telles que l'union, l'intersection et la différence.
- FrozenSet → similaire à un ensemble (set), mais immuable.
- Dict → structure de données avec des couples clé-valeur, où chaque clé est associée à une valeur unique, utilisé pour stocker et récupérer des données de manière associée.

## Caractéristiques des collections

| Collection   | Mutable | Ordonnées | Duplication | Syntaxe            |
|--------------|---------|-----------|-------------|--------------------|
| String       | ✗       | ✓         | ✓           | "abc"              |
| Liste        | ✓       | ✓         | ✓           | [0, 1, 2]          |
| Tuple        | ✗       | ✓         | ✓           | (0, 1, 2)          |
| Dictionnaire | ✓       | ✗         | Clé Unique  | {"clé" : "valeur"} |
| Set          | ✓       | ✗         | ✗           | 0, 2, 1            |
| Frozen Set   | ✗       | ✗         | ✗           | 0, 2, 1            |
| Array        | ✓       | ✓         | ✓           | [0, 0, 0]          |

- En Python le type **Array** n'est pas inclus nativement et nécessite l'utilisation d'une bibliothèque (numpy).
- Contrairement à une liste qui peut contenir des éléments de type différents celui-ci est moins flexible et ne peut contenir que des éléments de même type.



## Les méthodes propres aux collections mutable

- La méthode **append()** est utilisée pour ajouter un élément à la fin d'une liste.
- La méthode **remove()** est utilisée pour supprimer un élément d'une liste.
- L'instruction **del** est utilisée pour supprimer un élément d'une liste en utilisant son indice.

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

# Ajoute l'entier 4 à la liste
liste.append(4)
print(liste)  # Affiche : [1, 2, 3, 4]

# Supprime l'élément à l'indice 2
del liste[2]
print(liste)  # Affiche : [1, 2, 4]
```

```
# Supprime l'élément égal à 1
liste.remove(1)
print(liste) # Affiche : [2, 4]
```

## Les méthodes propres aux objets itérables : len

La méthode **len()** est une fonction qui est utilisée pour renvoyer la longueur d'une séquence, telle qu'une chaîne de caractères, une liste, un tuple, un dictionnaire, etc.

```
# Exemple avec une liste  
ma_liste = [1, 2, 3, 4, 5]  
print(len(ma_liste)) # Affiche : 5
```

```
# Exemple avec une chaîne de caractères  
ma_chaine = "Bonjour"  
print(len(ma_chaine)) # Affiche : 7 (caractères)
```

```
# Exemple avec un tuple  
mon_tuple = (10, 20, 30)  
print(len(mon_tuple)) # Affiche : 3
```

```
# Exemple avec un dictionnaire  
mon_dict = {'a': 1, 'b': 2, 'c': 3}  
print(len(mon_dict)) # Affiche : 3 (nombre de paires clé-valeur)
```

## Les méthodes propres aux objets itérables : any / all

Les méthodes **any()** et **all()** sont des fonctions utilisées pour évaluer des conditions sur une séquence d'éléments et renvoyer un booléen en fonction du résultat de cette évaluation.

- **any()** renvoi True si au moins un élément match avec la condition.
- **all()** renvoi True si tous les éléments match avec la condition.

```
# Exemple avec any
numbers = [1, 2, 3, 4, 5]
result_any = any(x > 3 for x in numbers)
print(result_any) # Output: True
```

```
# Exemple avec all
numbers = [1, 2, 3, 4, 5]
result_all = all(x > 3 for x in numbers)
print(result_all) # Output: False
```

## Les méthodes propres aux objets itérables : zip / map

- La fonction **map()** permet d'appliquer une fonction à chaque élément d'une ou plusieurs séquences sans avoir à utiliser une boucle explicite
- La fonction **zip()** permet de combiner les éléments de plusieurs séquences de manière ordonnée.

```
# Appliquer la fonction carré à chaque élément d'une liste
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x ** 2, numbers)
print(list(squared_numbers)) # Output: [1, 4, 9, 16, 25]
```

```
# Combinaison de deux listes en une liste de tuples
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]
combined_data = zip(names, ages)
print(list(combined_data))
# Output: [('Alice', 25), ('Bob', 30), ('Charlie', 35)]
```

## La fonction map : comparaison

```
# Utilisation de la fonction map
numbers = [1, 2, 3, 4, 5]
squared_with_map = list(map(lambda x: x ** 2, numbers))
print(squared_with_map) # Output: [1, 4, 9, 16, 25]
```

```
# Utilisation d'une boucle
squared_with_loop = []
for num in numbers:
    squared_with_loop.append(num ** 2)

print(squared_with_loop) # Output: [1, 4, 9, 16, 25]
```

## La fonction map : exemple

```
# Définition d'une fonction pour calculer le carré
def square(x):
    return x ** 2

# Utilisation de la fonction map avec la fonction square
numbers = [1, 2, 3, 4, 5]
squared_with_map = list(map(square, numbers))
print(squared_with_map) # Output: [1, 4, 9, 16, 25]
```

La fonction **map** fonctionne aussi bien avec une méthode déjà définie qu'avec une expression **lambda**. Une expression **lambda** est une fonction anonyme, c'est-à-dire une fonction sans nom. Elle est définie à l'aide du mot-clé **lambda** suivi d'une liste d'arguments, suivie d'une expression.

- `lambda argument : expression`
- `lambda argument_1, argument_2 : expression`

## La fonction map : fonctionnement

| Indice     | 0 | 1 | 2 | 3 | 4 |
|------------|---|---|---|---|---|
| Collection | 1 | 2 | 3 | 4 | 5 |

Méthode à appliquer : *Fonction carrée*  $\rightarrow$   $\text{valeur}^{**2}$

| Itération | Valeur d'origine | Résultat |
|-----------|------------------|----------|
| 0         | 1                | 1        |
| 1         | 2                | 4        |
| 2         | 3                | 9        |
| 3         | 4                | 16       |
| 4         | 5                | 25       |

## La fonction zip : comparaison

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]

# Utilisation de zip
combined_with_zip = list(zip(list1, list2))
print(combined_with_zip) # Output: [('a', 1), ('b', 2), ('c', 3)]

# Équivalent avec une compréhension de liste
combined_with_comprehension = [(list1[i], list2[i]) for i in range(min(len(list1), len(list2)))]
print(combined_with_comprehension) # Output: [('a', 1), ('b', 2), ('c', 3)]

# Utilisation de boucles classiques
combined_with_loops = []
for i in range(min(len(list1), len(list2))):
    combined_with_loops.append((list1[i], list2[i]))

print(combined_with_loops) # Output: [('a', 1), ('b', 2), ('c', 3)]
```

- Dans les exemples suivants où zip n'est pas utilisé, on applique un range avec comme fin `min(len(list1), len(list2))`. Ceci est utile dans le cas où les listes n'ont pas la même taille, ce qui n'est pas le cas ici. Ainsi, on aurait pu se limiter à un `range(len(list1))`.



# La fonction zip : fonctionnement

| Indice              | 0 | 1 | 2 | 3 | 4 | 5 |
|---------------------|---|---|---|---|---|---|
| <b>Collection 1</b> | 1 | 0 | 0 | 1 | 1 | 0 |
| <b>Collection 2</b> | 1 | 0 | 0 | 1 | 0 | 0 |

| Itération | Valeurs |
|-----------|---------|
| 0         | (1, 1)  |
| 1         | (0, 0)  |
| 2         | (0, 0)  |
| 3         | (1, 1)  |
| 4         | (1, 0)  |
| 5         | (0, 0)  |

# ZIP \*

## ZIP \*

En Python, l'opérateur `*` est utilisé pour débiller les éléments d'un itérable. Lorsqu'il est utilisé avec **zip**, il permet de débiller les éléments de chaque itérable passé à **zip** en tant qu'arguments distincts pour former des tuples.

```
listes = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Parcours toutes les sous-listes sans avoir à préciser le nom de chacune.
for elements in zip(*listes):
    print(elements) # Affiche : (1, 4, 7), (2, 5, 8), (3, 6, 9)
```

# La compréhension de liste à double résultat

```
liste1 = [1, 2, 3]  
liste2 = ['a', 'b', 'c']
```

```
resultat1, resultat2 = zip(*[(x, y) for x, y in zip(liste1, liste2)])  
print(resultat1) # Output: (1, 2, 3)  
print(resultat2) # Output: ('a', 'b', 'c')
```

## Explication

- Dans cet exemple, la compréhension de liste crée une liste de tuples en combinant les éléments de liste1 et liste2 ensemble grace à la méthode **zip()**.
- Ensuite, **zip(\*...)** est utilisé pour décompresser ces tuples en deux collections distinctes : resultat1 contenant les éléments de liste1 et resultat2 contenant les éléments de liste2.

# Les méthodes propres aux objets itérables : enumerate

## Enumerate

La fonction **enumerate()** est une fonction qui permet d'itérer simultanément sur les éléments d'une séquence et de récupérer à la fois l'indice de l'élément et la valeur de l'élément.

```
liste = ['a', 'b', 'c', 'd', 'e']  
  
for index, valeur in enumerate(liste):  
    print(f"Index : {index}, Valeur : {valeur}")
```

# Les méthodes propres aux objets itérables : slice

## slice

Utilisé pour extraire une portion spécifique d'une séquence, comme une liste, une chaîne de caractères, un tuple ou un array. La tranche commence à l'indice start, se termine avant l'indice stop et utilise un step facultatif pour spécifier l'incrément.

```
my_range = range(10)
slice_result = my_range[2:7:2]
print(list(slice_result)) # Output: [2, 4, 6]
```

```
my_string = "Hello, World!"
slice_result = my_string[7:12]
print(slice_result) # Output: "World"
```

## Les méthodes propres aux objets itérables : slice (2)

### Astuce

Si la valeur start n'est pas spécifiée, la tranche commencera depuis le début (0) de la séquence de même pour la fin. De plus l'utilisation d'un step négatif permet de parcourir la séquence en sens inverse.

```
original_list = [1, 2, 3, 4, 5]

# Utilisation de slice avec step -1 pour inverser la liste
reversed_list = original_list[::-1]

print(reversed_list)
```

## Les indices + / - en Python

|                         |    |    |    |    |    |    |
|-------------------------|----|----|----|----|----|----|
| <b>Indices Normaux</b>  | 0  | 1  | 2  | 3  | 4  | 5  |
| <b>Collection</b>       | A  | B  | C  | D  | E  | F  |
| <b>Indices Inversés</b> | -6 | -5 | -4 | -3 | -2 | -1 |

### Les indices inversés

En Python, il est possible de parcourir une collection en utilisant les indices inversés. Pour bien comprendre comment cela fonctionne, prenons une chaîne de caractères `message = "ABCDEF"`.

- `message[0:3]` → "ABC"
- `message[2::-1]` → "CBA"
- `message[3:]` → "DEF"
- `message[-1:2:-1]` → "FED"
- `message[::-1]` → "FEDCBA"

## Les méthodes propres au String : split et splitlines

- **split()** est utilisée pour diviser une chaîne de caractères en fonction d'un séparateur spécifié. Par défaut, le séparateur est l'espace (' ').
- **splitlines()** est utilisée pour diviser une chaîne de caractères en fonction des séparateurs de fin de ligne. Les séparateurs de fin de ligne peuvent être '\n', '\r' ou '\r\n' selon la plate-forme.

```
phrase = "Bonjour à tous"
mots = phrase.split(' ')
print(mots) # Affiche : ['Bonjour', 'à', 'tous']
```

```
texte = "Bonjour\nau\nmonde\n"
lignes = texte.splitlines()
print(lignes) # Affiche : ['Bonjour', 'au', 'monde']
```



# Les méthodes propres aux dictionnaires

- **keys()** renvoie une vue sur les clés du dictionnaire.
- **values()** renvoie une vue sur les valeurs du dictionnaire.
- **items()** renvoie une vue sur les paires (clé, valeur) du dictionnaire.

```
mon_dict = {'a': 1, 'b': 2, 'c': 3}

# Récupération des clés
cles = mon_dict.keys()
print(cles) # Affiche : dict_keys(['a', 'b', 'c'])

# Récupération des valeurs
valeurs = mon_dict.values()
print(valeurs) # Affiche : dict_values([1, 2, 3])

# Récupération des paires (clé, valeur)
paires = mon_dict.items()
print(paires) # Affiche : dict_items([('a', 1), ('b', 2), ('c', 3)])
```

## Les méthodes propres aux dictionnaires

- Pour ajouter une nouvelle paire clé-valeur à un dictionnaire, il suffit d'assigner une valeur à une nouvelle clé. De même, pour modifier la valeur associée à une clé existante, vous pouvez simplement réaffecter une nouvelle valeur à cette clé.
- La méthode **pop()** supprime un élément spécifique d'un dictionnaire en spécifiant la clé correspondante, et retourne également la valeur associée à cette clé.

```
mon_dictionnaire = {'a': 1, 'b': 2}
```

```
# Ajoute la clé c au dictionnaire avec pour valeur 3
```

```
mon_dictionnaire['c'] = 3
```

```
print(mon_dictionnaire) # Affiche : {'a': 1, 'b': 2, 'c': 3}
```

```
# Retire la clé b du dictionnaire ainsi que sa valeur associée  
valeur_supprimee = dictionnaire.pop('b')
```

## Les valeurs par défaut

| Nom                  | Type   | Valeur par défaut |
|----------------------|--------|-------------------|
| Entier               | int    | 0                 |
| Décimal              | float  | 0.0               |
| Booléen              | bool   | False             |
| Chaîne de caractères | str    | "                 |
| Liste                | list   | []                |
| Tableau              | array  | []                |
| Tuple                | tuple  | ()                |
| Dictionnaire         | dict   | {}                |
| Set                  | set    | {}                |
| Objet                | object | None              |

# Plan du Chapitre

- 1 Introduction
- 2 Propriétés du langage
- 3 Initiation à la programmation Python
- 4 Mécanisme d'exécution
  - Exécution / Fonctionnement
  - Installation et pré-requis
- 5 Sources

# Exécution d'un programme Python

## Exécution

- 1 Sauvegarder votre code Python avec l'extension `.py`
- 2 Ouvrez un terminal
- 3 Effectuez la commande suivante : `python nom_de_votre_script.py`

# Fonctionnement

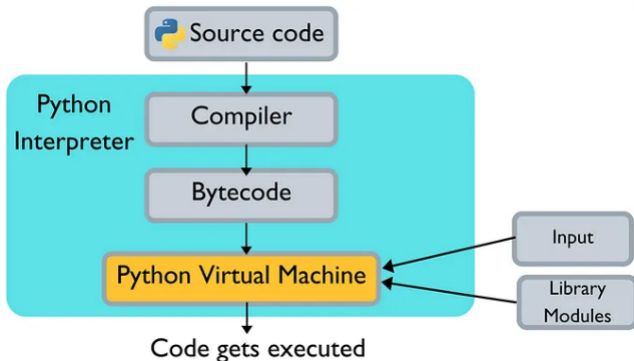


Figure: Fonctionnement de l'interpréteur python (Source)

# Installation

## **Besoin d'installer Python ?**

- Installation sous MacOS
- Installation sous Windows

## **Utiliser Python sans installation ?**

- Google Colab

## **IDE conseillé :**

- Pycharm

## Conflit : python2 & python3

Si vous possédez python 2 et python 3 installé sur votre machine :

- ❶ Précisez la version à l'exécution → `python3 nom_script.py`
- ❷ Configurez votre machine
  - Rendez vous dans votre `~/ .bashrc`
  - Placez vous tout en bas
  - Configurez un alias → `alias python='python3'`

### Connaître sa version ?

- Exécutez dans un terminal la commande suivante :  
`python --version`



# Plan du Chapitre

- 1 Introduction
- 2 Propriétés du langage
- 3 Initiation à la programmation Python
- 4 Mécanisme d'exécution
- 5 Sources

# Sources

- Python 2 vs 3: Everything You Need to Know
- Geekforgeek
- Wikipedia
- A basic introduction to Python
- ChatGPT

## TP 2 à 5

### Listes des TPs :

- TP 2 : <https://adventofcode.com/2017/day/1>
- TP 3 : <https://adventofcode.com/2018/day/1>
- TP 4 : <https://adventofcode.com/2020/day/1>
- TP 5 : <https://adventofcode.com/2017/day/2>