كلية العلوم والتقنيات بطنجة

Faculté des Sciences et Techniques de Tanger

Département Génie Informatique

# Module : Programmation avancée avec python

Filière : Master Sciences de données et Intelligence artificielle (IASD)

**Cours préparé et enseigné par :**

- Pr. Sanae KHALI ISSA

# Déroulement du cours

Introduction

Chapitre 1 : Les bases de la programmation python

- Instructions de base : lecture/écriture
- Les variables
- Les opérateurs
- Structure conditionnelle
- Structure itérative / boucle
- L'instruction : break
- L'instruction : continue

Chapitre 2 : Fonctions & Modules

- Modules
- Fonctions

Chapitre 3 : Structures de données

- Les types séquentiels : chaines de caractères, listes, tuples
- Les types de correspondance : dictionnaire

Chapitre 4 : Manipulation des fichiers

Chapitre 5 : Gestion des tableaux avec la bibliothèque **NumPy**

Chapitre 6 : Analyse de données avec la bibliothèque **Pandas**

Chapitre 7 : Gestion des graphiques avec le libraire **Matplotlib**

Projet de fin de module

# Introduction

# Introduction

## *Histoire du langage python*

- **1991 :** Guido van Rossum travaille aux Pays-Bas sur le projet AMOEBA : un système d'exploitation distribué. Il conçoit Python à partir du langage ABC et publie la version 0.9.0 sur un forum Usenet.
- **1996 :** sortie de **Numerical Python**, ancêtre de **numpy**
- **2001 :** naissance de la PSF (**Python Software Fundation**)

Les versions se succèdent... Un grand choix de modules est disponible, des colloques annuels sont organisés, Python est enseigné dans plusieurs universités et est utilisé en entreprise...

- **2006 :** première sortie de **IPython**
- **Fin 2008 :** sorties simultanées de **Python 2.6** et de **Python 3.0**
- **2013 :** versions en cours des branches 2 et 3 : **v2.7.3 et v3.3.0**
- **…**
- **2 Octobre 2023 :** sortie de la dernière version stable de python : **v3.12.0**

# Introduction

## Points forts du langage python

**Langage simple**

- Syntaxe claire et cohérente
- Gestion automatique de la mémoire
- Typage dynamique : pas de déclaration des types de données

**Open source**

- Python est libre et gratuit même pour les usages commerciaux
- Importante communauté de développeurs
- Nombreux outils standards disponibles

**Langage interprété**

- Langage interprété rapide
- De nombreux modules disponibles à partir des bibliothèques optimisées (souvent écrites en C ou C++)

**Orienté Objet**

- Modèle objet puissant mais pas obligatoire
- Structuration multifichier aisée des applications : facilite les modifications et les extensions

**Travail interactif**

- Nombreux interpréteurs interactifs disponibles
- Importante documentation disponible en ligne
- Développement rapide et incrémentiel
- Tests et déboggage outillés

**Ouverture au monde**

- Langage de haut niveau
- Interfaçable avec C/C++/FORTRAN
- Langage de script de plusieurs applications importantes : data science, développement web, application de bureau, etc.
- Excellente portabilité
- Indépendant de la machine

**Extensible**

- Disponibilité des bibliothèques et des packages dans tous les domaines
- Possibilité d'ajouter d'autres fonctions et modules.

# Introduction

## *Exécution d'un programme en python*

- Afin d'exécuter un programme en python, il faut installer un **interpréteur python** téléchargeable depuis le site : https://www.python.org/downloads/
- Et utiliser un **éditeur de texte** pour rédiger les script python ou utiliser un **IDE : Integrated Development Environment** tels que :

***Sublime Text***

***Visual Studio Code***


***Pycharm***


***Anaconda***


***Jupyter***

- Un script python est un ficher avec l'extension **.py**

# Chapitre 1 : Les bases de la programmation Python

# Chapitre 1 : les bases de la programmation python

## *La fonction : print*

C'est **une fonction** qui permet **d'afficher** les éléments passés comme arguments de la fonction

## *Syntaxe générale*

```
print (nom_variable)
print (nom_variable1, nom_variable2, ...)
print (" texte ")
print ( " texte " , nom_variable)
print ( f " texte  { nom_variable } " )
print ( " texte {} " .format(nom_variable) )
```

## *Exemple*

```
a=12
b=200
print(a)
print(a, b)
print("c'est un exemple pour la fonction print ")
print("a=",a)
print(f"a={a} et b={b}")
print("a={} et b={}".format(a,b))
```

# Chapitre 1 : les bases de la programmation python

## La fonction : input

- C'est une fonction qui permet à l'utilisateur d'un programme de saisir des valeurs à l'aide de son clavier.
- La fonction **input()** renvoie une valeur de type **chaine de caractères,** pour faire des calculs sur la valeur saisie, il faut la convertir à un entier avec la fonction **int (nom_variable)** ou un réel avec la fonction **float(nom_variable)**

## Syntaxe générale

nom_variable = **input ()**
nom_variable = **input** (" texte ")

## Exemple

```
a=input("entrer une première valeur : ")
b=input("entrer une deuxième valeur : ")
a=int(a)
b=int(b)
print(f"la somme de {a} et {b} est {a+b} ")
```

# Chapitre 1 : les bases de la programmation python

## Les variables

- Une variable est **un emplacement mémoire** dans lequel on peut mémoriser une valeur.
- Une variable est identifiée par **son nom** et **son adresse.**
- Le nom d'une variable doit :
  - Etre formé des lettres (A - Z) (a - z), des chiffres (1 - 9) et des lignes de soulignement ( _ )
  - Commencer par une lettre.
- En python, l**e typage des données est dynamique** : le type d'une variable est déterminée selon la valeur attribuée à cette variable.
  - Pour savoir le type d'une variable, il suffit d'utiliser la fonction **type (nom_variable)**

**Ex:**

```
a=3
print(type(a))
<class 'int'>
```

# Chapitre 1 : les bases de la programmation python

## Types natifs des variables

| Types prédéfinis | Signification | Exemples |
|---|---|---|
| **int** | Nombre entier | a=1, b=-2 |
| **float** | Nombre réel | a=1.3 |
| **complex** | Nombre complexe | a=3+5j |
| **bool** | Nombre logique | a=true , b=false |
| **str** | Chaine de caractères | a="Bonjour", b="s" |

# Chapitre 1 : les bases de la programmation python

## Opérateurs arithmétiques

| | |
|---|---|
| **x+y** | Somme de **x** et **y** |
| **x-y** | Différence de **x** et **y** |
| **x*y** | Produit de **x** et **y** |
| **x/y** | Quotient de **x** et **y** |
| **x//y** | Quotient entier de **x** et **y** |
| **x%y** | reste de la division euclidienne de **x** et **y** |
| **x**y** | **x** à la puissance **y** |

## Opérateurs de comparaison

| | |
|---|---|
| < | inférieur à |
| <= | inférieur ou égal à |
| > | supérieur à |
| >= | supérieur ou égal à |
| == | égal à |
| != | différent |

## Opérateurs logiques

| | |
|---|---|
| **and** | ET logique |
| **or** | OU logique |
| **not** | Négation logique |

# Chapitre 1 : les bases de la programmation python

## Opérateurs d'affectation

- C'est l'opération qui permet d'attribuer à une variable une valeur simple ou résultante d'une expression arithmétique :
  - Identificateur_variable = valeur simple
  - Identificateur_variable = expression arithmétique
- Une **affectation multiple** c'est attribuer à plusieurs variables une seule valeur avec une seule affectation.
  - Identificateur_variable1=Identificateur_variable2= Valeur
- Une **affectation parallèle** c'est affecter des valeurs à plusieurs variables en **parallèle**.
  - Identificateur_variable1, Identificateur_variable2= Valeur1, Valeur 2

## Exemple :

```
s=p*(r**2)
a=b=c=3
x,y,z=1,2,3
```

# Chapitre 1 : les bases de la programmation python

## Opérateurs d'assignation

| | | |
|---|---|---|
| **op1+= op2** | **op1=op1 +op2** | Additionne les deux valeurs **op1** et **op2, le résultat est stocké dans op1** |
| **op1-= op2** | **op1= op1 – op2** | Soustrait les deux valeurs **op1** et **op2, l**e résultat est stocké dans **op1** |
| **op1*= op2** | **op1= op1*op2** | Multiplie les deux valeurs **op1** et **op2**, le résultat est stocké dans **op1** |
| **op1/= op2** | **op1= op1/op2** | Divise les deux valeurs **op1** et **op2**, le résultat est stocké dans **op1** |
| **op1//=op2** | **op1=op1//op2** | Divise les deux valeurs **op1** et **op2**, la partie entière du résultat est stocké dans **op1** |
| **op1**=op2** | **op1=op1**op2** | Calculer **op1** à la puissance **op2** et mettre le résultat dans **op1** |

## Exemple :

```
A+=2 #Ajouter 2 à A puis stocker la nouvelle valeur dans A
B//=5 # Diviser B sur 5 puis stocker la partie entière du résultat dans B
C**=3 # calculer C à la puissance 3 puis stocker le résultat dans C
```

# Chapitre 1 : les bases de la programmation Python

## Expression logique simple

- C'est une **comparaison** de deux valeurs de même type en utilisant **un opérateur de comparaison**.
- La valeur d'une expression logique est de type **booléen** (vrai ou faux)

***Exemple***

A==B
A<=4

## Expression logique complexe

- C'est une **combinaison** entre deux expressions logiques simples en utilisant un **opérateur logique**.
- La valeur d'une expression logique complexe est de type **booléen** (vrai ou faux)

***Exemple***

Opérateur de comparaison — Opérateur logique — Opérateur de comparaison

$$(A + 3 < B * 2) \ \textbf{and} \ (A \ \% \ B == 0)$$

Expression logique simple — Expression logique simple

# Chapitre 1 : les bases de la programmation Python

## *La structure conditionnelle : if*

**if** expression logique **:**

Instruction 1
Instruction 2
…
Instruction n

Instructions à exécuter si l'expression logique est vraie

## Exemple

```
if  X % 2 == 0 :
      print ( " c'est un nombre pair " )
```

# Chapitre 1 : les bases de la programmation Python

## *La structure conditionnelle : if … else*

**if** expression logique **:**

Instruction 1
…
Instruction n

} à exécuter si l'expression logique est vraie

**else :**

Instruction 1
…
Instruction n

} à exécuter si l'expression logique est fausse

### Exemple

```
if X % 2 == 0 :
    print (" c'est un nombre pair ")
else :
    print (" c'est un nombre impair " )
```

# Chapitre 1 : les bases de la programmation Python

## *La structure conditionnelle : if … elif … else*

**if expression logique_1 :**
Instructions à exécuter si **l'expression logique_1** est vraie
**elif expression logique_2 :**
Instructions à exécuter si **l'expression logique_2** est vraie
…

**elif expression logique_n :**
Instructions à exécuter si **l'expression logique_n** est vraie
**else :**
Instructions à exécuter si toutes **les expressions logiques citées précédemment sont** fausses

# Chapitre 1 : les bases de la programmation Python

## *La structure conditionnelle : if … elif … else*

### Exemple

```
if  X>0 :
        print (″ C'est un nombre positif ″)
elif X < 0 :
        print (″ C'est un nombre négatif ″ )
else :
        print (″ C'est un nombre nul″ )
```

# Chapitre 1 : les bases de la programmation Python

## *L'instruction : match … case*

**match IdentificateurVariable :**

**case Valeur1 :**
Instructions à exécuter si **IdentificateurVariable = Valeur1**
…

**case ValeurN :**
Instructions à exécuter si **IdentificateurVariable = ValeurN**

**case other :**
Instructions à exécuter si **IdentificateurVariable** n'appartient pas à la liste des valeurs **{Valeur1, Valeur2, … , ValeurN}**

# Chapitre 1 : les bases de la programmation Python

## *L'instruction : match ... case*

### Exemple

```
choix = input ("entrer votre choix : ")
match choix :
   case "Lundi" :
       print("1er jour de la semaine ")
   case "Mardi" :
      print("2ème jour de la semaine ")
   case "Mercredi" :
      print("3ème jour de la semaine ")
   case "Jeudi" :
      print("4ème jour de la semaine « )
  case other :
      print("choix introuvable ")
```

# Chapitre 1 : structure conditionnelle

## *Exercice d'application*

Ecrire un script python qui permet d'afficher les mentions suivantes selon la valeur de la moyenne générale choisie par l'utilisateur.

- Si **moyenne >=16**, la mention affichée est : **Très Bien**
- Si **14 <= moyenne <16**, la mention affichée est : **Bien**
- Si **12 <= moyenne <14**, la mention affichée est **Assez Bien**
- Si **10 <= moyenne <12**, la mention affichée est **Passable**
- Si **moyenne <10**, l'étudiant a échoué

# Chapitre 1 : structure conditionnelle

## *Exercice d'application*

```
moyenne = float (input (" Entrer la moyenne générale :  ")
if moyenne >= 16 :
   print (" Vous avez une mention très bien ")
elif moyenne >=14 :
        print (" Vous avez une mention bien " )
elif moyenne >=12 :
        print (" Vous avez une mention assez bien " )
elif moyenne >=10 :
        print (" Vous avez une mention passable " )
else :
        print (" Vous avez échoué" )
```

# Chapitre 1 : les bases de la programmation Python

## *Les boucles*

Une boucle est un ensemble d'instructions qui se ***répètent*** un certain ***nombre de fois***.

Deux boucles sont utilisées en python :

- La boucle **for**
- La boucle **while**

# Chapitre 1 : les bases de la programmation Python

## *La boucle for*

```
for compteur in liste_valeurs :
    Instruction 1
    Instruction 2
    ...
    Instruction n
```

} à exécuter pour chaque valeur du compteur

## Exemple

```
for x in [1, 2, 3, 4, 5] :
    print (" x prend la valeur  ", x)
```

# Chapitre 1 : les bases de la programmation Python

## *La boucle for*

```
for i in  "Bonjour" :
        print (i, end = "  ") # le résultat d'exécution est l'affichage des caractères  B  o  n  j  o  u  r
for x in [4, 5, 6] :
        print (x, end = "  ") # le résultat d'exécution est l'affichage des valeurs  4  5  6
for y in ["B", 15, 3.6, "Et"] :
        print (y, end = "  ") # le résultat d'exécution est l'affichage des valeurs  B  15  3.6  Et
for j in range(5) :
        print (j, end = "  ") # le résultat d'exécution est l'affichage des valeurs  0  1  2  3  4
for k in range (1, 5) :
        print (k, end = "  ") # le résultat d'exécution est l'affichage des valeurs  1  2  3  4
for k in range (1, 10, 3) :
        print (k, end = "  ") # le résultat d'exécution est l'affichage des valeurs  1  4  7
```

# Chapitre 1 : les bases de la programmation Python

## *La boucle while*

**while** expression logique **:**

Instruction 1
Instruction 2
…
Instruction n

} à exécuter tant que l'expression logique est vraie

### Exemple

```
x=1
while x <=5 :
        print (" x prend la valeur ", x)
        x=x+1
```

# Chapitre 1 : les bases de la programmation Python

## *La boucle while*

### Exercice d'application

Ecrire un script python qui demande à l'utilisateur d'introduire un ensemble des valeurs numériques puis calculer et afficher leur carré.
Le script s'arrête une fois l'utilisateur introduit la valeur 0.

### Solution

```
N = int(input (" Entrer une valeur :"))
while N != 0 :
    Carre = N*N
    print (f " le carré du nombre saisi est {Carre}")
    N = int(input (" Entrer une valeur :"))
```

# Chapitre 1 : les bases de la programmation Python

## *L'instruction break*

L'instruction **break** permet de « casser » l'exécution d'une boucle (**while** ou **for**). C'est à dire, elle fait sortir de la boucle et passer à l'instruction suivante.

### Exemple avec la boucle for

```
for val in "Bonjour":
    if val == "j":
        break
    print(val)
```

**Résultat d'exécution**

```
B
o
n
```

### Exemple avec la boucle while

```
i = 0
while True:
    print(i)
    i = i + 1
    if i >= 5:
        break
```

**Résultat d'exécution**

```
0
1
2
3
4
```

# Chapitre 1 : les bases de la programmation Python

## *L'instruction continue*

Le mot-clé **continue** est utilisé pour terminer l'itération en cours dans une boucle for (ou une boucle while), et passer à l'itération suivante.

### Exemple avec la boucle for

```
for val in "Bon":
    if val == "o":
        continue
    print(val)
```

**Résultat d'exécution**

```
B
n
```

### Exemple avec la boucle while

```
i = 7
while i>0:
    i = i - 1
    if i == 5:
        continue
    print(i)
```

**Résultat d'exécution**

```
6
5
4
3
2
1
0
```

# Chapitre 2 : Fonctions & Modules

# Chapitre 2 : Fonctions

## *Définition*

- Une fonction est un **bloc d'instructions** regroupées sous un **même nom** (*le choix du nom de la fonction doit répondre aux mêmes contraintes pour choisir un identificateur d'une donnée*).
- Une fonction peut avoir des **paramètres/arguments**
- Une fonction peut **renvoyer une valeur** avec le mot clé **return**
- Pour exécuter le code d'une fonction, il suffit d'écrire **son nom** avec **ses paramètres** *( dans le cas d'une fonction ayant des paramètres)*

**Syntaxe générale d'une fonction avec des paramètres et ayant une valeur de retour**

```
def nom_fonction (liste des paramètres) :
        Instruction 1
        Instruction 2
        …
        Instruction N
        return (nom_variable)
```

# Chapitre 2 : Fonctions

## *Exemple 1*

```
def afficher_numeros () :
        n=int(input("entrer le nombre des valeurs à afficher : "))
        for i in range(n) :
                print(i)
afficher_numeros () # appel de la fonction afficher_numeros ()
```

**Résultat d'exécution**

```
Entrer le nombre des valeurs à afficher : 6
0
1
2
3
4
5
```

# Chapitre 2 : Fonctions

## *Exemple 2*

```
def afficher_numeros (n) :
        for i in range(n) :
            print(i)
N=int(input("entrer le nombre des valeurs à afficher : "))
afficher_numeros (N) # appel de la fonction avec un paramètre
```

**Résultat d'exécution**

```
Entrer le nombre des valeurs à afficher : 6
0
1
2
3
4
5
```

# Chapitre 2 : Fonctions

## *Exemple 3*

```
def afficher_numeros (n =3 ) :
        for i in range(n) :
                print(i)
N=4
afficher_numeros (N) # appel de la fonction avec un paramètre
afficher_numeros () # appel de la fonction sans paramètre permet de prendre la valeur par défaut
```

**Résultat d'exécution**

```
0
1
2
3
0
1
2
```

# Chapitre 2 : Fonctions

## *Exemple 4*

```
def somme (x, y) :
        s =x+y
        print ( "la somme des deux valeurs est : ", s )

a=int(input("entrer le premier nombre : "))
b=int(input("entrer le deuxième nombre : "))
somme (a, b) # appel de la fonction somme avec deux paramètres a et b
```

**Résultat d'exécution**

```
12
10
La somme des deux valeurs est : 22
```

# Chapitre 2 : Fonctions

## *Exemple 5*

```
def somme (x, y) :
        s =x+y
        return (s)

a=int(input("entrer le premier nombre : "))
b=int(input("entrer le deuxième nombre : "))
print ( "la somme des deux valeurs est : ", somme (a, b))  # appel de la fonction somme
```

**Résultat d'exécution**

```
12
10
La somme des deux valeurs est : 22
```

# Chapitre 2 : Fonctions

## *Les fonctions récursives*

**Une fonction récursive** est une fonction **ayant des paramètres** et **ayant aussi une valeur de retour** dans laquelle on fait appel à la **fonction ELLE-MÊME.**

**Exemple d'une fonction calculant la factorielle d'un entier**

**Méthode itérative**

```
def factorielle (x) :
        f=1
        if x==0 :
            return 1
        else :
            for i in range(2, x+1) :
                f=f*i
         return f
```

# Chapitre 2 : Fonctions

## *Les fonctions récursives*

**Une fonction récursive** est une fonction **ayant des paramètres** et **ayant aussi une valeur de retour** dans laquelle on fait appel à la **fonction ELLE-MÊME.**

**Exemple d'une fonction calculant la factorielle d'un entier**

**Méthode récursive**

```
def factorielle (x) :
    if x==0 :
        return 1
    else :
        return x * factorielle(x-1)
```

# Chapitre 2 : Modules

## *Définition*

- Un projet Python est généralement composé de **plusieurs fichiers sources** ayant l'extension **.py**,
- Un **module** est un **fichier script Python** permettant de définir des éléments de programme **réutilisables** dans d'autres scripts python. Ce mécanisme permet d'élaborer efficacement des bibliothèques de fonctions ou de classes.
- ***L'utilisation des modules peut avoir plusieurs avantages à savoir :***
  - La réutilisation du code ;
  - La possibilité d'intégrer la documentation et les tests au module ;
  - La réalisation de services ou de données partagés ;

## *Exemples des modules standards de python*

- Module **math** : il fournit un ensemble de fonctions permettant de réaliser des calculs mathématiques complexes
- Module **random** : il implémente des générateurs de nombres pseudo-aléatoires pour différentes distributions
- Module **datetime** : il fournit des classes pour manipuler de façon simple ou plus complexe des dates et des heures
- Module **sys** : il fournit un accès à certaines variables système utilisées et maintenues par l'interpréteur, et à des fonctions interagissant fortement avec ce dernier

# Chapitre 2 : Modules

## Importation des modules

- Afin d'importer le contenu d'un module:
  - **import nom_module** *# permet d'importer tout le contenu du module*
  - **import nom_module as nom_alias** *# permet d'importer tout le contenu du module avec la création d'un alias vers le nom de module*
  - **from nom_module import *** *# permet d'importer tous les éléments du module (méthodes et variables)*
  - **from nom_module import nom_element** *# permet d'importer un élément précis du module*

### Exemples

```
import math
x=int(input(" x= ")
print(" RC= " , math.sqrt(x))
```

```
import math as mt
x=int(input(" x= " )
print(" RC= " , mt.sqrt(x))
```

```
from math import *
x=int(input(" x= " )
y=int(input(" y= " )
print(" le pgcd des deux valeurs est :",
gcd(x, y)
```

# Chapitre 2 : Modules

## *Exemple de création et d'importation d'un module*

### *Module calcul.py*

```
def addition (x, y) :
    return (x+y)

def soustraction (x, y) :
    return (x-y)

def multiplication (x, y) :
    return (x*y)
```

### *Script principal*

```
from calcul import *
a=float(input(" Entrer une première valeur : " ))
b=float(input(" Entrer une deuxième valeur : " ))
s=addition (a, b)
print ("La somme des deux valeurs est ", s)
d=soustraction (a, b)
print ("La différence des deux valeurs est ", d)
p=multiplication (a, b)
print ("Le produit des deux valeurs est ", p)
```

# Chapitre 2 : Fonctions & Modules

## Exercices : Série N1

# Chapitre 3 : Structures de données

# Chapitre 3 : structures de données

## *Définition d'une structure de données*

- **Une structure de données** est **un ensemble d'objets/éléments** pouvant être :
  - De même ou de différents types.
  - Mutables ou immuables (les éléments sont changeables ou non)
  - Ordonnés ou non ordonnés (l'ordre des éléments est important ou non)
- On distingue trois types de structures de données : **séquentielles**, **ensemblistes** ou de **correspondance**.
- Une structure de données **séquentielle** peut être **une chaine de caractères, une liste** ou **un tuple.**
- Une structure de données de **correspondance** se présente sous forme d'**un dictionnaire.**
- Et **ensembliste** sous forme d'**un ensemble**.

# Chapitre 3 : chaines de caractères

## *Définition*

- Une chaine de caractère est un ensemble **ordonné** de caractères **non modifiables**.
- Les caractères peuvent être :
  - Des lettres en majuscules (A, B, …, Z ) ou en minuscules (a, b, …, z)
  - Des chiffres (0, 1, …, 9)
  - Des signes de ponctuation ( . : ; , ? ] [ } { ! … )
  - Des caractères spéciaux ( # @ & $ / + = - _ % ç à ° £ < é > § … )

## Syntaxe de définition des chaines de caractères

```
ch1 = " c'est une chaine de caractères "
ch2 = ' ceci est une chaine de caractères '
ch3 = ' c\'est une chaine de caractères '
ch4 = "" # déclaration d'une chaine vide
ch5 = r " c'est un texte1\n \t c'est un texte2 "  # ceci consiste à ignorer les caractères spéciaux \n \t
ch6 = """ chaine de caractères sur
plusieurs lignes """
```

# Chapitre 3 : chaines de caractères

## *Manipulation des chaines de caractères*

- Les opérateurs mathématiques qui peuvent être appliqués sur une chaine de caractères sont : * et +
- Pour la comparaison des chaines de caractères, on utilise les opérateurs : ==, **!**= , < , >

### Exemple

```
ch1 = "salut "
ch2 = "les programmeurs"
ch3 = ch1 + ch2
print (ch3) # permet d'afficher le texte : salut les programmeurs
ch4 = ch1*3
print (ch4) # permet d'afficher le texte : salut salut salut
print (ch1 == ch2 ) # renvoie la valeur False
print (ch1 > ch2 ) # renvoie la valeur True
```

# Chapitre 3 : chaines de caractères

## *Manipulation des chaines de caractères*

- Pour accéder à un caractère d'une chaine, il suffit de préciser sa position entre crochets
- Les indices de la position des caractères commencent par 0 ( de gauche à droite) et par -1 (de droite à gauche)
- C'est possible d'accéder aux éléments d'une chaine de caractères, via la méthode de **slicing**

**Exemple**

```
ch3 = "salut les programmeurs"
print (ch3[1] ) # permet d'afficher le caractère  'a'
print (ch3[-5] ) # permet d'afficher le caractère  'm'
print (ch3[: 5] ) # permet d'afficher les caractères ayant indice de 0 à 4 : 'salut'
print (ch3[10:17] ) # permet d'afficher les caractères ayant indice de 10 à 16: 'program'
print (ch3[-12:] ) # permet d'afficher les caractères ayant indice de -1 à -12 : 'programmeurs'
```

# Chapitre 3 : chaines de caractères

## *La fonction len (), bool ()*

- La fonction **len()** est une fonction utilisée pour le type de données **string**. Elle permet de renvoyer une valeur entière représentant **le nombre de caractères** d'une chaine de caractères (longueur d'une chaine).
- La fonction **bool()** permet de vérifier si une chaine est vide ou non, dans la cas vide, elle renvoie **False** sinon **True**

**Exemple**

```
ch1 = ""
ch2 = "salut les programmeurs"
L1=len(ch1)
L2=len(ch2)
print (L1) # permet d'afficher la valeur 0
print (L2) # permet d'afficher la valeur 22
print(bool(ch1)) # permet d'afficher False
```

# Chapitre 3 : chaines de caractères

## *Méthodes associées aux chaines de caractères*

- Une méthode est une fonction associée à un type de données : str, int, float, etc.
- Pour afficher la liste des méthodes associées au type **str**
  - print(dir(str)) ou bien print(dir(""))

## Exemple

**print(dir(str)**

['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

# Chapitre 3 : chaines de caractères

## *Les méthodes startswith(), endswith()*

- La fonction **startwith()** est une fonction qui permet de vérifier si une chaine de caractère commence par un ou plusieurs caractères, la méthode renvoie une valeur de **type booléen** : True ou False.
- La fonction **endswith()** est une fonction qui permet de vérifier si une chaine de caractère se termine par un ou plusieurs caractères, la méthode renvoie une valeur de **type booléen** : True ou False.

**Exemple**

```
chaîne = "234567891"
if chaîne.startswith ("2345") :
    print("ok") # permet d'afficher ok
if chaîne.endswith ("91") :
    print("ok") # permet d'afficher ok
```

# Chapitre 3 : chaines de caractères

## *Les méthodes : index (), rindex ()*

- La méthode **index()** permet de rechercher l'index de la **première occurrence** d'une valeur spécifiée dans une chaîne. S'il n'existe pas, une erreur est déclenchée (ValueError : sous-chaîne non trouvée)
- La méthode **rindex()** permet de rechercher l'index de la **dernière occurrence** d'une valeur spécifiée dans une chaîne. S'il n'existe pas, une erreur est déclenchée (ValueError : sous-chaîne non trouvée)
- Paramètres des fonctions **index (Valeur, Début, Fin) et rindex(Valeur, Début, Fin)**
  - **Valeur** (Obligatoire) : La valeur à rechercher
  - **Début** (Optionnel) : Où commencer la recherche. La valeur par défaut est 0
  - **Fin** (Optionnel) : Où terminer la recherche. La valeur par défaut est à la fin de la chaîne

**Exemple**

```
chaine1 = "Coucou tout le monde"
print(chaine1.index("t")) # permet d'afficher 7
print(chaine1.index("u", 7, 19)) # permet d'afficher 9
print(chaine1.rindex("o") # permet d'afficher 16
print(chaine1.rindex("o", 7, 19) # permet d'afficher 16
```

# Chapitre 3 : chaines de caractères

## *La méthode : count ()*

- La méthode **count()** permet de compter le nombre d'occurrences d'une sous-chaîne dans une chaîne.
- Paramètres de la fonction **count (Valeur, Début, Fin)**
  - **Valeur** (Obligatoire) : La valeur à rechercher
  - **Début** (Optionnel) : Où commencer la recherche. La valeur par défaut est 0
  - **Fin** (Optionnel) : Où terminer la recherche. La valeur par défaut est à la fin de la chaîne

**Exemple**

```
chaine1 = "Coucou tout le monde"
chaine2 = "ou"
print(chaine1.count(chaine2)) # permet d'afficher 3
print(chaine1.count("o", 3, 10)) # permet d'afficher 2
```

# Chapitre 3 : chaines de caractères

## *Les méthodes : find (), rfind()*

- La méthode **find()** permet de rechercher la position de la première occurence d'une sous-chaîne dans une chaîne.
- La méthode **rfind()** permet de rechercher la position de la dernière occurence d'une sous-chaîne dans une chaîne.
- Paramètres des fonctions **find (Valeur, Début, Fin) et rfind (Valeur, Début, Fin)**
  - **Valeur** (Obligatoire) : La valeur à rechercher
  - **Début** (Optionnel) : Où commencer la recherche. La valeur par défaut est 0
  - **Fin** (Optionnel) : Où terminer la recherche. La valeur par défaut est à la fin de la chaîne
- Les deux fonctions **find()** et **rfind()** renvoient -1 dans le cas où la valeur à rechercher n'existe pas

**Exemple**

```
chaine1 = "Salut tout le monde tout le monde"
chaine2 = "tout"
print(chaine1.find(chaine2)) # permet d'afficher 6
print(chaine1.rfind(chaine2)) # permet d'afficher 20
```

# Chapitre 3 : chaines de caractères

## *La méthode : replace ()*

- La méthode **replace()** permet de remplacer des caractères d'une chaine par d'autres.
- La méthode **replace(Ancienne, Nouvelle, nombre)** prend trois paramètres:
  - **Ancienne(Obligatoire) :** La chaîne à rechercher
  - **Nouvelle(Obligatoire) :** La nouvelle chaîne par laquelle remplacer l'ancienne chaîne
  - **Nombre (Optionnel) :** Un nombre spécifiant le nombre d'occurrences de l'ancienne chaîne souhaitant remplacer. La valeur par défaut est toutes les occurrences
- La méthode **replace()** renvoie une copie de la chaîne dans laquelle l'ancienne chaîne est remplacée par la nouvelle chaîne. La chaîne d'origine ne change pas.
- Si l'ancienne chaîne n'est pas trouvée, la méthode **replace ()** renvoie la copie de la chaîne d'origine.

**Exemple**

```
chaine1 = "Salut tout le monde"
chaine2="Bonjour"
print(chaine1.replace("Salut", chaine2)) # permet d'afficher 'Bonjour tout le monde'
```

# Chapitre 3 : chaines de caractères

## *Les méthodes : upper (), lower (), swapcase (), capitalize ()*

- La méthode **upper()** permet de convertir une chaine de caractères en **majuscules.**
- La méthode **lower()** permet de convertir une chaine de caractères en **minuscules.**
- La méthode **swapcase()** permet de convertir les lettres minuscules d'une chaine de caractères en majuscules et les lettres majuscules en minuscules.
- La méthode **capitalize()** permet de convertir **la première lettre** d'une chaîne en majuscule.

**Exemple**

```
string = "CE MATIN il fait beau"
print(string.upper()) #permet d'afficher 'CE MATIN IL FAIT BEAU '
string = "CE MATIN il fait beau"
print(string.lower()) #permet d'afficher 'ce matin il fait beau'
string = "ce mAtin IL Fait beau"
print(string.swapcase()) #permet d'afficher 'CE MaTIN il fAIT BEAU'
string = "ce matin il fait beau"
print(string.capitalize()) #permet d'afficher 'Ce matin il fait beau'
```

# Chapitre 3 : chaines de caractères

## *Les méthodes : strip (), rstrip (), lstrip ()*

- La méthode **strip()** permet de supprimer à partir d'une chaîne tous les caractères à **droite** et à **gauche** indiquées en paramètres de la méthode.
- La méthode **rstrip()** permet de supprimer à partir d'une chaîne tous les caractères à **droite** indiquées en paramètres de la méthode**.**
- La méthode **lstrip()** permet de supprimer à partir d'une chaîne tous les caractères à **gauche** indiquées en paramètres de la méthode**.**
- Si le paramètre de la méthode n'est pas précisé, la méthode supprime **les espaces** (***le paramètre par défaut***)
- Les trois méthodes **strip()**, **rstrip()**, **lstrip()** retournent une copie de la chaine après sa modification.

**Exemple**

```
chaine1 = "     c'est un test pour la méthode strip      "
print(chaine1.strip()) #permet d'afficher : c'est un test pour la méthode strip
chaine2 = "c'est un test pour la méthode rstrip :?!}#"
print(chaine2.rstrip(":#?!}")) #permet d'afficher : c'est un test pour la méthode rstrip
chaine3 = "c'est un test pour la méthode rstrip"
print(chaine3.lstrip("c'est ")) #permet d'afficher : un test pour la méthode lstrip
```

# Chapitre 3 : chaines de caractères

## Exercices : Série N2

# Chapitre 3 : les listes

## *Définition d'une liste*

- Une liste est **une séquence** d'éléments, **ordonnés**, **mutables**, de **même** ou de **différents types**.
- Une liste se présente sous le format suivant :

**nom_liste = [element1, element_2, …, element_n]**

## Exemple

```
Liste_1 = [] # permet de définir une liste vide
Liste_2 = [1, 2, 3, 4] # permet de définir une liste des entiers
Liste_3 = ["A", "B", "C"] # permet de définir une liste de caractères
Liste_4 = ["Lundi", "Mardi", "Mercredi", "Jeudi"] # permet de définir une liste de jours
Liste_5 = ["Lundi", "M", 3, 4.5, "?"] # permet de définir une liste hétérogène
```

# Chapitre 3 : les listes

## *Appliquer des opérations sur les listes*

Les deux opérateurs + et * sont utilisés avec les listes :

- Pour concaténer deux listes (*ajouter une liste à la fin d'une autre liste*) :

nom_liste1 = nom_liste2 + nom_liste3

- Pour multiplier le contenu d'une liste par une valeur numérique

nom_liste1= nom_liste * valeur_numérique

**Exemple**

```
Liste_1 = [13, 11] # permet de définir une liste des entiers
Liste_2 = ["A", "B", "C"] # permet de définir une liste de caractères
Liste_3 = Liste_1+Liste_2
print (Liste_3) # permet d'afficher : [13, 11,"A", "B", "C"]
Liste_3=Liste_3*2
print (Liste_3) # permet d'afficher : [13, 11, 'A', 'B', 'C', 13, 11, 'A', 'B', 'C']
```

# Chapitre 3 : les listes

## *Autres méthodes pour création de liste : range() et list ()*

- La fonction **range()** permet de générer un ensemble de valeurs comprises entre **valeur_1** et **valeur_2** (*valeur_2 non incluse*)

  **range (valeur_1, valeur_2, pas de changement)**

- La fonction **list()** permet de convertir un ensemble des valeurs (*string, tuple, set, dictionnaire*) vers une liste.

**Exemple**

```
Liste_1 = list(range (5)) # permet de définir une liste des entiers allant de 0 à 4
print (Liste_1) # permet d'afficher  la liste : [0, 1, 2, 3, 4]
L=list("Bonjour")
print(type(L)) # permet d'afficher  le type de L : <class 'list'>
print(L) # permet d'afficher  la liste : ['B', 'o', 'n', 'j', 'o', 'u', 'r']
```

# Chapitre 3 : les listes

## *Autres méthodes pour création de liste : split()*

- La fonction **split()** permet de transformer une chaine de caractères en **liste**.
- La méthode **split (séparateur, nbr_division)** prend deux paramètres:
  - séparateur (optionnelle) : spécifie le séparateur à utiliser lors de division de la chaîne. Par défaut, l'espace est un séparateur
  - nbr_division (optionnelle) : spécifie le nombre de division à effectuer. La valeur par défaut est -1, qui signifier « toutes les occurrences »

**Exemple**

```
chaine = "exemple de la fonction split"   # permet de définir une chaine de caractères
L1 = chaine.split ()
print(L1) # permet d'afficher  la liste : ['exemple', 'de', 'la', 'fonction', 'split']
L2 = chaine.split (" ", 2)
print(L2) # permet d'afficher  la liste : ['exemple', 'de', 'la fonction split']
```

# Chapitre 3 : les listes

## *Autres méthodes pour création de liste : join()*

- La méthode **join()** permet de créer des chaînes à partir d'un ensemble d'éléments (liste, tuple, dictionnaire, etc…) en utilisant un séparateur de chaîne.
- La méthode **join()** prend un seul paramètre qui est l'ensemble d'éléments à concaténer, et renvoie la chaîne concaténée.

**chaine = "séparateur_chaine".join (liste_elements)**

**Exemple**

```
Liste = ["B", "O", "N", "J", "O", "U", "R"]  # permet de définir une liste
chaine_1 = "/".join(Liste)
print(chaine_1) # permet d'afficher  la chaine :  'B/O/N/J/O/U/R'
chaine_2 = "".join(Liste)
print(chaine_2) # permet d'afficher  la chaine : BONJOUR
```

# Chapitre 3 : les listes

## *Accéder aux éléments d'une liste*

- Pour accéder aux éléments d'une liste, il est possible d'utiliser :
  - L'indiçage **positif** (0, 1, 2, etc.)
  - L'indiçage **négatif** (-1, -2, -3, etc.)
  - La méthode de **slicing** ( extraire une tranche d'éléments d'une liste)

## Exemple

```
Liste = ["Lundi", "Mardi", "Mercredi", "Jeudi" ]
print (Liste [1] ) # permet d'afficher la valeur : Mardi
print (Liste [-2] ) # permet d'afficher la valeur  : Mercredi
x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print (x[:]) # permet d'afficher : [0  1  2  3  4  5  6  7  8  9]
print (x[::2]) # permet d'afficher : [0  2  4  6  8]
print (x[1:6:3]) # permet d'afficher : [1  4 ]
print (x[1:-1]) # permet d'afficher : [1  2  3  4  5  6  7  8]
```

# Chapitre 3 : les listes

## *Parcourir une liste*

- Pour parcourir une liste, il est possible d'utiliser la boucle **for** selon la syntaxe suivante :

```
for nom_variable in nom_liste :
        Instruction_1
        ...
        Instruction_n
```

**Exemple**

```
Liste = ["Lundi", "Mardi", "Mercredi", "Jeudi"]
for i in Liste :
    print (i)
```

**Résultat d'exécution**

```
Lundi
Mardi
Mercredi
Jeudi
```

# Chapitre 3 : les listes

## *Afficher les éléments d'une liste*

- La méthode **enumerate()** permet d'afficher les éléments d'une liste associés à des index.

**for** index, nom_variable **in enumerate** (nom_liste) :
print(index, nom_variable)

**Exemple**

```
Liste_1 = ["Lundi", "Mardi", "Mercredi", "Jeudi"]
for i, x in enumerate(Liste_1) :
    print (i, x)
```

**Résultat d'exécution**

```
0 Lundi
1 Mardi
2 Mercredi
3 Jeudi
```

# Chapitre 3 : les listes

## *Afficher les éléments d'une liste*

- La méthode **zip()** permet de lier les éléments d'une liste avec une deuxième liste ou plus.

**for** valeur_1, valeur_2, ..., valeur_n **in zip** (liste_1, liste_2, ..., liste_n) :
print(valeur_1, valeur_2, ..., valeur_n)

**Exemple**

```
Liste_1 = ["Lundi", "Mardi", "Mercredi", "Jeudi"]
Liste_2 = [1, 2, 3, 4]
for i, x in zip(Liste_1, Liste_2) :
    print (i, x)
```

**Résultat d'exécution**

```
Lundi 1
Mardi 2
Mercredi 3
Jeudi 4
```

# Chapitre 3 : les listes

## *Méthodes associées aux listes : len(), count(), clear()*

- La fonction **len()** permet de calculer la longueur d'une liste (*le nombre d'éléments d'une liste*)
- La fonction **count()** permet de compter le nombre d'occurrence d'une valeur dans une liste.
- La fonction **clear()** permet de supprimer tous les éléments d'une liste. Elle permet de renvoyer une liste vide.

**Exemple**

```
L = ["Lundi", "Mardi", "Mercredi", "Jeudi" ]
print (len(L) ) # permet d'afficher la valeur : 4
print (L.count("Lundi") ) # permet d'afficher la valeur : 1
L.clear() # permet de supprimer tous les éléments d'une liste
```

# Chapitre 3 : les listes

## *Méthodes associées aux listes : remove(), pop(), del()*

- La fonction **remove()** permet de supprimer un élément à partir d'une liste **en introduisant sa valeur**.
- La fonction **pop()** permet de supprimer un élément à partir d'une liste **en utilisant son index.** Si le paramètre de la fonction n'est pas précisé, la fonction prend la valeur par défaut : -1 ( le dernier élément de la liste)
- La fonction **del()** permet de supprimer un élément à partir d'une liste **en utilisant son index** ou **une tranche de valeurs**.

**Exemple**

```
L = [1, 2, 3, 4, 5, 6, 7, 8, 9]
L.remove(5) # permet de supprimer l'élément 5 de la liste L
print (L) # permet d'afficher : [1, 2, 3, 4, 6, 7, 8, 9]
L.pop(3) #  permet de supprimer l'élément ayant l'indice 3 à partir de la liste  L
print (L) # permet d'afficher : [1, 2, 3, 6, 7, 8, 9]
del L[5] # permet de supprimer l'élément ayant l'indice 5 à partir de la liste L
del L[1:4] # permet de supprimer les éléments ayant les indices : 1, 2, 3  à partir de la liste L
print (L) # permet d'afficher : [1, 7, 9]
```

# Chapitre 3 : les listes

## *Méthodes associées aux listes : max(), min(), sum()*

- La fonction **max()** permet de rechercher la valeur maximale dans une liste.
- La fonction **min()** permet de rechercher la valeur minimale dans une liste.
- La fonction **sum()** permet de calculer la somme des éléments d'une liste.
- Les trois fonctions : **max()**, **min()**, **sum()** prennent un seul paramètre : **nom_liste** et renvoient une valeur numérique.

**Exemple**

```
L = [1, 2, 3, 4, 5, 6, 7, 8, 9]
print (max(L)) # permet d'afficher : 9
print (min(L)) # permet d'afficher : 1
print (sum(L)) # permet d'afficher : 45
```

# Chapitre 3 : les listes

## *Méthodes associées aux listes : append(), insert(), extend()*

- La fonction **append()** permet d'ajouter un élément à la fin d'une liste
- La fonction **insert()** permet d'insérer un élément dans une liste à une position donnée.
- La fonction **extend()** permet d'ajouter un ensemble d'éléments à la fin d'une liste.

**Exemple**

```
L = [1, 2, 3, 4, 5, 6, 7, 8, 9]
L.append(10)
print (L) # permet d'afficher : [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
L.insert(4, 101)
print (L) # permet d'afficher : [1, 2, 3, 4, 101, 5, 6, 7, 8, 9, 10]
L.extend ([10, 11, 12])
print (L) # permet d'afficher : [1, 2, 3, 4, 101, 5, 6, 7, 8, 9, 10, 10, 11, 12]
```

# Chapitre 3 : les listes

## *Méthodes associées aux listes : sort()*

- La fonction **sort()** permet de trier une liste selon un ordre croissant ou décroissant sans renvoyer une nouvelle liste, les modifications sont appliquées sur la liste d'origine.
- Les listes contenant des chaines de caractères sont triées selon l'ordre alphabétique.
- Pour spécifier le type de tri à appliquer (croissant ou décroissant), la méthode est utilisée avec un paramètre appelé **reverse** de type **bool**, sa valeur par défaut : **False**.

**reverse= True ( Trie décroissant), reverse = False (Trie croissant)**

### Exemple

```
L = [11, 2, 43, 114, 25, 6, 27, 18, 19]
L.sort(reverse=True)
print (L) # permet d'afficher :[114, 43, 27, 25, 19, 18, 11, 6, 2]
```

# Chapitre 3 : les listes

## *Méthodes associées aux listes : reverse(), index()*

- La fonction **reverse()** permet d'inverser les éléments d'une liste, la méthode **reverse()** ne renvoie aucune liste, les modifications sont appliquées sur la liste d'origine.
- La méthode **index()** permet de rechercher l'indice de la première occurence d'une valeur dans une liste *(dans le cas ou la valeur à rechercher n'existe pas , la méthode affiche une erreur)*

**Exemple**

```
L = ["S", "B", "A", "E", "AU", "ET"]
L.reverse()
print (L) # permet d'afficher : ['ET', 'AU', 'E', 'A', 'B', 'S']
print (L.index("E")) # permet d'afficher : 2
```

# Chapitre 3 : les listes

## *Liste d'une liste*

- Une liste d'une liste est un ensemble d'éléments dont **chaque élément** est **une liste.**
- Pour définir une liste, on utilise la syntaxe suivante :

nom_liste = [[element_1, element_2, …, element_n], [element_1, element_2, …, element_n], …, [element_1, element_2, …, element_n]]

## Exemple

```
Liste=[[12,"Janvier", 2000], [2,"Mai", 2005], [22,"Mars", 1999], [11,"Juin", 2002]]
print (Liste) # permet d'afficher : [[12, 'Janvier', 2000], [2, 'Mai', 2005], [22, 'Mars', 1999], [11, 'Juin', 2002]]
print(Liste[2]) # permet d'afficher : [22, 'Mars', 1999]
print(Liste[1][2]) # permet d'afficher : 2005
print(Liste[:][1]) # permet d'afficher : [2, 'Mai', 2005]
```

# Chapitre 3 : les listes

## *Compréhension de liste*

- Il s'agit d'une méthode simplifiée de créations de listes dans le but d'optimiser les programmes en python.
- **Syntaxe générale :**

nom_liste = [fonction(element) for element in nom_liste if condition]

### Exemple 1

```
liste_1=[1, 2, 3, 4, 5]
liste_2=[]
for x in liste_1 :
    liste_2.append(x*x)
print (liste_2) # permet d'afficher : [1, 4, 9, 16, 25]
```

Equivalent à :

```
liste_1=[1, 2, 3, 4, 5]
liste_2=[x*x for x in liste_1]
```

### Exemple 2

```
L1=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
L2=[]
for x in L1 :
    if x%2==0 :
        L2.append(x)
print (L2) # permet d'afficher : [2, 4, 6, 8, 10]
```

Equivalent à :

```
L1=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
L2=[x for x in L1 if x%2==0]
```

# Chapitre 3 : les listes

## *Exercice d'application*

On considère une liste L contenant les notes d'un groupe d'étudiants inscrits dans une formation donnée.

Écrire un script python qui permet de faire un ensemble de traitements selon un menu affiché à l'utilisateur. L'utilisateur quitte l'exécution du programme lorsqu'il choisit la valeur 11.

Le menu contient les éléments suivants :

1. Ajouter une note
2. Afficher la liste des notes des étudiants
3. Inverses la liste des notes des étudiants
4. Rechercher une note
5. Afficher la note maximale
6. Afficher la note minimale
7. Afficher la note moyenne du groupe
8. Trier la liste des notes d'une manière décroissante
9. Supprimer une note
10. Supprimer toutes les notes
11. Terminer l'exécution du programme

# Chapitre 3 : les tuples

## *Définition d'un tuple*

- C'est une structure de données assez semblable à une liste.
- Il s'agit d'**une séquence** d'éléments, **ordonnés**, de **même** ou de **différents types.** Mais **inchangeable** (non mutables).
- Un tuple se définit comme suit :

**nom_tuple = (element1, element_2, ..., element_n)**
**nom_tuple = element1, element_2, ..., element_n**

## Exemple

```
tuple_1 = () # permet de définir un tuple vide
tuple_2 = (1, 2, 3, 4) # permet de définir un tuple des entiers
print(tuple_2) # permet d'afficher : (1, 2, 3, 4)
tuple_3 = 1, 2, 3, 4 # permet de définir un tuple des entiers
tuple_4 = ("Lundi", 22, "Janvier", 2000) # permet de définir un tuple
print(tuple_4) # permet d'afficher : ("Lundi", 22, "Janvier", 2000)
```

# Chapitre 3 : les tuples

## *Opérations appliquées sur les tuples*

- Toutes les opérations appliquées sur les listes sont applicables sur les tuples : **+, ***
- Les méthodes utilisées pour manipuler les listes sont utilisées également avec les tuples à condition qu'elles modifient pas les éléments du tuple *(car un tuple est une séquence immuable)*
- Exemples des méthodes qui ne sont pas utilisées avec les tuples : *remove(), del, pop(), sort(), insert(), append, extend(), clear(), reverse(), etc.*

**Exemple**

```
tuple1, tuple2 = ("a","b"), ("c","d","e")
tuple3 = tuple1*4 + tuple2
print(tuple3) # permet d'afficher : ('a', 'b', 'a', 'b', 'a', 'b', 'a', 'b', 'c', 'd', 'e')
for x in tuple3 :
    print(x, end="-") # permet d'afficher : a-b-a-b-a-b-a-b-c-d-e-
```

# Chapitre 3 : les tuples

## *La méthode tuple()*

- La méthode **tuple()** est utilisée afin de convertir une liste ou une chaine de caractères vers le type **tuple**
- La syntaxe utilisée est la suivante :

**nom_tuple= tuple (nom_chaine)**

**nom_tuple=tuple(nom_liste)**

## Exemple

```
Liste_1 = [13, "B", 16.5, "AU", "et"]
Chaine_1 = "Bonjour"
tuple_1=tuple(Liste_1)
print(tuple_1) # permet d'afficher : (13, 'B', 16.5, 'AU', 'et')
tuple_2=tuple(Chaine_1)
print(tuple_2) # permet d'afficher : ('B', 'o', 'n', 'j', 'o', 'u', 'r')
```

# Chapitre 3 : le dictionnaire

## *Définition d'un dictionnaire*

- C'est une structure de données assez semblable à une liste.
- Il s'agit d'**une séquence** d'éléments, **mutables**, de **même** ou de **différents types.** Mais **non ordonnée**.
- Un dictionnaire se définit sous forme d'une **clef : valeur** :

  **nom_dictionnaire = { clef_1 : valeur, clef_2 : valeur, …, clef_n : valeur}**
- Les clefs d'un dictionnaire sont uniques

## Exemple

```
dico_1 = {} # permet de définir un dictionnaire vide
dico_2 = {1: "un", 2: "deux", 3: "trois", 4: "quatre"} # permet de définir un dictionnaire
print(dico_2) # permet d'afficher : {1: 'un', 2: 'deux', 3: 'trois', 4: 'quatre'}
dico_2[5] = "cinq" # permet d'ajouter l'élément 5 : "cinq" au dictionnaire dico_2
print(dico_2) # permet d'afficher : {1: 'un', 2: 'deux', 3: 'trois', 4: 'quatre', 5:'cinq'}
```

# Chapitre 3 : le dictionnaire

## *Manipulation de dictionnaires*

- Pour afficher les éléments d'un dictionnaire, trois méthodes sont utilisées : **items()**, **keys()** et **values()**.
- La méthode **items()** permet d'afficher les éléments d'un dictionnaire.
- La méthode **keyes()** permet d'afficher les clés d'un dictionnaire
- La méthode **values()** permet d'afficher les valeurs d'un dictionnaire

**Exemple**

```
dico = {1: "un", 2: "deux", 3: "trois", 4: "quatre"} # permet de définir un dictionnaire
print(dico) # permet d'afficher : {1: 'un', 2: 'deux', 3: 'trois', 4: 'quatre'}
print(dico.items()) # permet d'afficher : dict_items([(1, 'un'), (2, 'deux'), (3, 'trois'), (4, 'quatre')])
print(dico.keys())  # permet d'afficher : dict_keys([1, 2, 3, 4])
print(dico.values()) # permet d'afficher : dict_values(['un', 'deux', 'trois', 'quatre'])
```

# Chapitre 3 : le dictionnaire

## *Parcourir un dictionnaire*

### Exemple 1

```
dico = {1: "un", 2: "deux", 3: "trois"}
for x, y in dico.items() :
    print(x, y)
```

```
1 un
2 deux
3 trois
```

### Exemple 3

```
dico = {1: "un", 2: "deux", 3: "trois"}
for x in dico.values :
    print(x)
```

```
un
deux
trois
```

### Exemple 2

```
dico = {1: "un", 2: "deux", 3: "trois"}
for x in dico.keys :
    print(x)
```

```
1
2
3
```

# Chapitre 3 : le dictionnaire

## *La méthode dict()*

- La méthode **dict()** permet de convertir **une liste** *(liste de liste)* ou **un tuple** *(tuple de tuple)* vers un dictionnaire
- La méthode **dict()** est utilisée également pour créer un dictionnaire vide

**Exemple**

```
D0=dict() # permet de définir un dictionnaire vide
L=[[1, "Janvier"], [2, "Février"], [3, "Mars"]]
T=((4, "Avril"), (5, "Mai"), (6, "Juin"))
D1=dict(L)
D2=dict(T)
print(D1) # permet d'afficher : {1: 'Janvier', 2: 'Février', 3: 'Mars'}
print(D2) # permet d'afficher : {4: 'Avril', 5: 'Mai', 6: 'Juin'}
```

# Chapitre 3 : le dictionnaire

## *La méthode get()*

- La méthode **get()** renvoie la valeur de la clé donnée si elle est présente dans le dictionnaire. Sinon, il renverra None (si **get()** est utilisé avec un seul argument).
- La méthode get utilise la syntaxe suivante avec deux paramètres :

**nom_dictionnaire.get(clef, valeur_renvoyée)**

- **Clef :** le nom de clé de l'élément à renvoyer sa valeur
- **Valeur_renvoyée :** (facultatif) valeur à renvoyer si la clé n'est pas trouvée. La valeur par défaut est None.

**Exemple**

```
dico = {1: "un", 2: "deux", 3: "trois", 4: "quatre"} # permet de définir un dictionnaire
print(dico.get(2)) # permet d'afficher : deux
print(dico.get(5,"Element introuvable")) # permet d'afficher : Element Introuvable
```

# Chapitre 3 : le dictionnaire

## *La méthode pop()*

- La méthode **pop()** permet d'extraire la valeur équivalente à la clé précisée comme argument de la fonction et de la supprimer depuis les éléments du dictionnaire.
- La méthode **pop()** utilise la syntaxe suivante :

**Valeur_renvoyée=nom_dictionnaire.pop(clef)**

- **Clef :** le nom de clé de l'élément à supprimer
- **Valeur_renvoyée :** valeur équivalente au clef à supprimer, si la clé n'est pas trouvée. La valeur renvoyée sera None.

**Exemple**

```
dico = {1: "un", 2: "deux", 3: "trois", 4: "quatre"} # permet de définir un dictionnaire
print(dico.pop(2)) # permet d'afficher : deux
print(dico.get(5)) # permet d'afficher : None
print(dico) # permet d'afficher : {1: 'un', 3: 'trois', 4: 'quatre'}
```

# Chapitre 3 : le dictionnaire

## *Exercice d'application*

On considère un dictionnaire **Notes** contenant les notes obtenues par un groupe d'étudiants dans un module donnée.

Les clés de ce dictionnaire sont représentés par les noms des étudiants tandis que les valeurs des clés sont représentées par les moyennes générales obtenues.

Écrire un script python qui permet de :

1. Remplir ce dictionnaire par les noms ainsi que les notes de **N** étudiants *(N est choisi par l'utilisateur)*
2. Afficher la liste des étudiants avec leurs notes **triée par ordre alphabétique**
3. Diviser ce dictionnaire en deux sous dictionnaires (**valides**, **non_valides**) : le premier contient les étudiants ayant validé ce module, le deuxième contient les étudiants n'ayant pas validés ce module. Afficher les deux dictionnaires.
4. Rechercher puis afficher le nom de l'étudiant ayant **la note maximale**.

# Chapitre 3 : le dictionnaire

## *Solution*

```
# Question 1
Notes={}
N=int(input("Nombre des étudiants : "))
for i in range (N) :
   print("Etudiant " , i+1)
   nom=input("Nom = ")
   note=float(input("Note = "))
   Notes[nom]=note
# Question 2
print("Liste des étudiants par ordre alphabétique : ")
for x in sorted(Notes.keys()):
   print(x, Notes[x])
# Question 3
V={}
NV={}
for x,y in Notes.items():
   if y>=10:
      V[x]=y
   else :
      NV[x]=y
print("Les étudiants qui ont validé le module : ", V)
print("Les étudiants qui n'ont pas validé le module : ", NV)

# Question 4
def max_notes(dict_notes):
   L=[]
   for a,b in dict_notes.items() :
      if b==max(dict_notes.values()) :
         L.append(a)
   return L
print("Les étudiants ayant la note maximale sont : ")
print(max_notes(Notes))
```

# Chapitre 3 : le dictionnaire

## *Dictionnaire d'une structure de données*

- Les valeurs d'un dictionnaire peuvent être des valeurs simples pour chaque clef ou bien une structure de données *(liste, tuple ou même un dictionnaire)*
- La syntaxe utilisée pour créer un dictionnaire d'une structure de données

**nom_dictionaire = { clef_1 : liste_1, clef_2= liste_2,**

**...., clef_n : liste_n}**

## Exemple

```
dico_1 = {"Hiver": ["Décembre", "Janvier", "Février" ], "Printemps": ["Mars", "Avril", "Mai" ]}
dico_2={"Eté": ["Juin", "Juillet", "Août" ], "Automne": ["Septembre", "Octobre", "Novembre" ]}
dico_3={1: dico_1, 2 : dico_2}
print(dico_3)
""" permet d'afficher {1: {'Hiver': ['Décembre', 'Janvier', 'Février'], 'Printemps': ['Mars', 'Avril', 'Mai']}, 2: {'Eté': ['Juin', 'Juillet', 'Aôut'], 'Automne': ['Septembre', 'Octobre', 'Novembre']}} """
```

# Chapitre 3 : le dictionnaire

## *Exercice d'application*

- Ecrire un script python qui permet de saisir N valeurs numériques puis les classer dans un dictionnaire contenant deux listes : une liste des valeurs positives et une deuxième liste des valeurs négatives.
- Dictionnaire={
  "valeurs positives": [],
  "valeurs négatives": []
  }
- Le script affiche un message approprié dans le cas d'une valeur nulle

# Chapitre 3 : le dictionnaire

## *Solution*

```
Nombres={
    "Positif" : [],
    "Négatif" : []
    }
N=int(input("Entrer le nombre des valeurs à classer : "))
for i in range(N) :
    n=int(input ("entrer un nombre : "))
    if n>0 :
        Nombres["Positif"].append(n)
    elif n<0 :
        Nombres["Négatif"].append(n)
    else :
        print("Vous avez tapé une valeur nulle ")

print(Nombres)
```

# Chapitre 3 : structures de données

## Exercices : Série N3