# Formation Django

## Session I


Présenté par : **JADLI AISSAM**

/in/jadli-aissam-86a69843

Initiation au Langage Python

jadliaissam@gmail.com

# Présentation

Python est un langage de programmation interprété multiplateforme dont la première version a été proposée fin des années 1980 par le programmeur néerlandais **Guido Van Rossum**.

Python est un langage de programmation multiparadigme (programmation orientée objet et la programmation structurée). Il utilise le typage dynamique et la gestion de la mémoire automatique (via un garbage collector).

# Pourquoi

- Simplicité et Code moins volumineux

```c
#include <stdio.h>

int main(void) {
    printf("hello, world\n");
}
```

Langage C

```python
print("Bonjour Tout le Monde")
```

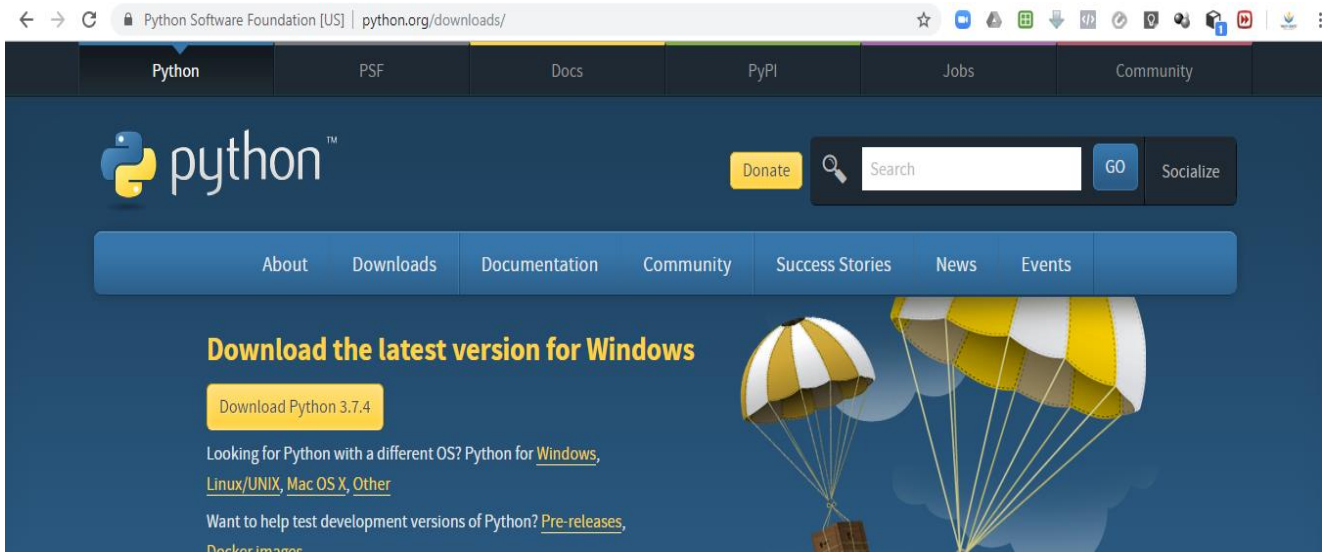
Langage Python

- Très bien documenté
- Grande communauté
- Utilisé partout (Administration système, applications bureau, web, mobile …etc.

# Installation

- Windows et MacOS : Télécharger l'exécutable sur https://www.python.org/downloads/ et installez-le.
- Linux : Exécuter les commandes suivantes dans le terminal avec les droits d'un administrateur.

```
$ sudo apt-get update
$ sudo apt-get install python3.6
```



# IDE


Pro & Community



Visual Studio Code

# Les versions de Python

- Python 3 a été sortie en 2008.
- La syntaxe de Python 3 **n'est pas rétro-compatible.**
- La version 2 existe toujours au sein des systèmes complexes et des systèmes d'exploitation mais **ne sera plus mis à jour et ne recevra plus de mises à jour de sécurité**.
- Les versions actuelles de Python sont 2.7.x et 3.9.x
- Il est possible d'installer plusieurs versions de python sur la même OS.

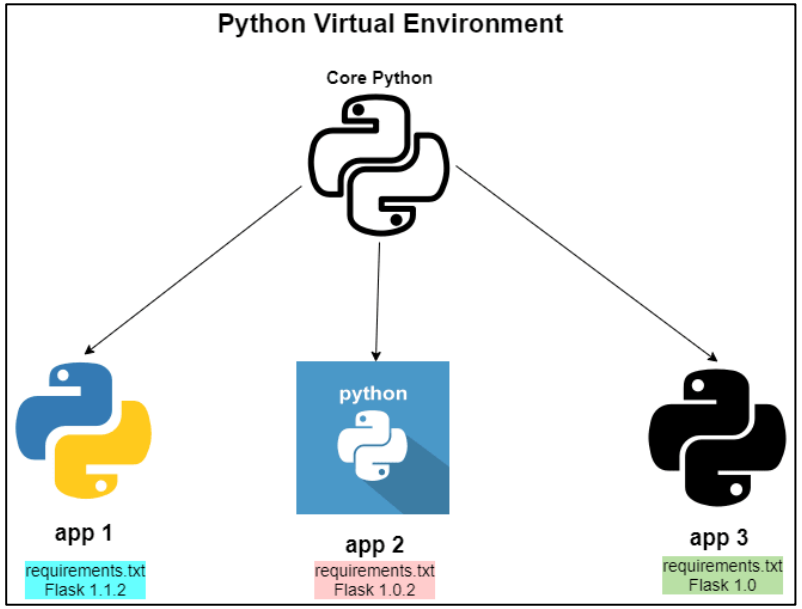
# Problématique

On souhaite travailler sur une machine sur deux projets distincts:

- Un projet utilise la librairie XXXX dans sa dernière version (**2.1**)
- Un projet utilise la librairie XXXX dans une version ancienne (**0.4**) pour des raisons de compatibilité avec un code ancien.

***Question*** : **Comment installer deux version de la même librairie sur le même interpréteur Python ?**

***Solution***: **Utiliser les environnements virtuels !!**



# Les environnements virtuels

La solution à ce problème consiste à créer un **environnement virtuel**, une arborescence de répertoires **autonome** qui contient une **installation Python pour une version particulière de Python**, ainsi qu'un certain nombre de packages supplémentaires.

Différentes applications peuvent ensuite utiliser différents environnements virtuels.

Si le projet B requiert la mise à niveau d'une bibliothèque vers la version 3.0, **cela n'affectera** pas l'environnement du projet A.

# Les gestionnaires d'environnement virtuels

Parmi Les gestionnaire d'environnement virtuels sous python :

- **`virtualenv`**
- **`pipenv`**
- **`conda`**
- **`poetry`**
- **`hatch`**



L'utilisation d'un environnement est composé de **deux étapes**:

- ❖ Activer l'environnement
- ❖ Désactiver l'environnement

# Virtualenv

- Installation

```
pipx install virtualenv
virtualenv --help
```

OU

```
python -m pip install --user virtualenv
python -m virtualenv --help
```

- **Virtualenv** a une seule commande de base :

```
virtualenv venv
```

Ceci va créer un nouveau dossier nommée **venv** (également nom de l'environnement) contenant une nouvelle installation autonome de Python.

Activation : utiliser l'exécutable venv/Scripts/activate

Désactivation : utiliser l'exécutable venv/Scripts/deactivate

# Variables

- Pas besoin de déclarer ou typer explicitement une variable avant de lui affecter une valeur (**Typage Dynamique**). Exemple :

  x = 120 x est de type (**int**)

  y = "Bonjour" y est de type (**str**)

- Le nom d'une variable peut commencer par n'importe lettre minuscule ou majuscule ou un '_', puis des lettres, des chiffres ou des '_'.
- Les noms de variables sont **sensibles à la casse** (age, Age et AGE sont trois variables différentes)
- Une variable sans valeur est définie par : **x = None** (None est l'équivalent de **null** dans d'autres langages).

# Types

| Liste des types basiques | |
|---|---|
| int | Nombre entier optimisé |
| float | Nombre à virgule flottante |
| complex | Nombre complexe |
| str | Chaîne de caractère |
| list | Liste de longueur variable |
| file | Fichier |
| bool | Booléen |
| module | module |

- Une variable définie en dehors de toutes les fonctions est globale.
- par défaut, une variable est toujours locale, donc disponible seulement dans la fonction dans laquelle est elle définie.

**N.B: pour connaître le type d'une variable X, utiliser la fonction type(X)**

# Opérateurs

- Un commentaire commence par le caractère # et s'étend jusqu' à la fin de la ligne

  ***Exemple*** : `>>> print(1+1)  # Ceci est un commentaire`
- Opérateurs logiques: **and**, **or**, **not**
- Opérateurs de comparaison: **>**, **>=**, **==**, **<=**, **<**, **!=**
- Opérations mathématiques: **+**, **-**, ***** , **/** , ****** (puissance) , **//** (division entière) , **%** (modulo).
- Opérations d'affections: **=**, **+=**, **-=**, ***=**, **/=**

# Fonctions E/S

- Python utilise la fonction **`print()`** pour afficher sur la sortie principale (généralement le Terminal ou le script a été lancé)

***Exemple***: `print("Hello World !!")` `print(1+1)` `print(myfunc())`

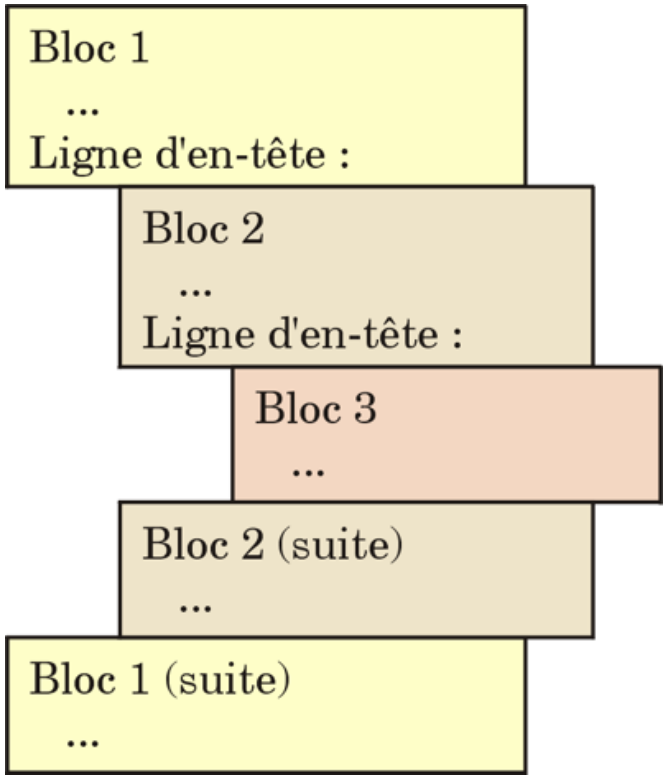
- Pour lire une valeur depuis le Terminal, la fonction utilisée est la fonction **`input()`**

***Exemple***: `num = input("Saisir un numéro de votre choix")`

**N.B: La variable retournée par input() est TOUJOURS de type str.**

# Notion de Bloc

- Un bloc d'instructions est une suite d'instructions qui est alignée sur la même tabulation.
- Python utilise **l'indentation** tandis que d'autres langages délimitent un bloc par les symboles {} , ou BEGIN - END
- Les blocs sont utilisés pour définir les corps des **fonctions**, des **boucles**, des **classes** ...




**N.B: Les variables définis dans un bloc sont locales pour ce bloc et ne sont accessible qu'à l'intérieur.**

# Les Conditions

L'instruction if permet de réaliser un traitement conditionnel selon une condition booléenne.

```
if condition_1:
        BLOC 1
elif condition_2:
        # Equivalent à else if
        BLOC 2
else:
        BLOC 3
```

```
x=0
if x < 0:
 print('negative')
elif x == 0:
 print('zero')
else:
 print('positive')
```

```
zero
```

Les conditions peuvent aussi être utilisés **inline** :

```
A = value1 if condition else value2
```

# Les Boucles : **for**

L'instruction **for** permet de répéter un traitement un nombre défini de fois soit en utilisant la fonction **range()** soit en itérant sur une liste.

```
for i in range (1,11) :
    if i % 2 == 0 :
        print ("Le nombre ", i, "est pair")
    else :
        print ("Le nombre ", i, "est impair")
```

```
Le nombre  1 est impair
Le nombre  2 est pair
Le nombre  3 est impair
Le nombre  4 est pair
Le nombre  5 est impair
Le nombre  6 est pair
Le nombre  7 est impair
Le nombre  8 est pair
Le nombre  9 est impair
Le nombre  10 est pair
```

Pour i dans l'intervalle de 1(**inclus**) à 11(**exclus**) afficher si i est pair ou impair.

la variable i est déclarée dans la boucle for et prend toutes les valeurs à chaque tour.

# Les Boucles : **while**

L'instruction **while** permet de répéter un traitement tant que la condition est évaluée à **True**.

```
while condition:
      INSTRUCTIONS
      INSTRUCTIONS
      INSTRUCTIONS
```

```
x=5
while x < 10:
 print(x)
 x+=1
```

```
5
6
7
8
9
```

# break et continue

L'instruction break permet d'arrêter une boucle avant sa fin.

```
while condition:
    if condition2:
       break  # si condition2 est vérifiée
              # on passe à instruction1
    BLOC D'INSTRUCTIONS
instruction 1
```

```
while 1: # 1 est toujours vrai -> boucle infinie
 lettre = input (" Tapez 'Q' pour quitter : ")
 if lettre == "Q":
  print (" Fin de la boucle ")
  break
```

```
Tapez 'Q' pour quitter :  1
Tapez 'Q' pour quitter :  3
Tapez 'Q' pour quitter :  Q
Fin de la boucle
```

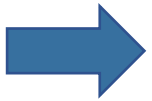
L'instruction continue est similaire, mais au lieu d'interrompre la boucle, on saute à la prochaine itération de la boucle.

# Les Listes

Les listes sont utilisées pour stocker plusieurs éléments dans une seule variable.

Les listes sont créées à l'aide de crochets.

```
thislist = ["apple", "banana", "cherry"]
print(thislist)
```



```
['apple', 'banana', 'cherry']
```

Les éléments de liste sont **ordonnées**, **changeables**, permettent des **valeurs en double** et **indexés**, le premier élément a l'index **[0]**, le deuxième élément a l'index **[1]** …etc.

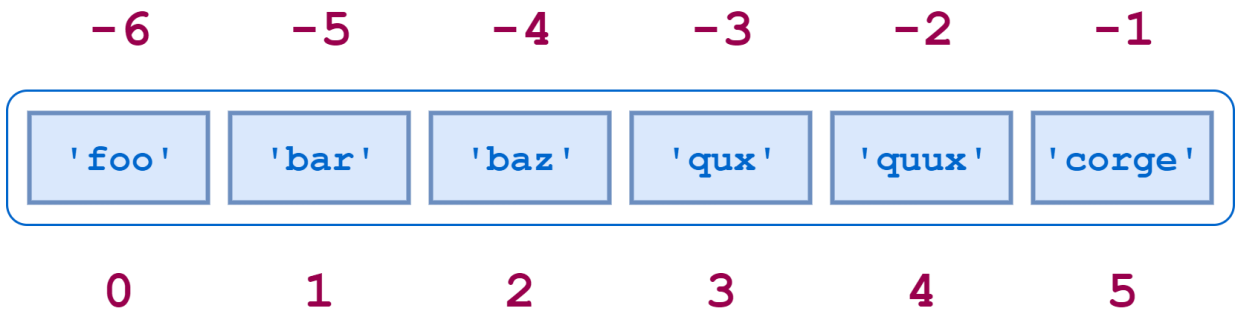
# Ordre

Lorsque nous disons que les listes sont ordonnées, cela signifie que les éléments ont un ordre défini, et que l'ordre ne changera pas.

Les nouveaux éléments seront placés à la fin de la liste.

Pour connaître la taille d'une liste, utiliser la méthode : **len(NOM_LISTE)**

```
thislist = ["apple", "banana", "cherry"]
thislist[1] = "blackcurrant"
print(thislist)
```



# Slicing

Vous pouvez spécifier une plage d'index en spécifiant où commencer et où terminer la plage (**Slicing**) :

**[ START : END : STEP ]**

N.B: Lors de la spécification d'une plage, la valeur de retour sera une nouvelle liste avec les éléments spécifiés.

List = [ 0, 1, 2, 3, 4, 5]

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

List[0] = 0

List[1] = 1

List[2] = 2

List[3] = 3

List[4] = 4

List[5] = 5

List[0:] = [0,1,2,3,4,5]

List[:] = [0,1,2,3,4,5]

List[2:4] = [2, 3]

List[1:3] = [1, 2]

List[:4] = [0, 1, 2, 3]

# Recherche

Vous pouvez rechercher si une valeur existe dans une liste en utilisant le mot-clé **in**. Pour récupérer l'index de la **première occurrence**, on utilise la méthode **index()**.

La méthode **count()** permet de connaître le nombre d'occurrence d'une valeur dans une liste.

```
>>> mylist = [12, 4, "ABX", True, 4]
>>> 4 in mylist
True
>>> 10 in mylist
False
>>> mylist.index(4)
1
>>> mylist.count(4)
2
>>> mylist.count(10)
0
>>> mylist.index(4)
1
>>> mylist.index(10)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: 10 is not in list
```

# Manipulation

Pour modifier la valeur des éléments dans une plage spécifique, définissez une liste avec les nouvelles valeurs et reportez-vous à la plage de numéros d'index dans laquelle vous souhaitez insérer les nouvelles valeurs.

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]
thislist[1:3] = ["blackcurrant", "watermelon"]
print(thislist)
```

Pour ajouter un élément à la fin de la liste, utilisez la méthode **append()** :

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
```

# Manipulation

Pour insérer un élément de liste à un index spécifié, utilisez la méthode **insert().**

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)
```

Pour ajouter des éléments d' une autre liste à la liste actuelle, utilisez la méthode **extend()**, ou simplement l'addition classique **list1 + list2.**

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
```

# Manipulation

Pour supprimer un élément d'une liste, on peut également :

- Utiliser La méthode **remove()** pour supprime l'élément spécifié en se basant sur **sa valeur**.
- Utiliser le mot clé **del** pour supprimer **par index**
- Utiliser la méthode **pop()** qui permet de supprimer une élément par **son index** et **retourne sa valeur**

```
>>> mylist = [12, 4, "ABX", True, 4]
>>> mylist.remove(12)
>>> mylist
[4, 'ABX', True, 4]
>>> del mylist[1]
>>> mylist
[4, True, 4]
>>> mavar = mylist.pop(2)
>>> mylist
[4, True]
>>> mavar
4
```

# Tri

Les objets de liste ont une méthode **sort()** qui triera la liste de manière alphanumérique, ascendante, par défaut.

```
thislist = ["orange", "mango", "kiwi", "pineapple", "banana"]
thislist.sort()
```

```
['banana', 'kiwi', 'mango', 'orange', 'pineapple']
```

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
```

```
[23, 50, 65, 82, 100]
```

Pour trier par **ordre décroissant**, utilisez l'argument mot-clé **reverse = True**

```
thislist = [100, 50, 65, 82, 23]
thislist.sort(reverse = True)
```

# Tri

La méthode **reverse()** inverse l'ordre de tri actuel des éléments.

```
thislist = ["banana", "Orange", "Kiwi", "cherry"]
thislist.reverse()
print(thislist)
```

Vous pouvez également personnaliser votre propre fonction en utilisant l'argument **key** (La fonction renverra un nombre qui sera utilisé pour trier la liste) (le plus petit nombre en premier).

```
def myfunc(n):
  return abs(n - 50)

thislist = [100, 50, 65, 82, 23]
thislist.sort(key = myfunc)
print(thislist)
```

# Copie

Vous ne pouvez pas copier une liste simplement en tapant **list2 = list1** car list2 ne sera qu'une **référence** à list1, et les modifications apportées dans list1 seront automatiquement également apportées dans list2.

```
>>> list1 = [1, 10, "Hello"]
>>> list2 = list1
>>> list2[2] = 30
>>> list2
[1, 10, 30]
>>> list1
[1, 10, 30]
```

Il existe plusieurs moyens de faire une copie :

- la méthode intégrée **copy()**
- La méthode **list()**
- L'addition avec une liste vide …

```
thislist = ["apple", "banana", "cherry"]
mylist = thislist.copy()
print(mylist)
```

# List Comprehension

La compréhension de liste offre une syntaxe plus courte lorsque vous souhaitez créer une nouvelle liste basée sur les valeurs d'une liste existante.

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)
```

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]

newlist = [x for x in fruits if "a" in x]

print(newlist)
```

```
newlist = [expression for item in iterable if condition == True]
```

# List Comprehension

## Exemples

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
```

```
newlist = [x for x in fruits if x != "apple"]
```

```
newlist = [x.upper() for x in fruits]
```

```
newlist = [x for x in range(10) if x < 5]
```

```
newlist = ['hello' for x in fruits]
```

```
newlist = [x for x in fruits]
```

```
newlist = [x for x in range(10)]
```

```
newlist = [x if x != "banana" else "orange" for x in fruits]
```

# Unpacking

en Python, nous sommes également autorisés à extraire les valeurs d'une liste dans des variables.

C'est ce qu'on appelle le « **Déballage** » (**Unpacking** en Anglais).

```
fruits = ["apple", "banana", "cherry"]

green, yellow, red = fruits

print(green)
print(yellow)
print(red)
```

**N.B: Le nombre de variables doit correspondre au nombre de valeurs de la liste, sinon, vous devez utiliser un astérisque pour collecter les valeurs restantes sous forme de liste.**

# Boucles

Vous pouvez parcourir les éléments de la liste en utilisant une boucle **for** classique.

```
thislist = ["apple", "banana", "cherry"]
for x in thislist:
  print(x)
```

Vous pouvez également parcourir les éléments de la liste et récupérer à la fois l'élément et son index en utilisant la méthode **enumerate()**

```
thislist = ["apple", "banana", "cherry"]
for (index, item) in enumerate(thislist):
  print(item)
```

# Méthodes Utiles

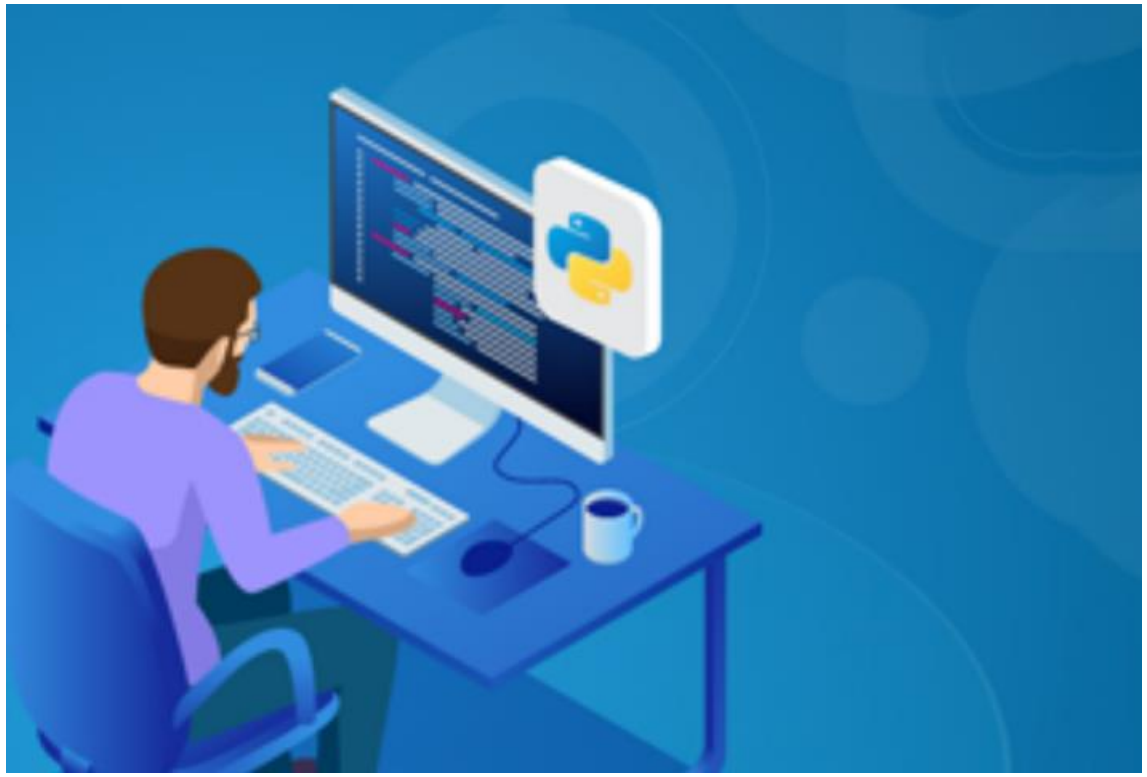
Il existe quelques méthodes qui permettent d'effectuer des opérations utiles sur les listes.

Par Exemple:

- **sum()**
- **max()**
- **min()**
- **statistics.mean()**
- **random.choice()**

```
>>> import statistics
>>> import random
>>> mylist = [100, 0, 3, 19, 23, 46, 4]
>>> sum(mylist)
195
>>> min(mylist)
0
>>> max(mylist)
100
>>> statistics.mean(mylist)
27.857142857142858
>>> random.choice(mylist)
23
>>> random.choice(mylist)
4
```

# TRAVAUX PRATIQUES



# Les Tuples

Un tuple est une collection ordonnée et immuable et acceptant les valeurs en double .

Les tuples sont écrits entre parenthèses.

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])
```

Toutes les propriétés des liste en terme d'indexation, recherche et Slicing sont valides avec les Tuples (ainsi que les méthodes **count()** et **index()**).

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])
```

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

# Manipulation

Les tuples ne sont pas modifiables, ce qui signifie que vous ne pouvez pas modifier, ajouter ou supprimer des éléments une fois le tuple créé.

Mais il existe quelques solutions de contournement.

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)
```

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
```

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
```

# Les Ensembles (Sets)

Un ensemble (**Set**) est une collection à la fois non ordonnée et non indexée .

Les ensembles sont écrits avec des accolades.

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Les ensembles **ne sont pas modifiables**, ce qui signifie que nous ne pouvons pas modifier les éléments une fois l'ensemble créé.

# Accès et Recherche

Vous **ne pouvez pas** accéder aux éléments d'un ensemble en vous référant à un index ou à une clé.

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

Vous pouvez parcourir les éléments de l'ensemble en utilisant une boucle **for**, ou demander si une valeur spécifiée est présente dans un ensemble, en utilisant le mot-clé **in**.

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

# Manipulation

Une fois qu'un ensemble est créé, vous ne pouvez pas modifier ses éléments, mais vous pouvez ajouter de nouveaux éléments avec la méthode **add().**

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

Pour ajouter des éléments d'un autre ensemble dans l'ensemble actuel, utilisez la méthode **update()**.

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)
```

```
thisset = {"apple", "banana", "cherry"}
mylist = ["kiwi", "orange"]

thisset.update(mylist)
```

# Manipulation

Pour supprimer un élément d'un ensemble, utilisez la méthode **remove()** ou **pop()** ou **discard()**.

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")
```

N.B: si l'élément à supprimer n'existe pas, **remove()** génère une erreur tandis que **discard()** ne génère aucune erreur.

```
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
```

# Les Dictionnaires

Les dictionnaires sont utilisés pour stocker les valeurs de données dans des paires : **clé-valeur**.

Un dictionnaire est une collection de données qui est **ordonnée***, **modifiable** et **n'autorise pas les doublons**.

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
print(thisdict["brand"])
```

Les valeurs en double remplaceront les valeurs existantes:

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964,
  "year": 2020
}
print(thisdict)
```

*Depuis Python 3.7

# Méthodes

Il existe une méthode appelée **get()** qui donne la valeur en se basant sur la clé.

La méthode **keys()** retournera une liste de toutes les clés du dictionnaire.

La méthode **values()** retournera une liste de toutes les valeurs du dictionnaire.

La méthode **items()** retournera les éléments d'un dictionnaire, sous forme de tuples.

```
>>> thisdict = {
...     "model": "Clio",
...     "year": 2020,
...     "brand": "Renault"
... }
>>> thisdict.keys()
dict_keys(['model', 'year', 'brand'])
>>> thisdict.values()
dict_values(['Clio', 2020, 'Renault'])
>>> thisdict.items()
dict_items([('model', 'Clio'), ('year', 2020),
>>> thisdict.get('model')
'Clio'
>>> thisdict.get('modell')
>>> thisdict.get('modell', "Default")
'Default'
```

# Manipulation

Vous pouvez modifier la valeur d'un élément spécifique en vous référant à sa clé.

La méthode **update()** mettra à jour le dictionnaire avec les éléments de l'argument donné.

Pour supprimer des éléments d'un dictionnaire :

- **del**
- **pop()**

```
thisdict = {
  "brand": "Ford",
  "model": "Mustang",
  "year": 1964
}
thisdict["year"] = 2018
```

```
thisdict.update({"year": 2020})
```

```
del thisdict["model"]
```

```
thisdict.pop("model")
```

# Manipulation

Lorsque vous parcourez un dictionnaire en boucle, les valeurs de retour sont les **clés du dictionnaire**, mais on peut également travailler sur les valeurs en utilisant la méthode **`values()`**.

```
>>> thisdict = {
...     "model": "Clio",
...     "year": 2020,
...     "brand": "Renault"
... }
>>> for i in thisdict:
...     print(i)
...
model
year
brand
>>> for i in thisdict.values():
...     print(i)
...
Clio
2020
Renault
>>> for k,v in thisdict.items():
...     print(k,v)
...
model Clio
year 2020
brand Renault
```

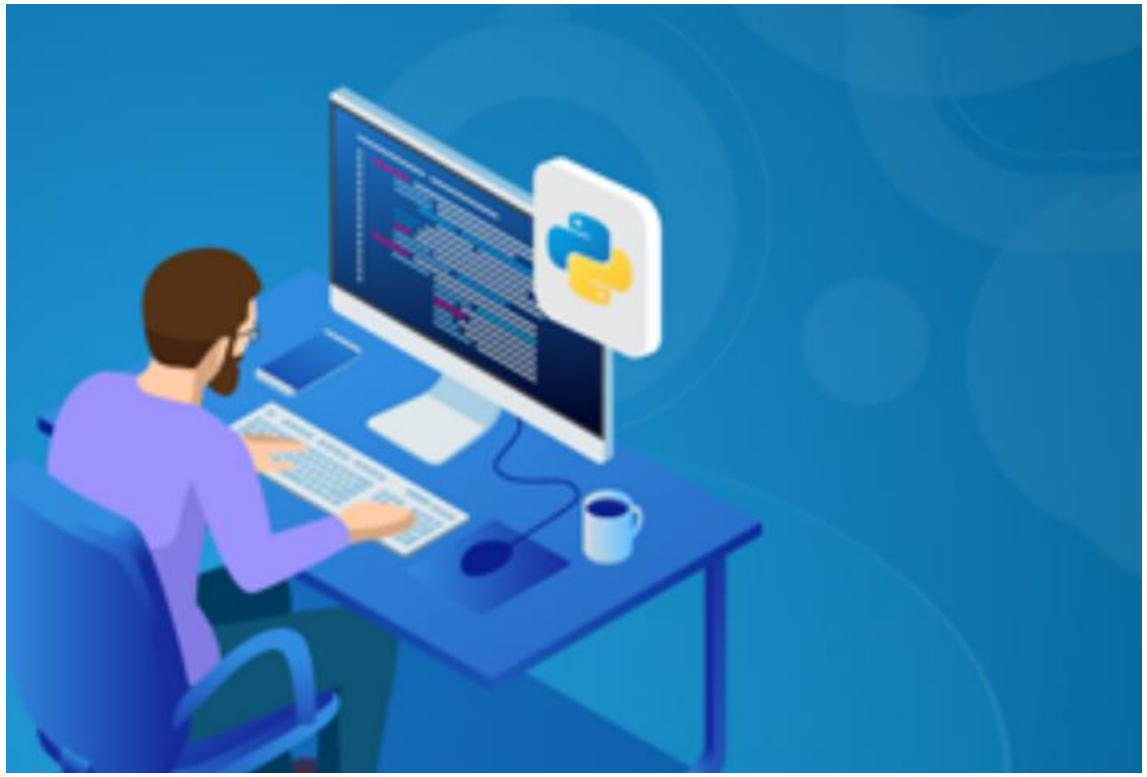
# Copie

Vous ne pouvez pas copier un dictionnaire simplement avec **dict2 = dict1**, car **dict2** ne sera qu'une référence à **dict1**, et les modifications apportées dans **dict1** seront automatiquement également apportées dans **dict2**.

Il existe des moyens de faire une copie, l'un d'entre eux consiste à utiliser la méthode intégrée **copy()**.

```
>>> thisdict = {
...     "model": "Clio",
...     "year": 2020,
...     "brand": "Renault"
... }
>>> mydict = thisdict
>>> mydict["year"] = 2021
>>> mydict
{'model': 'Clio', 'year': 2021, 'brand': 'Renault'}
>>> thisdict
{'model': 'Clio', 'year': 2021, 'brand': 'Renault'}
>>> mydict2 = thisdict.copy()
>>> mydict2['year'] = 2022
>>> mydict2
{'model': 'Clio', 'year': 2022, 'brand': 'Renault'}
>>> thisdict
{'model': 'Clio', 'year': 2021, 'brand': 'Renault'}
```

# TRAVAUX PRATIQUES



# Chaînes de Caractères

Comme beaucoup d'autres langages de programmation populaires, les chaînes en Python sont des tableaux d'octets représentant des caractères Unicode.

```
a = "Hello, World!"
print(a[1])
```

Pour obtenir la longueur d'une chaîne, utilisez la fonction **len()**

```
a = "Hello, World!"
print(len(a))
```

Pour vérifier si une certaine phrase ou caractère est présent dans une chaîne, nous pouvons utiliser le mot-clé **in**.

```
txt = "The best things in life are free!"
print("free" in txt)
```

# Manipulation des Strings

Python a un ensemble de méthodes intégrées que vous pouvez utiliser sur des chaînes.

La méthode **upper()** renvoie la chaîne en majuscules.

```
a = "Hello, World!"
print(a.upper())
```

La méthode **lower()** renvoie la chaîne en minuscules.

```
a = "Hello, World!"
print(a.lower())
```

La méthode **strip()** supprime tous les espaces du début ou de la fin.

```
a = " Hello, World! "
print(a.strip()) # returns
```

La méthode **replace()** remplace une chaîne par une autre chaîne

```
a = "Hello, World!"
print(a.replace("H", "J"))
```

# Manipulation des Strings

La méthode **split()** renvoie une liste dans laquelle le texte entre le séparateur spécifié devient les éléments de la liste. L'inverse est la méthode **join()**.

```
a = "Hello"
b = "World"
c = a + b
print(c)
```

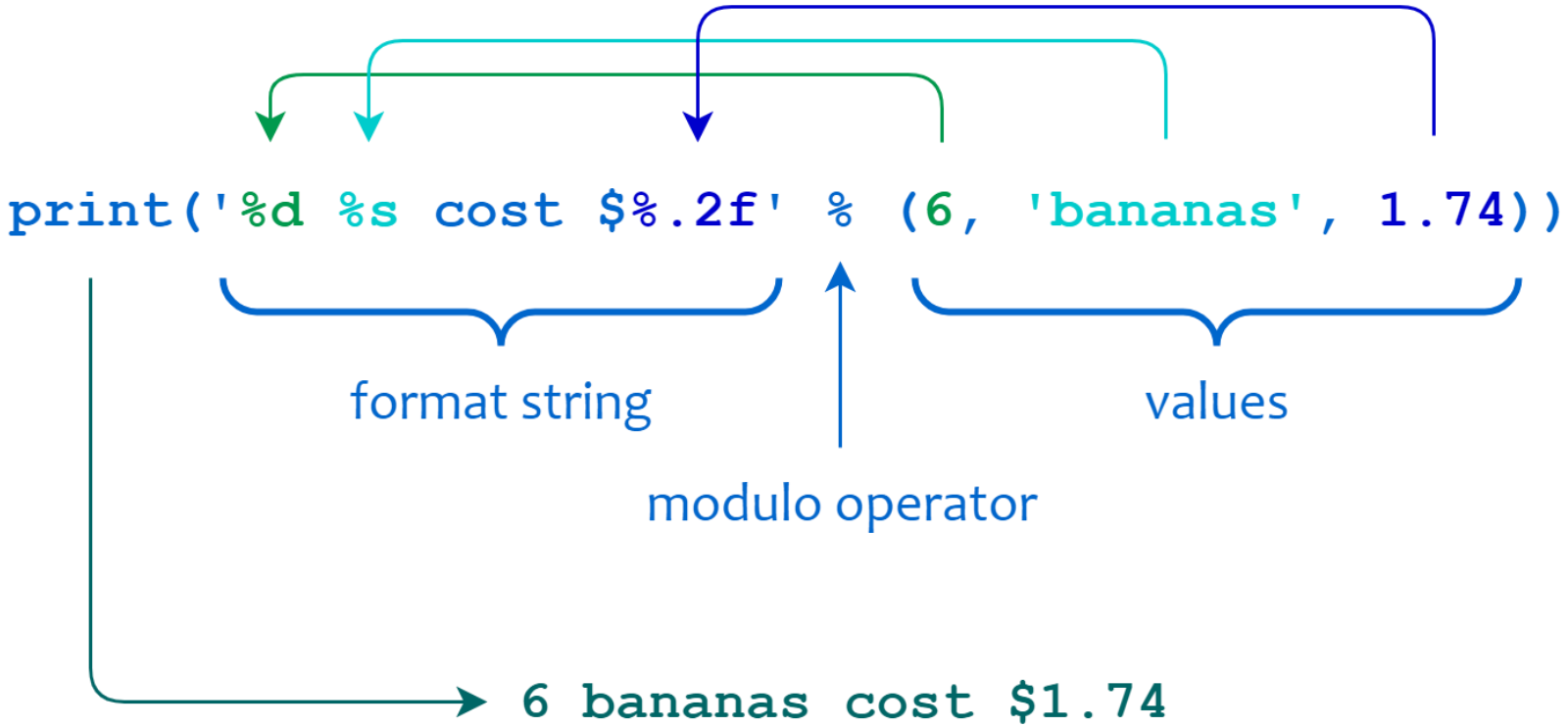
Pour concaténer ou combine deux chaînes ou plus, utiliser l'opérateur **+**

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

**N.B : La concaténation entre string et nombre donne une erreur.**

# Formatage

Utilisez l'opérateur modulo (**%**) pour insérer des nombres dans des chaînes.



# Formatage

Utilisez la méthode **format()** pour insérer des nombres dans des chaînes.

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

Vous pouvez utiliser des numéros d'index **{0}** pour vous assurer que les arguments sont placés dans les bons espaces réservés.

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

# Formatage

Dans Python, f-strings est un mécanisme de mise en forme de chaînes qui permet d'intégrer des expressions Python dans des chaînes de caractères.

```
name = "Emily"
age = 22

# Old % python string formatting, that isn't very clear:
print("My name is %s and I am %i years old." % (name, age))

# .format() python string formatting, which is newer & more user friendly
print("My name is {} and I am {} years old.".format(name, age))

# f-string formatting, which is very clear and was introduced in Python 3.6:
print(f"My name is {name} and I am {age} years old.")
```

```
executed in 5ms, finished 13:24:03 2019-09-12
My name is Emily and I am 22 years old.
My name is Emily and I am 22 years old.
My name is Emily and I am 22 years old.
```

# Les Fonctions

Une fonction est un bloc de code qui ne s'exécute que lorsqu'elle est appelée.

Vous pouvez transmettre des données, appelées paramètres, à une fonction.

Une fonction peut renvoyer des données en conséquence.

En Python, une fonction est définie à l'aide du mot-clé **def** :

```
def my_function():
  print("Hello from a function")

my_function()
```

# Arguments (args)

Les informations peuvent être transmises aux fonctions en tant qu'arguments.

Par défaut, une fonction doit être **appelée avec le nombre correct d'arguments**. Cela signifie que si la fonction attend 2 arguments, elle doit être appelée avec 2 arguments, pas plus, ni moins.

```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

# Arguments arbitraires (*args)

Si le nombre d'arguments de la fonction est inconnus on utilise le symbole «*» avant le nom du paramètre dans la définition de la fonction.

De cette façon, la fonction recevra **un tuple d'arguments** et pourra accéder aux éléments en conséquence:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

# Arguments de mots-clés (kwargs)

On peut également envoyer des arguments avec la syntaxe **clé = valeur** .

De cette façon, l'ordre des arguments n'a pas d'importance.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

# Arguments arbitraires de mots-clés (**kwargs)

Si le nombre d'arguments de la fonction est inconnus on utilise le symbole «**» avant le nom du paramètre dans la définition de la fonction.

De cette façon, la fonction recevra un **dictionnaire d'arguments** et pourra accéder aux éléments en conséquence:

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

# Valeurs par Défaut

On peut spécifier une valeur par défaut pour un argument donné. Cette valeur sera utilisée si aucune valeur de cet argument n'est fournie.

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

N.B : Les arguments à valeurs par défaut doivent être obligatoirement **placé en dernier**.

# Valeur de retour

Pour laisser une fonction renvoyer une valeur, utilisez l' instruction **return** :

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

N.B : Si aucune valeur n’est retournée par la fonction, La valeur **None** est retournée par défaut.

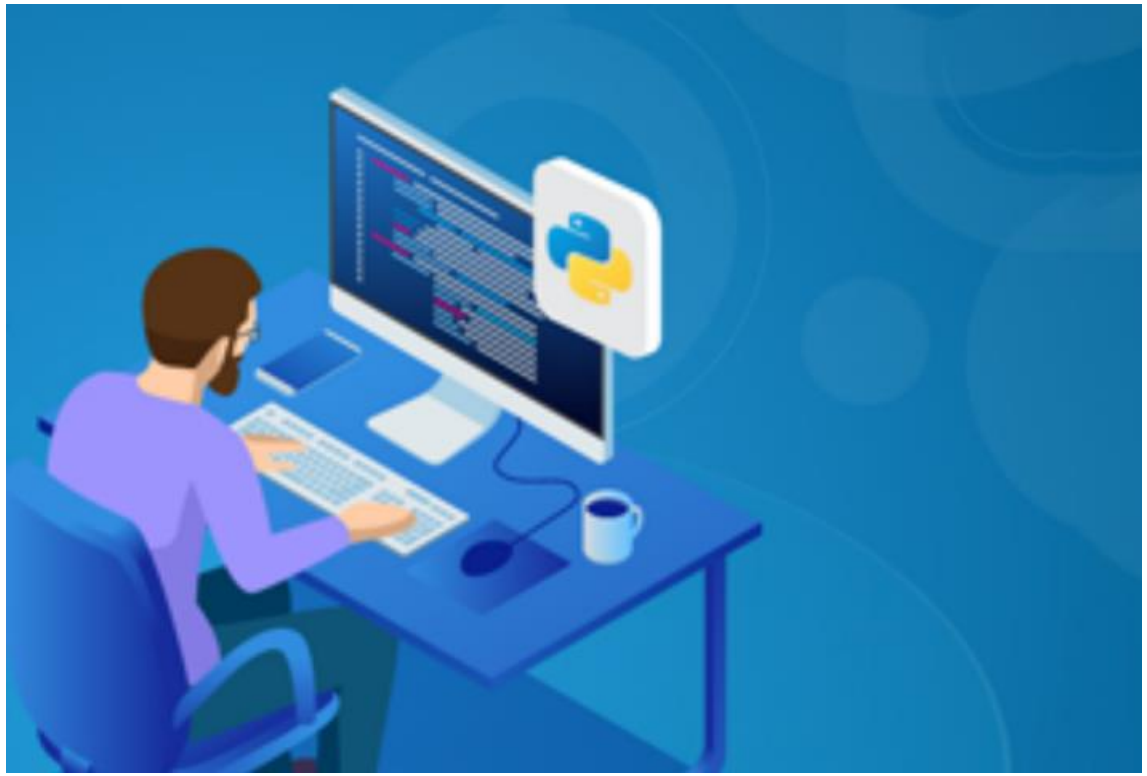
# Les fonctions Lambda

Une fonction lambda est une petite fonction anonyme qui peut prendre n'importe quel nombre d'arguments, mais **ne peut avoir qu'une seule expression**.

Utilisez les fonctions lambda lorsqu'une fonction anonyme est requise pendant une courte période.

```
>>> liste = [12, 34, 40, 1]
>>> def increment(x):
...     return x + 1
...
>>> liste2 = map(increment, liste)
>>> list(liste2)
[13, 35, 41, 2]
>>> liste3 = map(lambda x:x+1, liste)
>>> list(liste3)
[13, 35, 41, 2]
```

# TRAVAUX PRATIQUES



# Les modules

Un module est identique à une bibliothèque de codes.

Un fichier contenant un ensemble de fonctions que vous souhaitez inclure dans un autre script/application est considéré également comme un module.

```
def greeting(name):
  print("Hello, " + name)
```

mymodule.py

```
import mymodule

mymodule.greeting("Jonathan")
```

```
from mymodule import person1

print (person1["age"])
```

# Les modules

Vous pouvez créer un alias lorsque vous importez un module, en utilisant le mot clé « **as** ».

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

Il existe une fonction intégrée nommé **dir()** pour lister tous les noms de fonctions (ou noms de variables) dans un module.

```
import platform

x = dir(platform)
print(x)
```

```
['DEV_NULL', '_UNIXCONFDIR', 'WIN32_CLIENT_RELEASES', 'WIN32_SERVER_RELEASES', '__builtins__', '__cached__', '__copyright__', '__doc__', '__file_
```

# PIP

PIP est un gestionnaire de packages pour les packages/ modules Python.

Pour vérifiez si PIP est installé

```
pip --version
```

Pour téléchargez un package

```
pip install camelcase
```

Pour supprimer un package

```
pip uninstall camelcase
```

Pour lister les packages

```
pip list
```

```
Package         Version
----------------------
camelcase       0.2
mysql-connector 2.1.6
pip             18.1
pymongo         3.6.1
setuptools      39.0.1
```

# Gestion des Fichiers

La gestion des fichiers est une partie importante de toute application Web/Desktop

Python a plusieurs fonctions pour créer, lire, mettre à jour et supprimer des fichiers.

La fonction clé pour travailler avec des fichiers en Python est la fonction **open()**.

La fonction **open()** prend deux paramètres :

- Nom ou chemin de fichier
- Mode d'ouverture .

```
f = open("demofile.txt", "rt")
```

# Modes d'ouverture

"r" - Lire - Valeur par défaut. Ouvre un fichier en lecture, erreur si le fichier n'existe pas

"a" - Ajouter - Ouvre un fichier à ajouter, crée le fichier s'il n'existe pas

"w" - Ecrire - Ouvre un fichier pour l'écriture, crée le fichier s'il n'existe pas

"x" - Créer - Crée le fichier spécifié, renvoie une erreur si le fichier existe

"t" - Texte - Valeur par défaut. Mode texte

"b" - Binaire - Mode binaire (par exemple images)

# Lecture de Fichier

La fonction **open()** renvoie un objet fichier, qui a une méthode **read()** pour lire le contenu du fichier.

Vous pouvez renvoyer une ligne en utilisant la méthode **readline().**

En parcourant le fichier avec une boucle **for**, vous pouvez lire l'intégralité du fichier, ligne par ligne.

Il est recommandé de toujours **fermer le fichier** à la fin.

```
f = open("demofile.txt", "r")
print(f.read())
```

```
f = open("demofile.txt", "r")
print(f.readline())
```

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

# Ecriture dans un Fichier

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

```
Hello! Welcome to demofile2.txt
This file is for testing purposes.
Good Luck!Now the file has more content!
```

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

```
Woops! I have deleted the content!
```

# Suppression d'un Fichier

Pour supprimer un fichier, vous devez importer le module OS et exécuter la fonction **os.remove().**

```
import os
os.remove("demofile.txt")
```

N.B: La suppression d'un fichier inexistant soulève une exception.

Pour supprimer un dossier entier, utilisez la méthode **os.rmdir()** ou **os.removedirs()**

```
import os
os.rmdir("myfolder")
```

N.B: vous ne pouvez supprimer que les dossiers vides .

# Manipulation des Fichiers

Utiliser le module **shutil** pour :

- Copier un Fichier : **shutil.copy()**
- Déplacer un Fichier: **shutil.move()**

Utiliser le module **path** pour:

- Vérifier l'existence d'un fichier/dossier : **exists**, **isfile()**, **isdir()**, …etc.
- Toutes les opérations concernant les chemins de fichiers : **abs()**, **dirname()**, …etc.

# Manipulation des Fichiers

Lister le contenu d'un dossier en utilisant la méthode **os.listdir().**

Créer un dossier en utilisant **os.mkdir()** ou **os.makedirs().**

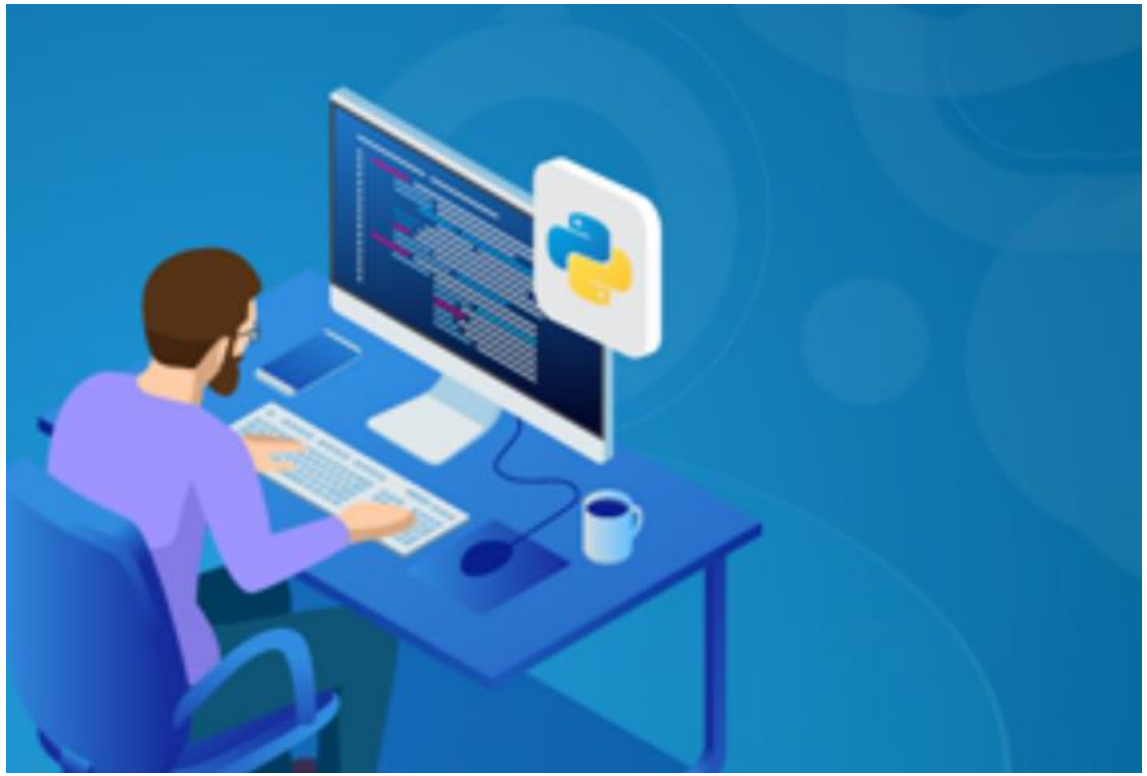
Renommer un fichier en utilisant la méthode **os.rename().**

```
import os


items = os.listdir(".")


data = []
for names in items:
    if names.endswith(".json"):
        data.append(names)
print(data)
```

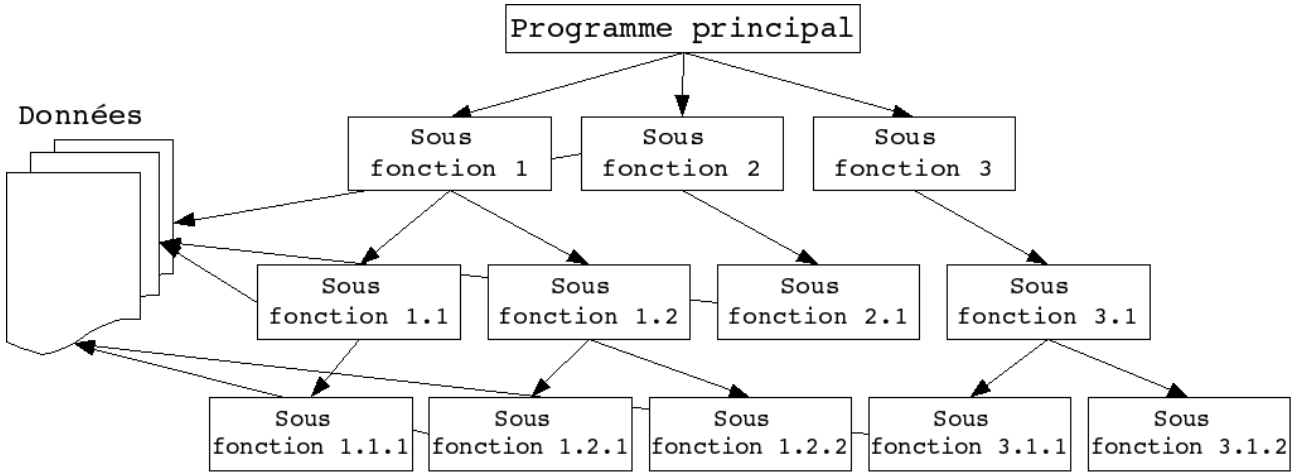
# TRAVAUX PRATIQUES



# La Programmation Structurée

La programmation fonctionnelle (également qualifiées de structurées) met en évidence **les fonctions à assurer** et proposent une approche hiérarchique **descendante** et **modulaire**.

L'approche fonctionnelle dissocie le problème de la représentation des données, du problème du traitement de ces données.



# La Programmation Orientée Objet

La P.O.O considère le logiciel comme une **collection d'objets dissociés, identifiés et possédant des caractéristiques**.

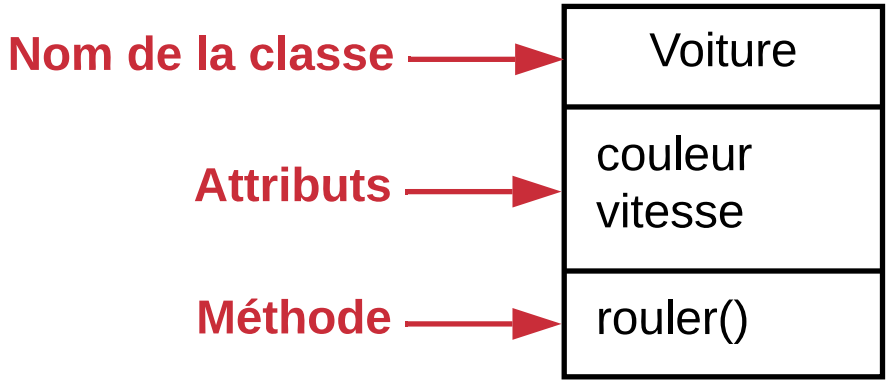
Une caractéristique est soit un **attribut** (i.e. une donnée caractérisant l'état de l'objet), soit une **entité comportementale** de l'objet (i.e. une fonction). La fonctionnalité du logiciel émerge alors de l'interaction entre les différents objets qui le constituent.

L'une des particularités de cette approche est qu'elle **rapproche les données et leurs traitements** associés au sein d'un unique objet.

# Caractéristiques de la P.O.O

## ❑ Notion de classe

Une classe est un type de données abstrait qui précise des caractéristiques (**attributs et méthodes**) communes à toute une famille d'objets et qui permet de créer (instancier) des objets possédant ces caractéristiques.



# Caractéristiques de la P.O.O

## ❑ Notion d'objet

On dit qu'un objet est une instance de classe. Nous pouvons donc avoir plusieurs objets pour une même classe. Chacun des objets a des valeurs qui lui sont propres pour les attributs.

```
class Point:
    "Definition d'un point geometrique"
```

```
>>> p = Point()
```

```
>>> print(p)
<__main__.Point instance at 0x012CAF30>
```

# Caractéristiques de la P.O.O

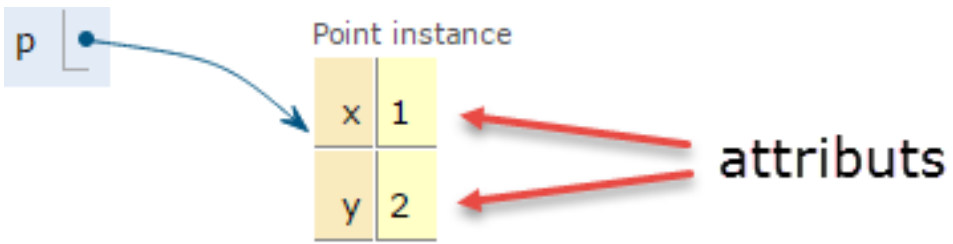
## ❑ Définition des attributs

En Python les attributs peuvent être ajoutés **dynamiquement** même après l'instanciation de l'objet.

Pour accéder à un attribut, on utilise la variable qui contient la référence à l'objet et on met un point « . » puis le nom de l'attribut.

```
class Point:
    "Definition d'un point geometrique"

p = Point()
p.x = 1
p.y = 2
print("p : x =", p.x, "y =", p.y)
```



# Caractéristiques de la P.O.O

## ❑ Définition des méthodes

Pour définir une méthode, il faut :

- indiquer son nom (ici **deplace()**).
- indiquer les arguments entre des parenthèses.

**N.B: Le premier argument d'une méthode doit être obligatoirement self**.

```
class Point:
    def deplace(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

a = Point()
a.x = 1
a.y = 2
print("a : x =", a.x, "y =", a.y)
a.deplace(3, 5)
print("a : x =", a.x, "y =", a.y)
```

# Caractéristiques de la P.O.O

## ❑ Le Constructeur

Un constructeur est une méthode, sans valeur de retour, qui porte un nom imposé par le langage Python : **__init__()**.

Cette méthode sera appelée lors de **la création de l'objet**. Le constructeur peut disposer d'un nombre quelconque de paramètres, éventuellement aucun.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def deplace(self, dx, dy):
        self.x = self.x + dx
        self.y = self.y + dy

a = Point(1, 2)
b = Point(3, 4)
print("a : x =", a.x, "y =", a.y)
print("b : x =", b.x, "y =", b.y)
a.deplace(3, 5)
b.deplace(-1, -2)
print("a : x =", a.x, "y =", a.y)
print("b : x =", b.x, "y =", b.y)
```

# Caractéristiques de la P.O.O

## ❑ La notion d'encapsulation

Le concept d'encapsulation est un concept très utile de la POO. Il permet en particulier **d'éviter une modification par erreur des données d'un objet**.

En effet, il n'est alors pas possible d'agir directement sur les données d'un objet; il est nécessaire de passer par ses méthodes qui jouent le rôle **d'interface obligatoire**.

# Caractéristiques de la P.O.O

## ❑ Définition d'attributs privés

On réalise la protection des attributs de notre classe Point grâce à l'utilisation d'attributs privées. Pour avoir des attributs privés, leur nom doit débuter par __ (double underscore).

**Il n'est alors plus possible de faire appel aux attributs privés depuis l'extérieur de la classe.**

```
class Point:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
```

```
>>> p = Point(1, 2)
>>> p.__x

Traceback (most recent call last):
  File "<pyshell#9>", line 1, in
    p.__x
AttributeError: Point instance has no attribute '__x'
```

# Caractéristiques de la P.O.O

## ❑ Accesseurs et mutateurs

Il faut disposer de méthodes qui vont permettre par exemple de modifier ou d'afficher les informations associées aux variables privées (accesseurs et mutateurs).

```
class Point:
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def deplace(self, dx, dy):
        self.__x = self.__x + dx
        self.__y = self.__y + dy

    def affiche(self):
        print("abscisse =", self.__x, "ordonnee =", self.__y)

a = Point(2, 4)
a.affiche()
a.deplace(1, 3)
a.affiche()
```

```
    def get_y(self):
        return self.__y

    def set_x(self, x):
        self.__x = x
```

# Caractéristiques de la P.O.O

## ❑ Attributs et méthodes de classe

Parfois il est utile de **partager des attributs et méthodes entre toutes les instances de la même classe**. Dans ce cas, il serait préférable d'utiliser des attributs et méthodes de classe.

```
class A:
    nb = 0

    def __init__(self):
        print("creation objet de type A")
        A.nb = A.nb + 1
        print("il y en a maintenant ", A.nb)

    @classmethod
    def get_nb(cls):
        return A.nb
```

```
class A:
    nb = 0

    def __init__(self, x):
        print("creation objet de type A")
        self.x = x
        A.nb = A.nb + 1

print("A : nb = ", A.nb)
print("Partie 1")
a = A(3)
print("A : nb = ", A.nb)
print("a : x = ", a.x, " nb = ", a.nb)
print("Partie 2")
b = A(6)
print("A : nb = ", A.nb)
print("a : x = ", a.x, " nb = ", a.nb)
print("b : x = ", b.x, " nb = ", b.nb)
c = A(8)
print("Partie 3")
print("A : nb = ", A.nb)
print("a : x = ", a.x, " nb = ", a.nb)
print("b : x = ", b.x, " nb = ", b.nb)
print("c : x = ", c.x, " nb = ", c.nb)
```

# Caractéristiques de la P.O.O

## ❑ Héritage Python

L'héritage nous permet de définir une classe qui hérite de toutes les méthodes et propriétés d'une autre classe.

- La **classe parente** est la classe héritée, également appelée **classe de base**.
- La **classe enfant** est la classe qui hérite d'une autre classe, également appelée **classe dérivée**.

# Caractéristiques de la P.O.O

## ❑ Héritage Python : Exemple

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object

x = Person("John", "Doe")
x.printname()
```

```
class Student(Person):
  pass
```

```
x = Student("Mike", "Olsen")
x.printname()
```

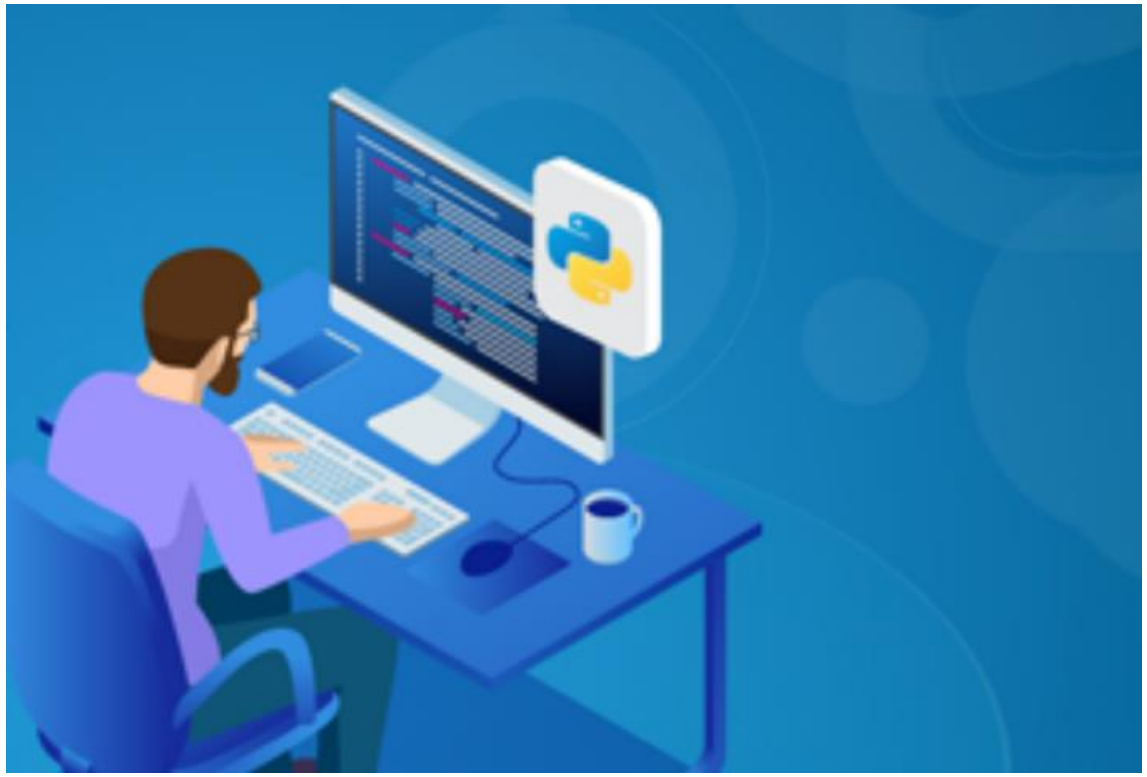
# Caractéristiques de la P.O.O

## ❑ Héritage Python : super()

Python a également une fonction **super()** qui remplace le nom de la classe parente dans la classe enfant.

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
```

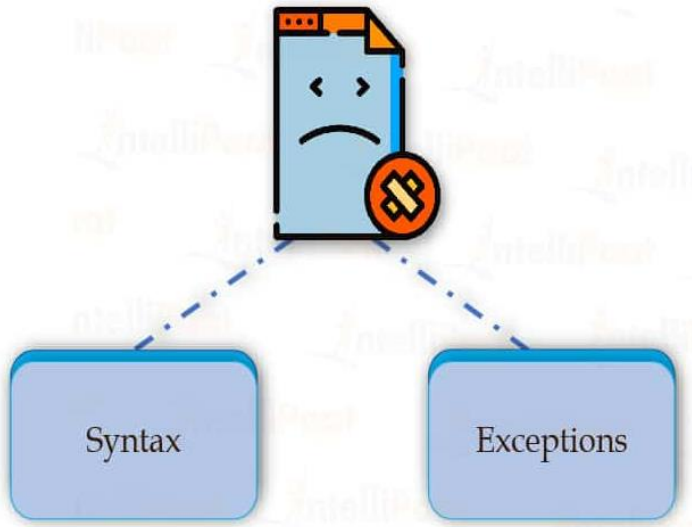
# TRAVAUX PRATIQUES



# Gestion des Exceptions

## ❑ Qu'est-ce qu'une exception?

Une exception est un événement qui se produit pendant l'exécution d'un programme qui **perturbe le flux normal des instructions du programme**.




En général, lorsqu'un script Python rencontre une situation qu'il ne peut pas gérer, **il déclenche une exception**.

Une exception est un **objet** Python qui représente une erreur.

# Gestion des Exceptions

## ❑ Bloc try … except

Si vous avez du code suspect qui peut soulever une exception, vous pouvez défendre votre programme en plaçant le code suspect dans un bloc **try** … **except**, suivie d'un bloc de code qui gère le problème le plus élégamment possible.

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

# Gestion des Exceptions

## ❑ Bloc finally

Le bloc **finally**, s'il est spécifié, sera exécuté indépendamment du fait que le bloc d'essai soulève ou non une erreur.

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

# Gestion des Exceptions

## ❑ Lever une Exception avec raise

En tant que développeur Python, vous pouvez choisir de lancer une exception en cas de condition. Pour lancer (ou relancer) une exception, utilisez le mot clé **raise.**

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

# Gestion des Exceptions

## ❑ Exception Courantes

- **IOError**: si le fichier ne peut pas être ouvert
- **KeyboardInterrupt**: lorsqu'une touche non requise est enfoncée par l'utilisateur
- **ValueError**: lorsque une fonction reçoit un argument incorrect.
- **AttributeError** : lorsqu'on souhaite accéder à un attribut qui n'existe pas.
- **ImportError**: s'il est impossible de trouver le module
- **MemoryError** : Cette erreur est générée lorsqu'une opération manque de mémoire.
- **KeyError** : Lors qu'on essaie d'accéder à une clé inexistante dans un dictionnaire
- **IndexError** : pour un index inexistante dans une liste.
- **TypeError** : quand une opération individuelle est effectuée sur un type inattendu

# Gestion des Exceptions

## ❑ Exceptions définies par l'utilisateur

Python vous permet également de créer vos propres exceptions en dérivant des classes à partir des exceptions intégrées standard.

La définition d'exceptions personnalisés permet de personnaliser les traitements à effectuer en cas de problèmes (Ecriture dans un fichier log, stockage dans une base de donnée, etc.)

```
class Networkerror(RuntimeError):
    def __init__(self, arg):
        self.args = arg
```

# Gestion des Dates

## ❑ Python Dates

Une date dans Python n'est pas un type de données qui lui est propre, mais nous pouvons importer un module nommé **datetime** pour travailler avec les dates.

Les 4 principales classes d'objets utilisées dans le module **datetime** sont:

- **datetime**
- **date**
- **time**
- **timedelta**

# Gestion des Dates

## ❑ Datetime et strftime

Le module **datetime** contient la classe **datetime** qui se charge des opérations sur les dates nécessitant les date et le temps.

L'objet **datetime** a une méthode pour formater les objets de date en chaînes lisibles.

La méthode est appelée **strftime()** , et prend un paramètre, **format**, pour spécifier le format de la chaîne retournée.

```
import datetime

x = datetime.datetime.now()
print(x)
```

```
import datetime

x = datetime.datetime.now()

print(x.year)
print(x.strftime("%A"))
```

# Gestion des Dates

## ❑ Datetime et strptime

L'objet **datetime** a une méthode nommée **strptime()** pour convertir une date depuis **une chaîne de caractères vers un objet datetime**.

La méthode accepte deux paramètres : la chaîne de caractères et le format.

```
my_string = '2019-10-31'

# Create date object in given time format yyyy-mm-dd
my_date = datetime.strptime(my_string, "%Y-%m-%d")

print(my_date)
print('Type: ',type(my_date))
```

```
2019-10-31 00:00:00 Type:
```

# Gestion des Dates

## ❑ Timedelta

Un objet **timedelta** représente la durée entre deux dates ou heures. Nous pouvons l'utiliser pour mesurer des intervalles de temps, ou manipuler des dates ou des heures en les ajoutant et en les soustrayant, etc.

```
#import datetime
from datetime import timedelta
# create timedelta object with difference of 2 weeks
d = timedelta(weeks=2)


print(d)
print(type(d))
print(d.days)
```

```
14 days, 0:00:00 <class 'datetime.timedelta'> 14
```

# Gestion des Dates

## ❑ Directives de Format

| Directive | Description | Example |
|---|---|---|
| %a | Weekday, short version | Wed |
| %A | Weekday, full version | Wednesday |
| %w | Weekday as a number 0-6, 0 is Sunday | 3 |
| %d | Day of month 01-31 | 31 |
| %b | Month name, short version | Dec |
| %B | Month name, full version | December |
| %m | Month as a number 01-12 | 12 |
| %y | Year, short version, without century | 18 |
| %Y | Year, full version | 2018 |
| %H | Hour 00-23 | 17 |
| %I | Hour 00-12 | 05 |
| %p | AM/PM | PM |
| %M | Minute 00-59 | 41 |
| %S | Second 00-59 | 08 |
| %f | Microsecond 000000-999999 | 548513 |

# TRAVAUX PRATIQUES