

# PYTHON ET FRAMEWORK - DJANGO



Les structures de contrôles, les collections et les classes python dans les Framework python, l'accès aux données, la gestion des vues, la création des Template et des formulaires.

- 1) Élément du langage python
- 2) L'accès aux données et les ORM
- 3) Gestion des vues et les Template
- 4) Gestion des formulaires



# AGENDA

- **Introduction: Rappel sur les notions de base**
- **Partie 1: Le Framework Django (Distribution des PFM)**
- **Partie 2: Les URLs et les vues**
- **Partie 3: L'accès aux données et les ORM**
- **Partie 4: Gestion des vues**
- **Partie 5: Template**
- **Partie 6: Les formulaires**
- **La note des comptes rendus des TP inclus dans la note de PFM**

# AGENDA

- **Introduction: Rappel sur les notions de base**
- Partie 1: Le Framework Django (**Distribution des PFM**)
- Partie 2: Les URLs et les vues
- Partie 3: L'accès aux données et les ORM
- Partie 4: Gestion des vues
- Partie 5: Template
- Partie 6: Les formulaires
- **La note des comptes rendus des TP inclus dans la note de PFM**

# INTRODUCTION: RAPPEL SUR LES NOTION DE BASE

## SÉANCE 1

- 1) Préparation de l'environnement Python**
- 2) Les variables
- 3) L'affichage
- 4) Les listes
- 5) Les boucles, comparaison et Test
- 6) Les Tests et les fichiers
- 7) Les modules
- 8) Les fonctions
- 9) Les Containers et les dictionnaires et les collections

# 1) PRÉPARATION DE L'ENVIRONNEMENT PYTHON

## 1.1. Configuration de la machine pour le développement python : (1/2)

Les étapes ci-dessous sont disponible en détails sur le site: <https://code.visualstudio.com/docs/python/python-tutorial>

### ✓ L'environnement de développement Python:

1

Editeur (IDE): VS, VS Code, Vim, Atom, Brackets, Sublime Text, ....



Microsoft Visual Code: <https://visualstudio.microsoft.com/>

2

Interpréteur Python



<https://www.python.org/downloads/>

### ✓ Vérification de l'installation Python: (ex: sur Windows)

```
C:\Users\userName> py -3 --version  
Python 3.10.7
```

# 1) PRÉPARATION DE L'ENVIRONNEMENT PYTHON

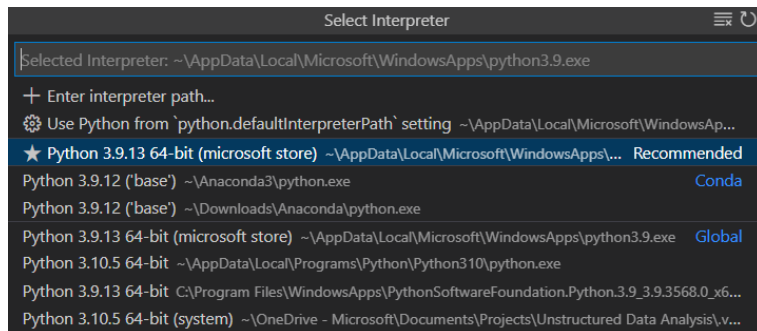
## 1.1. Configuration de la machine pour le développement python : (2/2)

✓ **Création du projet python dans un Workspace:** (ex: sur Windows)

 Démarrer VS Code dans un dossier de projet (espace de travail)

```
C:\Users\userName> cd "arborescence du dossier"  
« arborescence du dossier »> code .
```

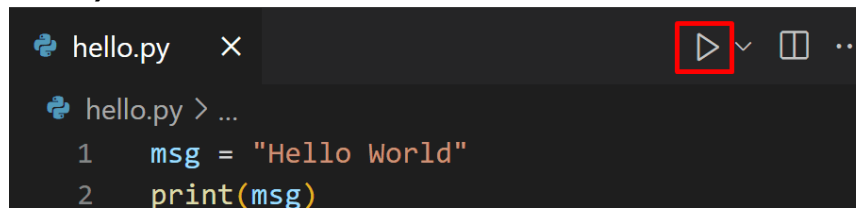
 Sélectionnez un interpréteur Python à l'aide du raccourcis clavier: **Ctrl+Shift+P** pour activer l'**IntelliSense**



Configuration des environnement Python

 Créer un fichier de code source Python

 Exécution du programme



# 1) PRÉPARATION DE L'ENVIRONNEMENT PYTHON

## 1.2. Les environnements Python: (1/2)

### ✓ L'environnement globale:

- Par défaut, tout interpréteur Python installé s'exécute dans son propre **environnement global**. Ils ne sont pas spécifiques à un projet particulier.
  - Avec le temps, cet environnement deviendra encombré et il sera difficile de tester correctement une application.

### ✓ Les environnements virtuel:

- L'**environnement virtuel** est un dossier qui contient une copie (ou un lien **symbolique - SymLink**) d'un interpréteur spécifique.
  - Lorsque vous **installez dans un environnement virtuel**, tous les packages que vous installez sont **installés uniquement dans ce sous-dossier**.
  - Lorsque vous **exécutez** ensuite un programme Python dans cet environnement, vous savez **qu'il ne s'exécute que sur ces packages spécifiques**.

✱ **Remarque** : bien qu'il soit possible d'ouvrir un dossier d'environnement virtuel en tant qu'espace de travail, cela n'est pas recommandé et peut entraîner des problèmes lors de l'utilisation de l'extension Python.

# 1) PRÉPARATION DE L'ENVIRONNEMENT PYTHON

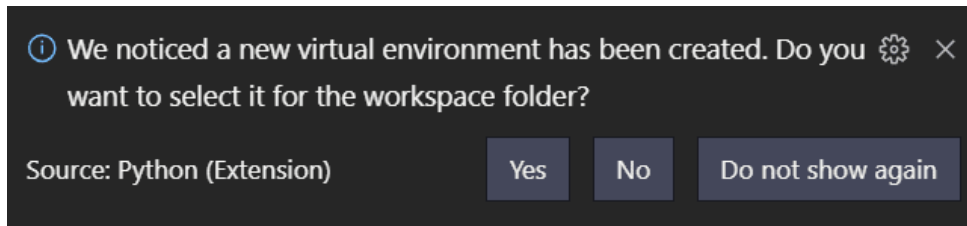
## 1.2. Les environnements Python: (2/2)

### ✓ Création d'un environnement Python:

- La création d'environnements virtuels se fait en exécutant la commande **venv** :

```
py -m venv \arborescence\de\l'environnement  
\arborescence\de\l'environnement\scripts\activate
```

- lorsque vous créez un nouvel environnement virtuel, une invite s'affiche pour vous permettre de le sélectionner pour l'espace de travail.



### ✓ Suppression d'un environnement Python:

- Il suffit de supprimer le dossier contenant l'environnement virtuel

```
rd /s /arborescence/de/l'environnement
```



# 1) PRÉPARATION DE L'ENVIRONNEMENT PYTHON

## 1.3. Installation et utilisation des packages (1/2)

Exemple :

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 20, 100) # Create a list of evenly-spaced numbers over the range
plt.plot(x, np.sin(x))      # Plot the sine of each x point
plt.show()                  # Display the plot
```



"ModuleNotFoundError: No module named 'matplotlib'".

# 1) PRÉPARATION DE L'ENVIRONNEMENT PYTHON

## 1.3. Installation et utilisation des packages (2/2)

 Créer et activer l'environnement virtuel

 Sélectionnez l'interpréteur Python de l'environnement crée à l'aide du raccourcis clavier: Ctrl+Shift+P.

 Installer les packages:

Exemple :

```
python -m pip install matplotlib
```

✓ Réexécutez le programme maintenant (avec ou sans le débogueur)

 Une fois que vous avez terminé, tapez deactivates dans la fenêtre du terminal pour désactiver l'environnement virtuel.

# INTRODUCTION: RAPPEL SUR LES NOTION DE BASE

## SÉANCE 1

- 1) Préparation de l'environnement Python
- 2) Les variables**
- 3) L'affichage
- 4) Les listes
- 5) Les boucles, comparaison et Test
- 6) Les Tests et les fichiers
- 7) Les modules
- 8) Les fonctions
- 9) Les Containers et les dictionnaires et les collections

## 2) LES VARIABLES

### 2.1. Définition des variables

#### ✓ Déclaration et initialisation:

- Une variable est une **zone de la mémoire de l'ordinateur** dans laquelle une valeur est stockée:
  - ☑ cette variable est définie par **un nom**, alors que pour l'ordinateur, il s'agit en fait d'une **adresse**.
- En Python, la **déclaration** d'une variable et son **initialisation** se font en même temps:

```
>>> x = 2
>>> x
2
```

#### ✓ Les types des variables:

Les trois principaux types dont nous aurons besoin dans un premier temps sont:

- ✓ Les **entiers** (integer ou **int**),
- ✓ Les **nombres décimaux** que nous appellerons **floats**
- ✓ Les **chaînes de caractères** (**string** ou **str**).

## 2) LES VARIABLES

### 2.2. Les opérations sur les variables

#### ✓ Opération sur les types numériques:

+	Addition
-	Soustraction
*	Multiplication
/	Division
% ou //	Modulo (reste de la division entière)
**	puissance

#### ✓ Opération sur les chaines de caractères:

+	Concaténation entre deux chaines de caractères
*	Duplication d'une chaine de caractère

#### ✓ Savoir le type d'une variable:

```
>>> type(nom_variable)
```

## 2) LES VARIABLES

### 2.3. Conversion de type des variables:

- Dans Python la conversion de type se fait à l'aides des fonctions **int()**, **float()** et **str()**.

Exemple :

```
>>> i = 3
>>> str (i)
'3'

>>> i = '456'
>>> int (i)
456

>>> float (i)
456.0

>>> i = '3.1416 '
>>> float (i)
3.1416
```

## 2) LES VARIABLES

### 2.4. Minimum et le maximum de deux variables:

- Python propose les fonctions **min()** et **max()** qui renvoient respectivement le minimum et le maximum de plusieurs **entiers** et / ou **floats** :

Exemple :

```
>>> min (1, -2, 4)
-2
>>> pi = 3.14
>>> e = 2.71
>>> max (e, pi)
3.14
>>> max (1, 2.4 , -6)
2.4
```

# INTRODUCTION: RAPPEL SUR LES NOTION DE BASE

## SÉANCE 1

- 1) Préparation de l'environnement Python
- 2) Les variables
- 3) L'affichage**
- 4) Les listes
- 5) Les boucles, comparaison et Test
- 6) Les Tests et les fichiers
- 7) Les modules
- 8) Les fonctions
- 9) Les Containers et les dictionnaires et les collections



## 3) L’AFFICHAGE

### 3.1. la fonction print():

- la fonction **print()** affiche l’argument qu’on lui passe entre parenthèses et un retour à ligne.

#### Exemples :

```
>>> print (" Hello world !")  
Hello World!
```

```
>>> var = 3  
>>> print ( var )  
3
```

```
>>> x = 32  
>>> nom = " John "  
>>> print (nom , "a", x, "ans")  
Jhon a 32 ans
```

```
>>> x = 32  
>>> nom = " John "  
>>> print (nom , "a", x, " ans ", sep  
="")  
Jhona30ans
```

```
>>> print (nom , "a", x, " ans ", sep ="  
-")  
Jhon-a-30-ans
```

## 3) L’AFFICHAGE

### 3.2. l’écriture formatée (*f-strings*)

- ***f-string*** est le diminutif de « formatted string literals ».
- L’équivalent de ***f-string*** est une chaîne de caractères précédée du caractère ***f*** sans espace entre les deux

```
>>>f" Une chaîne de caractères"
```

- Ce caractère ***f*** avant les guillemets va indiquer à Python qu’il s’agit d’une ***f-string*** permettant de mettre en place le **mécanisme de l’écriture formatée**.

#### Exemples :

```
>>> x = 32
>>> nom = " John "
>>> print (f"{ nom } a {x} ans ")
4 John a 32 ans
```

```
>>> print (f"J'affiche l'entier {10},le float {3.14} et la chaîne {'Python'}")
J'affiche l'entier 10, le float 3.14 et la chaîne python
```

# INTRODUCTION: RAPPEL SUR LES NOTION DE BASE

## SÉANCE 1

- 1) Préparation de l'environnement Python
- 2) Les variables
- 3) L'affichage
- 4) Les listes**
- 5) Les boules, comparaison et Test
- 6) Les Tests et les fichiers
- 7) Les modules
- 8) Les fonctions
- 9) Les Containers et les dictionnaires et les collections

## 4) LES LISTES

### 4.1. Utilisation des listes

- Une liste est une **structure de données** qui contient une **série de valeurs**.
  - Python autorise la construction de **liste contenant des valeurs de types différents**.

#### Exemples :

```
>>> animaux = [" girafe ", " tigre ", " singe ", " souris "]
>>> tailles = [5, 2.5 , 1.75 , 0.15]
>>> mixte = [" girafe ", 5, " souris ", 0.15]
>>> animaux
['girafe ', 'tigre ', 'singe ', 'souris ']
>>> tailles
[5, 2.5 , 1.75 , 0.15]
>>> mixte
['girafe ', 5, 'souris ', 0.15]
```

- On peut appeler les éléments d'une liste par leur position (indice /index) ou indilage négatif

```
>>> animaux = [0]
girafe
```

```
>>> animaux = [-4]
girafe
```

## 4) LES LISTES

### 4.2. Opération sur les listes

- Tout comme les chaînes de caractères, les listes supportent **l'opérateur + de concaténation**, ainsi que **l'opérateur \* pour la duplication**.

#### Exemples :

```
>>> ani1 = [" girafe ", " tigre "]
>>> ani2 = [" singe ", " souris "]
>>> ani1 + ani2
['girafe ', 'tigre ', 'singe ', 'souris ']
>>> ani1 * 3
['girafe ', 'tigre ', 'girafe ', 'tigre ', 'girafe ', 'tigre ']
```

- Pour la concaténation vous pouvez également utiliser la méthode **append()**

#### Exemples :

```
>>> a = [8]
>>> a.append(1)
>>> a.append(17)
```

```
>>> a
[8, 1, 17]
```

## 4) LES LISTES

### 4.3. Les tranches des listes

- Les listes de python nous permet de **sélectionner une partie d'une liste** en utilisant un indilage construit sur le modèle **[m:n+1]** pour récupérer tous les éléments, du **m<sup>ème</sup>** au **n<sup>ème</sup>** (de l'élément **m inclus** à l'élément **n+1 exclu**).

#### Exemples 1:

```
>>> animaux = [" girafe ", " tigre ", " singe ", "
souris "]
>>> animaux [0:2]
['girafe ', 'tigre ']
>>> animaux [0:3]
['girafe ', 'tigre ', 'singe ']
>>> animaux [0:]
['girafe ', 'tigre ', 'singe ', 'souris ']
>>> animaux [:]
['girafe ', 'tigre ', 'singe ', 'souris ']
>>> animaux [1:]
['tigre ', 'singe ', 'souris ']
>>> animaux [1: -1]
['tigre ', 'singe ']
```

## 4) LES LISTES

### 4.3. Les tranches des listes

- On peut préciser **le pas** en ajoutant **un symbole deux-points supplémentaire** et en indiquant **le pas par un entier**.

#### Exemples 2:

```
>>> animaux = [" girafe ", " tigre ", " singe ", " souris "]
>>> animaux [0:3:2]
['girafe ', 'singe ']
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x [::1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x [::2]
[0, 2, 4, 6, 8]
>>> x [::3]
[0, 3, 6, 9]
>>> x [1:6:3]
[1, 4]
```

## 4) LES LISTES

### 4.4. les fonctions des listes

✓ La fonction **len()** :

- L'instruction **len()** vous permet de connaître **la longueur d'une liste**, c'est-à-dire le nombre d'éléments que contient la liste.

Exemples :

```
>>> animaux = [" girafe ", " tigre ", " singe ", " souris "]
>>> len ( animaux )
4
>>> len ([1 , 2, 3, 4, 5, 6, 7, 8])
8
```



## 4) LES LISTES

### 4.4. les fonctions des listes

#### ✓ La fonction `list()` :

- La méthode `list()` prend une séquence (**string**, **tuples**) ou collection (**set**, **dictionary**) ou tout objet itérateur et la **convertit en liste**.

#### ✓ La fonction `range()` :

- L'instruction `range()` est une fonction spéciale en Python qui **génère des nombres entiers compris** dans un intervalle: `range([début,] fin[, pas])`.
- Lorsqu'elle est utilisée en combinaison avec la fonction `list()`, on obtient une liste d'entiers.

#### Exemples :

```
>>> list( range(10))  
[0, 1, 2, 3, 4, 5, 6, 7, 8,  
9]  
>>> list ( range (0, 5))  
[0, 1, 2, 3, 4]
```

```
>>> list ( range (15 , 20))  
[15 , 16, 17, 18, 19]  
>>> list ( range (0, 1000 , 200))  
[0, 200 , 400 , 600 , 800]  
>>> list ( range (2, -2, -1))  
[2, 1, 0, -1]
```

## 4) LES LISTES

### 4.4. Listes de listes:

- Dans python on peut **construire** une liste des listes

Exemples :

```
>>> enclos1 = [" girafe ", 4]
>>> enclos2 = [" tigre ", 2]
>>> enclos3 = [" singe ", 5]
>>> zoo = [ enclos1 , enclos2 , enclos3 ]
>>> zoo
[[' girafe ', 4], ['tigre ', 2], ['singe ', 5]]
```

- Pour **accéder** à un élément de la liste, on utilise l'indilage habituel
- Pour **accéder** à un élément de la sous-liste, on utilise un double indilage

Exemples :

```
>>> zoo [1]
['tigre ', 2]
>>> zoo [1][0]
'tigre '
>>> zoo [1][1]
2
```

## 4) LES LISTES

### 4.5. Minimum, maximum et somme d'une liste:

- Les fonctions `min()`, `max()` et `sum()` renvoient respectivement le minimum, le maximum et la somme d'une liste passée en argument

#### Exemples :

```
>>> liste = list ( range (10))
>>> liste
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sum ( liste )
45
>>> min ( liste )
0
>>> max ( liste )
9
```

\* Même si en théorie ces fonctions peuvent prendre en argument une **liste de strings**, on les utilisera la plupart du temps avec des **types numériques** (liste d'entiers et / ou de floats).

# INTRODUCTION: RAPPEL SUR LES NOTION DE BASE

## SÉANCE 1

- 1) Préparation de l'environnement Python
- 2) Les variables
- 3) L'affichage
- 4) Les listes
- 5) Les boules, comparaison et Test**
- 6) Les Tests et les fichiers
- 7) Les modules
- 8) Les fonctions
- 9) Les Containers et les dictionnaires et les collections

## 5) LES BOUCLES, COMPARAISON ET TEST

### 5.1. La Comparaison dans python :

- Python est capable d'effectuer toute une **série de comparaisons** entre le contenu de deux variables, telles que :

<code>==</code>	Égal à
<code>!=</code>	différent de
<code>&gt;</code>	supérieur à
<code>&gt;=</code>	supérieur ou égal à
<code>&lt;</code>	inférieur
<code>&lt;=</code>	Inférieur ou égal à

#### Exemples :

```
>>> x = 5
>>> x == 5
True
>>> x > 10
False
```

```
>>> animal = " tigre "
>>> animal == " tig "
False
>>> animal != " tig "
True
```

```
>>> " ali " < " alo"
True
>>> " abb " < " ada"
True
```

## 5) LES BOUCLES, COMPARAISON ET TEST

### 5.2. La boucle **for** : (1/3)

#### ✓ Utilisation de la boucle **for** ( ) :

- L'instruction **for** est une instruction composée:
  - Une instruction dont l'en-tête se termine par deux-points : , suivie d'un bloc indenté qui constitue le corps de la boucle.

#### Exemples :

```
>>> animaux = [" girafe ", " tigre ", " singe ", "
souris "]
>>> for animal in animaux :
...     print ( animal )
...
girafe
tigre
singe
souris
```

## 5) LES BOUCLES, COMPARAISON ET TEST

### 5.2. La boucle **for** : (2/3)

✓ La fonction **range()** et **len()** :

Exemples :

```
>>> for i in range (2):  
...     print (i)  
...  
0  
1
```

```
>>> animaux = [" girafe ", " tigre ", " singe ", " souris "  
>>> for i in range ( len ( animaux )):  
...     print (f"L'animal {i} est un(e) { animaux [i ]}")  
...  
L' animal 0 est un(e) girafe  
L' animal 1 est un(e) tigre  
L' animal 2 est un(e) singe  
L' animal 3 est un(e) souris
```

## 5) LES BOUCLES, COMPARAISON ET TEST

### 5.2. La boucle **for** : (3/3)

#### ✓ La fonction **enumerate()** :

- Python possède toutefois la fonction **enumerate()** qui permet d'itérer sur les indices et les éléments eux-mêmes.

#### Exemples :

```
>>> animaux = [" girafe ", " tigre ", " singe ", " souris "]
>>> for i, animal in enumerate ( animaux ):
...     print (f"L' animal {i} est un(e) { animal }")
...
L' animal 0 est un(e) girafe
L' animal 1 est un(e) tigre
L' animal 2 est un(e) singe
L' animal 3 est un(e) souris
```



## 5) LES BOUCLES, COMPARAISON ET TEST

### 5.3. La boucle **while** :

- Dans la boucle **while** une série d'instructions est exécutée tant qu'une condition est vraie.

#### Exemples :

```
>>> i = 1
>>> while i <= 4:
...     print (i)
...     i = i + 1
...
1
2
3
4
```

```
>>> i = 0
>>> while i < 10:
...     reponse = input (" Entrez un entier supérieur à 10 : ")
...     i = int ( reponse )
...
Entrez un entier supérieur à 10 : 4
Entrez un entier supérieur à 10 : -3
Entrez un entier supérieur à 10 : 15
>>> i
15
```

## 5) LES BOUCLES, COMPARAISON ET TEST

### 5.4. Les tests: (1/3)

- Les **tests** sont des éléments essentiels à tout langage informatique. Python utilise l'instruction **if** ainsi qu'une comparaison.

#### Exemples :

```
>>> x = 2
>>> if x == 2:
...     print (" Le test est vrai !")
...
Le test est vrai !
```

- il est pratique de tester si la condition est vraie ou si elle est fausse dans une même instruction

#### Exemples :

```
>>> x = '2'
>>> if x == 2:
...     print (" Le test est vrai !")
... else :
...     print (" Le test est faux !")
...
Le test est faux !
```

## 5) LES BOUCLES, COMPARAISON ET TEST

### 5.4. Les tests: (2/3)

- On peut utiliser **une série de tests** dans la même instruction **if**, notamment pour tester plusieurs valeurs d'une même variable.

#### Exemples :

```
>>> import random
>>> base = random . choice ([" a", "t", "c", "g"])
>>> if base == "a":
...     print (" choix d'une adénine ")
... elif base == "t":
...     print (" choix d'une thymine ")
... elif base == "c":
...     print (" choix d'une cytosine ")
... elif base == "g":
...     print (" choix d'une guanine ")
...
choix d'une cytosine
```

## 5) LES BOUCLES, COMPARAISON ET TEST

### 5.4. Les tests: (3/3)

#### ✓ Les tests multiples

- Les tests multiples permettent de **tester plusieurs conditions** en même temps en utilisant des **opérateurs booléens**.

True	and	True	True
True	and	false	False
False	and	True	False
False	and	False	False

True	or	True	True
True	or	false	True
False	or	True	True
False	or	False	False

Exemples :

```
>>> x = 2
>>> y = 2
>>> if x == 2 and y == 2:
...     print (" le test est vrai ")
...
le test est vrai
```

## 5) LES BOUCLES, COMPARAISON ET TEST

### 5.4. Instructions `break` et `continue` :

- Ces deux instructions permettent de **modifier le comportement d'une boucle** (for ou while) avec un test.
  - ☑ L'instruction **`break`** stoppe la boucle.
  - ☑ L'instruction **`continue`** saute à l'itération suivante, sans exécuter la suite du bloc d'instructions de la boucle.

#### Exemples :

```
>>> for i in range (5):  
...     if i > 2:  
...         break  
...     print (i)  
...  
0  
1  
2
```

```
>>> for i in range (5):  
...     if i == 3:  
...         continue  
...     print (i)  
...  
0  
1  
2  
4
```

# INTRODUCTION: RAPPEL SUR LES NOTION DE BASE

## SÉANCE 2

- 1) Préparation de l'environnement Python
- 2) Les variables
- 3) L'affichage
- 4) Les listes
- 5) Les boucles, comparaison et Test
- 6) Les fichiers**
- 7) Les modules
- 8) Les fonctions
- 9) Les Containers et les dictionnaires et les collections

## 6) LES FICHIERS

### 6.1. Ouvrir et fermer un fichier :

#### ✓ La méthode **open()** :

- Pour ouvrir un fichier python propose la méthode **open()**:

Syntaxe : `file = open ("chemin du fichier", "type d'ouverture")`

- r** : ouverture en lecture (READ).
- w** : ouverture en écriture (WRITE), à chaque ouverture le contenu du fichier est écrasé ou crée.
- x** : crée un nouveau fichier et l'ouvre pour écriture
- a** : pour une ouverture en mode ajout à la fin du fichier (APPEND).

#### ✓ La méthode **close()** :

- Il faut refermer le fichier une fois les instructions terminées à l'aide de méthode **close()**:

Syntaxe : `file.close()`

## 6) LES FICHIERS

### 6.2. Lecture dans un fichier : (1/4)

#### ✓ La méthode `.readlines()` :

- La méthode `.readlines()` agit sur l'objet « fichier ouvert » en déplaçant le curseur de lecture du début à la fin du fichier, puis elle renvoie une liste contenant toutes les lignes du fichier.

Exemple :

```
>>> file = open ("zoo.txt", "r")
>>> lignes = file.readlines()
>>> lignes
[' girafe \n', ' tigre \n', ' singe \n', ' souris \n ']
>>> for ligne in lignes :
        print ( ligne )

girafe
tigre
singe
souris
file . close ()
```



## 6) LES FICHIERS

### 6.2. Lecture dans un fichier : (2/4)

- ✓ La méthode `.readlines()` avec le mot-clé `with` :
- le mot-clé **with** permet **d'ouvrir** et de **fermer** un fichier de manière efficace.

Exemple :

```
>>> with open (" zoo. txt ", 'r ') as file :  
...     lignes = file . readlines ()  
...     for ligne in lignes :  
...         print ( ligne )  
...  
girafe  
tigre  
singe  
souris
```

## 6) LES FICHIERS

### 6.2. Lecture dans un fichier : (3/4)

#### ✓ La méthode `.read()` :

- la méthode `.read()` lit tout le contenu d'un fichier et renvoie une chaîne de caractères unique.

Exemple :

```
>>> with open (" zoo. txt ", 'r ') as file :  
...     file . read ()  
...  
'girafe \ ntigre \ nsinge \ nsouris \n'
```

#### ✓ La méthode `.readline()` :

- La méthode `.readline()` lit **une ligne d'un fichier** et la renvoie sous forme de chaîne de caractères.
  - À chaque nouvel appel de `.readline()`, la ligne suivante est renvoyée

Exemple :

```
>>> with open (" zoo. txt ", "r") as file :  
...     ligne = file . readline ()  
...     while ligne != "":  
...         print ( ligne )  
...         ligne = file . readline ()
```

## 6) LES FICHIERS

### 6.2. Lecture dans un fichier : (4/4)

#### ✓ Itération directe sur un fichier :

- L'objet file est « **itérable** », ainsi la boucle **for** peut demander à Python d'aller lire le fichier ligne par ligne.

#### Exemple :

```
>>> with open (" zoo. txt ", "r") as file :  
...     for ligne in file :  
...         print ( ligne )  
...  
girafe  
tigre  
singe  
souris  
  
>>>
```

## 6) LES FICHIERS

### 6.3. Ecriture dans un fichier :

- La méthode `.write()` permet d'écrire dans un fichier.

#### Exemple :

```
>>> animaux2 = ["poisson", "abeille", "chat"]
>>> with open (" zoo2.txt", "w") as filout :
...     for animal in animaux2 :
...         filout . write (f"{ animal }\n")
...
8
8
5
```

## 6) LES FICHIERS

### 6.4. Ouverture de plusieurs fichiers avec l'instruction `with`:

- l'instruction **`with`** ouvrir plusieurs fichiers en même temps.

Exemple :

```
>>> with open ("zoo.txt", "r") as file1, open ("zoo2.txt", "w") as file2 :  
    for ligne in file1 :  
        file2 . write ("*" + ligne )
```

# INTRODUCTION: RAPPEL SUR LES NOTION DE BASE

## SÉANCE 2

- 1) Préparation de l'environnement Python
- 2) Les variables
- 3) L'affichage
- 4) Les listes
- 5) Les boucles, comparaison et Test
- 6) Les fichiers
- 7) Les modules**
- 8) Les fonctions
- 9) Les Containers et les dictionnaires et les collections

# 7) LES MODULES

## 7.1. Présentation:

- Les **modules** sont des programmes Python qui contiennent des fonctions que l'on est amené à réutiliser souvent (on les appelle aussi bibliothèques ou **libraries**).
  - Ce sont des « **boîtes à outils** » qui vont vous être très utiles.

### Exemple :

- ❑ **Math** : fonctions et constantes mathématiques de base (sin, cos, exp, pi, ...).
- ❑ **sys** : interaction avec l'interpréteur Python, passage d'arguments.
- ❑ **os** : dialogue avec le système d'exploitation.
- ❑ **Random** : génération de nombres aléatoires.
- ❑ **Time** : accès à l'heure de l'ordinateur et aux fonctions gérant le temps.
- ❑ **Urllib** : récupération de données sur internet depuis Python.
- ❑ **Tkinter** : interface python avec **Tk**. Création d'objets graphiques.
- ❑ **Re** : gestion des expressions régulières.
- ❑ **Numpy** : Utiliser pour travailler avec des array. Il a également des fonctions pour travailler dans le domaine de l'algèbre linéaire, de la transformée de Fourier et des matrices....

# 7) LES MODULES

## 7.2. Importation des modules: (1/2)

- l'utilisation de la syntaxe **import module** permet d'importer toute une série de fonctions organisées par « **thèmes** ».

Exemple :

```
>>> import math
>>> math.cos (math.pi / 2)
6.123233995736766e-17
>>> math.sin (math.pi / 2)
1.0
```

- Pour importer une ou plusieurs fonctions spécifique d'un module, on peut utiliser **from module import function**.

Exemple :

```
>>> from random import randint
>>> randint (0 ,10)
7
```



# 7) LES MODULES

## 7.2. Importation des modules: (2/2)

- Pour importer une ou plusieurs fonctions spécifique d'un module, on peut utiliser **from module**

**Exemple :**

```
>>> from random import *
>>> uniform (0 ,2.5)
0.64943174760727951
```

```
>>> from random import *
>>> uniform (0 ,2.5)
0.64943174760727951
```

- Il est possible de définir un alias (un nom plus court) pour un module

**Exemple :**

```
>>> import random as rand
>>> rand.randint (1, 10)
6
```

- Pour vider la mémoire d'un module déjà chargé, on peut utiliser l'instruction **del** :

**Exemple :**

```
>>> import random
>>> random . randint (0 ,10)
2
>>> del random
```

# INTRODUCTION: RAPPEL SUR LES NOTION DE BASE

## SÉANCE 2

- 1) Préparation de l'environnement Python
- 2) Les variables
- 3) L'affichage
- 4) Les listes
- 5) Les boucles, comparaison et Test
- 6) Les fichiers
- 7) Les modules
- 8) Les fonctions**
- 9) Les Containers et les dictionnaires et les collections

## 8) LES FONCTIONS

### 8.1. Définition :

- Pour définir une fonction, Python utilise le mot-clé **def**. Si on souhaite que la fonction renvoie quelque chose, il faut utiliser le mot-clé **return**.
- On peut passer un argument, retourner une valeur.
- La valeur retournée par une fonction est récupérable dans une variable.

Exemple :

```
>>> def carre(x):  
...     return x**2  
>>> print(carre(2))  
4
```

```
>>> def hello():  
...     print("bonjour")  
>>> hello()  
bonjour
```

```
>>> def carre(x):  
...     return x**2  
>>> y=carre(3)  
>>> y  
9
```

## 8) LES FONCTIONS

### 8.2. Passage d'arguments :

- Le **nombre d'arguments** que l'on peut passer à une fonction est **variable**.
- Python est un langage « typage dynamique »: Vous n'êtes pas obligé de **préciser le type des arguments** que vous lui passez, dès lors que les opérations que vous effectuez avec ces arguments sont valides.

Exemple :

```
>>> def fois(x,y):  
...     return x*y  
...  
>>> fois(2, 3)  
6  
>>> fois(3.1415 , 5.23)  
16.430045000000003  
>>> fois("to", 2)  
'toto '  
>>> fois([1 ,3], 2)  
[1, 3, 1, 3]
```

## 8) LES FONCTIONS

### 8.3. Renvoi de résultats:

- En Python les fonctions sont **capables** de renvoyer plusieurs objets à la fois
- En réalité Python **ne renvoie qu'un seul objet**, mais celui-ci peut être séquentiel: contenir lui même d'autres objets.
- Renvoyer un **tuple** ou une **liste** de deux éléments (ou plus) est très pratique en **conjonction avec l'affectation multiple**.
  - Cela permet de **recupérer plusieurs valeurs renvoyées** par une fonction et de les **affecter à la volée à des variables différentes**.

Exemple:

```
>>> def carre_cube (x):  
...     return  
x**2,x**3  
...  
>>> carre_cube(2)  
(4, 8)
```

```
>>> def carre_cube2  
(x):  
...     return [x**2,x**3]  
...  
>>> carre_cube2 (3)  
[9, 27]
```

```
>>> z1,z2=carre_cube(3)  
>>> z1  
9  
>>> z2  
27
```

## 8) LES FONCTIONS

### 8.4. Arguments positionnels et arguments par mot-clé:

- Lorsqu'on définit une fonction **def fct(x,y):** les arguments **x** et **y** sont appelés arguments positionnels (positional arguments).
- Il est aussi possible de passer un ou plusieurs argument(s) de **manière facultative** et de leur attribuer une valeur par défaut:
  - Un argument défini avec une syntaxe **def fct(arg=val):** est appelé argument par mot-clé (keyword argument).
  - Python permet même de rentrer les arguments par mot-clé dans un ordre arbitraire

#### Exemple :

```
>>> def fct(x=1):  
...     return x  
>>> fct ()  
1  
>>> fct (10)  
10
```

```
>>> def fct (x=0, y=0, z =0):  
...     return x, y, z  
>>> fct (z=10 , x=3, y =80)  
(3, 80, 10)  
>>> fct (z=10 , y =80)  
(0, 80, 10)
```

## 8) LES FONCTIONS

### 8.5. Variables locales et variables globales:

- Une variable est dite **locale** lorsqu'elle est créée dans une fonction:
  - Elle n'existera et ne sera visible que lors de l'exécution de ladite fonction.
- Une variable est dite **globale** lorsqu'elle est créée dans le programme principal.
  - Elle sera visible partout dans le programme.

#### Exemple :

```
# définition d'une fonction carre ()
def carre (x):
    y = x **2
    return y

# programme principal
z = 5
resultat = carre (z)
print(resultat)
```

## 8) LES FONCTIONS


### 8.7. Documentation des fonctions:

- Il est toujours important de **commenter** notre développement pour faciliter la réutilisation des fonctions et le commit sur GitHub.

Exemple : NumPy/SciPy Docstrings

```
def nom_fonction(nom_argument):  
    """description de la fonction  
  
    Parameters  
    -----  
    nom_argument : type d'argument  
        description de l'argument  
  
    Returns  
    -----  
    type de retour  
        description du retour
```

```
    Raises  
    -----  
    type du raise(ex:  
    NotImplementedError)  
        description de l'erreur  
  
    """  
  
    #list d'instructions
```





# INTRODUCTION: RAPPEL SUR LES NOTION DE BASE

## SÉANCE 2

- 1) Préparation de l'environnement Python
- 2) Les variables
- 3) L'affichage
- 4) Les listes
- 5) Les boucles, comparaison et Test
- 6) Les fichiers
- 7) Les modules
- 8) Les fonctions
- 9) Les Containers et les dictionnaires et les collections**

## 9) LES CONTAINERS ET LES DICTIONNAIRES ET LES COLLECTIONS

### 9.1. Les containers:

- Un **container** est un nom générique pour définir un objet Python qui contient une collection d'autres objets. Exemple : les listes, les chaînes de caractères et les objets de type **range()**
- ✓ **Propriétés des containers :**
  - Capacité à supporter le test **d'appartenance**.
  - Capacité à supporter la fonction **len()** renvoyant la longueur du container.
  - **Ordonné** (ordered) : il y a un ordre précis des éléments.
  - **Indexable** (subscriptable) : on peut retrouver un élément par son indice ou plusieurs éléments avec une tranche.
  - **Itérable** (iterable) : on peut faire une boucle dessus: objet séquentiel.
  - **Non modifiable** : On parle aussi d'objet **immuable** (immutable). Cela signifie qu'une fois créé, Python ne permet plus de le modifier par la suite.
  - **Hachable**: il est possible de calculer une valeur de hachage sur celui-ci avec la fonction interne **hash()**

# 9) LES CONTAINERS ET LES DICTIONNAIRES ET LES COLLECTIONS

## 9.2. Les dictionnaires: (1/6)

- Les **dictionnaires** sont des **collections ordonnée d'objets**. Il ne s'agit pas d'objets séquentiels comme les listes ou chaînes de caractères, mais plutôt d'objets dits de correspondance (**mapping objects**) ou tableaux associatifs.
  - On accède aux valeurs d'un dictionnaire par des **clés**.
  - on définit un dictionnaire vide avec les accolades **{}**: on remplit le dictionnaire avec différentes clés auxquelles on affecte des valeurs.
  - On peut aussi initialiser toutes les clés et les valeurs d'un dictionnaire en une seule opération, et d'ajouter une clé et une valeur supplémentaire.

Exemple :

```
>>> ani1 = {}
>>> ani1 ["nom"] = "girafe"
>>> ani1 ["taille"] = 5.0
>>> ani1 ["poids"] = 1100
>>> ani1
{'nom': 'girafe ', 'taille': 5.0, 'poids': 1100}
>>> ani2 = {" nom ": " singe ", " poids ": 70, " taille ": 1.75}
>>> ani2 ["age"] = 15
```

## 9) LES CONTAINERS ET LES DICTIONNAIRES ET LES COLLECTIONS

### 9.2. Les dictionnaires: (2/6)

#### ✓ Itération sur les clés pour obtenir les valeurs :

- Si on souhaite voir toutes les associations **clés / valeurs**, on peut itérer sur un dictionnaire de la manière suivante:

Exemple :

```
>>> ani2 = {'nom' : 'singe', 'poids' : 70, 'taille' : 1.75}
>>> for key in ani2 :
...     print (key , ani2 [key])
...
nom singe
poids 70
taille 1.75
```

#### ✓ Existence d'une clé ou d'une valeur :

- Pour vérifier **si une clé existe dans un dictionnaire**, on peut utiliser le test d'appartenance avec l'opérateur **in** qui renvoie un booléen.

Exemple :

```
>>> if " poids " in ani2 :
...     print (" La clé 'poids ' existe pour ani2 ")
```

## 9) LES CONTAINERS ET LES DICTIONNAIRES ET LES COLLECTIONS

### 9.2. Les dictionnaires: (3/6)

✓ Les méthodes `.keys()`, `.values()`, `.items()`:

- Les méthodes `.keys()` et `.values()` renvoient les clés et les valeurs d'un dictionnaire :

Exemple :

```
>>> ani2 . keys ()
dict_keys(['poids ', 'nom ', 'taille '])
>>> ani2 . values ()
dict_values([70 , 'singe ', 1.75])
```

- La méthode `.items()` renvoie un nouvel objet `dict_items` : Celui-ci n'est pas indexable, mais il est itérable.

- Il rassemble à la méthode `enumerate()`

Exemple :

```
>>> dico = {0: "t", 1: "o", 2: "t", 3: "o"}
>>> dico . items ()
dict_items([(0, 't '), (1, 'o '), (2, 't '), (3, 'o ')])
>>> for key , val in dico . items ():
...     print (key , val )
```

## 9) LES CONTAINERS ET LES DICTIONNAIRES ET LES COLLECTIONS

### 9.2. Les dictionnaires: (4/6)

#### ✓ La méthode `.get()` :

- Si on demande la valeur associée à une clé qui n'existe pas, Python renvoie une erreur :
  - La méthode `.get()` extraie la valeur associée à une clé mais ne renvoie pas d'erreur si la clé n'existe pas

#### Exemple :

```
>>> ani2 = {'nom' : 'singe ', 'poids' : 70, 'taille' : 1.75}
>>> ani2 . get ( " nom " )
'singe '
>>> ani2 . get ( " age " )
>>>
```

- On peut indiquer à `.get()` une valeur par défaut si la clé n'existe pas:

#### Exemple :

```
>>> ani2 . get ( " age ", 12)
12
```

## 9) LES CONTAINERS ET LES DICTIONNAIRES ET LES COLLECTIONS

### 9.2. Les dictionnaires: (5/6)

#### ✓ Tri par clés :

- On peut utiliser la fonction **sorted()** pour trier un dictionnaire par ses clés :

#### Exemple :

```
>>> ani2 = {'nom': 'singe', 'taille': 1.75, 'poids': 70}
>>> sorted(ani2)
['nom', 'poids', 'taille']
```

#### ✓ Tri par valeurs :

- Pour trier un dictionnaire par ses valeurs, il faut utiliser la fonction **sorted** avec l'argument **key**

#### Exemple :

```
>>> dico = {"a": 15, "b": 5, "c": 20}
>>> sorted(dico, key=dico.get)
['b', 'a', 'c']
```

## 9) LES CONTAINERS ET LES DICTIONNAIRES ET LES COLLECTIONS

### 9.2. Les dictionnaires: (6/6)

#### ✓ Liste de dictionnaires :

- En créant une liste de dictionnaires qui possèdent les mêmes clés, on obtient une structure qui ressemble à une base de données:

#### Exemple :

```
>>> animaux = [ani1 , ani2 ]  
>>> animaux  
[{'nom': 'girafe', 'poids': 1100, 'taille': 5.0} , {'nom': 'singe', 'poids': ...}]
```

#### ✓ Fonction `dict()` :

- La fonction **`dict()`** va convertir l'argument qui lui est passé en dictionnaire:
  - L'argument qui lui est passé doit avoir une forme particulière : un objet séquentiel contenant d'autres objets séquentiels de 2 éléments.

#### Exemple :

```
>>> liste_animaux = [{" girafe ", 2}, {" singe ", 3}]  
>>> dict ( liste_animaux )  
{ 'girafe ' : 2, 'singe ' : 3 }
```



## 9) LES CONTAINERS ET LES DICTIONNAIRES ET LES COLLECTIONS

### 9.3. Les tuples:

- Les tuples (« n-uplets ») sont des **objets séquentiels** correspondant aux listes (itérables, ordonnés et indexables) mais ils sont toutefois non modifiables:
  - L'intérêt des tuples par rapport aux listes réside dans leur **immutabilité**. Cela, accélère considérablement la manière dont Python accède à chaque élément et ils prennent moins de place en mémoire.
  - L'affectation et l'indigage fonctionnent comme avec les **listes**. Mais si on essaie de modifier un des éléments du tuple, Python renvoie un message d'erreur.
  - Les opérateurs + et \* fonctionnent comme pour les listes (concaténation et duplication)
  - On peut utiliser la fonction **tuple(sequence)** qui fonctionne exactement comme la fonction **list()**,

#### Exemple :

```
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
>>> t [2]
3
```

```
>>> t [0:2]
(1, 2)
>>> t = t + (2, )
>>> t
(1, 2, 3, 2)
```

```
>>> tuple ([1, 2, 3])
(1, 2, 3)
>>> tuple ("ATGCC")
('A', 'T', 'G', 'C', 'C')
```

## 9) LES CONTAINERS ET LES DICTIONNAIRES ET LES COLLECTIONS

### 9.4. Les sets :

- Les sets ont la particularité d'être modifiables, non hachables, non ordonnés, non indexables et de ne contenir qu'une seule copie maximum de chaque élément.
  - Les containers de type **set** sont très utiles pour rechercher les **éléments uniques** d'une suite d'éléments.
  - Les sets permettent l'évaluation d'union ou d'intersection mathématiques en conjonction avec les opérateurs respectivement **|** et **&** ou les méthodes **.union** et **.intersection**
  - L'instruction **s1.difference(s2)** renvoie sous la forme d'un nouveau set les éléments de **s1** qui ne sont pas dans **s2**.
  - La méthode **.issubset()** indique si un set est inclus dans un autre **set**. La méthode **isdisjoint()** indique si un **set** est disjoint d'un autre set.

#### Exemple :

```
>>> s = {4, 5, 5, 12}
>>> s
{12, 4, 5}
```

```
>>> liste_1 = [3, 3, 5, 1, 3, 4, 1, 1, 4, 4]
>>> liste_2 = [3, 0, 5, 3, 3, 1, 1, 1, 2, 2]
>>> set ( liste_1 ) | set ( liste_2 )
{0, 1, 2, 3, 4, 5}
```

## 9) LES CONTAINERS ET LES DICTIONNAIRES ET LES COLLECTIONS

### 9.5. les collections:

- Le module **collections** implémente des types de données de conteneurs spécialisés qui apportent des alternatives aux conteneurs natifs de Python plus généraux **dict**, **list**, **set** et **tuple**.
  - Les **dictionnaires ordonnés** : qui se comportent comme les dictionnaires classiques mais qui sont ordonnés ;
  - Les **defaultdicts** : permettant de générer des valeurs par défaut quand on demande une clé qui n'existe pas (cela évite que Python génère une erreur) ;
  - Les **namedtuples** : que nous évoquerons au chapitre 19 Avoir la classe avec les objets.
  - Les **compteurs** : crée des compteurs à partir d'un objets itérable

#### Exemple :

```
>>> import collections
>>> compo_seq = collections . Counter (" aatctccgatcgatcgatcgatgc ")
>>> compo_seq
Counter ({'a ': 7, 't ': 7, 'c ': 7, 'g ': 5})
>>> compo_seq["a"]
7
>>> compo_seq["n"]
0
```

# INTRODUCTION: RAPPEL SUR LES NOTION DE BASE

## SÉANCE 3

- 2) Les variables
- 3) L'affichage
- 4) Les listes
- 5) Les boucles, comparaison et Test
- 6) Les fichiers
- 7) Les modules
- 8) Les fonctions
- 9) Les Containers et les dictionnaires et les collections
- 10)Création des modules**

## 10) CRÉATION DES MODULES

### 10.1. Utilité des modules:

- Une **fonction bien écrite** pourrait être judicieusement réutilisée dans un autre programme Python.
  - C'est l'intérêt de la **création d'un module**: On y met un ensemble de fonctions que l'on peut être amené à utiliser souvent.
  - En général, les modules sont regroupés autour d'un thème précis.
- ✓ **Création d'un module :**
  - Pour créer un module il suffit d'écrire un ensemble de **fonctions** (et/ou de **constantes**) dans un fichier, puis d'enregistrer ce dernier avec une extension « .py ».

```
""" Module inutile qui affiche des messages. """  
  
DATE = 22102022  
  
def bonjour ( nom ) :  
    """ Dit Bonjour . """  
    return " Bonjour " + nom  
  
def hello ( nom ) :  
    """ Dit Hello . """  
    return " Hello " + nom
```

## 10) CRÉATION DES MODULES

### 10.2. Utilisation de son propre modules:

- Pour appeler une fonction ou une variable d'un module, il faut que le fichier **nom\_module.py** soit:
  - dans le répertoire courant.
  - dans un répertoire listé par la variable d'environnement **PYTHONPATH** de votre système d'exploitation.
- Ensuite, il suffit d'importer le module et toutes ses fonctions (et constantes) vous sont alors accessibles.

#### Exemple :

```
>>> import message
>>> message . hello (" Joe ")
'Hello Joe '
>>> message . bonjour (" Monsieur ")
'Bonjour Monsieur '
>>> message . DATE
22102022
```

## 10) CRÉATION DES MODULES

### 10.3. La documentation des modules:

- Lorsqu'on écrit un module, il est important de créer de la **documentation** pour expliquer ce que fait le module et **comment utiliser chaque fonction**.
- Les chaînes de caractères entre triple guillemets situées en début du module et de chaque fonction sont là pour cela, on les appelle **docstrings**.
- Ces **docstrings** permettent de fournir de l'aide lorsqu'on invoque la commande **help()**
- Pour quitter l'aide, pressez la touche Q.

```
>>> help ( message )
Help on module message :

NAME
    message - Module inutile qui affiche
              des messages.

FUNCTIONS
    bonjour (nom)
        Dit Bonjour .

    hello (nom)
        Dit Hello .

DATA
    DATE = 22102022

FILE
    /home/message.py
```

## 10) CRÉATION DES MODULES

### 10.4. Module ou script:

- Un module typiquement ne contient que des fonctions et une constante.
  - Si on l'exécutait comme un script classique, cela n'afficherait rien.
  - A l'inverse Si on l'importe comme une module le code principale vas s'exécuter.
- Afin de pouvoir utiliser un code Python en tant que **module** ou en tant que **script**, voici la structure a suivre:

```
""" Script et module de test . """

def bonjour ( nom ) :
    """ Dit Bonjour . """
    return " Bonjour " + nom

if __name__ == "__main__" :
    print ( bonjour ( " Joe " ) )
```



# **INTRODUCTION:** RAPPEL SUR LES NOTION DE BASE

## **SÉANCE 1: TRAVAUX PRATIQUE**

### **TP 1:** Éléments du langage Python

# INTRODUCTION: RAPPEL SUR LES NOTION DE BASE

## SÉANCE 2

- 3) L'affichage
- 4) Les listes
- 5) Les boucles, comparaison et Test
- 6) Les fichiers
- 7) Les modules
- 8) Les fonctions
- 9) Les Containers et les dictionnaires et les collections
- 10)Création des modules
- 11)La POO dans Python**

## 11) LA POO DANS PYTHON

- ❑ La **programmation orientée objet** (POO) est un concept de programmation très puissant qui permet de structurer ses programmes d'une manière logique.
  - En **POO**, on définit un « **objet** » qui peut contenir des « **attributs** » ainsi que des « **méthodes** » qui agissent sur lui-même.
  - Une **classe** définit des **objets** qui sont des **instances** (des représentants) de cette classe.

# 11) LA POO DANS PYTHON

## 11.1. Les classes:

- ❑ Une **classe** regroupe des fonctions et des attributs qui définissent un objet.

Exemple :

```
class Voiture:  
  
    def __init__(self):  
        self.nom = "Ferrari"
```

- ❑ La méthode **\_\_init\_\_()** est appelée lors de la création d'un objet.
  - ❑ **self.nom** est une manière de stocker une information dans la classe.
    - ❑ On parle **d'attribut de classe**.
    - ❑ Dans notre cas, on stock le nom dans l'attribut nom .

# 11) LA POO DANS PYTHON

## 11.2. Les objets:

- ❑ Un **objet** est une instance d'une classe.
  - ✓ On peut créer autant d'objets que l'on désire avec une classe.

Exemple : `ma_voiture = Voiture()`

## 11.3. Les attributs de classe:

- ❑ Les **attributs de classe** permettent de stocker des informations au niveau de la classe.

Exemple : `ma_voiture = Voiture()  
print(ma_voiture.nom)`

- ❑ On peut créer un attribut pour un objet, et le lire aussi.

Exemple : `ma_voiture.modele = "250"  
print(ma_voiture.modele)`

# 11) LA POO DANS PYTHON

## 11.4. Les méthodes:

- ❑ Les **méthodes** sont des fonctions définies dans une classe.

Exemple :

```
class Voiture:
    def __init__(self):
        self.nom =
        "Ferrari"

    def nom_modele(self):
        return self.nom
```

```
from ex1 import Voiture
m_ = Voiture()
print(m_.nom_modele())
```

## 11.5. La fonction dir:

- ❑ Parfois il est intéressant de décortiquer un **objet** pour résoudre à un bug ou pour comprendre un script.
  - La fonction **dir** nous donne un aperçu des méthodes de l'objet:

Exemple :

```
print(dir(ma_voiture))
```

# 11) LA POO DANS PYTHON

## 11.6. Les propriétés: (1/2)

- ❑ Il est de préférence de passer par des **propriétés** pour changer les valeurs des attributs.
- ❑ Il existe une convention de passer par des **getter** et des **setter** pour changer la valeur d'un attribut.

Exemple:

```
class Voiture():
    def __init__(self):
        self._roues=4

    def _get_roues(self):
        return self._roues

    def _set_roues(self, v):
        self._roues = v

    roues=property(_get_roues,
set_roues)
```

```
class Voiture():
    def __init__(self):
        self._roues=4

    @property
    def roues(self):
        return self._roues

    @roues.setter
    def roues(self, v):
        self._roues = v
```

# 11) LA POO DANS PYTHON

## 11.6. Les propriétés: (2/2)

Exemple :

```
ma_voiture.roues=10  
print(ma_voiture.roues)
```

## 11.7. L'attribut spécial `__dict__` :

❑ Cet attribut nous donne les **valeurs des attributs de l'instance** sous forme d'un dictionnaire

Exemple :

```
>>> ma_voiture.__dict__  
{'nom': 'Ferrari'}
```



# 11) LA POO DANS PYTHON

## 11.8. L'héritage de classe:

❑ L'héritage permet de **créer de nouvelles classes** mais avec **une base existante**.

Exemple :

```
class Voiture:
    roues = 4
    moteur = 1

    def __init__(self):
        self.nom = " vide "

class VoitureSport(Voiture):

    def __init__(self):
        self.nom = "Ferrari"
```

☑ On a indiqué que **VoitureSport** a hérité de classe **Voiture**, elle récupère donc toutes ses méthodes et ses attributs.

```
>>> ma_voiture=Voiture()
>>> ma_voiture.nom
'vide'
>>> ma_voiture.roues
4
```

```
>>>
ma_voiture_sport=VoitureSport()
>>> ma_voiture_sport.nom
'Ferrari'
>>> ma_voiture_sport.roues
4
```

# 11) LA POO DANS PYTHON

## 11.9. Polymorphisme / surcharge de méthode: (1/2)

❑ Il est possible d'écraser la méthode de la classe parente en la redéfinissant.

- On parle alors de **surcharger une méthode**

Exemple :

```
class Voiture:
    roues = 4
    moteur = 1

    def __init__(self):
        self.nom = "A déterminer"

    def allumer(self):
        print("La voiture
démarre")
```

```
class VoitureSport(Voiture):

    def __init__(self):
        self.nom = "Ferrari"

    def allumer(self):
        print("La voiture de sport
démarre")
```

# 11) LA POO DANS PYTHON

## 11.9. Polymorphisme / surcharge de méthode: (2/2)

- ❑ Il est possible d'appeler la **méthode du parent** puis de faire la spécificité de la méthode.
  - On peut d'ailleurs appeler n'importe quelle autre méthode.

### Exemple :

```
class Voiture:
    roues = 4
    moteur = 1

    def __init__(self):
        self.nom = "vide"

    def allumer(self):
        print("La voiture
démarré")
```

```
class VoitureSport(Voiture):

    def __init__(self):
        self.nom = "Ferrari"

    def allumer(self):
        Voiture.allumer(self)
        print("La voiture de sport
démarré")
```

# 11) LA POO DANS PYTHON

## 11.10. Les classes et les *docstring*:

- ❑ Les classes peuvent bien sûr contenir des **doctrings** comme les fonctions et les modules.

```
class Citron :  
    """ Voici la classe Citron .  
    .....  
    """  
  
    saveur = " acide "  
    def __init__ (self , couleur =" jaune ", taille =" standard "):  
        """ Constructeur de la classe Citron . """  
        self . couleur = couleur  
        self . taille = taille  
  
    def __str__ ( self ):  
        """ Redéfinit le comportement avec print (). """  
        return f"saveur: {self.saveur}, couleur: {self.couleur}, taille:  
{self.taille}"
```

# **INTRODUCTION:** RAPPEL SUR LES NOTION DE BASE

## **SÉANCE 2: TRAVAUX PRATIQUE**

**TP 2:** Éléments du langage Python – La POO