



# Formation Python Session 1

30/03/2023

# SOMMAIRE

1

Introduction

2

Environnement de développement

3

Variables et types de données

4

Structure de contrôle: if/else, for, while

5

Fonctions

6

Classes et objets

7

Décorateurs

8

Fichiers et exceptions

9

Tests unitaires







# 1- Introduction

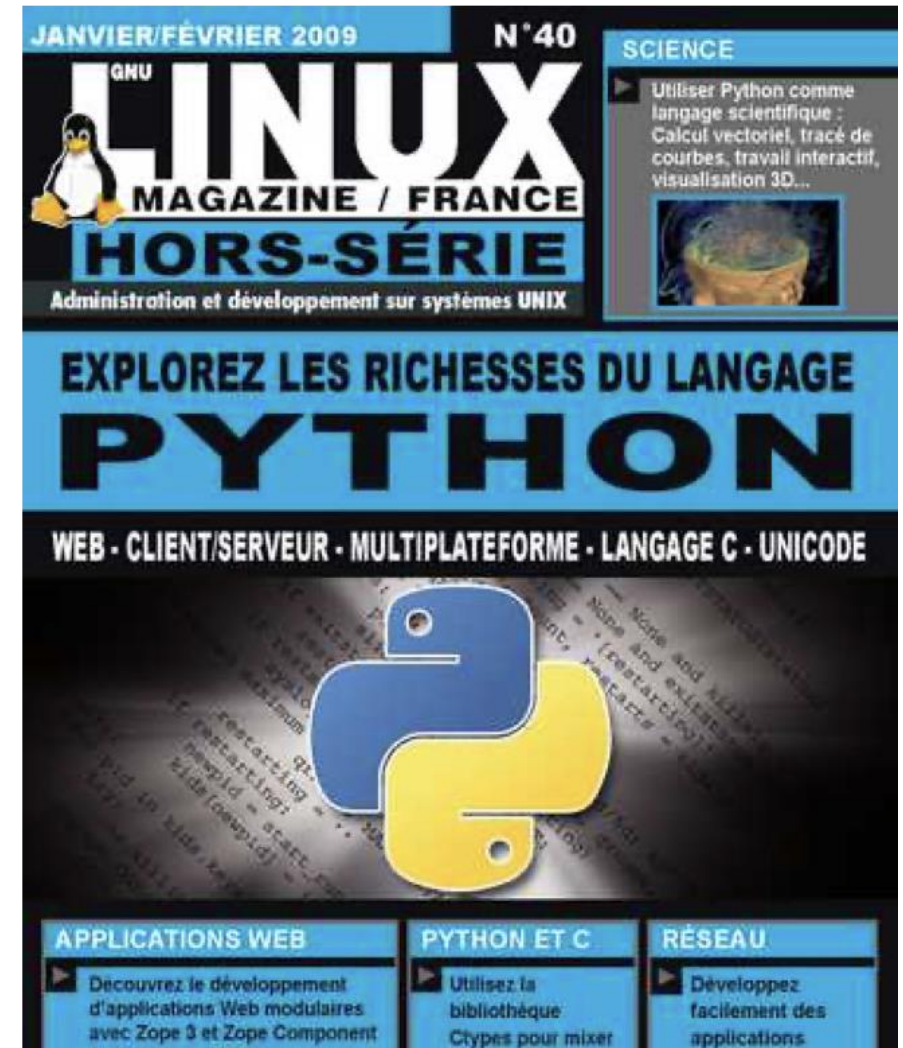
---

Python est un langage de programmation généraliste populaire et open-source, qui a été créé par Guido van Rossum en 1989 et a connu une croissance exponentielle depuis lors. Python est apprécié pour sa syntaxe simple et lisible, ce qui facilite l'apprentissage et la compréhension du code par les développeurs de tous niveaux.






Python est également connu pour sa facilité d'utilisation et son flexibilité. Il peut être utilisé pour une grande variété de tâches, allant de la création de scripts automatisant des tâches simples à la création de programmes complexes pour les domaines de l'intelligence artificielle, de la science des données, de la finance, de la bio-informatique et plus encore.



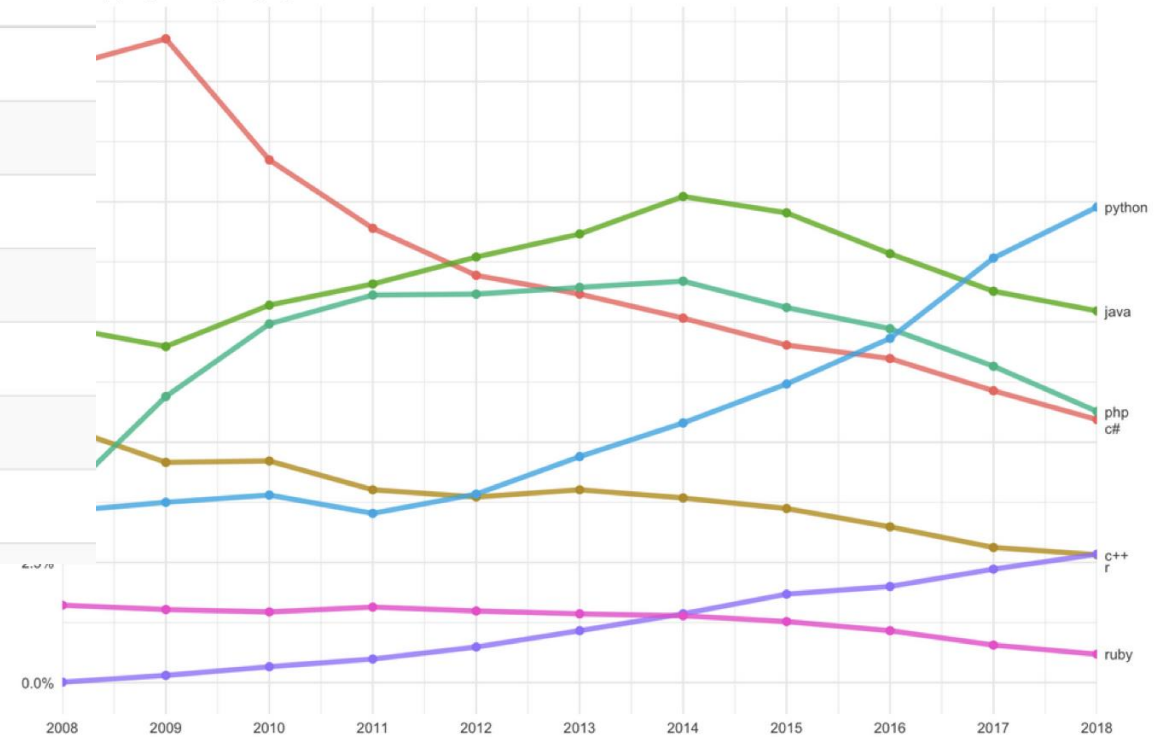
- **1990**
  - décembre: Guido commence l'écriture de Python
- **1991**
  - février: version 0.9.0
- **2000:**
  - 16 octobre: Python 2.0
- **2008**
  - 1er octobre: Python 2.6
  - 3 octobre: Python 3.0
- **2014**
  - 16 mars: Python 3.4 et asyncio



# 1- Introduction: Popularité

Feb 2022	Feb 2021	Change	Programming Language	Ratings	Change
1	3	▲	 Python	15.33%	+4.47%
2	1	▼	 C	14.08%	-2.26%
3	2	▼	 Java	12.13%	+0.84%
4	4		 C++	8.01%	+1.13%
5	5		 C#	5.37%	+0.93%
6	6		 Visual Basic	5.23%	+0.90%
7	7		 JavaScript	1.83%	-0.45%

Fraction of total questions per year in Stack Overflow  
top programming languages







# 1- Introduction: Définition

---

Python est un langage **interprété**, ce qui signifie que le code peut être exécuté directement sans avoir besoin d'être compilé. Python est également **multiplateforme**, ce qui signifie que les programmes écrits en Python peuvent être exécutés sur divers systèmes d'exploitation, y compris Windows, MacOS et Linux.

Enfin, Python possède une large communauté de développeurs, qui contribuent régulièrement à son développement et à son amélioration. Cela signifie que les bibliothèques et les outils disponibles pour Python sont nombreux et en constante évolution, ce qui en fait un choix populaire pour les projets de développement logiciel modernes.



# 1- Introduction: Applications en python



Voici quelques exemples d'applications et de sites populaires codés en Python :

Instagram : Instagram utilise Python pour gérer les interactions entre les utilisateurs, gérer les notifications et les recommandations.

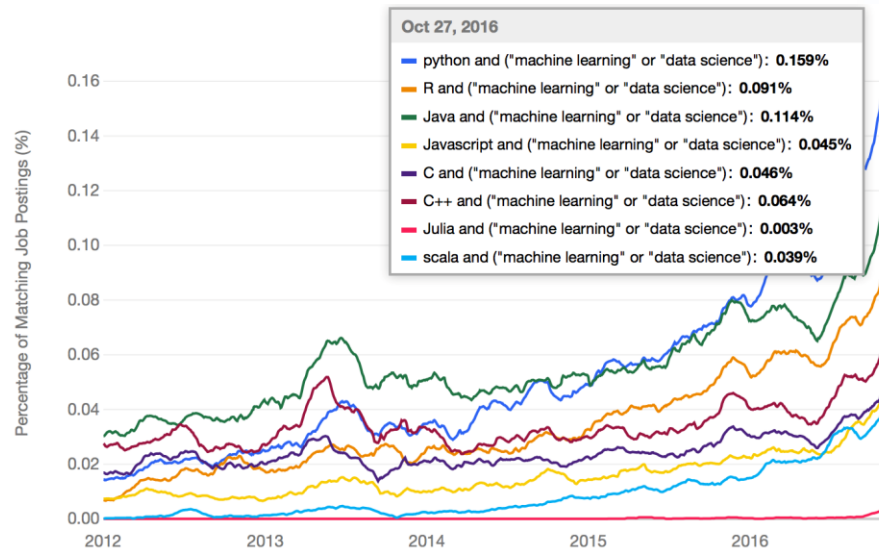
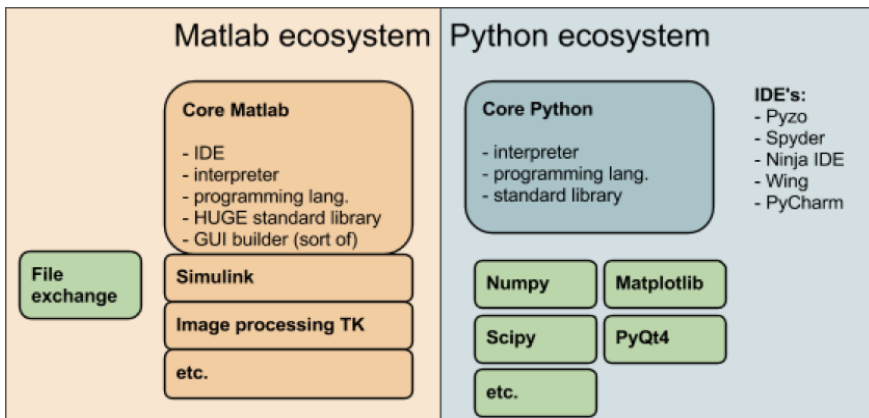
Pinterest : Pinterest utilise Python pour gérer les recommandations de contenu, les recherches et la collecte de données d'utilisation.

Spotify : Spotify utilise Python pour gérer les recommandations de musique, la gestion de la playlist et le traitement des données de lecture.

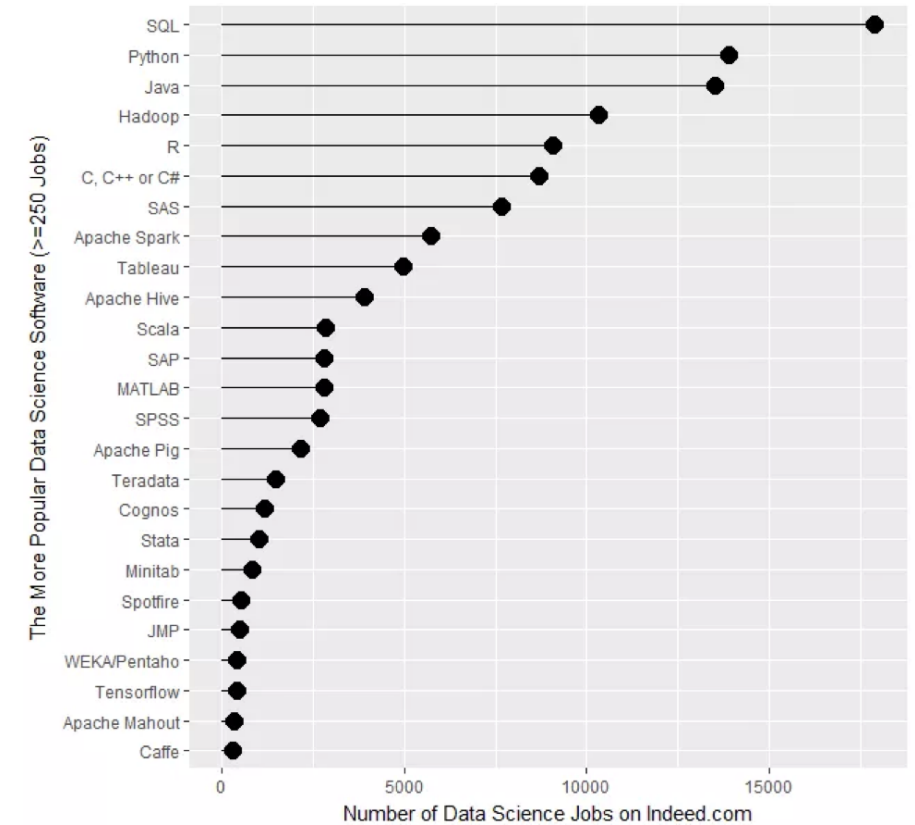
Uber : Uber utilise Python pour gérer le calcul des itinéraires, la mise en correspondance des chauffeurs avec les passagers et le traitement des données de voyage.

NASA : La NASA utilise Python pour gérer les données et les images des missions spatiales, les simulations et les analyses scientifiques.

# 1- Introduction: Applications



TOP PYTHON MACHINE LEARNING LIBRARIES





# Python vs Java

## Python

- Langage **interprété**
- **Language à typage dynamique**: les types sont découverts à l'exécution.

## Java

- Langage **compilé**
- **Language à typage statique**: les types sont fixés à la compilation.

**Python et Java sont deux langage fortement typés.**

exemple: si je veux manipuler un entier comme une chaine de caractère, je vais devoir le convertir en chaine de caractère d'abord.

Alors qu'en Javascript, langage faible typé, "2" + 4 -> "24"



# Python vs Java



---

L'un des inconvénients de Python, c'est qu'il est connu pour être relativement lent comparé à d'autres langages notamment Java.

Plusieurs solution permettent d'améliorer les performances tel que **Numba** :

Numba, compilateur Just-In-Time (JIT) pour Python qui permet d'accélérer l'exécution de code Python en le traduisant en code machine optimisé. Il est principalement utilisé pour accélérer les calculs numériques et scientifiques en Python, mais il peut également être utilisé pour accélérer d'autres types de calculs.

Numba fonctionne en analysant le code Python et en compilant les parties les plus intensives en calcul en code machine optimisé. Il utilise le compilateur LLVM pour générer le code machine, ce qui permet d'obtenir des performances proches de celles du code C ou Fortran.






# Implémentations

Il existe plusieurs implémentations de Python, qui sont des versions du langage Python écrites dans différents langages et qui peuvent avoir des fonctionnalités et des performances différentes. Voici quelques-unes des implémentations les plus courantes :

- **CPython** : CPython est la principale implémentation de Python, écrite en langage C. Il est considéré comme la référence pour le langage Python, car il est maintenu par la communauté Python et est utilisé comme l'interpréteur de Python par défaut dans la plupart des systèmes d'exploitation.
- **Jython** : Jython est une implémentation de Python qui s'exécute sur la machine virtuelle Java (JVM). Il permet d'utiliser Python dans des applications Java et offre une intégration transparente avec les bibliothèques Java.
- **IronPython** : IronPython est une implémentation de Python pour le framework .NET de Microsoft. Il permet d'utiliser Python dans des applications .NET et offre une intégration transparente avec les bibliothèques .NET.
- **PyPy** : PyPy est une implémentation de Python qui utilise un compilateur Just-In-Time (JIT) pour améliorer les performances des applications Python. Il est particulièrement utile pour les applications intensives en calcul.
- **MicroPython** : MicroPython est une implémentation de Python qui a été optimisée pour fonctionner sur des microcontrôleurs et des plates-formes de l'Internet des objets (IoT). Il est capable de s'exécuter sur des ressources très limitées, ce qui le rend particulièrement utile pour les projets d'IoT.
- **Stackless Python** : Stackless Python est une implémentation de Python qui prend en charge la programmation concurrente et parallèle, ainsi que la continuité de tâche. Il permet également d'effectuer des opérations en continu sans avoir à utiliser des threads.

Ces implémentations de Python offrent des fonctionnalités supplémentaires et des performances différentes, ce qui peut les rendre plus adaptées à certaines situations ou projets.







# Distributions



Différentes distributions sont disponibles, qui incluent parfois beaucoup de packages dédiés à un domaine donné :

- **CPython** : CPython est la distribution de référence de Python, qui est développée et maintenue par la communauté Python. Il est écrit en langage C et est utilisé comme interpréteur de Python par défaut dans de nombreux systèmes d'exploitation.
- **Anaconda** : Anaconda est une distribution Python populaire utilisée pour l'analyse de données et la science des données. Il est livré avec un grand nombre de bibliothèques scientifiques pré-installées, ainsi qu'un gestionnaire de paquets et un environnement virtuel.
- **Miniconda**: une version minimaliste de Anaconda et embarque un Python de base ainsi que le gestionnaire de packages Conda.
- **Intel Distribution for Python**: distribution basée sur Anaconda, intégrant notamment la bibliothèque MKL (en) d'Intel afin d'accélérer les calculs numériques de bibliothèques telles que NumPy et SciPy, intégrées à la distribution. Elle est disponible gratuitement seule, ou bien intégrée à Intel Parallel Studio (en), qui nécessite une licence payante.
- **Pyzo** : « Python to the people », destinée à être facile d'utilisation.

Ce ne sont pas des implémentations différentes du langage Python : elles sont basées sur CPython, mais sont livrées avec un certain nombre de bibliothèques préinstallées.





# Versions



- Python 1 : sortie en 1994, c'est la première version publique de Python.
- Python 2 : sortie en 2000, c'est une version majeure qui a introduit de nombreuses fonctionnalités, mais elle n'est plus maintenue depuis le 1er janvier 2020.
- Python 3 : sortie en 2008, c'est la version actuelle de Python et elle est en développement continu. Python 3 apporte de nombreuses améliorations et de nouvelles fonctionnalités par rapport à Python 2, mais il y a aussi des changements importants dans la syntaxe et le comportement de certaines fonctions.

Dernière version stable à ce jour : Python 3.10.10







# IDEs



Il y a plusieurs IDE (Integrated Development Environment) populaires pour Python, voici quelques exemples :

- **PyCharm** : un IDE populaire avec une interface utilisateur conviviale pour les développeurs Python professionnels. Il est développé par JetBrains et est disponible en versions gratuites et payantes.
  - **Spyder** : un IDE open-source pour Python qui est conçu spécifiquement pour les scientifiques et les ingénieurs. Il fournit un environnement de développement intégré pour l'analyse de données, la visualisation et le débogage.
  - **Visual Studio Code** : un éditeur de code multiplateforme qui offre une expérience de développement de qualité pour les développeurs Python.
  - **Sublime Text** : un éditeur de code très personnalisable et rapide pour les développeurs Python. Il est livré avec un grand nombre de plugins pour améliorer la productivité.
  - **Jupyter Notebook** : une application Web open-source qui permet de créer et de partager des documents contenant du code, des équations, des visualisations et des textes. C'est un excellent outil pour l'exploration de données et l'enseignement de la programmation.
  - **IDLE** : l'environnement de développement intégré (IDE) officiel de Python. C'est un environnement léger et simple qui peut être utilisé pour écrire et exécuter des programmes Python.
- 
- 




## 2- Environnement virtuel

---

En Python, un environnement virtuel est un espace de travail isolé qui contient une version spécifique de Python et des packages installés. Les environnements virtuels permettent aux développeurs de travailler sur plusieurs projets simultanément, chacun avec ses propres dépendances, sans interférer avec d'autres projets ou avec le système global de Python.

Les environnements virtuels permettent aux développeurs de travailler avec des versions différentes de Python et des packages spécifiques à chaque projet.





## 2- Comment configurer un environnement virtuel ?

---

1- Tout d'abord, installez **pipenv** en utilisant **pip** (le gestionnaire de paquets de Python) en exécutant la commande suivante dans votre terminal :

**pip install pipenv**

2- Naviguez jusqu'au répertoire de votre projet en utilisant votre terminal.

3- Exécutez la commande **pipenv shell** pour activer l'environnement virtuel. Cela créera automatiquement un nouveau fichier **Pipfile** et un fichier **Pipfile.lock** pour votre projet.

4- Utilisez la commande **pipenv install** pour installer les dépendances de votre projet dans l'environnement virtuel. Vous pouvez spécifier des packages et des versions spécifiques dans le fichier Pipfile en utilisant la syntaxe appropriée.

5- Pour désactiver l'environnement virtuel, utilisez la commande **exit**.







# 3- Variables et types de donnée

---

Python prend en charge plusieurs types de données, chacun ayant ses propres propriétés et méthodes.

En Python, les types de données sont classés en deux catégories : **mutables** et **immuables**. Les types de données **immuables** sont des types de données qui ne peuvent pas être modifiés une fois qu'ils ont été créés.

Les types de données **mutables**, en revanche, sont des types de données qui peuvent être modifiés après leur création.

Nous allons détailler pour chaque type de donnée.





# 3- Variables et types de donnée

---

Python prend en charge plusieurs types de données, chacun ayant ses propres propriétés et méthodes.

En Python, les types de données sont classés en deux catégories : **mutables** et **immuables**. Les types de données **immuables** sont des types de données qui ne peuvent pas être modifiés une fois qu'ils ont été créés.

Les types de données **mutables**, en revanche, sont des types de données qui peuvent être modifiés après leur création.

Nous allons détailler pour chaque type de donnée.



# 3- Variables: Normes de nommage

- Les noms de variable doivent être en minuscules et séparés par des underscores (\_). Par exemple : **my\_variable**, **my\_other\_variable**.
- Les noms de variable ne doivent pas commencer par un chiffre, mais peuvent contenir des chiffres. Par exemple, `my_variable_1` est un nom de variable valide.
- Les noms de variable ne doivent pas contenir d'espaces. Si vous avez besoin d'utiliser plusieurs mots dans le nom de la variable, utilisez plutôt des underscores pour les séparer.
- Les noms de variable peuvent inclure des lettres minuscules et majuscules, ainsi que des chiffres et des underscores. Cependant, il est généralement recommandé d'utiliser uniquement des lettres minuscules et des underscores pour des raisons de lisibilité.
- Les noms de variable ne doivent pas être des mots-clés réservés du langage Python.

Mots réservés en Python				
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	

# 3-1 Les booléens

un booléen est un type de données qui représente deux valeurs possibles : True (vrai) et False (faux).

En Python, **True** et **False** sont des mots-clés réservés, ce qui signifie qu'ils ont une signification spéciale dans le langage et ne peuvent pas être utilisés comme noms de variables.

Les booléens sont **immuables**.

Voici quelques exemples pour mieux comprendre les booléens en Python

```
# Exemple 1 : déclaration de variables booléennes
est_vrai = True
est_faux = False

# Exemple 2 : utilisation dans des conditions
if est_vrai:
    print("La condition est vraie")
else:
    print("La condition est fausse")

# Exemple 3 : utilisation dans des opérations booléennes
a = True
b = False
c = a and b  # Résultat : False
d = a or b   # Résultat : True
e = not a    # Résultat : False
```

## 3-2 Les entiers

En Python, les entiers sont un type de données qui représentent des nombres entiers. Les entiers peuvent être positifs, négatifs ou nuls.

Les entiers sont **immuables**.

Dans CPython (l'implémentation de référence de Python), la taille maximale d'un entier dépend de la plateforme sur laquelle Python est en cours d'exécution. En général, sur une plateforme 32 bits, la taille maximale d'un entier est  **$2^{31}-1$  (2147483647)**, tandis que sur une plateforme 64 bits, la taille maximale d'un entier est  **$2^{63}-1$  (9223372036854775807)**.

```
# Exemple 1 : déclaration de variables entières
```

```
a = 10
```

```
b = -5
```

```
c = 0
```

```
# Exemple 2 : utilisation dans des opérations mathématiques
```

```
d = a + b    # Résultat : 5
```

```
e = a * c    # Résultat : 0
```

```
f = b ** 2   # Résultat : 25
```

```
# Exemple 3 : conversion d'une chaîne de caractères en entier
```

```
chaine_entier = "42"
```

```
entier = int(chaine_entier) # entier vaut 42
```

## 3-2 Les ranges

Le type range représente une séquence immuable de nombres et est couramment utilisé pour itérer un certain nombre de fois dans les boucles for.

range(stop)

range(start, stop[, step])

start: point de départ (0 par défaut)

stop: valeur de fin de séquence (exclut)

step: pas

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

## 3-2 Les entiers

Opérations	Résultat
$x + y$	
$x - y$	
$x * y$	
$x / y$	division 'flottante'
$x // y$	division entière
$x \% y$	le reste de la division entière
$-x$	la négation de $x$
$+x$	$x$ non-changé
<code>abs(x)</code>	valeur absolue de $x$
<code>int(x)</code>	<i><math>x</math> est convertit en entier</i>
<code>divmod(x, y)</code>	la paire $(x // y, x \% y)$
<code>pow(x, y)</code>	$x$ à la puissance $y$
$x ** y$	$x$ à la puissance $y$

## 3-3 Les nombres flottants

Les flottants en Python ont une précision limitée en raison de la façon dont ils sont stockés en mémoire.

Cela signifie que les opérations sur les flottants peuvent parfois entraîner des erreurs d'arrondi. Il est donc important d'être conscient de ces limitations lors de l'utilisation de flottants dans des programmes.

Les flottants sont **immuables**.

La plage de valeurs possibles va d'environ  $2.2e-308$  à  $1.8e308$  (environ)

En général, les flottants ont une précision d'environ 15 à 17 chiffres décimaux significatifs, ce qui signifie que les nombres plus grands ou plus petits que cela peuvent perdre de la précision.

```
# Exemple 1 : déclaration de variables flottantes
```

```
a = 3.14159
```

```
b = -2.5
```

```
c = 0.0
```

```
# Exemple 2 : utilisation dans des opérations mathématiques
```

```
d = a + b    # Résultat : 0.64159
```

```
e = a * c    # Résultat : 0.0
```

```
f = b ** 2   # Résultat : 6.25
```

```
# Exemple 3 : conversion d'une chaîne de caractères en flottant
```

```
chaine_flottant = "3.14"
```

```
flottant = float(chaine_flottant) # flottant vaut 3.14
```



## 3-3 Les nombres flottants: affichage

Pour l'affichage nous allons utiliser la fonction: `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`

-> sep correspond au séparateur entre les valeurs

-> end correspond au caractère ajouté à la fin (par défaut, un retour chariot)

`print` en Python est l'équivalent de en Java de `System.out.print()`

L'affichage d'un float avec beaucoup de chiffres après la virgule en Python est possible grâce à la méthode `format()` qui permet de spécifier le nombre de chiffres après la virgule à afficher.

Voici un exemple pour afficher un float avec 10 chiffres après la virgule :

```
pi = 3.141592653589793238462643383279502884197169399375105820974944592307816406286
print("La valeur de pi est : {:.10f}".format(pi))
```

Le résultat de l'exécution sera:

```
La valeur de pi est : 3.1415926536
```

## 3-4 Les complexes

En Python, un nombre complexe est un nombre qui a une partie réelle et une partie imaginaire, et qui est représenté par la lettre **j** après la partie imaginaire.

On peut effectuer des opérations mathématiques sur des nombres complexes, par exemple l'addition, la soustraction, la multiplication et la division. Voici quelques exemples :

On peut également accéder à la partie réelle et à la partie imaginaire d'un nombre complexe à l'aide des attributs **real** et **imag**. Voici un exemple :

```
z = 2 + 3j
print(z.real) # Output: 2.0
print(z.imag) # Output: 3.0
```

```
# Addition de nombres complexes
a = 2 + 3j
b = 4 - 1j
c = a + b
print(c) # Output: (6+2j)

# Soustraction de nombres complexes
a = 2 + 3j
b = 4 - 1j
c = a - b
print(c) # Output: (-2+4j)

# Multiplication de nombres complexes
a = 2 + 3j
b = 4 - 1j
c = a * b
print(c) # Output: (11+10j)

# Division de nombres complexes
a = 2 + 3j
b = 4 - 1j
c = a / b
print(c) # Output: (0.56+0.77j)
```

## 3-5 Le module math

En Python, le module **math** est une bibliothèque standard qui fournit des fonctions mathématiques de base. Pour l'utiliser, il suffit d'importer le module en utilisant la commande `import math`.

Voici quelques exemples d'utilisation de ce module :

Fonctions	Documentation
<code>math.sin(x)</code>	sinus
<code>math.cos(x)</code>	cosinus
<code>math.tan(x)</code>	tangente
<code>math.asin(x)</code>	arcsinus
<code>math.acos(x)</code>	arccosinus
<code>math.atan(x)</code>	arctangente
<code>math.atan2(y, x)</code>	arctangente de y/x
<code>math.sinh(x)</code>	sinus hyperbolique
<code>math.cosh(x)</code>	cosinus hyperbolique
<code>math.tanh(x)</code>	tangente hyperbolique
<code>math.asinh(x)</code>	arcsinus hyperbolique
<code>math.acosh(x)</code>	arccosinus hyperbolique
<code>math.atanh(x)</code>	arctangente hyperbolique

Fonctions	Documentation
<code>math.ceil(x)</code>	retourne n entier tel que $n-1 \leq x < n$
<code>math.floor(x)</code>	retourne n entier tel que $n \leq x < n+1$
<code>math.fabs(x)</code>	la valeur absolue de x
<code>math.factorial(x)</code>	factorielle de x
<code>math.gcd(a, b)</code>	plus grand commun diviseur de a et b
<code>math.isclose(a, b, *, rel_tol=1e-09, abs_tol=0.0)</code>	renvoie True si a et b sont proches
<code>math.isfinite(x)</code>	True si x est fini
<code>math.isinf(x)</code>	True si x est infini
<code>math.isnan(x)</code>	True si x n'est pas un nombre
<code>math.exp(x)</code>	exponentielle de x
<code>math.log(x [, base])</code>	logarithme népérien ou de base quelconque
<code>math.log10(x)</code>	logarithme en base 10
<code>math.sqrt(x)</code>	racine carrée de x
<code>math.gamma(x)</code>	gamma de x
<code>math.lgamma(x)</code>	logarithme naturel de gamma(x)

## 3-6 Les comparaisons

Opération	Résultat
<	strictement inférieur à
<=	inférieur ou égal
>	strictement supérieur à
>=	supérieur ou égal à
==	égal
!=	différent
is	identité d'objet
is not	négation de l'identité d'objet

Python supporte des écritures comme:

$x < y \leq z$

qui est équivalent à:

$x < y$  **and**  $y \leq z$

# 3-7 Les listes

les listes sont un type de données qui permettent de stocker une collection **ordonnée** d'éléments.

Les éléments d'une liste peuvent être de différents types de données, tels que des entiers, des flottants, des chaînes de caractères, des booléens, etc.

Les listes sont **mutables**, ce qui signifie qu'il est possible de modifier leurs éléments.

```
# Exemple 1 : déclaration de listes
liste1 = [1, 2, 3, 4, 5]
liste2 = ["Bonjour", "tout", "le", "monde"]

# Exemple 2 : accès aux éléments d'une liste
premier_element = liste1[0] # premier_element vaut 1
dernier_element = liste2[-1] # dernier_element vaut "monde"

# Exemple 3 : modification d'une liste
liste1[2] = 10 # La liste devient [1, 2, 10, 4, 5]
liste2.append("!") # La liste devient ["Bonjour", "tout", "le", "monde", "!"]
```

# 3-7 Les listes

Opération	Résultat
<code>x in s</code>	True si un élément de <code>s</code> est égal à <code>x</code> , sinon False
<code>x not in s</code>	False si un élément de <code>s</code> est égal à <code>x</code> , sinon True
<code>s + t</code>	la concaténation de <code>s</code> et <code>t</code>
<code>s * n</code> or <code>n * s</code>	équivalent à ajouter <code>s</code> <code>n</code> fois à lui même
<code>s[i]</code>	<code>i</code> ème élément de <code>s</code> en commençant par 0
<code>s[i:j]</code>	tranche ( <i>slice</i> ) de <code>s</code> de <code>i</code> à <code>j</code>
<code>s[i:j:k]</code>	tranche ( <i>slice</i> ) de <code>s</code> de <code>i</code> à <code>j</code> avec un pas de <code>k</code>
<code>s.append(x)</code>	ajoute <code>x</code> à la fin de <code>s</code>
<code>s[i] = x</code>	élément <code>i</code> (basée à partir de 0) de <code>s</code> est remplacé par <code>x</code>
<code>s[i:j] = t</code>	tranche de <code>s</code> de <code>i</code> à <code>j</code> est remplacée par le contenu de l'itérable <code>t</code>
<code>s[i:j:k] = t</code>	les éléments de <code>s[i:j:k]</code> sont remplacés par ceux de <code>t</code>
<code>del s[i:j]</code>	identique à <code>s[i:j] = []</code>
<code>del s[i:j:k]</code>	identique à <code>s[i:j:k] = []</code>
<code>s.clear()</code>	supprime tous les éléments de <code>s</code> (équivalent à <code>del s[:]</code> )
<code>len(s)</code>	longueur de <code>s</code>
<code>min(s)</code>	plus petit élément de <code>s</code>
<code>max(s)</code>	plus grand élément de <code>s</code>
<code>s.index(x[, i[, j]])</code>	indice de la première occurrence de <code>x</code> dans <code>s</code> (à ou après l'indice <code>i</code> et avant indice <code>j</code> )
<code>s.count(x)</code>	nombre total d'occurrences de <code>x</code> dans <code>s</code>
<code>s.extend(t)</code> or <code>s += t</code>	étend <code>s</code> avec le contenu de <code>t</code> (proche de <code>s[len(s):len(s)] = t</code> )
<code>s *= n</code>	met à jour <code>s</code> avec son contenu répété <code>n</code> fois
<code>s.insert(i, x)</code>	insère <code>x</code> dans <code>s</code> à l'index donné par <code>i</code> (identique à <code>s[i:i] = [x]</code> )
<code>s.pop([i])</code>	recupère l'élément à <code>i</code> et le supprime de <code>s</code>
<code>s.remove(x)</code>	supprime le premier élément de <code>s</code> pour qui <code>s[i] == x</code>
<code>s.reverse()</code>	inverse sur place les éléments de <code>s</code>

## 3-7 Les listes: Python vs Java

En comparaison, en Java, les listes sont implémentées en utilisant des classes telles que **ArrayList** ou **LinkedList**.

Les listes en Java sont également mutables et peuvent contenir des éléments de différents types de données.

	Python	Java
Add item to list	<code>lis.append(1)</code>	<code>lis.add(1)</code>
Getting value from index	<code>lis[0]</code>	<code>lis.get(0)</code>
Length	<code>len(lis)</code>	<code>lis.size()</code>
Counting occurrences in list	<code>list.count(1)</code>	-
finding first-found index of "apple"	<code>lis.index("apple")</code>	<code>lis.indexOf("apple")</code>
boolean value checking if "apple" exists in list	<code>"apple" in lis</code>	<code>lis.contains("apple")</code>
sorting elements of lis from small to large	<code>lis.sort()</code>	<code>collections.sort(lis)</code>

## 3-7 Les list comprehension

En Python, les listes de compréhension sont une façon concise et expressive de créer des listes en utilisant une syntaxe simplifiée.

La structure de base d'une liste de compréhension est la suivante :

```
[expression for item in iterable if condition]
```

où :

- expression est une expression Python qui sera évaluée pour chaque élément dans l'itérable,
- item est une variable qui représente chaque élément de l'itérable,
- iterable est un objet itérable comme une liste, un tuple ou une chaîne de caractères,
- condition est une expression Python optionnelle qui filtre les éléments en fonction d'une condition.





## 3-7 Les list comprehension: Exercice

---

Enoncé: Créer une liste qui contient uniquement les nombres pairs de 0 à 11.



## 3-7 Les list comprehension: Exercice

Enoncé: Créer une liste qui contient uniquement les nombres pairs de 0 à 11.

Solution:

```
>>> even_numbers = [x for x in range(11) if x % 2 == 0]
>>> even_numbers
[0, 2, 4, 6, 8, 10]
```

## 3-8 Les tuples

Le tuple est une liste **immuable**: que l'on ne peut pas modifier. Une fois créée, on ne peut pas y ajouter de nouveaux éléments ni en modifier les existants.

Dans son fonctionnement, le tuple est similaire à la liste, sauf pour les fonctions de modification qui déclencheront des exceptions si on les utilisent.

```
# Exemple 1 : déclaration de tuples
tuple1 = (1, 2, 3, 4, 5)
tuple2 = ("Bonjour", "tout", "le", "monde")

# Exemple 2 : accès aux éléments d'un tuple
premier_element = tuple1[0] # premier_element vaut 1
dernier_element = tuple2[-1] # dernier_element vaut "monde"

# Exemple 3 : tentative de modification d'un tuple (erreur)
tuple1[2] = 10 # lève une erreur TypeError: 'tuple' object does not support
```

## 3-10 Les chaînes de caractères

Python gère les chaînes de caractères comme le propose les langages C/C++ ou Java mais propose aussi des chaînes multi-lignes.

Celles-ci sont totalement intégrées et très utiles notamment pour définir les chaînes d'aide.

Par contre, celle-ci sont implémentées comme un tuple de caractères: donc **immuables**.

Toutes les opérations valides sur les tuples sont valables sur les chaînes.

```
a='Tobby est un "chien"'
b="Tobby est un \"chien\""
c="Rex est un autre 'chien'"
d='Rex est un autre \'chien\' '
e="""Ceci est un essai de chaîne multilignes
pour vérifier: "Tobby" est un 'chien' """
f="""Ceci est un essai de chaîne multilignes
pour vérifier: "Tobby" est un 'chien' """
```

## 3-10 Les chaînes de caractères

Opération	Résultat
<code>str(x)</code>	convertit x en chaîne de caractères
<code>len(s)</code>	donne la longueur de la chaîne s
<code>s1+s2</code>	concaténation de s1 et s2
<code>s*n</code>	concatène n fois s
<code>s.lower()</code>	retourne s convertit en minuscule
<code>s.upper()</code>	retourne s convertit en majuscule
<code>s.capitalize()</code>	retourne s convertit en minuscule sauf le premier caractère en majuscule
<code>s.encode(encoding="utf-8", errors="strict")</code>	retourne une copie encodée de s
<code>';'.join(['a', 1, 5.5])</code>	retourne 'a;1;5.5'
<code>"a;b;c".split(';')</code>	retourne ['a', 'b', 'c']
<code>'spacious'.strip("\t\r\n")</code>	retourne 'spacious'
<code>'spacious'.lstrip("\t\r\n")</code>	retourne 'spacious '
<code>'spacious'.rstrip("\t\r\n")</code>	retourne ' spacious'
<code>s.find('toto')</code>	retourne l'index de la première occurrence de 'toto' dans s; -1 si non trouvé
<code>s.rfind('toto')</code>	retourne l'index de la première occurrence de 'toto' en partant de la fin dans s; -1 si non trouvé

# 3-11 Les sets

un set est un type de données qui permet de stocker une **collection non ordonnée d'éléments uniques et immuables**. Les éléments d'un set peuvent être de différents types de données, tels que des entiers, des flottants, des chaînes de caractères, des booléens, etc.

Les sets sont utiles pour vérifier rapidement si un élément fait partie de la collection ou pour éliminer les doublons.

```
# Exemple 1 : déclaration de sets
set1 = {1, 2, 3, 4, 5}
set2 = {"Bonjour", "tout", "le", "monde"}

# Exemple 2 : ajout d'éléments à un set
set1.add(6) # set1 contient maintenant {1, 2, 3, 4, 5, 6}

# Exemple 3 : suppression d'éléments d'un set
set2.remove("le") # set2 contient maintenant {"Bonjour", "tout", "monde"}

# Exemple 4 : opérations sur les sets
set3 = set1.union(set2) # set3 contient la réunion de set1 et set2
set4 = set1.intersection(set2) # set4 contient l'intersection de set1 et set2
```



## 3-11 Les sets: Python vs Java

---

En Java, un set est également un type de données qui permet de stocker une collection d'éléments uniques. Cependant, en Java, les sets sont ordonnés, ce qui signifie que les éléments sont stockés dans un ordre particulier et prévisible. Les sets en Java sont implémentés par les classes `java.util.HashSet` et `java.util.TreeSet`.

# 3-12 Les dictionnaires

En Python, un dictionnaire est un type de données qui permet de stocker une collection d'éléments clé-valeur.

Les clés sont **uniques et immuables**, et les valeurs peuvent être de différents types.

Les dictionnaires sont utiles pour stocker des données qui doivent être associées à des clés spécifiques.

Les dictionnaires sont mutables

```
# Exemple 1 : déclaration d'un dictionnaire
dict1 = {"nom": "Jean", "age": 30, "ville": "Paris"}

# Exemple 2 : accès aux éléments d'un dictionnaire
nom = dict1["nom"]      # nom contient "Jean"
age = dict1.get("age")  # age contient 30

# Exemple 3 : modification d'un élément d'un dictionnaire
dict1["age"] = 35        # dict1 contient maintenant {"nom": "Jean", "age": 35, "ville": "Paris"}

# Exemple 4 : suppression d'un élément d'un dictionnaire
del dict1["ville"]       # dict1 contient maintenant {"nom": "Jean", "age": 35}
```



## 3-12 Les dictionnaires

Opérations	Résultat
<code>dict( one=1, two=2, three=3 )</code>	<code>{'one': 1, 'three': 3, 'two': 2}</code>
<code>dict(zip(['one', 'two', 'three'], [1, 2, 3]))</code>	<code>{'one': 1, 'three': 3, 'two': 2}</code>
<code>dict([('two', 2), ('one', 1), ('three', 3)])</code>	<code>{'one': 1, 'three': 3, 'two': 2}</code>
<code>len(d)</code>	nombre de couples dans le dictionnaire
<code>d[key]</code>	retourne la valeur associée à la clé ou une exception <code>KeyError</code>
<code>d[key] = value</code>	associe value à la clé key
<code>del d[key]</code>	Supprime <code>d[key]</code> de <code>d</code> , ou une exception
<code>key in d</code>	retourne <code>True</code> si key est dans <code>d</code>
<code>key not in d</code>	retourne <code>True</code> si key n'est pas dans <code>d</code>
<code>d.clear()</code>	vide le dictionnaire
<code>d.get( key, default)</code>	si key est dans <code>d</code> retourne la valeur associé sinon retourne default
<code>d.items()</code>	retourne un itérateur sur les couples
<code>d.keys()</code>	retourne un itérateur sur les clés
<code>d.values()</code>	retourne un itérateur sur les valeurs
<code>d.update(other_dir)</code>	ajoute les couples d' <code>other_dir</code> dans <code>d</code> et écrase en cas de conflits



## 3-12 Les dictionnaires: Python vs Java

---

En Java, un dictionnaire est également un type de données qui permet de stocker des paires clé-valeur, appelé Map. Les Map en Java sont ordonnées et peuvent être mutables ou immuables. Les clés et les valeurs d'une Map peuvent être de n'importe quel type de données, à condition qu'ils respectent les contraintes de mutabilité et d'unicité des clés.







## 3-12 Les dict comprehension

En Python, un "dict comprehension" est un moyen concis et efficace de créer un dictionnaire en Python. Il fonctionne de manière similaire à une "list comprehension", mais au lieu de créer une liste, il crée un dictionnaire.

Voici la syntaxe générale pour créer un "dict comprehension" en Python:

```
{clé: valeur pour élément in iterable}
```

La partie "clé: valeur" définit chaque paire clé-valeur dans le dictionnaire que vous créez. La partie "pour élément in iterable" définit la façon dont vous créez ces paires en itérant à travers un iterable (comme une liste, un tuple, un set ou un autre dictionnaire) et en appliquant une expression à chaque élément de l'itérable pour définir la clé et la valeur correspondantes.





# 3-12 Les dict comprehension: Exercice

Enoncé:

Etant donné la liste de nom suivante:

```
noms = [ « Alice », « Bob », « Charlie », « David »]
```

Générez un dictionnaire avec en clé le nom et en valeur la longueur du nom.





# 3-12 Les dict comprehension: Exercice

Enoncé:

Etant donné la liste de nom suivante:

```
noms = [ « Alice », « Bob », « Charlie », « David »]
```

Générez un dictionnaire avec en clé le nom et en valeur la longueur du nom.

```
noms = ['Alice', 'Bob', 'Charlie', 'David']  
longueurs = {nom: len(nom) for nom in noms}  
print(longueurs)  
# Output: {'Alice': 5, 'Bob': 3, 'Charlie': 7, 'David': 5}
```



## 3-13 Python vs Java

Type de données	Python	Java
Entiers	int	int
Longs (entiers longs)	int (pas de type long explicite)	long
Flottants (réels)	float	float
Doubles (réels)	float (pas de type double explicite)	double
Booléens	bool	boolean
Chaînes de caractères	str	String
Caractères	str (représenté comme une chaîne de longueur 1)	char
Tableaux	list	Array
Tuples	tuple	Pas d'équivalent
Dictionnaires	dict	Map
Ensembles	set	Set



## 3-14 D'autres types

---


Le module **datetime** : permet la manipulation de dates et d'heures. il permet de représenter des dates, des heures, des intervalles de temps, des différences de temps, etc.

Datetime propose deux types d'objets principaux :

**datetime** : qui représente une date et une heure précise (année, mois, jour, heure, minute, seconde, microseconde).

**timedelta** : qui représente une durée entre deux dates ou heures.

Le module '**collections**' fournit nombre de conteneurs utiles:

- namedtuple
  - deque
  - ChainMap
  - Etc....
- 
- 



## 3-14 D'autres types

---

Le module '**collections**' fournit nombre de conteneurs utiles:

- namedtuple
- deque
- ChainMap
- Etc....

Le module '**decimal**' permet de travailler avec des flottants avec un nombre de décimales fixées. Il est utile pour travailler en finance...







# 4-1 Structures de Contrôles: if/elif/else

---

Les instructions conditionnelles "if", "elif" et "else" en Python sont utilisées pour exécuter différents blocs de code en fonction d'une ou plusieurs conditions.  
Voici la syntaxe générale pour une instruction conditionnelle en Python :

```
if condition1:
    # instructions si la condition1 est vraie
elif condition2:
    # instructions si la condition2 est vraie
else:
    # instructions si aucune des conditions précédentes n'est vraie
```



# 4-1 Structures de Contrôles: if/elif/else



il est possible d'écrire une instruction if-else sur une seule ligne pour définir la valeur d'une variable en fonction d'une condition. C'est ce qu'on appelle une expression conditionnelle.

Voici un exemple :

```
age = 25  
est_majeur = True if age >= 18 else False
```

L'équivalent en java:

```
int age = 25;  
boolean estMajeur = age >= 18 ? true : false;
```



## 4-2 Structures de Contrôles: for

```
for i in range(0, 10, 2):
    print(i)

a = [1, 3, 5, 7]
for i in a:
    print(i)

d = {"one": 1, "two": 2, "three": 3}
for k in d.keys():
    print("d[" + k + "] = ", d[k])
for k, v in d.items():
    print("d[" + k + "] = ", v)

for c in "abcdefghi":
    print(c)

for i in range(0, 10, 2):
    if i > 5:
        break
    print(i)
```

on itère une séquence

l'action est dans un nouveau bloc (indentation →)

ne pas oublier le ':'

On peut quitter la boucle en utilisant l'instruction **break** ou sauter une itération en utilisant **continue**

## 4-2 Structures de Contrôles: for

Comment parcourir une séquence?

```
a = [ 1, 3, 7., "toto", 4. ]  
for i in range( len( a ) ):  
    print( i, '>', a[ i ] )
```

```
a = [ 1, 3, 7., "toto", 4. ]  
i = 0  
for v in a:  
    print( i, '>', v )  
    i += 1
```

Comment faire pour avoir à la fois l'index et la valeur?

```
a = [ 1, 3, 7., "toto", 4. ]  
for i, v in enumerate( a ):  
    print( i, '>', v )
```

## 4-3 Structures de Contrôles: while

```
a = [1, 3, 5, 7, 11, 13]
i = 0
while i < len(a):
    print(i, ">", a[i])
    i += 1
print()
```

```
a = [i * i for i in range(100)]
i = 0
while i < len(a):
    if a[i] > 20:
        break
    print(i, ">", a[i])
    i += 1
```

```
a = [i * i for i in range(100)]
i = 0
while i < len(a):
    if a[i] > 20:
        continue # attention à la boucle infinie
    print(i, ">", a[i])
    i += 1
```

## 4-3 Structures de Contrôles: match

L'opérateur **match** est une nouvelle fonctionnalité introduite dans Python 3.10 qui permet d'écrire des expressions conditionnelles plus expressives et plus concises. L'opérateur **match** est similaire à l'opérateur **switch** que l'on trouve dans d'autres langages de programmation.

```
def test(x):  
    match x:  
        case 0:  
            print("La valeur de x est zéro")  
        case 1:  
            print("La valeur de x est un")  
        case _:  
            print("La valeur de x est autre chose")  
  
test(0) # affiche "La valeur de x est zéro"  
test(1) # affiche "La valeur de x est un"  
test(42) # affiche "La valeur de x est autre chose"
```

# 5-1 Les fonctions: Syntaxe

Voici la syntaxe d'une fonction python:

- Le mot clé **def** pour déclarer la fonctions
- Nom de la fonction qui doit suivre la norme...
- Les paramètres
- Le **return** pour retourner un output.

Important: Une fonction peut ne pas avoir de **return**.

```
def nom_de_la_fonction(paramètres):  
    """Docstring (optionnel)"""  
    # Instructions de la fonction  
    return valeur_de_retour
```



# 5-1 Les fonctions: Paramètres

---

Il existe plusieurs types de paramètres que vous pouvez utiliser en Python :

**Paramètres positionnels** : les paramètres positionnels sont les paramètres les plus courants en Python. Ils sont définis dans l'ordre dans lequel ils sont passés à la fonction.

**Paramètres nommés** : les paramètres nommés permettent de spécifier les paramètres dans n'importe quel ordre, en les nommant explicitement. Les paramètres nommés sont définis avec une valeur par défaut.

```
def ma_fonction(param1, param2="valeur_par_defaut"):
    print(f"Le premier paramètre est {param1}")
    print(f"Le deuxième paramètre est {param2}")
```





# 5-1 Les fonctions: args, kwargs

---

En Python, **args** et **kwargs** sont des paramètres spéciaux que vous pouvez utiliser dans une fonction pour accepter un nombre variable d'arguments.

**args** est un paramètre spécial qui permet de passer un nombre variable d'arguments non nommés à une fonction sous la forme d'un tuple.

Vous pouvez utiliser **\*args** pour définir une fonction qui peut accepter un nombre variable d'arguments positionnels. Voici un exemple :

```
def ma_fonction(*args):  
    for arg in args:  
        print(arg)  
  
ma_fonction('a', 'b', 'c')
```



# 5-1 Les fonctions: args, kwargs

---

**kwargs** est un autre paramètre spécial qui permet de passer un nombre variable d'arguments nommés à une fonction sous la forme d'un dictionnaire.

Vous pouvez utiliser **\*\*kwargs** pour définir une fonction qui peut accepter un nombre variable d'arguments nommés. Voici un exemple :

```
def ma_fonction(**kwargs):  
    for cle, valeur in kwargs.items():  
        print(f'{cle}: {valeur}')
```

```
ma_fonction(a=1, b=2, c=3)
```





# 5-1 Les fonctions: args, kwargs

---

A noter:

Il est possible d'utiliser `*args` et `**kwargs` simultanément pour définir une fonction qui peut accepter un nombre variable d'arguments positionnels et nommés.

`args` et `kwargs` ne sont que des conventions de nommage et que vous pouvez utiliser d'autres noms pour ces paramètres. Cependant, l'utilisation de ces noms couramment acceptés facilite la compréhension et la lisibilité de votre code par d'autres programmeurs Python.







# 5-1 Les fonctions: args, kwargs: Exercice

---

## Enoncé:

comment réaliser une fonction `g` avec args, kwargs pour afficher "je pratique Python aujourd'hui." par l'appel `g("je", "partique", cours='Python', temps='aujourd'hui')`







# 5-1 Les fonctions: args, kwargs: Solution

---

```
def g(*args, **kwargs):  
    result = ''  
    for c in args:  
        result = result + ' ' + c  
  
    for k in kwargs.values():  
        result = result + ' ' + k  
  
    print(result[1:])
```

On peut améliorer en utilisant le **join**:

```
def g(*args, **kwargs):  
    print(' '.join(args) + ' ' + ' '.join(kwargs.values()))
```





# 5-1 Les fonctions: local, nonlocal, global


Python, les termes "**global**" et "**local**" font référence à la portée d'une variable, c'est-à-dire à l'endroit où cette variable peut être utilisée ou modifiée dans le code.

Une variable locale est définie à l'intérieur d'une fonction et n'existe que dans le cadre de cette fonction. Elle n'est pas accessible en dehors de cette fonction. Par exemple :

```
def ma_fonction():  
    x = 5 # variable locale  
    print(x)  
  
ma_fonction() # affiche 5  
print(x) # erreur : la variable x n'est pas définie dans ce contexte
```

En revanche, une variable globale est définie en dehors de toute fonction et est accessible à l'ensemble du programme. Elle peut être modifiée à l'intérieur d'une fonction, mais il est nécessaire d'indiquer explicitement que l'on souhaite modifier la variable globale, à l'aide du mot-clé "global". Par exemple :

```
x = 5 # variable globale  
  
def ma_fonction():  
    global x  
    x = 10 # modification de la variable globale
```





# 5-1 Les fonctions: local, nonlocal, global

En Python, il existe trois niveaux de portée pour les variables : **local**, **nonlocal** et **global**.

Une variable **locale** est définie à l'intérieur d'une fonction et n'est accessible qu'à l'intérieur de cette fonction. Elle n'existe pas en dehors de la fonction.

Une variable **non locale** est définie à l'intérieur d'une fonction imbriquée (fonction à l'intérieur d'une autre fonction) et est accessible dans la fonction interne ainsi que dans la fonction externe qui la contient. Elle n'est pas accessible en dehors de ces deux fonctions.

Une variable **globale** est définie en dehors de toutes les fonctions et est accessible à toutes les fonctions du programme.

```
x = "global" # variable globale

def test_nonlocal():
    x = "nonlocal" # variable non locale

    def fonction_interne():
        nonlocal x
        x = x + " appelée en interne"

    fonction_interne()
    print(x)

def test_global():
    global x
    print(x)

test_nonlocal()
test_global()
```



## 5-2 Modules et packages

---



En Python, un **module** est simplement un fichier Python contenant des définitions de fonctions, de classes et d'autres objets. Les modules sont utilisés pour organiser le code en différents fichiers pour faciliter la lecture, la maintenance et la réutilisation du code.

Pour importer un module dans votre code Python, vous pouvez utiliser l'instruction "import" suivie du nom du module. Par exemple, si vous avez un fichier Python appelé "mon\_module.py" contenant des fonctions utiles, vous pouvez l'importer dans un autre fichier Python en utilisant l'instruction "import" :

```
import mon_module  
mon_module.ma_fonction()
```

Les **packages**, quant à eux, sont simplement des collections de modules. Les packages sont utilisés pour organiser les modules en hiérarchies et pour éviter les conflits de noms de modules.

Pour créer un package, vous devez créer un dossier contenant un fichier « **\_\_init\_\_.py** ». Le fichier « **\_\_init\_\_.py** » est utilisé pour indiquer à Python que le dossier contient un package et pour spécifier les modules qui se trouvent dans le package.





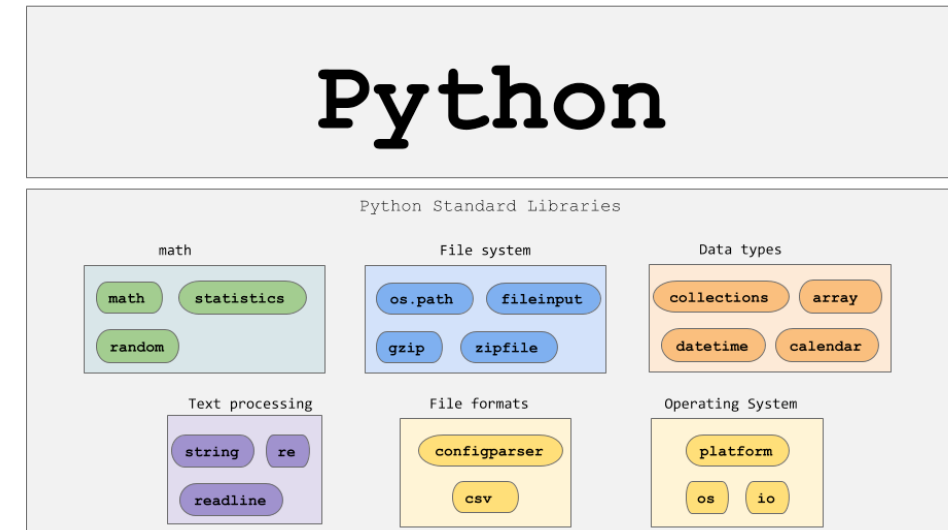
## 5-2 Modules et packages

Les deux fonctions suivantes permettent d'obtenir des informations sur les modules:

La fonction **help()** est une fonction intégrée en Python qui permet d'obtenir une aide sur les fonctions, les méthodes, les classes et les modules.

Lorsque vous appelez **help()** avec un objet en tant qu'argument, elle affiche une documentation détaillée sur l'objet. Cette documentation est généralement incluse dans la documentation de Python.

La fonction **dir()** est également une fonction intégrée en Python qui permet de lister les attributs et les méthodes d'un objet. Elle peut être utilisée avec des modules, des classes, des instances de classe et des objets intégrés.





## 5-2 Modules et packages





On peut distinguer deux grandes catégories de packages :

**1. Les packages standard** : Ce sont les packages qui sont inclus dans la distribution Python officielle et qui sont disponibles sur toutes les plateformes. Ils sont généralement utilisés pour des tâches de base telles que la manipulation de fichiers, la gestion de la mémoire, la gestion de processus, etc. Exemples de packages standard : os, sys, re, datetime, math, etc.

**2. Les packages tiers** : Ce sont les packages qui ne sont pas inclus dans la distribution Python officielle, mais qui peuvent être installés à partir de sources externes, telles que PyPI (Python Package Index) ou des référentiels Git. Les packages tiers sont généralement utilisés pour des tâches plus spécifiques, telles que la manipulation de données, l'analyse de données, la visualisation de données, etc. Exemples de packages tiers : NumPy, Pandas, Matplotlib, TensorFlow, Django, Flask, etc.

Il est important de noter que les packages standard sont installés automatiquement lors de l'installation de Python, tandis que les packages tiers doivent être installés manuellement en utilisant des outils tels que pip, conda, ou en installant directement à partir du code source.



## 5-3 Les fonctions lambda

Les fonctions lambda sont des fonctions anonymes en Python qui peuvent être utilisées pour créer des fonctions simples sans avoir à les définir explicitement,

```
x = lambda x: x*x  
print(x(3))
```

```
f = lambda a,b: a if (a > b) else b  
print(f(5, 3))
```

Les fonctions lambda peuvent souvent être plus rapides et plus efficaces que les fonctions régulières, car elles ne nécessitent pas la définition d'une fonction séparée et peuvent être utilisées directement à l'endroit où elles sont nécessaires.





## 5-4 Les générateurs

---

En Python, les générateurs sont des objets qui permettent de générer une séquence de valeurs d'une manière efficace en termes de mémoire et de temps d'exécution. Contrairement aux fonctions normales, qui retournent une valeur et terminent leur exécution, les générateurs suspendent leur état d'exécution après avoir produit une valeur et attendent d'être appelés à nouveau avant de poursuivre.

Les générateurs sont définis à l'aide de la syntaxe **yield**. Lorsqu'un générateur rencontre une instruction `yield`, il renvoie la valeur spécifiée et se suspend jusqu'à ce qu'il soit appelé à nouveau. Lorsqu'il est appelé à nouveau, il reprend l'exécution à partir de l'endroit où il s'était arrêté et poursuit jusqu'à la prochaine instruction `yield`.



## 5-4 Les générateurs

Voici un exemple simple de générateur qui produit une séquence d'entiers pairs :

```
def generate_even_numbers(maximum):  
    n = 0  
    while n < maximum:  
        yield n  
        n += 2
```

## 5-4 Les générateurs

En Python, **yield from** est une fonctionnalité introduite dans la version 3.3, qui permet de déléguer la génération d'une séquence à un autre générateur.

Concrètement, **yield from** peut être utilisé pour simplifier le code qui utilise des générateurs imbriqués. Au lieu d'écrire du code pour gérer les itérations sur chaque générateur imbriqué, **yield from** permet de déléguer cette tâche à un autre générateur.

```
def generator_one():  
    yield 1  
    yield 2  
    yield 3  
  
def generator_two():  
    yield from generator_one()  
    yield 4  
    yield 5  
    yield 6  
  
for i in generator_two():  
    print(i)
```

# 6- Classes et objets

Pour définir une classe en Python, on utilise le mot-clé **class**, suivi du nom de la classe et des deux points.

Voici un exemple de définition d'une classe simple.

Le constructeur `__init__` est une méthode spéciale qui est appelée lors de la création d'une nouvelle instance de la classe.

```
class NomDeLaClasse:
    """Docstring (optionnel)"""

    def __init__(self, paramètres):
        """Constructeur """
        self.attribut1 = valeur1
        self.attribut2 = valeur2

    def methode1(self, paramètres):
        # Instructions de la méthode

    def methode2(self, paramètres):
        # Instructions de la méthode
```

# 6- Classes et objets: Déclaration en Java vs Python

Java

```
1 public class Car {
2     private String color;
3     private String model;
4     private int year;
5
6     public Car(String color, String model, int year) {
7         this.color = color;
8         this.model = model;
9         this.year = year;
10    }
11
12    public String getColor() {
13        return color;
14    }
15
16    public String getModel() {
17        return model;
18    }
19
20    public int getYear() {
21        return year;
22    }
23 }
```

Python

```
1 class Car:
2     def __init__(self, color, model, year):
3         self.color = color
4         self.model = model
5         self.year = year
```





# 6- Classes et objets: méthodes

## 1- Les méthodes d'instance :

Elles sont associées à une instance de la classe et ont accès aux attributs de cette instance via le mot clé **self**. Les méthodes d'instance sont appelées avec l'instance comme premier argument, et elles peuvent être utilisées pour modifier les attributs de l'instance ou pour effectuer des opérations qui dépendent de l'état de l'instance.

## 2- Les méthodes de classe :

Les méthodes de classe sont associées à la classe elle-même plutôt qu'à une instance spécifique. Elles sont décorées avec le décorateur **@classmethod** et ont accès à la classe elle-même via le mot clé **cls**. Les méthodes de classe sont appelées avec la classe comme premier argument, et elles sont souvent utilisées pour effectuer des opérations qui concernent la classe plutôt que des instances individuelles,

## 3- Les méthodes statiques :

Les méthodes statiques sont similaires aux méthodes de classe, mais elles n'ont pas accès à la classe ou à l'instance de la classe. Elles sont décorées avec le décorateur **@staticmethod** et sont souvent utilisées pour effectuer des opérations qui sont liées à la classe mais qui n'ont pas besoin de modifier l'état de la classe ou de l'instance.


```
class NomDeLaClasse:
```

```
    def __init__(self, paramètres):  
        self.attribut1 = valeur1
```

```
    def instance_method(self, paramètres):  
        # Instructions de la méthode
```

```
    @classmethod  
    def class_method(cls, paramètres):  
        # Instructions de la méthode
```

```
    @staticmethod  
    def static_method(paramètres):  
        # Instructions de la méthode
```





# 6- Classes et objets: dunder methods



## 1- Les méthodes d'instance :

Elles sont associées à une instance de la classe et ont accès aux attributs de cette instance via le mot clé **self**. Les méthodes d'instance sont appelées avec l'instance comme premier argument, et elles peuvent être utilisées pour modifier les attributs de l'instance ou pour effectuer des opérations qui dépendent de l'état de l'instance.

## 2- Les méthodes de classe :

Les méthodes de classe sont associées à la classe elle-même plutôt qu'à une instance spécifique. Elles sont décorées avec le décorateur **@classmethod** et ont accès à la classe elle-même via le mot clé **cls**. Les méthodes de classe sont appelées avec la classe comme premier argument, et elles sont souvent utilisées pour effectuer des opérations qui concernent la classe plutôt que des instances individuelles,

## 3- Les méthodes statiques :

Les méthodes statiques sont similaires aux méthodes de classe, mais elles n'ont pas accès à la classe ou à l'instance de la classe. Elles sont décorées avec le décorateur **@staticmethod** et sont souvent utilisées pour effectuer des opérations qui sont liées à la classe mais qui n'ont pas besoin de modifier l'état de la classe ou de l'instance.


```
class NomDeLaClasse:
```

```
    def __init__(self, paramètres):  
        self.attribut1 = valeur1
```

```
    def instance_method(self, paramètres):  
        # Instructions de la méthode
```

```
    @classmethod  
    def class_method(cls, paramètres):  
        # Instructions de la méthode
```

```
    @staticmethod  
    def static_method(paramètres):  
        # Instructions de la méthode
```





# 6- Classes et objets: attributs



## 1- Les attributs d'instance :

Les attributs d'instance sont des attributs qui sont spécifiques à une instance particulière d'une classe. Ils sont définis dans le constructeur (`__init__`) en utilisant le mot clé `self`, et peuvent être accédés à partir de cette instance

## 2- Les attributs de classe :

Les attributs de classe sont des attributs qui sont partagés par toutes les instances d'une classe. Ils sont définis directement sur la classe plutôt que sur une instance spécifique, et peuvent être accédés à partir de la classe ou de n'importe quelle instance de la classe. Les attributs de classe sont souvent utilisés pour stocker des informations qui concernent la classe dans son ensemble, plutôt que des instances individuelles.

En Python, il n'existe pas d'attribut privé ou protégé.

Il existe cependant une norme d'écriture pour définir les attributs protégé/privé

Pour contrôler l'accès et la modification des attributs, vous devez utiliser les **propriétés**.

```
class NomDeLaClasse:
```

```
    class_attribute = « .. »
```

```
    def __init__(self, value):
```

```
        self.instance_attribute = value
```

```
        self._protected_attribute = ...
```

```
        self.__private_attribute = ...
```



# 6- Classes et objets: propriétés

Les propriétés en Python sont des méthodes spéciales qui permettent de définir des attributs de classe qui se comportent comme des attributs d'instance, mais qui sont en réalité calculés dynamiquement en utilisant des méthodes. Les propriétés peuvent être utilisées pour encapsuler l'accès aux attributs d'instance et pour exécuter des opérations supplémentaires lors de la récupération ou de la définition de la valeur d'un attribut.

Pour définir une propriété dans une classe, vous devez utiliser le décorateur **@property** devant une méthode. Cette méthode est appelée lorsque vous accédez à la propriété.

Voici un exemple de définition de propriété :

```
class Rectangle:
    def __init__(self, length, width):
        self._length = length
        self._width = width

    @property
    def area(self):
        return self._length * self._width

r = Rectangle(5, 2)

print(r.area)
10
```

# 6- Classes et objets: propriétés

En plus de la méthode **@property**, il existe deux autres méthodes spéciales que vous pouvez utiliser pour définir une propriété :

**@property.setter** : Cette méthode est appelée lorsque vous affectez une valeur à une propriété. Elle permet de valider ou de modifier la valeur avant de l'assigner à l'attribut d'instance correspondant.

**@property.deleter** : Cette méthode est appelée lorsque vous supprimez une propriété à l'aide de l'instruction **del**. Elle permet de nettoyer ou de supprimer l'attribut d'instance correspondant.

Voici un exemple de définition de propriétés avec des méthodes **getter**, **setter** et **deleter** :

```
class Rectangle:
    def __init__(self, length, width):
        self._length = length
        self._width = width

    @property
    def area(self):
        return self._length * self._width

    @property
    def length(self):
        return self._length

    @length.setter
    def length(self, value):
        if value < 0:
            raise ValueError("Length must be positive")
        self._length = value

    @length.deleter
    def length(self):
        del self._length
```

# 6- Classes et objets: Héritage en Java vs Python

```
Java

public class Vehicle {

    private String color;
    private String model;

    public Vehicle(String color, String model) {
        this.color = color;
        this.model = model;
    }

    public String getColor() {
        return color;
    }

    public String getModel() {
        return model;
    }
}

public interface Device {
    int getVoltage();
}

public class Car extends Vehicle implements Device {

    private int voltage;
    private int year;

    public Car(String color, String model, int year) {
        super(color, model);
        this.year = year;
        this.voltage = 12;
    }

    @Override
    public int getVoltage() {
        return voltage;
    }

    public int getYear() {
        return year;
    }
}
```

```
Python

class Vehicle:
    def __init__(self, color, model):
        self.color = color
        self.model = model

class Device:
    def __init__(self):
        self._voltage = 12

class Car(Vehicle, Device):
    def __init__(self, color, model, year):
        Vehicle.__init__(self, color, model)
        Device.__init__(self)
        self.year = year

    @property
    def voltage(self):
        return self._voltage

    @voltage.setter
    def voltage(self, volts):
        print("Warning: this can cause problems!")
        self._voltage = volts

    @voltage.deleter
    def voltage(self):
        print("Warning: the radio will stop working!")
        del self._voltage
```

# 6- Classes et objets: duck typing

Le **duck typing** est une technique de programmation qui consiste à déterminer le type d'un objet en examinant les méthodes et les attributs qu'il expose plutôt qu'en se basant sur son type réel. En d'autres termes, si un objet marche comme un canard et fait "coin coin" comme un canard, alors il est considéré comme un canard, peu importe son type réel.

En Python, le "duck typing" est souvent utilisé pour implémenter le **polymorphisme**. Plutôt que de vérifier si un objet est d'un certain type pour décider quelle méthode appeler, on vérifie simplement si cet objet a une méthode spécifique, et si c'est le cas, on l'appelle. Par exemple, considérons le code suivant :

Java

```
public class Rhino {  
  
    public class Main{  
        public static void charge(Device device) {  
            device.getVoltage();  
        }  
  
        public static void main(String[] args) throws Exception {  
            Car car = new Car("yellow", "beetle", 1969);  
            Rhino rhino = new Rhino();  
            charge(car);  
            charge(rhino);  
        }  
    }  
}
```

Main.java

Error:(43, 11) java: incompatible types: Rhino cannot be converted to Device

Python

```
>>> def charge(device):  
...     if hasattr(device, '_voltage'):  
...         print(f"Charging a {device._voltage} volt device")  
...     else:  
...         print(f"I can't charge a {device.__class__.__name__}")  
...  
>>> class Phone(Device):  
...     pass  
...  
>>> class Rhino:  
...     pass  
...  
>>> my_car = Car("yellow", "Beetle", "1966")  
>>> my_phone = Phone()  
>>> my_rhino = Rhino()  
  
>>> charge(my_car)  
Charging a 12 volt device  
>>> charge(my_phone)  
Charging a 12 volt device  
>>> charge(my_rhino)  
I can't charge a Rhino
```

# 6- Classes et objets: Python vs Java

	Python	Java
Syntaxe pour définir une classe	<code>class NomDeLaClasse:</code>	<code>public class NomDeLaClasse {</code>
Héritage	<code>class NomDeLaClasse(Parent):</code> Héritage multiple autorisé	<code>public class NomDeLaClasse extends Parent {</code> <b>Héritage multiple impossible</b>
Méthode constructeur	<code>def __init__(self, paramètres):</code>	<code>public NomDeLaClasse(paramètres) {</code>
Encapsulation	Pas de mot-clé pour spécifier la visibilité	<code>public, private, protected</code>
Référencer l'instance	<code>self</code>	<code>this</code>
Conversion en str	<code>def __str__(self)</code>	<code>public String toString() {</code>
Accesseurs et Mutateurs	Pas de syntaxe particulière, mais les attributs peuvent être accédés et modifiés directement	Les méthodes <code>get</code> et <code>set</code> sont utilisées pour accéder et modifier les attributs
Méthodes de classe	<code>@classmethod</code>	<code>static</code>
Méthodes d'instance	<code>def methode(self, paramètres):</code>	<code>public void methode(paramètres) {</code>
Polymorphisme	Polymorphisme de méthode (overloading et overriding)	Polymorphisme de méthode (overloading et overriding)
Interfaces	Pas de syntaxe particulière, mais une classe peut implémenter une interface	<code>public class NomDeLaClasse implements Interface {</code>





# 6- Classes et objets: Exercices

---

Enoncé:

1- Créez une classe qui représente un cercle.

Il doit avoir un rayon avec une valeur à 1 par défaut. Il devra également avoir un bel affichage quand on le print,

2- Ajouter une propriété pour pouvoir récupérer la surface. On ne devrait pas pouvoir la changer



# 6- Classes et objets: Solution

Question 1:

```
import math

class Circle:

    """Circle with radius, area, and diameter."""

    def __init__(self, radius=1):
        self.radius = radius

    def __repr__(self):
        return f'Circle({self.radius})'
```

# 6- Classes et objets: Solution

## Question 2:

```
class Circle:

    """Circle with radius, area, and diameter."""

    def __init__(self, radius=1):
        self.radius = radius

    def __repr__(self):
        return f'Circle({self.radius})'

    @property
    def area(self):
        return math.pi * self.radius ** 2
```

# 7- Décorateurs

Un décorateur en Python est une fonction qui prend une autre fonction en argument, ajoute une fonctionnalité à cette fonction, puis retourne la fonction modifiée. Les décorateurs sont couramment utilisés pour modifier ou étendre le comportement des fonctions existantes sans avoir à les modifier directement. Ils sont souvent utilisés pour ajouter des fonctionnalités de journalisation, de temporisation, de mémoire cache ou de contrôle d'accès à une fonction existante.

```
def decorateur(fonction):  
    def wrapper():  
        print("Avant l'appel de la fonction")  
        fonction()  
        print("Après l'appel de la fonction")  
    return wrapper  
  
@decorateur  
def ma_fonction():  
    print("Ma fonction est appelée")
```



# 7- Décorateurs: Exemples

---



**@staticmethod** : Ce décorateur est utilisé pour marquer une méthode de classe comme une méthode statique, ce qui signifie qu'elle peut être appelée directement à partir de la classe sans avoir besoin d'une instance de la classe.

**@classmethod** : Ce décorateur est utilisé pour marquer une méthode de classe comme une méthode de classe, ce qui signifie qu'elle prend la classe elle-même en tant que premier argument, plutôt qu'une instance de la classe.

**@property** : Ce décorateur est utilisé pour créer une propriété à partir d'une méthode, ce qui permet d'accéder et de modifier une variable de classe à travers une syntaxe d'attribut, plutôt qu'une syntaxe de méthode.

**@abstractmethod** : Ce décorateur est utilisé pour créer une méthode abstraite, qui doit être implémentée dans les classes dérivées.

**@functools.wraps** : Ce décorateur est utilisé pour copier les métadonnées d'une fonction décorée à la fonction de décoration, ce qui permet de conserver les noms, la documentation, etc.



## 7- Décorateurs: Exercice

Implémentez une fonction décorateur cache qui prend en entrée une fonction et retourne une nouvelle fonction qui utilise une mémoire cache pour stocker les résultats précédents. La nouvelle fonction doit d'abord vérifier si le résultat de la fonction pour les mêmes paramètres a déjà été calculé et stocké dans la mémoire cache. Si c'est le cas, elle doit retourner ce résultat. Sinon, elle doit appeler la fonction originale et stocker le résultat dans la mémoire cache avant de le renvoyer.

```
def fibonacci(n):  
    if n < 2:  
        return n  
    else:  
        return fibonacci(n-1) + fibonacci(n-2)
```

# 7- Décorateurs: Solution

```
def cache(func):  
    cached_results = {}  
  
    def new_func(*args):  
        if args in cached_results:  
            return cached_results[args]  
        else:  
            result = func(*args)  
            cached_results[args] = result  
            return result  
  
    return new_func  
  
@cache  
def fibonacci(n):  
    if n < 2:  
        return n  
    else:  
        return fibonacci(n-1) + fibonacci(n-2)
```

## 8- Fichiers & exceptions

Python permet de lire et d'écrire des fichiers en utilisant les fonctions et les objets de la bibliothèque standard de Python. Les fichiers peuvent être ouverts en mode lecture, écriture ou ajout. Le mode par défaut est la lecture ('r')

```
# Ouvrir un fichier en mode lecture
fichier = open('monfichier.txt', 'r')

# Lire le contenu du fichier
contenu = fichier.read()

# Fermer le fichier
fichier.close()
```





# 8- Fichiers & exceptions

---

Le module **OS** expose un grand nombre de fonctions qui peuvent être utilisées pour effectuer des tâches liées au système d'exploitation, telles que la manipulation de fichiers et de répertoires, l'exécution de commandes système, la gestion des environnements de processus, et plus encore.

Voici quelques exemples d'utilisation de fonctions du module OS :

La fonction **os.getcwd()** renvoie le répertoire de travail actuel.

La fonction **os.mkdir()** permet de créer un répertoire.

La fonction **os.listdir()** renvoie la liste des fichiers et répertoires dans un répertoire.

La fonction **os.remove()** permet de supprimer un fichier.



# 8- Fichiers & exceptions

**Scandir**, quant à lui, est une fonction ajoutée dans Python 3.5 pour faciliter la manipulation des fichiers et des répertoires. Elle permet de récupérer des informations sur les fichiers et les répertoires, comme leur nom, leur taille, leur type, leur date de modification, etc.

**Scandir** est plus performant que **os.listdir()** car il ne génère pas immédiatement la liste complète des fichiers et répertoires dans un répertoire, mais renvoie un itérateur de type **os.DirEntry** qui permet de parcourir les fichiers et répertoires un par un. Il utilise également moins de mémoire que **os.listdir()** pour les répertoires contenant un grand nombre de fichiers.

Voici un exemple d'utilisation de scandir :

```
import os

with os.scandir('/chemin/vers/mon/repertoire/') as it:
    for entry in it:
        if entry.is_file():
            print(entry.name)
```

# 8- Fichiers & exceptions

Python permet de lire et d'écrire des fichiers en utilisant les fonctions et les objets de la bibliothèque standard de Python. Les fichiers peuvent être ouverts en mode lecture, écriture ou ajout. Le mode par défaut est la lecture ('r')

```
# Ouvrir un fichier en mode lecture
fichier = open('monfichier.txt', 'r')

# Lire le contenu du fichier
contenu = fichier.read()

# Fermer le fichier
fichier.close()
```

Que se passe-t-il si le fichier n'existe pas ou si on oublie de fermer le fichier ?

# 8- Exceptions

Il existe de nombreux types d'exceptions prédéfinies en Python, qui peuvent être levées pour différentes raisons. Voici quelques exemples d'exceptions courantes :

**ZeroDivisionError** : levée lorsque vous essayez de diviser par zéro.

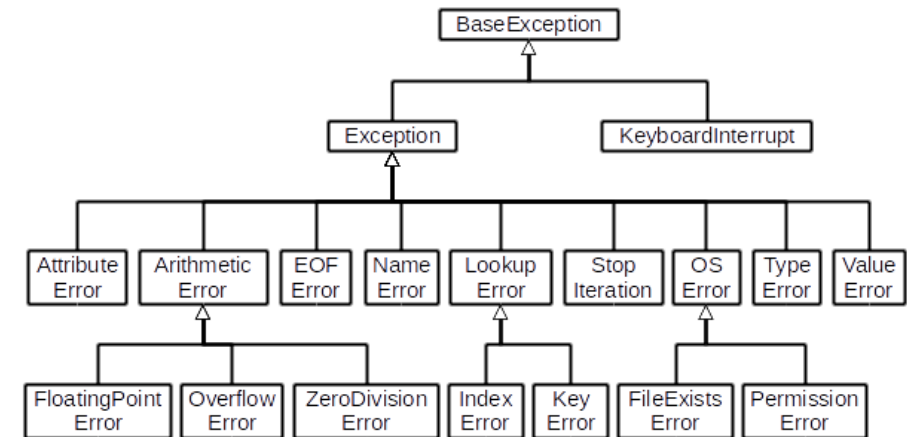
**NameError** : levée lorsque vous essayez d'accéder à une variable qui n'a pas été définie.

**TypeError** : levé lorsque vous essayez d'effectuer une opération sur des objets de types incompatibles.

**ValueError** : levée lorsque vous essayez d'utiliser une valeur inappropriée pour un argument ou un paramètre.

**FileNotFoundError**: levée lorsque vous essayez d'accéder à un fichier non existant

Il est également possible de définir vos propres exceptions personnalisées en Python, en créant une nouvelle classe qui hérite de la classe d'exception de base (**Exception**).





## 8- Context manager



---

Les **context managers** sont des objets qui définissent un contexte d'exécution dans Python.

Ils sont utilisés pour gérer les ressources système telles que les fichiers, les connexions réseau et les verrous de thread. Les context managers permettent de s'assurer que les ressources sont correctement gérées et libérées après leur utilisation, même en cas d'exception ou d'erreur.

En Python, les context managers sont implémentés en utilisant deux méthodes spéciales : `__enter__` et `__exit__`.

`__enter__` est appelée lorsque le contexte est créé, et elle renvoie l'objet qui sera utilisé dans le contexte. `__exit__` est appelée lorsque le contexte est quitté, et elle est utilisée pour nettoyer les ressources et gérer les exceptions.





## 8- Context manager



---

Les **context managers** sont des objets qui définissent un contexte d'exécution dans Python.

Ils sont utilisés pour gérer les ressources système telles que les fichiers, les connexions réseau et les verrous de thread. Les context managers permettent de s'assurer que les ressources sont correctement gérées et libérées après leur utilisation, même en cas d'exception ou d'erreur.

En Python, les context managers sont implémentés en utilisant deux méthodes spéciales : `__enter__` et `__exit__`.

`__enter__` est appelée lorsque le contexte est créé, et elle renvoie l'objet qui sera utilisé dans le contexte. `__exit__` est appelée lorsque le contexte est quitté, et elle est utilisée pour nettoyer les ressources et gérer les exceptions.





## 8- Context manager



---

Les **context managers** sont des objets qui définissent un contexte d'exécution dans Python.

Ils sont utilisés pour gérer les ressources système telles que les fichiers, les connexions réseau et les verrous de thread. Les context managers permettent de s'assurer que les ressources sont correctement gérées et libérées après leur utilisation, même en cas d'exception ou d'erreur.

En Python, les context managers sont implémentés en utilisant deux méthodes spéciales : `__enter__` et `__exit__`.

`__enter__` est appelée lorsque le contexte est créé, et elle renvoie l'objet qui sera utilisé dans le contexte. `__exit__` est appelée lorsque le contexte est quitté, et elle est utilisée pour nettoyer les ressources et gérer les exceptions.



# 8- Context manager

## Implémentation du context manger

```
class MyContext:
    def __init__(self, param):
        self.param = param

    def __enter__(self):
        return ...

    def __exit__(self, exc_type, exc_val, exc_tb):
        return ...
```

## Usage du context manger

```
with MyContext(param=« param ») as mc:
    #Instruction ....
```



## 8- Context manager

Si on reprend notre premier exemple, et qu'on utilise un context manager (open), on obtient ceci:

```
# Ouvrir un fichier en mode lecture
fichier = open('monfichier.txt', 'r')

# Lire le contenu du fichier
contenu = fichier.read()

# Fermer le fichier
fichier.close()
```



```
with open('monfichier.txt', 'r') as fichier:
    contenu = fichier.read()
    print(contenu)
```



## 8- Context manager: Exercice

---

Enoncé: Implémentez un context manager qui peut être utilisé pour mesurer le temps d'exécution d'une fonction.

Output voulu :

```
with Timer() as t:
```

```
    # Exécuter la fonction
```

```
# Afficher le temps d'exécution
```

```
print(f« La fonction a été exécuté en {t.interval} secondes.»)
```

A faire: Implémentez la classe Timer 😊







# 9- Tests unitaires

---

## Frameworks de tests:

- **unittest:** framework de test unitaire inclus dans la bibliothèque standard de Python. Il fournit des fonctionnalités pour écrire, organiser et exécuter des tests unitaires pour des modules Python.
  - **pytest:** framework de test pour Python qui permet d'écrire des tests unitaires de manière plus simple et plus expressive que le module de test standard de Python.
- 
- 

# 9-1- Tests unitaires : unittest

## Etapes pour écrire un test unitaire:

- 1- Importez le module **unittest** :
- 2- Créez une classe pour le test qui hérite de **unittest.TestCase** :
- 3- Écrivez une ou plusieurs méthodes de test. Chaque méthode de test doit avoir un nom qui commence par "test\_" et doit effectuer une assertion pour vérifier que le code testé se comporte comme prévu.
- 4- Exécutez les tests en appelant **unittest.main()** à la fin du fichier de test. Vous pouvez également exécuter les tests depuis la ligne de commande en tapant **python -m unittest nom\_du\_fichier.py**.

```
import unittest

def addition(a, b):
    return a + b

class TestMyCode(unittest.TestCase):
    def test_addition(self):
        resultat = addition(1, 2)
        self.assertEqual(resultat, 3)

if __name__ == '__main__':
    unittest.main()
```

## 9-1- Tests unitaires: @patch

**@patch** : ce décorateur permet de remplacer temporairement un objet ou une fonction par un objet simulé. Cela peut être utile lorsqu'une fonctionnalité dépend d'un objet externe qui n'est pas facilement accessible ou facile à tester. En utilisant @patch, vous pouvez remplacer cet objet par un objet simulé afin de pouvoir tester votre fonctionnalité plus facilement

```
from unittest.mock import patch

@patch('module.objet_a_remplacer')
def test_fonctionnalite(mock_objet):
    mock_objet.method.return_value = 'valeur_simulee'
    assert fonctionnalite() == 'valeur_simulee'
```

## 9-1- Tests unitaires: @mock

**@mock** : ce décorateur permet de créer un objet simulé à utiliser dans un test unitaire. Contrairement à @patch, @mock ne remplace pas un objet existant, mais crée un nouvel objet simulé à partir d'une classe ou d'une fonction

```
from unittest.mock import Mock

def test_fonctionnalite():
    mock_objet = Mock()
    mock_objet.method.return_value = 'valeur_simulee'
    assert fonctionnalite(mock_objet) == 'valeur_simulee'
```



# 9-1- Tests unitaires: monkeypatch

---

**monkeypatch** : ce module fournit une interface pour remplacer des variables et des fonctions à l'aide de valeurs simulées.

Contrairement à **@patch** et **@mock**, **monkeypatch** peut être utilisé à l'intérieur du corps d'un test unitaire plutôt qu'en tant que décorateur.

```
def test_fonctionnalite(monkeypatch):  
    def fonction_simulee():  
        return 'valeur_simulee'  
    monkeypatch.setattr('module.objet_a_remplacer', fonction_simulee)  
    assert fonctionnalite() == 'valeur_simulee'
```