

كلية العلوم والتقنيات بطنجة

Faculté des Sciences et Techniques de Tanger

Département Génie Informatique

# Module : Algorithmique avancé & Programmation python

Filière : Licence Analytique de Données

**Cours préparé et enseigné par :**

- Pr. Sanae KHALI ISSA

# Déroulement du cours

## Généralités

- Chapitre 1 : Bases de la programmation Python - **TD1**
  - Instructions de base : lecture/écriture/affectation
  - Structure conditionnelle
  - Structure itérative / boucle
  - L'instruction : break
  - L'instruction : continue
- Chapitre 2 : Fonctions & Modules - **TD2**
- Chapitre 3 : Chaines de caractères - **TD3**
- Chapitre 4 : Structures de données - **TD4**
  - Les types séquentiels : listes, tuples
  - Les types de correspondance : dictionnaire
- Chapitre 5 : Fichiers - **TD5**
- Chapitre 6 : Programmation orienté objet (POO) - **TD6**
- Chapitre 7 : Tableaux - **TD7**

## Projet de fin de module

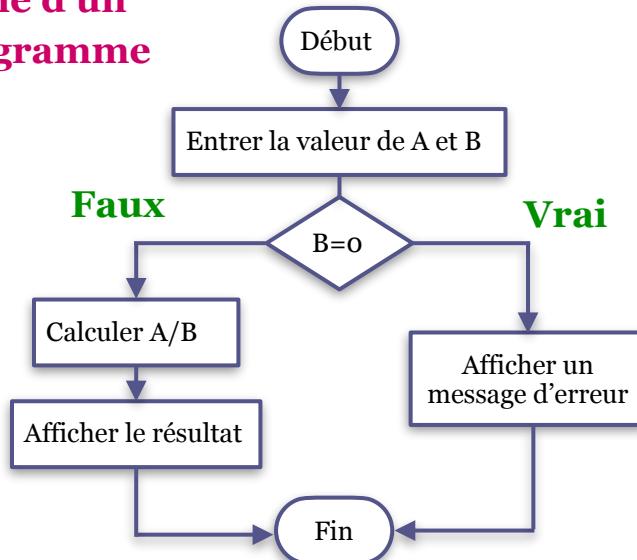
# Généralités

# Généralités

## Algorithme

- Une **description formelle** d'un procédé de traitement qui permet, à partir d'un ensemble d'informations initiales, d'obtenir des informations déduites;
- Un algorithme peut être représenté sous forme d'un **organigramme** ou d'un **pseudocode**

### Exemple d'un organigramme



### Exemple d'un pseudocode

```

Algorithme Division ;
Variables A, B : réel ;
Début
Ecrire ('Entrer les deux valeurs : ') ;
Lire (A,B);
if (B=0)
    Ecrire ('impossible de diviser par 0 ') ;
else
    Ecrire ('le résultat est : ', A/B) ;
Fin
  
```

# Généralités

## Programme

- **Programmer une machine** : expliquer en détail ce que doit faire une machine pour atteindre un but ou résoudre un problème
- Un **programme** est une suite d'instructions écrites dans un code bien précis appelé **langage de programmation** pouvant être compilés, interprétés ou exécutés pour donner un résultat bien déterminé.

## Types des programmes

- **Systèmes d'exploitation** : un ensemble des programmes qui gèrent les ressources matérielles et logicielles. Il propose une aide au dialogue entre l'utilisateur et l'ordinateur à travers une interface textuelle (**interpréteur de commande**) ou graphique (**gestionnaire de fenêtres**). Il est souvent **multitâche** et parfois **multiutilisateur** (*ex: Ms-Dos, windows, MacOs, Linux, etc.*)
- **Programmes d'application** : ils sont dédiés à des tâches particulières, formés d'une série de commandes contenues dans un programme source qui est transformé pour être exécuté par l'ordinateur (*ex: calculatrice, Excel, photoshop, etc.*)

# Généralités

## Langage de programmation

C'est une **notation conventionnelle** destinée à **formuler des algorithmes** afin de produire des **programmes informatiques**.

Un langage de programmation est construit à partir d'une **grammaire formelle**, qui inclut des **symboles** et des **règles** syntaxiques.

## Types des langage de programmation

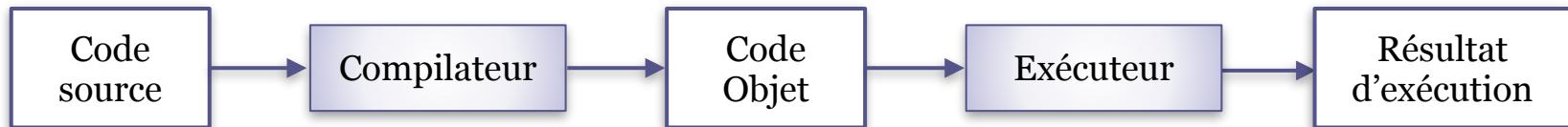
- **Langage machine** : un langage proche de la machine, formé de 0 et 1, propre à chaque processeur, il n'est pas portable.
- **Langage d'assemblage** : un codage alphanumérique du langage machine. plus lisible que le langage machine, mais n'est toujours pas portable. traduit en langage machine par un **assembleur**
- **Langage de haut niveau** : compréhensible par l'être humain, portable d'une machine à une autre, traduit en langage machine par un  **compilateur** ou un **interpréteur**.

# Généralités

## Compilation

C'est la traduction du code source (rédigé en utilisant un langage de programmation) en langage objet (langage binaire). Elle comprend au moins quatre phases :

- Phase d'analyse lexicale,
- Phase d'analyse syntaxique
- Phase d'analyse sémantique
- Et une phase de production de code objet



**Ex :**  
**Langage**  
**C**

**Langage  
binaire  
0 et 1**

# Généralités

## Interprétation

C'est l'analyse et l'exécution de chaque ligne du code source (rédigé en utilisant un langage de programmation) sans la génération du code objet.



**Ex :**

**Langage  
python**

# Généralités

## **Difference entre Interpréteur et Compilateur**

Interpréteur	Compilateur
<ul style="list-style-type: none"> <li>Convertit le programme en prenant une seule ligne à la fois.</li> </ul>	<ul style="list-style-type: none"> <li>Analyse l'ensemble du programme et le traduit dans son ensemble en code machine.</li> </ul>
<ul style="list-style-type: none"> <li>L'analyse du code source prend moins de temps, mais le temps d'exécution global est plus lent.</li> </ul>	<ul style="list-style-type: none"> <li>L'analyse du code source prend beaucoup de temps, mais le temps d'exécution global est comparativement plus rapide.</li> </ul>
<ul style="list-style-type: none"> <li>Aucun code d'objet intermédiaire n'est généré, la mémoire est donc efficace.</li> </ul>	<ul style="list-style-type: none"> <li>Génère du code d'objet intermédiaire qui nécessite en outre une liaison, nécessite donc davantage de mémoire.</li> </ul>
<ul style="list-style-type: none"> <li>Continue de traduire le programme jusqu'à ce que la première erreur soit rencontrée. Par conséquent, le débogage est facile.</li> </ul>	<ul style="list-style-type: none"> <li>Il génère le message d'erreur uniquement après avoir analysé l'ensemble du programme. Par conséquent, le débogage est relativement difficile.</li> </ul>

# Généralités

## Type de programmation

Généralement, on distingue entre deux types de programmation : **programmation procédurale** et **programmation orienté objet**

### Programmation procédurale

- La programmation procédurale est un paradigme de programmation qui repose sur le concept de procédures ou fonctions.
- Dans la programmation procédurale, un programme est divisé en petites unités autonomes appelées procédures contenant des séquences d'instructions à exécuter. Ces procédures peuvent être appelées dans un ordre spécifique pour accomplir une tâche plus complexe.

#### Exemple :

**Ada, FoxPro, Modula-2, BASIC, Pascal, C, ALGOL, COBOL, Fortran**

# Généralités

## Type de programmation

Généralement, on distingue entre deux types de programmation : **programmation procédurale** et **programmation orienté objet**

### Programmation orienté objet

- La programmation orientée objet (POO) est un paradigme de programmation qui utilise des **objets**, qui sont des **instances de classes**, pour organiser et structurer le code.
- Elle repose sur plusieurs concepts clés à savoir : **l'encapsulation, l'abstraction, l'héritage, le polymorphisme**, etc.

#### Exemple :

**C++, C, Java, Python, Ruby, Javascript, etc.**

# Généralités

## Classe

- Un type d'une variable complexe
- Un modèle ou un plan pour créer des objets. Elle définit les propriétés (**attributs**) et les comportements (**méthodes**) communs à tous les objets créés à partir de cette classe.

## Exemple d'une classe en python

```
class Etudiant
    def __init__(self, nom, age, note):
        self.nom = nom
        self.age = age
        self.note = note

    def afficher_informations(self):
        print(f"Nom: {self.nom}")
        print(f"Âge: {self.age}")
        print(f"Note: {self.note}")

    def modifier_note(self, nouvelle_note):
        self.note = nouvelle_note
```

# Généralités

## Objet

- Une instance concrète d'une classe
- Il représente une entité du monde réel et possède des caractéristiques (**attributs**) et des actions (**méthodes**) associées à la classe à partir de laquelle il a été créé.

## Création d'un objet en python

```
# Création d'une instance de la classe Etudiant
etudiant1 = Etudiant("Alice", 20, 85.5)

# Affichage des informations initiales
print ("Informations initiales de l'étudiant : ")
etudiant1.afficher_informations()

# Mise à jour de la note de l'étudiant
etudiant1.modifier_note(90.0)
```

# Généralités

## Présentation du langage python

### Langage simple

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

### Open source

- Licence Open Source
- Python est libre et gratuit même pour les usages commerciaux
- Importante communauté de développeurs
- Nombreux outils standard disponibles

### Langage interprété

- Langage interprète rapide
- Interprétation du bytecode compilé
- De nombreux modules sont disponibles à partir de 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

# Généralités

## Présentation du langage python

### Travail interactif

- Nombreux interpréteurs interactifs disponibles
- Importantes documentations en ligne
- Développement rapide et incrémentiel
- Tests et déboggage outillés
- Analyse interactive de données

### 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
- Plusieurs milliers de packages sont disponibles dans tous les domaines
- Possibilité d'ajouter d'autres fonctions et modules

# Généralités

## **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 Foundation**)

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**
- **6 septembre 2022** : sortie de la dernière version stable de python : **v3.7. 14**

# Généralités

## 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.



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

# Chapitre 1 : Bases de la programmation Python

# Chapitre 1 : instructions de base

## Structure d'un programme python

- Un programme python ne possède pas une structure bien précise
- Un programme python contient plusieurs types d'instructions qui répondent au problème défini.
  - Instructions d'affichage
  - Instructions de lecture
  - Instructions d'affectation et de calcul
  - Instructions alternativs
  - Instructions répétitives
  - Des commentaires, ...

- **Pour ajouter un commentaire sur une seule ligne**

# exemple de commentaire

- **Pour ajouter un commentaire sur plusieurs lignes**

""" première ligne de commentaire

deuxième ligne de commentaire """

# Chapitre 1 : instructions de base

## Mots réservés au langage python

False	class	from	or	None	continue
global	pass.	True	def	if	raise
and	del	return	break	for	not
as	elif	in	try		
assert	else	is	while		
async	except	lambda	with		
await	finally	nonlocal	yield		

**Pour afficher l'aide sur un keyword ou une fonction python, il suffit de taper la commande :**

**>>> help(nom du keyword ou nom de la fonction)**

# Chapitre 1 : instructions de base

## Définition d'une donnée

- C'est **un emplacement mémoire** dans lequel on peut mémoriser une valeur.
- Une donnée est caractérisée par **un identificateur, une valeur, et un type**.
- Une donnée peut être
  - **variable** : s'elle change de valeurs dans un programme
  - **Constante** : s'elle garde la même valeur tout au long du programme

# Chapitre 1 : Instructions de base

## Identificateur des données

**L'identificateur** est un nom choisi pour désigner une donnée,

### Il doit :

- Être formé des **lettres** (A - Z) (a - z), des **chiffres** (1 - 9) et des **lignes de soulignement** (\_)
- Commencer obligatoirement par une lettre.

### Il ne doit pas

- Contenir des espaces
- Être un mot réservé du langage python

# Chapitre 1 : Instructions de base

## Types natifs de données

- Il représente le **type de l'ensemble des valeurs** que peut prendre une donnée.
- En python, le **typage des données est dynamique** : le type d'une donnée est déterminée selon la valeur attribuée à cette donnée
- Pour savoir le type d'une donnée, il suffit d'utiliser la fonction **type** :

```
>>> type (nom_donnee)
```

### Ex:

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

# Chapitre 1 : Instructions de base

## Types natifs des données

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

# Chapitre 1 : instructions de base

## Opérateurs arithmétiques

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

## Opérateurs logiques

<b>and</b>	ET logique
<b>or</b>	OU logique
<b>not</b>	Négation logique

# Chapitre 1 : instructions de base

## Opérateurs de comparaison

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

## Opérateurs de conversion

<b>int(x)</b>	Convertir x vers un nombre entier
<b>float (x)</b>	Convertir x vers un nombre réel
<b>str(x)</b>	Convertir x vers une chaîne de caractères

# Chapitre 1 : instructions de base

## 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
```

### NB :

Lorsque les types des deux opérandes sont différents il y'a conversion implicite vers le type de la variable résultante.

# Chapitre 1 : instructions de base

## Opérateurs d'assignation

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

### 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 : instructions de base

## Instruction d'écriture : print

C'est la fonction qui permet d'afficher des messages ou les valeurs des données à l'utilisateur.

### Syntaxe générale

```
print (identificateur_donnee)
print (iden_donnee1, iden_donnee2, ...)
print (" message ")
print ( " message " , identificateur_donnee)
print ( f " message { identificateur_donnee } " )
```

### Résultat d'exécution



### Exemple

```
print ( " La somme est : ")
print (S)
print ( f " La somme est : {S} " )
```

# Chapitre 1 : instructions de base

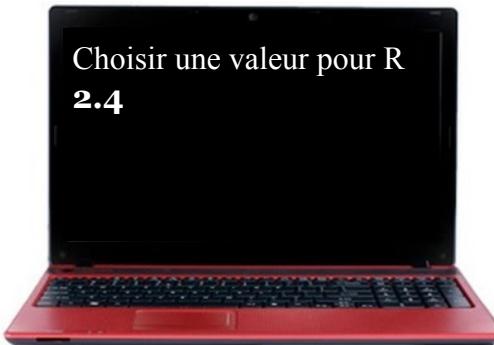
## Instruction de lecture des variables : input

- C'est la fonction qui permet à l'utilisateur de fournir au programme les valeurs des variables.
- La fonction **input()** renvoie une valeur de type chaîne 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

```
iden_var = input ()  
***  
iden_var = input ( "message ")
```

### Résultat d'exécution



### Exemple

```
R=float(input ('Choisir une valeur pour R'))
```

# Chapitre 1 : instructions de contrôle

## 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) **and** (A % B == 0)

Expression logique  
simple

Expression logique  
simple

# Chapitre 1 : structure conditionnelle

## **L'instruction : if**

**if** expression logique :

Instruction 1

Instruction 2

...

Instruction n

Instructions à exécuter si  
l'expression logique est  
vraie

## Exemple

**if** X > 0 :

    print ( " c'est un nombre positif " )

# Chapitre 1 : structure conditionnelle

## **L'instruction : 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 > 0 :

```
print (" c'est un nombre positif")
```

else :

```
print (" c'est un nombre négatif  
ou nul " )
```

# Chapitre 1 : structure conditionnelle

## *L'instruction : 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 : structure conditionnelle

## 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 : structure conditionnelle

## *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**}

à

# Chapitre 1 : structure conditionnelle

## 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 : structure itérative

## 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 : structure itérative

## La boucle for

```
for compteur in liste_valeurs :
```

    Instruction 1

    Instruction 2

    ...

    Instruction n

à exécuter pour  
chaque valeur du  
compteur

## Résultat d'exécution

```
x prend la valeur 1
x prend la valeur 2
x prend la valeur 3
x prend la valeur 4
x prend la valeur 5
```



## Exemple

```
for x in [1, 2, 3, 4, 5] :
```

```
    print (" x prend la valeur ", x)
```

# Chapitre 1 : structure itérative

## 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 : structure itérative

## **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 : structure itérative

## **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 : l'instruction break

## 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 : l'instruction continue

## 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 1 : Bases de la programmation Python

## Exercices : Série N1

# 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

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

## Résultat d'exécution

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

# Chapitre 2 : Fonctions

## Exemple 2

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

## 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

## 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 attributs)
  - **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

## 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

## 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 N2

# Chapitre 3 : Chaines de caractères

# Chapitre 3 : chaines de caractères

## Définition

- Une chaîne 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 chaînes de caractères

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

# Chapitre 3 : chaines de caractères

## Manipulation des chaines de caractères

- Les opérateurs mathématiques qui peuvent être appliqués sur une chaîne de caractères sont : \* et +
- Pour la comparaison des chaînes 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 chaîne, 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 chaîne 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'  
print(ch3[-12:-1])  
# permet d'afficher les caractères ayant indice de -2 à -12 : 'programmeur'
```

# 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 chaîne de caractères (longueur d'une chaîne).
- La fonction **bool()** permet de vérifier si une chaîne est vide ou non, dans le 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 : chaînes de caractères

## **Méthodes associées aux chaînes 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 : chaînes de caractères

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

- La fonction **`startswith()`** est une fonction qui permet de vérifier si une chaîne 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 chaîne 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 : chaînes 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 : chaînes de caractères

## ***La méthode : count ()***

- La méthode **count()** permet de compter le nombre d'occurrences d'une sous-chaîne dans une chaîne.
- Paramètres des fonctions **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 : chaînes de caractères

## ***Les méthodes : `find()`, `rfind()`***

- La méthode **`find()`** permet de rechercher la position de la première occurrence d'une sous-chaîne dans une chaîne.
- La méthode **`rfind()`** permet de rechercher la position de la dernière occurrence 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 O***

- La méthode **replaceO** permet de remplacer des caractères d'une chaîne 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 **replaceO 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 O** 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 chaîne de caractères en **majuscules**.
- La méthode **lower()** permet de convertir une chaîne de caractères en **minuscules**.
- La méthode **swapcase()** permet de convertir les lettres minuscules d'une chaîne 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 : chaînes 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 chaîne** 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 lstrip"
print(chaine3.lstrip("c'est ")) #permet d'afficher : un test pour la méthode lstrip
```

# Chapitre 3 : chaines de caractères

## Exercices : Série N3

# Chapitre 4 : Structures de données

# Chapitre 4 : 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 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 4 : 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 = [element<sub>1</sub>, element<sub>2</sub>, ..., element<sub>n</sub>]**

## 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
```

# Chapitre 4 : 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 vide  
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 4 : 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 4 : les listes

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

- La fonction **split()** permet de transformer une chaîne 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 signifie « toutes les occurrences »

### Exemple

```
chaine = "exemple de la fonction split" # permet de définir une chaîne 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 4 : 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 4 : les listes

## Accéder aux éléments d'une liste

- Pour accéder aux éléments d'une liste, il est possible d'utiliser :
  - L'indication **positif** (0, 1, 2, etc.)
  - L'indication **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 4 : 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 4 : 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 4 : 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 4 : 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 4 : 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 4 : 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 4 : 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, 11, 12]
```

# Chapitre 4 : 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 chaînes de caractères sont triées selon l'ordre alphabétique.
- Pour spécifier le type de tri à appliquer (croissant ou décroissant), on peut utiliser la méthode `sort()` 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 4 : 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 occurrence d'une valeur dans une liste

### 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 4 : 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 4 : 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 4 : les listes

## ***Exercice d'application***

On considère une liste Notes 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 7.

Le menu contient les éléments suivants :

1. Ajouter une note
2. Afficher la liste des notes des étudiants
3. Afficher la note maximale
4. Afficher la note minimale
5. Afficher la note moyenne du groupe
6. Trier la liste des notes d'une manière décroissante
7. Terminer l'exécution du programme

# Chapitre 4 : les listes

## Solution

```

L=[]
def menu():
    print("\n1: Ajouter une note")
    print("2: Afficher la liste des notes des étudiants")
    print("3: Afficher la note maximale ")
    print("4: Afficher la note minimale")
    print("5: Afficher la note moyenne du groupe")
    print("6: Trier la liste des notes d'une manière
décroissante")
    print("7: Terminer l'exécution du programme")
    ch=int(input("Entrer votre choix :"))
    return(ch)
def afficher_liste(nom_liste):
    for x in nom_liste :
        print(x, end=" ")
choix=0
while choix!=7:
    choix=menu()
    match choix :

```

```

case 1:
    note=float(input("entrer votre note :"))
    L.append(note)
    print("note ajoutée avec succès")
case 2:
    if len(L)==0 :
        print("Liste vide ")
    else :
        print("Liste des notes :")
        afficher_liste(L)
case 3:
    print("la note maximale est ", max(L))
case 4:
    print("la note minimale est ", min(L))
case 5:
    print("la note moyenne est ", sum(L)/len(L))
case 6:
    L.sort(reverse=True)
    print("Liste triée avec succès")
case 7:
    print("Fin du programme")
case other :
    print("Veuillez saisir une valeur comprise entre 1 et 7 ")

```

# Chapitre 4 : 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 4 : 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 4 : les tuples

## ***La méthode tuple()***

- La méthode **tuple()** est utilisée afin de convertir une liste ou une chaîne 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 4 : 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 **clé : 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 4 : 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 4 : 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 4 : 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 4 : 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 4 : 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(2)) # permet d'afficher : None
print(dico) # permet d'afficher : {1: 'un', 3: 'trois', 4: 'quatre'}
```

# Chapitre 4 : 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 4 : 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 4 : 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_dictionnaire = { 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', 'Août'], 'Automne': ['Septembre', 'Octobre', 'Novembre']}} """

```

# Chapitre 4 : 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 4 : 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 4 : structures de données

## Exercices : Série N4

# Chapitre 5 : Les fichiers

# Chapitre 5 : les fichiers

## Définition

Un fichier est un **ensemble structuré de données** regroupées et enregistrées sur un support de stockage

Un fichier est identifié par **son nom** et **son format** (*txt, csv, json, xlsx, docx, rtf, xml, etc.*)

Chaque fichier est localisé grâce à **son chemin absolu** ou **son chemin relatif**

- **Le chemin relatif** d'un fichier est déterminé à partir d'un emplacement donné emplacement de lecture (ex : programs/ex1.py)
- **Le Chemin absolu** est un chemin complet déterminé à partir de la racine (ex : "/Users/sanakh/Desktop/exemples programmes python/fichier.py")

# Chapitre 5 : les fichiers

## **Vérifier l'existence d'un fichier**

Afin de vérifier l'existence d'un fichier ou non, on peut utiliser la fonction **exists()** du module **path** qui appartient au module Python standard **os**.

Cette fonction renvoie :

- **True** si le chemin du fichier passé est un chemin qui existe
- **False** si le chemin n'existe pas.

## Exemple

```
import os
if os.path.exists("/Users/sanakh/Desktop/exemples programmes python/somme.py") == True :
    print("Le fichier existe")
else :
    print("Le fichier n'existe pas ")
```

# Chapitre 5 : les fichiers

## ***La méthode `getcwd()` et `chdir()`***

- Les deux méthodes **`getcwd()`** et **`chdir()`** existent dans le module standard **os** de python.
- La méthode **`getcwd()`** sert à afficher le répertoire courant
- La méthode **`chdir(nouveau_chemin)`** permet de changer le chemin du répertoire courant vers un autre utilisé comme argument de la fonction.

### Exemple

```
import os
print(os.getcwd()) # permet d'afficher : /Users/sanakh
os.chdir("/Users/sanakh/Desktop/exemples programmes python")
print(os.getcwd()) # permet d'afficher : /Users/sanakh/Desktop/exemples programmes python
```

# Chapitre 5 : les fichiers

## ***Supprimer un fichier***

Pour supprimer un fichier à partir d'un emplacement donnée, on peut utiliser la fonction **remove()** du module Python standard **os**.

La méthode **remove()** renvoie une erreur si le fichier à supprimer n'existe pas.

### **Exemple**

```
import os
if os.path.exists("/Users/sanakh/Desktop/exemples programmes python/fich.py") :
    os.remove("/Users/sanakh/Desktop/exemples programmes python/fich.py")
    print("Fichier supprimé avec succès ")
else :
    print("Le fichier à supprimer n'existe pas ")
```

# Chapitre 5 : les fichiers

## ***Ouverture et fermeture d'un fichier***

- Pour ouvrir un fichier, on peut utiliser la fonction **open()**
- La méthode **open()** utilise deux arguments : le chemin du fichier à ouvrir et le mode d'ouverture

```
descripteur_fichier = open("nom_fichier", "mode_ouverture")
```

**Avec :**

**nom\_fichier** : une chaîne de caractère constante ou bien une variable de type **string**

**mode\_ouverture** : un paramètre servant à déterminer le mode d'ouverture de fichier

- Pour fermer un fichier, on utilise la fonction **close()**

# Chapitre 5 : les fichiers

## **Modes d'ouverture d'un fichier**

Mode	Description
r : read	Ouvrir un fichier déjà existant en lecture seule. Le pointeur interne est placé au début du fichier.
w : write	Ouvrir un fichier en écriture seule. Si le fichier existe, les informations existantes seront écrasées. Sinon, le fichier sera créé.
a : append	Ouvrir un fichier en écriture seule en conservant les données existantes. Le pointeur interne est placé en fin de fichier et les nouvelles données seront donc ajoutées à la fin. Si le fichier n'existe pas, il sera créé.

## Exemple

```
id_fichier=open("/Users/sanakh/Desktop/exemples programmes python/somme.py", "r")
print (id_fichier)
"""
Permet d'afficher <_io.TextIOWrapper name='/Users/sanakh/Desktop/exemples programmes python/
somme.py' mode='r' encoding='UTF-8'> « """
id_fichier.close() # permet de fermer le fichier identifié par son descripteur id_fichier
```

# Chapitre 5 : les fichiers

## **Ouverture d'un fichier**

- Afin d'ouvrir un fichier, on peut utiliser la fonction **open()** associé avec le mot **with**
- La méthode **open()** utilise deux arguments : le chemin du fichier à ouvrir et le mode d'ouverture

**with open("nom\_fichier", "mode\_ouverture") as descripteur\_fichier :**

**Avec :**

**nom\_fichier** : une chaîne de caractère constante ou bien une variable de type **string**

**mode\_ouverture** : un paramètre servant à déterminer le mode d'ouverture de fichier

## Exemple

```
with open("/Users/sanakh/Desktop/exemples programmes python/somme.py", "a") as d_fichier :  
    print (d_fichier) # permet d'afficher les informations sur le fichier identifié par d_fichier
```

# Chapitre 5 : les fichiers

## **Lire le contenu d'un fichier**

- Avant de commencer la lecture d'un fichier, il faut l'ouvrir en mode lecture.
- Pour lire le contenu d'un fichier, on peut utiliser la boucle **for** qui permet de parcourir le fichier *ligne par ligne*.

**for nom\_variable in descripteur\_fichier :**

print (nom\_variable)

### Exemple

```
id_fichier=open("/Users/sanakh/Desktop/exemples programmes python/somme.py", "r")
for x in id_fichier :
    print (x) # permet d'afficher le contenu du fichier somme.py
id_fichier.close()
```

# Chapitre 5 : les fichiers

## **Lire le contenu d'un fichier : `read()`**

- Avant de commencer la lecture d'un fichier, il faut l'ouvrir en mode lecture.
- Pour lire le contenu d'un fichier, on peut utiliser la fonction **`read()`**

**`valeur_renvoyée = id_fichier.read()`**

Avec : valeur\_ renvoyée contient le contenu du fichier à lire

### Exemple

```
id_fichier=open("/Users/sanakh/Desktop/exemples programmes python/somme.py", "r")
print (id_fichier.read()) # permet d'afficher le contenu du fichier somme.py
id_fichier.close()
```

### Résultat d'exécution

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

# Chapitre 5 : les fichiers

## **Lire le contenu d'un fichier : readlines()**

- Avant de commencer la lecture d'un fichier, il faut l'ouvrir en mode lecture.
- La fonction **readlines()** permet de lire le contenu d'un fichier ligne par ligne puis stocker ces lignes dans une liste (*chaque ligne représente un élément de la liste*)

**nom\_liste = id\_fichier.readlines()**

### Exemple

```
id_fichier=open("/Users/sanakh/Desktop/exemples programmes python/semaine.txt », "r")
Liste_1= id_fichier.readlines()
print (Liste_1)
# permet d'afficher : ['Lundi 28 Novembre 2022\n', 'Mardi 29 Novembre 2022\n', 'Mercredi 30 Novembre 2022\n',
'Jeudi 01 Décembre 2022\n', 'Vendredi 02 Décembre 2022\n', 'Samedi 03 Décembre 2022\n', 'Dimanche 04
Décembre']
id_fichier.close()
```

# Chapitre 5 : les fichiers

## **Ecrire dans un fichier : write()**

- Avant de commencer l'écriture dans un fichier, il faut l'ouvrir en mode **d'écriture** ou mode **ajout**.
- La fonction **write()** permet d'ajouter un texte dans un fichier  
**valeur\_renvoyée = write(texte à ajouter)**

Avec:

**valeur\_renvoyée** : elle représente le nombre de caractères ajoutés dans le fichier

**Texte à ajouter** : l'élément à ajouter dans le fichier, il peut être une valeur constante ou une variable de type str

## **Exemple**

```
id_fichier=open("/Users/sanakh/Desktop/exemples programmes python/semaine.txt ", "w")
v=id_fichier.write("c'est un texte à ajouter dans le fichier texte")
print (f"le nombre de caractères ajoutés au fichier est {v}")
id_fichier.close()
```

# Chapitre 5 : les fichiers

## ***Calculer la taille d'un fichier avec la méthode getsize()***

- La méthode **getsize()** appartient au module **path** du module standard **os de python.**
- La fonction **getsize()** permet de calculer la taille d'un fichier en octets

**valeur\_renvoyée = getsize(nom\_fichier)**

**Avec:**

**valeur\_renvoyée :** elle représente le nombre d'octets du fichier

## **Exemple**

```
import os  
x=os.path.getsize("exemple_fichier.txt")  
print("le fichier que vous avez choisi est de taille ", x, "octets")
```

## Chapitre 5 : les fichiers

### Exercices : Série N5

# Chapitre 6 : Programmation orientée objet

# Chapitre 6 : Programmation orientée objet

## *C'est quoi la programmation orientée objet ?*

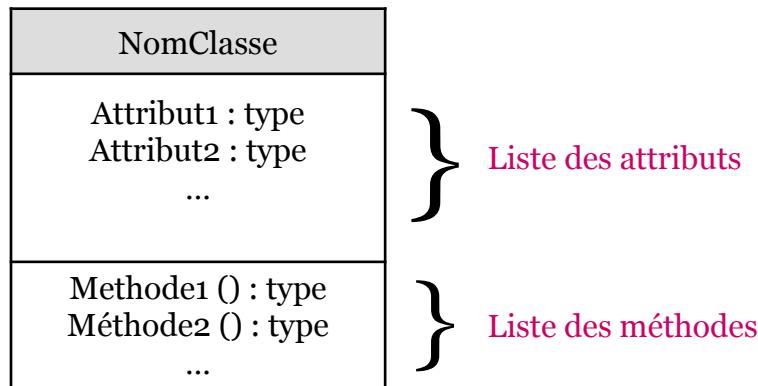
- **La programmation orientée objet (POO)** est une méthode pour concevoir des programmes informatiques d'une manière organisée et bien structurée.
- **La programmation orientée objet** est un paradigme de programmation basé sur deux notions extrêmement puissants : **objet** et **Classe**.
  - **Objet** : instantiation d'une classe (**représentation concrète**)
  - **Classe** : Modèle abstrait qui regroupe une famille des objets ayant même structure et même actions (**représentation abstraite**)
    - **Ex :** modèle d'une voiture, modèle d'un ordinateur, modèle d'un appartement. Etc.
- La POO repose sur plusieurs concepts clés à savoir : **l'encapsulation, l'héritage, l'abstraction et le polymorphisme**

# Chapitre 6 : Programmation orientée objet

## C'est quoi une classe ?

- Une classe sert à créer de nouveaux types des variables
- Elle permet en fait de regrouper un ensemble des variables (appelées **attributs**) et un ensemble des fonctions (appelées **méthodes**) dans un même type.
- Une classe se considère comme un modèle ou un plan pour créer des objets.

### Représentation graphique d'une classe



### Exemple d'une classe

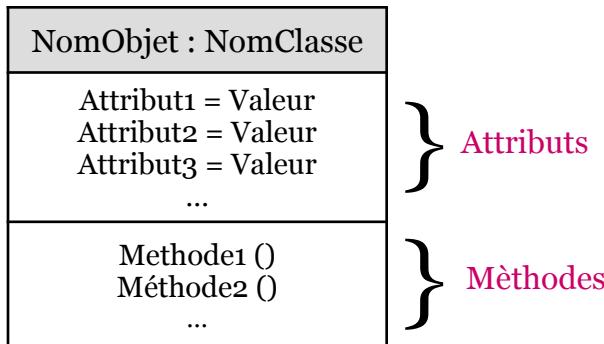
Voiture
Matricule : string Marque : string Modèle : int Couleur : string
Afficher_info () Démarrer () Arrêter ()

# Chapitre 6 : Programmation orientée objet

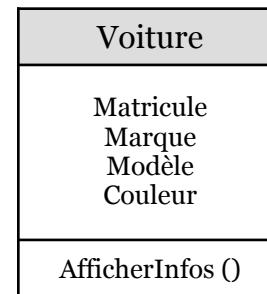
## C'est quoi un objet ?

- Une instance concrète d'une classe
- Il représente une entité du monde réel qui possède des caractéristiques (**attributs**) et des actions (**méthodes**) associées à la classe à partir de laquelle il a été créé.

### Représentation graphique d'un objet

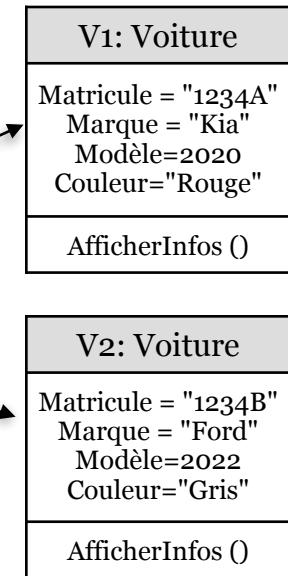


### Exemple des objets



Instantiation

Instantiation

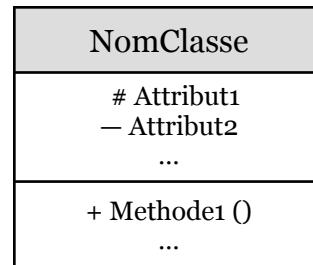


# Chapitre 6 : Programmation orientée objet

## Encapsulation

- L'encapsulation est l'un des principes fondamentaux de la programmation orientée objet.
- Elle permet de regrouper des attributs et des méthodes dans une classe et de cacher leurs visibilité.
  - **Visibilité par défaut** : aucun modificateur de visibilité n'est indiqué.
  - **Visibilité publique (+)** : les fonctions de toutes les classes peuvent accéder aux données ou aux méthodes d'une classe définie avec le niveau de visibilité « public ».
  - **Visibilité protégée (#)** : l'accès aux données est réservé aux fonctions des classes héritières, c'est-à-dire par les fonctions membres de la classe et des classes dérivées.
  - **Visibilité privée (-)** : l'accès aux données est limité aux méthodes de la classe elle-même.

### Représentation graphique d'une classe



### Exemple d'une classe

Stagiaire
— CINStagiaire
+ NomStagiaire
# AgeStagiaire
AfficherInfos ()

# Chapitre 6 : Programmation orientée objet

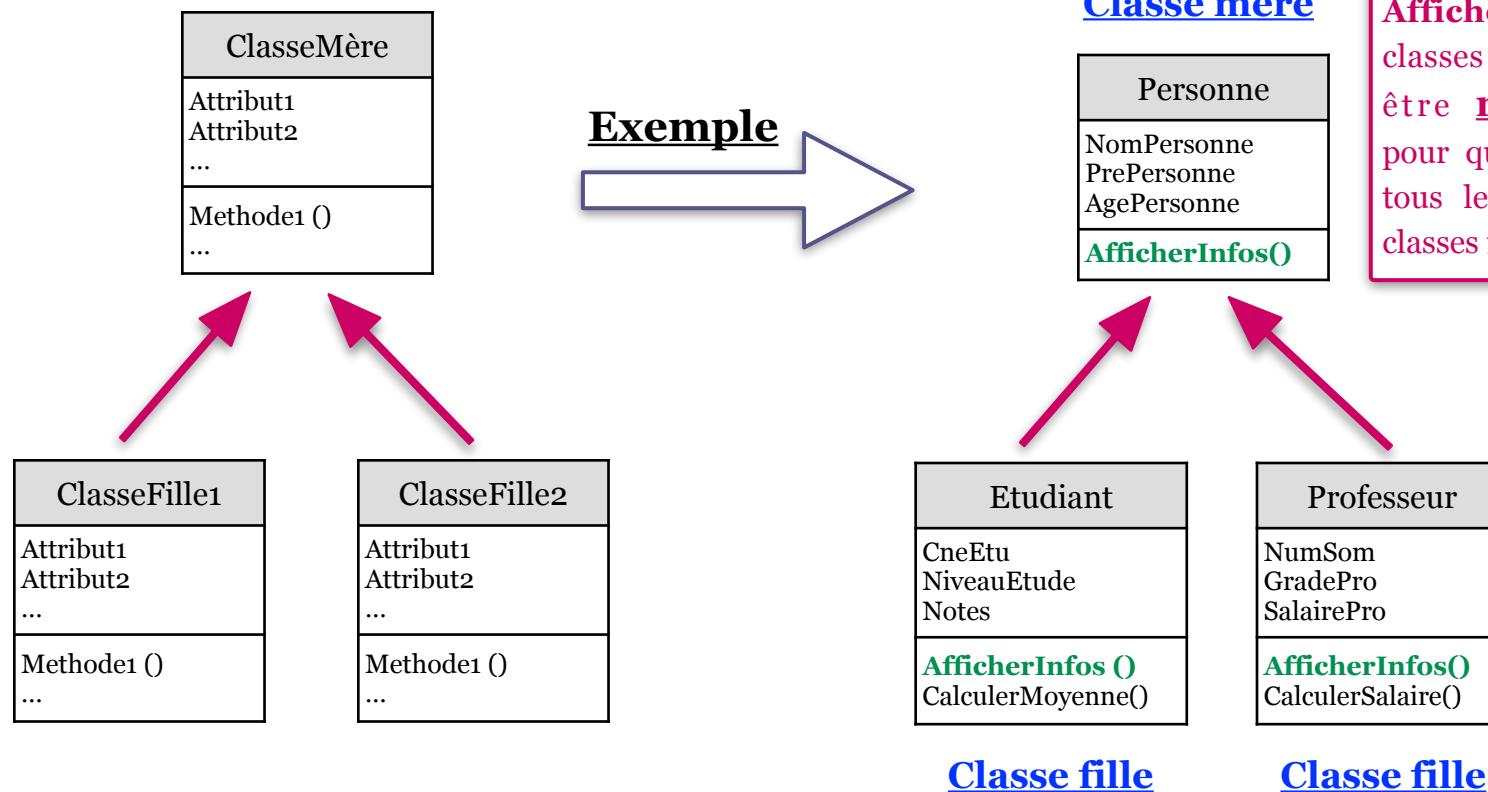
## Héritage

- L'héritage est un concept fondamental de la programmation orientée objet (POO) qui permet à une classe, appelée « **classe dérivée** », « **sous-classe** » ou « **classe fille** », d'hériter des attributs et des méthodes d'une autre classe, appelée « **classe de base** », « **superclasse** » ou « **classe mère** ».
  - Classe de base (classe mère) **donne TOUT** aux classes filles (méthodes et attributs)
  - Classes dérivées (classes filles) :
    - **héritent TOUT** de ses classes mères ( méthodes et attributs)
    - peuvent **avoir D'AUTRES** attributs et méthodes,
    - peuvent **MODIFIER** ce qu'elles ont hérité.
- On distingue entre deux types d'héritage :
  - **Heritage simple** : une classe qui hérite des attributs et des méthodes d'une seule super classe
  - **Heritage multiple** : une classe qui hérite des méthodes et des attributs de plusieurs super classes

# Chapitre 6 : Programmation orientée objet

## Héritage simple

### Représentation graphique de l'héritage simple



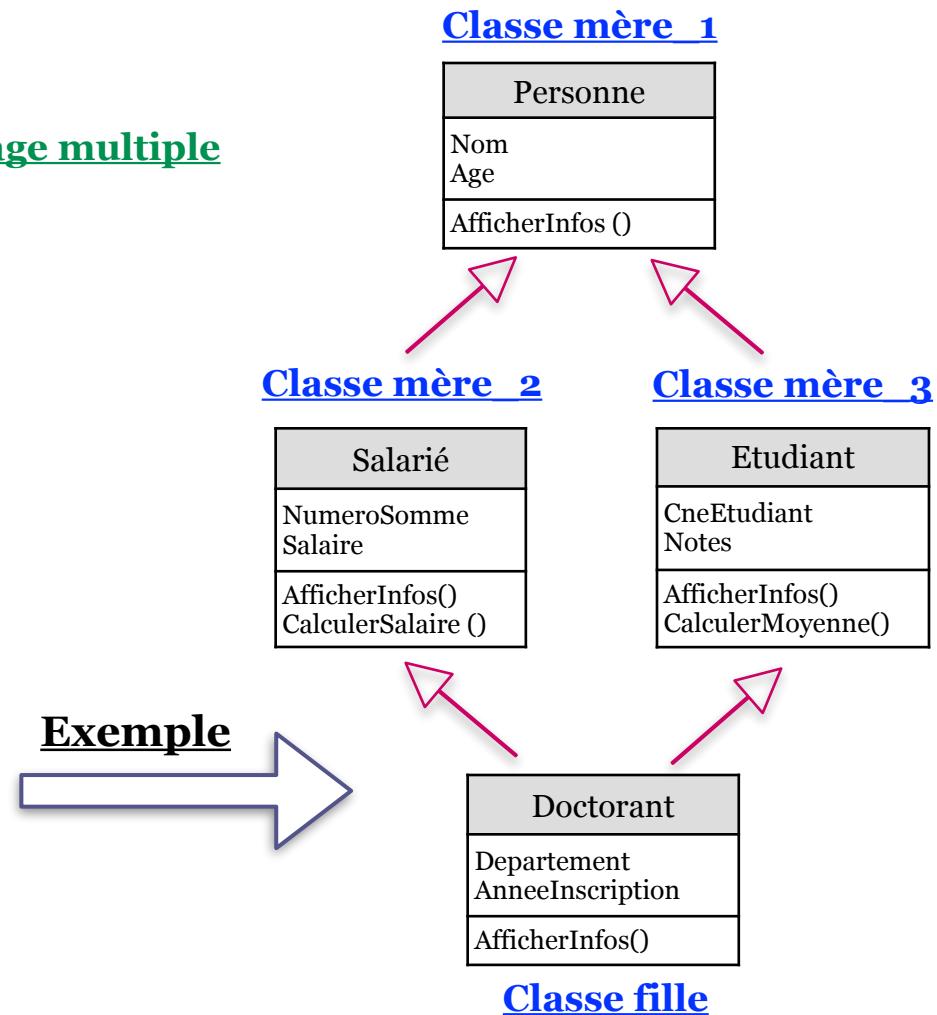
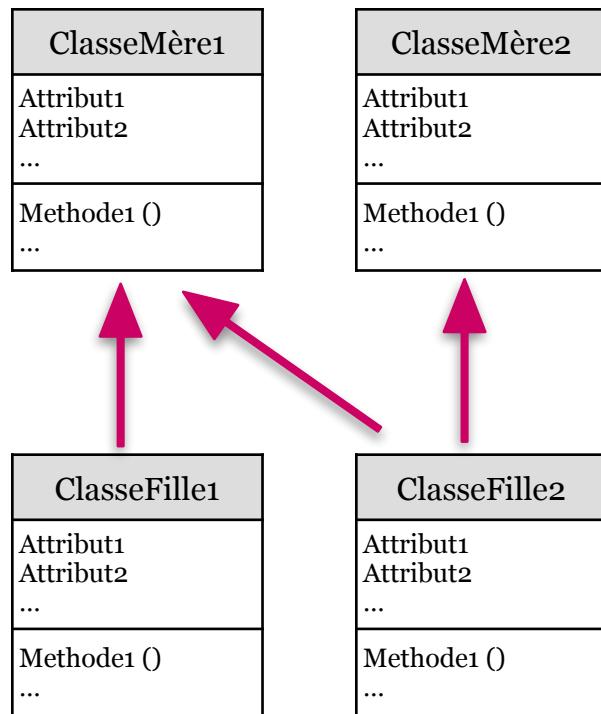
Les méthodes

**AfficherInfos()** des classes filles doivent être **redéfinies** pour qu'elles affichent tous les attributs des classes filles

# Chapitre 6 : Programmation orientée objet

## Héritage multiple

### Représentation graphique de l'héritage multiple



# Chapitre 6 : Programmation orientée objet

## Polymorphisme

- Le polymorphisme est un concept très important dans la **POO**, elle consiste à définir des méthodes portant **même nom** mais ayant des **comportements différents**.
- On distingue entre **trois types de polymorphisme** :
  - **Redéfinition des méthodes** : lorsqu'on redéfinit les mêmes méthodes d'une classe mère dans ses classes filles (cas de l'héritage).
  - **Surcharge des méthodes** : appelé aussi « overloading », c'est définir deux ou plus des méthodes dans une même classe mais avec des paramètres différents
  - **Surcharge des opérateurs** : lorsqu'on redéfinit le code **des méthodes magiques** liées aux opérateurs arithmétiques tels que **`_add_`**, **`_mul_`** ou logiques tels que **`_lt_`** :

# Chapitre 6 : Programmation orientée objet

## Abstraction

- L'abstraction est un concept très important dans la **POO**, elle consiste à simplifier la complexité d'un système. L'abstraction est souvent mise en œuvre à l'aide de **classes abstraites ou d'interfaces**
- Une **classe abstraite** est une classe qui ne peut pas être instantiée directement, mais qui sert de base pour d'autres **classes concrètes**.
- Une classe est dite abstraite si au moins elle contient une méthode abstraite.
- Une méthode abstraite est une méthode qui ne contient aucun traitement. Pourtant on définit juste son prototype.

# Chapitre 6 : Programmation orientée objet

## **Comment créer une classe en python ?**

- Pour définir une classe ayant des attributs et des méthodes en python, on utilise la syntaxe suivante :

```
class NomClasse :  
    def __init__(self, Parametre1, Parametre2, ...):  
        self.Attribut1=Parametre1  
        self.Attribut2=Parametre2  
        ...  
    def NomMethode1(self, ....):  
        # code de la méthode1  
    def NomMethode2(self, ....):  
        # code de la méthode2
```

### Avec :

- **def \_\_init\_\_(self, Parametre1, Parametre2, ...)** : Il s'agit du constructeur de la classe. Il est appelé lors de la création d'une nouvelle instance de la classe. La méthode **\_\_init\_\_** est utilisée pour initialiser les attributs de l'instance.
- **self** : c'est une convention en Python pour faire référence à l'instance de la classe (objet en cours de création). Il est utilisé à l'intérieur des méthodes pour accéder aux attributs et aux autres méthodes de l'instance.

# Chapitre 6 : Programmation orientée objet

## Comment créer une classe en python ?

### Exemple de création d'une classe

```
class Etudiant:  
    def __init__(self, nom, age, note):  
        self.nom = nom  
        self.age = age  
        self.note = note  
  
    def AfficherInformations(self):  
        print(f"Nom: {self.nom}")  
        print(f"Age: {self.age}")  
        print(f>Note: {self.note}")  
  
    def ModifierNote(self, NouvelleNote):  
        self.note = NouvelleNote
```

### Exemple d'instantiation d'un objet

```
# Création d'une instance de la classe Etudiant  
E1 = Etudiant("Ali", 23, 18)  
  
# Affichage des informations initiales  
print("Informations initiales de l'étudiant : ")  
E1.AfficherInformations()  
  
# Mise à jour de la note de l'étudiant  
E1.ModifierNote(90.0)
```

# Chapitre 6 : Programmation orientée objet

## *Exercice d'application*

### Question 1

- Définir une classe **Stagiaire** contenant les attributs suivants : **num**, **nom**, **age** et une méthode **afficher\_infos()** qui affiche les informations d'un stagiaire donné.
- Créer un stagiaire puis afficher ses informations.

### Question 2

- Ajouter aux attributs de la classe **Stagiaire** les attributs suivants : **note\_1** et **note\_2**.
- Ajouter également une méthode **calculer\_moyenne()** qui permet de calculer puis afficher la moyenne générale des notes obtenues pour un stagiaire donné.
- Créer deux stagiaires puis afficher le nom de celui qui a obtenu la moyenne générale maximale.

# Chapitre 6 : Programmation orientée objet

## Solution

### Question 1

```
class Stagiaire :
    def __init__(self, num, nom, age):
        self.num=num
        self.nom=nom
        self.age=age

    def afficher_infos(self):
        print(f"Numéro de stagiaire : {self.num}")
        print(f"Nom de stagiaire : {self.nom}")
        print(f"Age de stagiaire : {self.age}")

s1=Stagiaire("A1", "Adam", 22)
print("Les informations du premier stagiaire : ")
s1.afficher_infos()
```

# Chapitre 6 : Programmation orientée objet

## Solution

### Question 2

```
class Stagiaire :
    def __init__(self, num, nom, age, note_1, note_2):
        self.num=num
        self.nom=nom
        self.age=age
        self.note_1=note_1
        self.note_2=note_2

    def afficher_infos(self):
        print(f"Numéro de stagiaire : {self.num}")
        print(f"Nom de stagiaire : {self.nom}")
        print(f"Age de stagiaire : {self.age}")
        print(f"Note 1 de stagiaire : {self.note_1}")
        print(f"Note 2 de stagiaire : {self.note_2}")

    def calculer_moyenne(self):
        M=(self.note_1+self.note_2)/2
        return (M)
```

# Chapitre 6 : Programmation orientée objet

## Solution

### Question 2 (suite)

```
s1=Stagiaire("A1", "Adam", 22, 15, 12)
print("Les informations du premier stagiaire : ")
s1.afficher_infos()
s2=Stagiaire("B1", "Sara", 24, 13, 11)
print("Les informations du deuxième stagiaire : ")
s2.afficher_infos()

if s1.calculer_moyenne()>s2.calculer_moyenne() :
    print("le stagiaire ayant la moyenne maximale s'appelle ", s1.nom)
    print("Moyenne générale obtenue ", s1.calculer_moyenne())
elif s1.calculer_moyenne()<s2.calculer_moyenne() :
    print("le stagiaire ayant la moyenne maximale s'appelle ", s2.nom)
    print("Moyenne générale obtenue ", s2.calculer_moyenne())
else :
    print("les deux stagiaires ont obtenu la même moyenne : ", s2.calculer_moyenne())
```

# Chapitre 6 : Programmation orientée objet

## **Comment implémenter l'encapsulation en python ?**

- Tous les attributs et les méthodes d'une classe sont **publiques** (accessibles en dehors de leur classe)
- Pour rendre un attribut ou une méthode **privée (private)**, il suffit de précéder son nom par \_\_ :
   
**\_\_NomAttribut ou \_\_ NomMethode()**
- Pour rendre un attribut ou une méthode **protégée (protected)**, il suffit de précéder son nom par \_ :
   
**\_NomAttribut ou \_NomMethode()**
- Pour accéder aux attributs privés en dehors de leur classe, on peut utiliser deux méthodes : **get** et **set**.
  - **getNomAttribut()** : méthode qui renvoie l'attribut **privé** ou **protégé** en dehors de sa classe.
  - **setNomAttribut()** : méthode qui permet de modifier la valeur de l'attribut **privé** ou **protégé** en dehors de la classe où il a été créé.

### **Définition des attributs privés et protégés**

```
def __init__(self, p1, p2, ...):
    self.__NomAttribut1= p1 # attribut privé
    self._NomAttribut2= p2 # attribut protégé
    ....
```

### **Définition des méthodes get et set**

```
def getNomAttribut(self):
    return (self.__NomAttribut)
def setNomAttribut(self, NouvelleValeur):
    self.__NomAttribut = NouvelleValeur
```

# Chapitre 6 : Programmation orientée objet

## *Exercice d'application*

En se basant sur l'exercice d'application précédent

### Question 3

- Modifier la visibilité de tous les attributs de la classe **Stagiaire** en choisissant de les rendre **privés**.
- Utiliser les méthodes **get** pour récupérer les valeurs de tous les attributs de la classe **Stagiaire**
- Utiliser les méthodes **set** pour pouvoir faire des modifications par rapport aux valeurs des attributs **note\_1** et **note\_2** d'un stagiaire donné.
- Créer deux stagiaires puis afficher le nom de celui qui a obtenu la moyenne générale maximale.

# Chapitre 6 : Programmation orientée objet

## Solution

```
class Stagiaire :  
    def __init__(self, num, nom, age, note_1, note_2):  
        self.__num=num  
        self.__nom=nom  
        self.__age=age  
        self.__note_1=note_1  
        self.__note_2=note_2  
  
    def getnum(self):  
        return(self.__num)  
  
    def getnom(self):  
        return (self.__nom)  
  
    def getage(self):  
        return(self.__age)  
  
    def getnote_1(self):  
        return (self.__note_1)  
  
    def getnote_2(self):  
        return (self.__note_2)  
  
    def setnote_1(self, note_1):  
        self.__note_1=note_1  
  
    def setnote_2(self, note_2):  
        self.__note_2=note_2  
  
    def calculer_moyenne(self):  
        return (self.__note_1+self.__note_2)/2
```

# Chapitre 6 : Programmation orientée objet

## Solution (suite)

```
# creer un nouveau stagiaire
s1=Stagiaire("A1", "Adam", 22, 15, 12)
# Afficher les informations du 1er stagiaire
print("Les informations du premier stagiaire : ")
print("Num : ", s1.getnum())
print("Nom : ", s1.getnom())
print("Age : ", s1.getage())
print("Note 1 : ", s1.getnote_1())
print("Note 2 : ", s1.getnote_2())
print("Moyenne générale : ", s1.calculer_moyenne())

# creer un deuxième stagiaire
s2=Stagiaire("B1", "Sara", 24, 16, 17)

# modifier les notes du deuxième stagiaire
s2.setnote_1(11)
s2.setnote_2(12)
```

# Chapitre 6 : Programmation orientée objet

## Solution (suite)

```
# afficher le nom du stagiaire ayant la moyenne générale maximale
if s1.calculer_moyenne()>s2.calculer_moyenne() :
    print("le stagiaire ayant la moyenne maximale s'appelle ", s1.getnom())
    print("Moyenne générale obtenue ", s1.calculer_moyenne())
elif s1.calculer_moyenne()<s2.calculer_moyenne() :
    print("le stagiaire ayant la moyenne maximale s'appelle ", s2.getnom())
    print("Moyenne générale obtenue ", s2.calculer_moyenne())
else :
    print("les deux stagiaires ont obtenu la même moyenne : ", s2.calculer_moyenne())
```

# Chapitre 6 : Programmation orientée objet

## **Comment implémenter l'héritage en python ?**

### Syntaxe de définition d'un héritage simple

```
class SuperClasse :
    # attributs et méthodes

class SousClasse (SuperClasse) :
    # attributs et méthodes
```

### Syntaxe de définition d'un héritage multiple

```
class SuperClasse1 :
    # attributs et méthodes

class SuperClasse2 :
    # attributs et méthodes

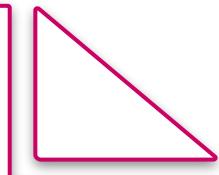
class SousClasse (SuperClasse1, SuperClasse2) :
    # attributs et méthodes
```

Dans le constructeur de la classe fille `__init__()`, il faut appeler le constructeur de la classe mère avec l'une des méthodes suivantes :

`super().__init__(p1, p2, ..)`  
`SuperClasse.__init__(self, p1, p2, ..)`

#### Avec

p1, p2, ... sont les paramètres de la classe mère



# Chapitre 6 : Programmation orientée objet

## Exemple d'heritage simple

```
class Personne:
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def afficher_infos(self):
        print(f"Nom: {self.nom}, Age: {self.age}")

class Etudiant(Personne):
    def __init__(self, nom, age, niveau):
        super().__init__(nom, age)
        self.niveau = niveau

    def afficher_infos(self):
        super().afficher_infos()
        print(f"Niveau: {self.niveau}")
```

Définition sous classe Etudiant

Définition super classe Personne

Définition sous classe Employé

```
class Employe(Personne):
    def __init__(self, nom, age, poste):
        super().__init__(nom, age)
        self.poste = poste

    def afficher_infos(self):
        super().afficher_infos()
        print(f"Poste: {self.poste}")
```

# Test de l'héritage  
etudiant = Etudiant("Ali", 20, "Première année")  
employe = Employe("Adam", 30, "Ingénieur")

# Appel des méthodes spécifiques à chaque classe  
etudiant.afficher\_infos()  
employe.afficher\_infos()

# Chapitre 6 : Programmation orientée objet

## *Exemple d'héritage multiple*

```
import datetime

class Personne:
    def __init__(self, n, p, d):
        self.nom=n
        self.prenom=p
        self.dat_nai=d

    def afficher(self):
        print("Nom : ", self.nom)
        print("Prénom : ", self.prenom)
        print("Date de naissance : ", self.dat_nai)

class Employe :
    def __init__(self, c, s):
        self.__code=c
        self.__salaire=s

    def afficher (self) :
        print("Code : ", self.__code)
        print("Salaire : ", self.__salaire)
```

The diagram illustrates multiple inheritance. It shows two base classes, `Personne` and `Employe`, each with its own set of methods and attributes. A single derived class, represented by a large rectangle containing both code snippets, inherits from both. Two blue arrows point from the right side of the `Personne` class definition to the right edge of the derived class rectangle, labeled Classe mère 1. Another two blue arrows point from the right side of the `Employe` class definition to the right edge of the derived class rectangle, labeled Classe mère 2.

# Chapitre 6 : Programmation orientée objet

## Exemple d'héritage multiple

```
class Professeur (Personne, Employe):  
    def __init__(self, n,p,d,c,s,sp):  
        Personne.__init__(self, n,p,d)  
        Employe.__init__(self,c,s)  
        self.__specialite=sp  
  
    def afficher(self):  
        Personne.afficher(self)  
        Employe.afficher(self)  
        print("Spécialité : ", self.__specialite)  
  
p1=Professeur("Alaoui", "Sanae", datetime.date(1988, 12,10), "A1", 12000, "Info")  
print("Affichage des informations du professeur :")  
p1.afficher()
```

Classe fille

Redéfinition de la fonction afficher()

# Chapitre 6 : Programmation orientée objet

## **Comment implémenter le polymorphisme en python ?**

### Surcharge des méthodes

```
class Employe:  
    def __init__(self, s):  
        self.salaire=s  
  
    def CalculSalaire(self, comission) :  
        print("Salaire total : ", self.salaire+comission)  
  
    def CalculSalaire(self) :  
        print("Salaire total : ", self.salaire)  
  
E1=Employe(12000)  
E1.CalculSalaire() # out : 12000  
E1.CalculSalaire(2500)  
  
# TypeError: Employe.CalculSalaire() takes 1 positional argument but 2 were given
```



Erreur parce que la méthode qui sera utilisée est **la dernière** (avec deux paramètres)

# Chapitre 6 : Programmation orientée objet

## ***Comment implémenter le polymorphisme en python ?***

### **Surcharge des méthodes**

```
class Employe:  
    def __init__(self, s):  
        self.salaire=s  
    def CalculSalaire(self,comission=None) :  
        if comission==None :  
            print("Salaire total : ", self.salaire)  
        else :  
            print("Salaire total : ", self.salaire+comission)  
  
E1=Employe(12000)  
E1.CalculSalaire() # out: Salaire total : 12000  
E1.CalculSalaire(2500) # out : Salaire total : 14500
```

# Chapitre 6 : Programmation orientée objet

**Comment implémenter le polymorphisme en python ?**

## Surcharge des opérateurs

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

    def __add__(self,y):
        return(self.x+y)

    def __mul__(self,y):
        return(self.x*y)

    def __sub__(self, y):
        return(self.x-y)
```

```
N1=entier(3)
print(N1+2) # out : 5
print(N1*3) # out : 9
print(N1-5) # out : -2
```

Code de la méthode magique : \_\_add\_\_

Code de la méthode magique : \_\_mul\_\_

Code de la méthode magique : \_\_sub\_\_

# Chapitre 6 : Programmation orientée objet

## Comment implémenter le polymorphisme en python ?

### Surcharge des opérateurs

```
class Employe:  
    def __init__(self, s):  
        self.salaire=s  
  
    # surcharge de l'opérateur +  
    def __add__(self,e):  
        return (self.salaire+e.salaire)  
  
    # surcharge de l'opérateur <  
    def __lt__(self, e):  
        if self.salaire<e.salaire:  
            return True  
        else :  
            return False
```

```
E1=Employe(15000)  
E2=Employe(13000)  
  
print(E1+E2)  
  
if E1<E2 :  
    print("E1 a le salaire le plus petit ")  
else :  
    print("E2 a le salaire le plus petit ")
```

# Chapitre 6 : Programmation orientée objet

## Comment implémenter l'abstraction en python ?

### Syntaxe de définition d'une classe abstraite

```
from abc import ABC, abstractmethod
class MaClasseAbstraite(ABC):
    @abstractmethod
    def methode_abstraite(self):
        pass
    @abstractmethod
    def autre_methode_abstraite(self):
        pass
```

Classe mère abstraite  
(au moins une méthode abstraite)

Classe fille

### Syntaxe d'utilisation d'une classe abstraite

La redéfinition des méthodes abstraites est obligatoire dans les classes filles

```
class MaClasseConcrete(MaClasseAbstraite):
    def methode_abstraite(self):
        print("Implémentation de methode_abstraite")

    def autre_methode_abstraite(self):
        print("Implémentation de autre_methode_abstraite")

# Instanciation de la classe concrète
objet_concret = MaClasseConcrete()

# Appel des méthodes abstraites
objet_concret.methode_abstraite()
objet_concret.autre_methode_abstraite()
```

# Chapitre 6 : Programmation orientée objet

## *Exemple d'abstraction*

```
from abc import ABC, abstractmethod

# classe abstraite Vehicule
class Vehicule(ABC):
    def __init__(self, marque, modele):
        self.marque = marque
        self.modele = modele

    @abstractmethod
    def deplacer(self):
        pass

# classe concrète Voiture
class Voiture(Vehicule):
    def __init__(self, marque, modele, couleur):
        super().__init__(marque, modele)
        self.couleur = couleur

    def deplacer(self):
        return f"La voiture {self.marque} {self.modele} de couleur {self.couleur} roule."
```

# Chapitre 6 : Programmation orientée objet

## *Exemple d'abstraction (suite)*

```
# classe concrète Avion
class Avion(Vehicule):
    def __init__(self, marque, modèle, compagnie):
        super().__init__(marque, modèle)
        self.compagnie = compagnie

    def déplacer(self):
        return f"L'avion {self.marque} {self.modèle} de la compagnie {self.compagnie} décolle."

# Exemple d'utilisation
V1 = Voiture("Kia", "2020", "Rouge")
A1 = Avion("Boeing", "747", "RAM")

print(V1.déplacer())
print(A1.déplacer())
```

# Chapitre 6 : Programmation orientée objet

## ***Attributs et méthodes des classes***



C'est quoi la différence entre les attributs et les méthodes des objets et les attributs et les méthodes des classes ?



# Chapitre 6 : Programmation orientée objet

## Exercice 1

Ecrire un script python qui permet de calculer puis afficher la surface des formes géométriques suivantes : **disque**, **rectangle** et **Triangle** (*les dimensions de ces formes sont choisies par l'utilisateur*)

Le script à rédiger devrait contenir une classe mère nommée « **Forme** » et trois sous classes filles : « **Rectangle** », « **Triangle** » et « **Disque** ».

Chaque classe devrait contenir les **attributs** et les **méthodes** nécessaires.

**NB :** L'utilisateur n'a pas le droit de créer d'autres formes autre que **Disque**, **Rectangle** et **Triangle**

## Exercice 2

L'objectif de cet exercice est de rédiger un script python qui implémente les concepts clés de la programmation orienté objet. L'exercice consiste à appliquer un certain nombre de traitement sur des livres.

Chaque livre est caractérisé par les attributs **privés** suivants : son **ISBN**, son **titre**, le **nom de son auteur** (ou les noms des ses auteurs), **date de sa publication**, son **prix** et le non de **sa maison d'édition**.

1. Développer la classe **Livre** qui devrait contenir en plus de ses attributs, les méthodes suivantes :

- **ComparerLivres(..)** qui permet de comparer deux livres en fonction de leurs **ISBN**
- **AfficherInfos(...)** qui permet d'afficher les informations d'un livre.
- **AppliquerPromotion(...)** qui permet de réduire le prix d'un livre selon un taux de réduction.

2. Enregistrer les informations de 10 livres. Puis afficher les livres d'un auteur donné.

# Chapitre 6 : Programmation orientée objet

## Exercices : Série N6

# Chapitre 7 : Les tableaux

# Chapitre 7 : les tableaux

## Définition d'un tableau

- C'est une séquence d'éléments **ordonnés, modifiables** et de **même ou de différents types (int, float ou str)**
- Un tableau peut avoir **une seule dimension, deux dimensions** ou **N dimensions**.



Tableau à une  
seule dimension  
**1D**

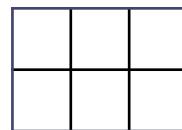


Tableau à deux  
dimensions  
**2D**

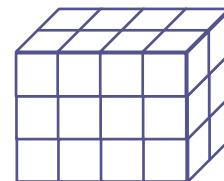


Tableau à trois  
dimensions  
**3D**

.....



Tableau à N  
dimensions  
**nD**

# Chapitre 7 : les tableaux

## ***Création d'un tableau***

- Un tableau en python se définit comme un type particulier appelé **ndarray**
- Le type **ndarray** s'exécute plus rapidement par rapport au type **list** et il occupe moins de mémoire (*meilleur pour les calculs scientifiques*)
- La bibliothèque **numpy** permet d'effectuer des calculs numériques avec Python. Elle introduit une gestion facilitée des tableaux de nombres.
- Plusieurs méthodes du module **numpy** sont disponibles pour la création des tableaux de différentes dimensions.
- **Parmi ces méthodes :**
  - array()
  - arange(), linspace()
  - randn(), randint()
  - ones(), zeros(), full(),
  - Etc.

# Chapitre 7 : les tableaux

## ***La fonction array()***

- La méthode **array(...)** du module **numpy** permet de convertir une liste vers un tableau, elle prend un seul paramètre :
  - Une liste pour un tableau d'une seule dimension
  - Une liste de listes pour un tableau de deux dimensions

## Exemple

```
from numpy import *
T1 = array ([1, 2, 3, 4, 5]) # permet de définir un tableau d'une seule dimension
print (T1) # permet d'afficher : [1 2 3 4 5]
print (type(T1)) # permet d'afficher <class 'numpy.ndarray'>
T2= array ([[1, 2] , [ 3, 4]]) # permet de définir un tableau de deux dimensions
print (T2)
« """ permet d'afficher :
[1 2],
[ 3 4]
"""»
```

# Chapitre 7 : les tableaux

## ***La fonction arange()***

- La fonction **arange(..)** du module **numpy** permet de créer un tableau à une seule dimension.
- La méthode **arange(début, fin, pas)** prend trois paramètres :
  - **début** : première valeur du tableau
  - **fin** : dernière valeur du tableau (avec dernière valeur non incluse)
  - **pas** : valeur de pas pour changer les valeurs du tableau

## Exemple

```
from numpy import *
T1 = arange (6) # permet de définir un tableau d'une seule dimension composé des valeurs 0, 1, 2, 3, 4 et 5
print (T1) # permet d'afficher : [0 1 2 3 4 5]
T2= arange (2, 10, 2)
print (T2) # permet d'afficher : [2 4 6 8]
print (type(T2)) # permet d'afficher : <class 'numpy.ndarray'>
```

# Chapitre 7 : les tableaux

## ***La fonction linspace()***

- La fonction **linspace(...)** du module **numpy** sert à créer un tableau (*une seule dimension*) à partir d'une liste des valeurs aléatoires.
- La fonction **linspace(début, fin, n)** prend trois paramètres :
  - début : première valeur du tableau
  - fin : dernière valeur du tableau (avec dernière valeur incluse)
  - n : nombre des valeurs aléatoires à créer entre début et fin

### Exemple

```
from numpy import *
T1 = linspace (1,6, 4) # permet de définir un tableau d'une seule dimension
print (T1) # permet d'afficher : [1.      2.66666667    4.33333333    6.     ]
```

# Chapitre 7 : les tableaux

## ***La fonction randn()***

- La fonction **randn(...)** du module **numpy.random** sert à créer un tableau rempli par des valeurs aléatoires.
- La fonction **random.randn(n\_lignes, n\_colonnes)** prend deux paramètres :
  - **n\_lignes** : nombre de lignes du tableau à créer
  - **n\_colonnes** : nombre de colonnes du tableau à créer

## **Exemple**

```
from numpy import *
T1 = random.randn (2, 3) # permet de définir un tableau de 2 linges et 3 colonnes
print (T1)
"""permet d'afficher :
[[ 1.09666015 -1.46504893  1.73101646]
 [-0.6840089  0.2359427  0.36864442]]"""

```

# Chapitre 7 : les tableaux

## ***La fonction randint()***

- La fonction **`randint(...)`** du module **`numpy.random`** sert à créer un tableau rempli par des valeurs aléatoires de type **int** et comprises entre **deux valeurs**.
- La fonction **`random.randint(valeur_initiale, valeur_finale, shape)`** prend trois paramètres :
  - **valeur\_initiale** : la valeur minimale des éléments du tableau
  - **valeur\_finale** : la valeur maximale des éléments du tableau (*valeur\_finale non incluse*)
  - **shape** : Le nombre de lignes et de colonnes du tableau à créer

## Exemple

```
from numpy import *
T1 = random.randint (1, 10, (2,6)) # permet de définir un tableau de 2 linges et 6 colonnes
print (T1)
"""permet d'afficher :
[[5 8 4 6 2 2]
 [9 9 1 8 9 9]]""""
```

# Chapitre 7 : les tableaux

## Autres fonctions pour créer un tableau

- La fonction **ones(shape)** permet de créer un tableau rempli par la valeur **1**.
- La fonction **zeros(shape)** permet de créer un tableau rempli par la valeur **0**.
- La fonction **full(shape, V)** permet de créer un tableau rempli par une valeur **V**.

Avec **shape** : (*nombre\_lignes, nombre\_colonnes*)

## Exemple

```
from numpy import *
T1 = ones ((2, 3)) # permet de définir un tableau de 2 linges et 3 colonnes rempli par la valeur 1
T2 = zéros ((3, 5)) # permet de définir un tableau de 3 linges et 5 colonnes rempli par la valeur 0
T3 = full ((2, 3), 12) # permet de définir un tableau de 2 linges et 3 colonnes rempli par la valeur 12
```

# Chapitre 7 : les tableaux

## ***Les méthodes : ndim(), shape() et size()***

- La fonction **ndim(...)** permet d'afficher la dimension d'un tableau.
- La fonction **shape(...)** sert à afficher le nombre de lignes et de colonnes d'un tableau.
- La fonction **size(...)** sert à afficher le nombre d'éléments d'un tableau.

### Exemple

```
from numpy import *
T1 = array ([1, 2, 3, 4, 5])
T2= array ([[1, 2, 3] , [ 4, 5, 6]])
print(ndim(T1))# permet d'afficher 1
print(ndim(T2))# permet d'afficher 2
print(size(T2))# permet d'afficher 6
print (shape(T2)) # permet d'afficher : (2, 3)
print (T2.shape[0]) # permet d'afficher 2
print (T2.shape[1]) # permet d'afficher 3
```

1	2	3
4	5	6

T2

ndim(T2) = 2  
shape(T2) = (2,3)  
size(T2) = 6

1	2	3	4	5
---	---	---	---	---

T1

ndim(T1) = 1  
shape(T1) = (1,)  
size(T1) = 5

# Chapitre 7 : les tableaux

## Exercice d'application 1

- Ecrire un script qui permet de créer puis afficher les éléments d'un tableau rempli par des valeurs aléatoires de type **int**.
  - Les valeurs maximale et minimale du tableau ainsi que le nombre de lignes et de colonnes sont déterminés par l'utilisateur.

## Solution

```
from numpy import *
def afficher_tableau(Tab) :
    for i in range(Tab.shape[0]) :
        for j in range(Tab.shape[1]) :
            print(Tab[i,j], end = " ")
    print("")
```

```
v_min=int(input("Entrer la valeur manimale : "))
v_max=int(input("Entrer la valeur maximale : "))
n_lig=int(input("Entrer le nombre de lignes : "))
n_col=int(input("Entrer le nombre de colonnes : "))
T=random.randint(v_min, v_max, (n_lig, n_col))
afficher_tableau(T)
```

# Chapitre 7 : les tableaux

## **Accéder aux éléments d'un tableau**

Pour accéder aux éléments d'un tableau, il est possible d'utiliser :

- **L'indication positif** (0, 1, 2, etc.)
- **L'indication négatif** (-1, -2, -3, etc.)
- La méthode de **slicing** ( extraire une tranche d'éléments d'un tableau)

## Exemple

```
from numpy import *
T1 = array ([1, 2, 3, 4, 5]) # permet de définir un tableau d'une seule dimension
print (T1[1], T1[-2]) # permet d'afficher : 2 4
T2= array ([[1, 2, 3] , [ 4, 5, 6]]) # permet de définir un tableau de deux dimensions
print (T2[1, 1]) # permet d'afficher : 5
print (T2[0][2]) # permet d'afficher : 3
```

# Chapitre 7 : les tableaux

## ***La méthode de slicing pour les tableaux***

- La méthode de **slicing** est une méthode d'extraction d'une trame d'éléments à partir d'un tableau
- La méthode de **slicing** appliquée pour un tableau est différente à celle appliquée aux listes et aux chaînes de caractères.
- La syntaxe générale utilisée est la suivante :

nom\_tableau [début\_l : fin\_l , début\_f : fin\_f, pas ]

### Exemple

```
from numpy import *
T= array ([[1, 2, 3, 4], [ 5, 6, 7, 8], [ 9, 10, 11, 12]])
T1=T[1:2, 1:3]
T2= T[:, 2:]
```

1	2	3	4
5	6	7	8
9	10	11	12

T1

1	2	3	4
5	6	7	8
9	10	11	12

T2

# Chapitre 7 : les tableaux

## *La fonction append()*

- La fonction **append(...)** permet d'ajouter des éléments à la fin d'un tableau en renvoyant une copie de ce tableau.
- La fonction **append(...)** utilise un argument appelé **axis** qui prend soit la valeur 1 ou 0.
  - axis =0 : pour ajouter une ligne
  - axis =1 : pour ajouter une colonne

### Exemple 1

```
from numpy import *
T1 = ones((2,5))
T2=append(T1,[[1,2,3,4,5]], axis=0)
print(T2)
```

T2

1	1	1	1	1
1	1	1	1	1
1	2	3	4	5

### Exemple 2

```
from numpy import *
T1 = ones((3,5))
T3=append(T2,[[1],[2],[3]], axis=1)
print(T3)
```

T3

1	1	1	1	1	1
1	1	1	1	1	2
1	1	1	1	1	3

# Chapitre 7 : les tableaux

## ***La fonction insert()***

- La fonction **insert(...)** permet d'insérer des éléments à une position donnée. La fonction renvoie une copie du tableau.
- La fonction **insert(nom\_tableau, indice\_position, valeur\_insérer, axis)** utilise un argument appelé **axis** qui prend soit la valeur 1 ou 0.
  - **indice\_position** : l'indice de la position
  - **valeur\_insérer** : les valeurs à insérer dans le tableau
  - **axis =0** : pour insérer une ligne
  - **axis =1** : pour insérer une colonne

## Exemple

```
from numpy import *
T= array ([[1, 2, 3, 4], [ 5, 6, 7, 8], [ 9, 10, 11, 12]])
print(T)
T1=insert(T, 2, [1, 1, 1, 1], axis=0)
print(T1)
```

T1

1	2	3	4
5	6	7	8
1	1	1	1
9	10	11	12

# Chapitre 7 : les tableaux

## ***La fonction delete()***

- La fonction **delete(...)** permet de supprimer des éléments à partir d'un tableau (lignes ou colonnes) en renvoyant une copie du tableau.
- La fonction **delete(nom\_tableau, objet, axis)** utilise trois arguments :
  - **objet** : les indices des éléments à supprimer
  - **axis** : indicateur pour préciser l'élément à supprimer.
    - **axis = 0** : pour supprimer une ligne
    - **axis = 1** : pour supprimer une colonne

## Exemple

```
from numpy import *
T= array ([[1, 2, 3, 4], [ 5, 6, 7, 8], [ 9, 10, 11, 12]])
T1=delete(T, [1, 3], axis=1) # permet de supprimer les colonnes ayant les indices 1 et 3
print(T1)
```

1	3
5	7
9	11

**T1**

# Chapitre 7 : les tableaux

## **Appliquer des opérations mathématiques sur les tableaux**

- C'est possible d'appliquer les opérateurs mathématiques suivants sur les tableaux : +, -, \*, \*\*, /, //, %
- **Les dimensions des tableaux doivent être égales.**

### Exemple

```
from numpy import *
T1 = full((2, 4), 3) # permet de définir un tableau de 2 lignes et 4 colonnes rempli par 3
T2= ones((2, 4)) # permet de définir un tableau de 2 lignes et 4 colonnes rempli par 1
T3=T1+T2
print (T3)
"""permet d'afficher
[4. 4. 4. 4.]
[4. 4. 4. 4.]"""

```

# Chapitre 7 : les tableaux

## ***Les fonctions : sum(), prod(), max(), min() et mean()***

- La fonction **sum()** permet de calculer la somme des éléments d'un tableau.
- Les deux fonctions **max()** et **min()** permettent respectivement de chercher **le maximum** et **le minimum** d'un tableau.
- La fonction **mean()** permet de calculer la moyenne des éléments d'un tableau.

### Exemple

```
from numpy import *
T1 = random.randint(1, 10, (4,5))
print (T1.sum())
print (T1.prod())
print (T1.max())
print (T1.min())
print (T1.mean())
```

3	3	4	1	7
7	8	7	7	1
5	7	8	2	6
8	2	5	7	4

T1

T1.sum()=102  
 T1.prod()=5204415283200  
 T1.max()= 8  
 T1.min()=1  
 T1.mean()=5.1

# Chapitre 7 : les tableaux

## *Exercice d'application 2*

- Ecrire un script python qui permet de créer un tableau T, de shape (n, m).
- Le script contient les fonctions suivantes :
  - Une fonction **somme()** qui permet de calculer la somme des éléments de chaque ligne du tableau **T**.
  - Ecrire une fonction **produit()** qui permet de calculer le produit des éléments de chaque colonne du tableau **T**
  - Ecrire une fonction **maximum()** qui permet de chercher et afficher la valeur maximale de chaque ligne du tableau **T**.
  - Ecrire une fonction **minimum()** qui permet de chercher et afficher la valeur minimale de chaque colonne du tableau **T**.

# Chapitre 7 : les tableaux

## **Les fonctions : sum(), max(), min() et mean()**

- C'est possible d'appliquer ces fonctions pour **chaque ligne** ou **chaque colonne** en ajoutant l'argument **axis**.
  - axis = 0 : appliquer les fonctions pour chaque colonne
  - axis=1 : appliquer les fonctions pour chaque ligne

### Exemple

```
from numpy import *
T=array([[1.24, 2.3, 16, 4, 5],[124, 12.3, 18, 2, 6], [24, 12, 19, 7, 3]])
T1=T.sum(axis=0)
print(T1)
T2=T.max(axis=1)
print(T2)
T3=T.min(axis=0)
print(T3)
T4=T.mean(axis=1))
print(T4)
```

					<b>axis = 1</b>
					↑
					↓
1.24	2.3	16	4	5	
124	12.3	18	2	6	
24	12	19	7	3	
T					

T1	149.24	26.6	53	13	14
T2	16	124	24		
T3	1.24	2.3	16	2	3
T4	5.708	32.46	13		

# Chapitre 7 : les tableaux

## Exercice d'application 3

Ecrire un script python qui permet de créer un tableau T, de shape (n, m). Après le remplissage du tableau T, le script calcule et affiche sa variance et son écart type.

- Pour calculer la variance d'un tableau.

$$V = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2.$$

- Pour calculer l'écart type d'un tableau.

$$\sigma = \sqrt{V} = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2}$$

Avec : n : nombre d'éléments du tableau T

xi : l'ensemble des valeurs du tableau T

$\bar{x}$  : la moyenne des valeurs du tableau T

# Chapitre 7 : les tableaux

## ***Les fonctions : std(), var()***

- La méthode **var()** permet de calculer la variance d'un tableau.
- La méthode **std()** permet de calculer l'écart type d'un tableau.
- C'est possible d'appliquer ces fonctions pour **chaque ligne** ou **chaque colonne** en ajoutant l'argument **axis**.
  - **axis = 0** : appliquer les fonctions pour chaque colonne
  - **axis=1** : appliquer les fonctions pour chaque ligne

## Exemple

```
from numpy import *
T= random.randint(1, 20, (4,5))
print (T.var())
print (T.std())
```

2	19	2	7
12	14	13	12
14	7	18	16

T

T.var() = 29.555555555555557

T.std() = 5.436502143433364

# Chapitre 7 : les tableaux

## ***La fonction sort()***

- La méthode **sort(...)** permet de trier les éléments d'un tableau. Elle permet de renvoyer une copie du tableau trié.
- En ajoutant le paramètre axis, ça permet de trier le tableau selon les lignes ou les colonnes (axis=0 : trier les éléments de chaque colonne - axis =1 : trier les éléments de chaque ligne)

### Exemple

```
from numpy import *
T1= random.randint(1, 20, (3,4))
print (T1)
T2=sort(T1, axis=0)
print (T2)
T3=sort(T1, axis=1)
print (T3)
```

T1      

14	8	2	15
5	8	16	5
11	4	4	19

      T2      

5	4	2	5
11	8	4	15
14	8	16	19

T3      

2	8	14	15
5	5	8	15
4	4	11	19

# Chapitre 7 : les tableaux

## *Exercice d'application 4*

Ecrire un script python qui permet de créer un tableau **M** de **shape (n, m)**. Après le remplissage du tableau **M** par **n x m** valeurs aléatoires de type **int**, le script calcule et affiche **son transposé Tr**.

### Exemple

**M**       $\Rightarrow$       **Tr(M)**

3	6	4	9
4	3	8	2
12	9	7	5

3	4	12
6	3	9
4	8	7
9	2	5

# Chapitre 7 : les tableaux

## ***Calculer le transposé d'une matrice***

- La méthode **transpose(...)** permet de calculer le transposé d'un tableau en renvoyant une copie du tableau.

### **Exemple**

```
import numpy as np  
T=np.array([[3, 6, 4, 5],[4, 3, 8, 2], [12, 9, 7, 3]])  
T1=np.transpose(T)  
print(T2)
```

T       T1

3	6	4	5
4	3	8	2
12	9	7	3

3	4	12
6	3	9
4	8	7
5	2	3

# Chapitre 7 : les tableaux

## ***La fonction concatenate()***

- La méthode **concatenate(...)** permet de concatener deux tableaux selon une **colonne** ou une **ligne**
- La méthode **concatenate((tableau\_1, tableau\_2), axis)** prend trois paramètres
  - tableau\_1 : nom du premier tableau
  - tableau\_2 : nom du deuxième tableau
  - axis : paramètre pour déterminer le type de la concaténation à faire :
    - *axis = 0 : concaténation selon la ligne*
    - *axis=1 : concaténation selon la colonne*

# Chapitre 7 : les tableaux

## ***La fonction concatenate()***

### Exemple 1

```
import numpy as np
T1=np.full((3,4),6)
T2=np.full((2,4),2)
T3=np.concatenate((T1,T2), axis=0)
print(T3)
```

Le **nombre de colonnes** des tableaux à concaténer doit être **le même**

6	6	6	6
6	6	6	6
6	6	6	6
=>			
2	2	2	2
2	2	2	2
T3			

T1

T2

### Exemple 2

```
import numpy as np
M1=np.full((4,3),4)
M2=np.full((4,2),3)
M3=np.concatenate((M1,M2), axis=1)
print(M3)
```

Le **nombre de lignes** des tableaux à concaténer doit être **le même**

4	4	4
4	4	4
4	4	4
4	4	4
=>		
3	3	
3	3	
3	3	
3	3	
M3		

M1

M2

# Chapitre 7 : les tableaux

## Exercice d'application 5

Ecrire un script python qui permet de créer deux tableaux **T1** de **shape (n, p)** et **T2** de **shape (p, m)** . Après le remplissage des deux tableaux **T1** et **T2**, le script calcule et affiche le produit matriciel de deux tableaux.

3	x	4	+	6	x	12	+	4	x	24	+	5	x	7	->	215
---	---	---	---	---	---	----	---	---	---	----	---	---	---	---	----	-----

3	6	4	5
4	3	8	2
12	9	7	3

T1

x

4	2
12	18
24	9
7	3

T2

=&gt;

215	165
258	140
345	258

T3

# Chapitre 7 : les tableaux

## Exercice d'application 5

Ecrire un script python qui permet de créer deux tableaux **T1** de **shape (n, p)** et **T2** de **shape (p, m)** . Après le remplissage des deux tableaux **T1** et **T2**, le script calcule et affiche le produit matriciel de deux tableaux.

3	x	2	+	6	x	18	+	4	x	9	+	5	x	3	->	165
---	---	---	---	---	---	----	---	---	---	---	---	---	---	---	----	-----

3	6	4	5
4	3	8	2
12	9	7	3

T1

x

4	2
12	18
24	9
7	3

T2

=&gt;

215	165
258	140
345	258

T3

# Chapitre 7 : les tableaux

## Exercice d'application 5

Ecrire un script python qui permet de créer deux tableaux **T1** de **shape (n, p)** et **T2** de **shape (p, m)** . Après le remplissage des deux tableaux **T1** et **T2**, le script calcule et affiche le produit matriciel de deux tableaux.

4	x	4	+	3	x	12	+	8	x	24	+	2	x	7	->	258
---	---	---	---	---	---	----	---	---	---	----	---	---	---	---	----	-----

3	6	4	5
4	3	8	2
12	9	7	3

T1

x

4	2
12	18
24	9
7	3

T2

=&gt;

215	165
258	140
345	258

T3

# Chapitre 7 : les tableaux

## Exercice d'application 5

Ecrire un script python qui permet de créer deux tableaux **T1** de **shape (n, p)** et **T2** de **shape (p, m)** . Après le remplissage des deux tableaux **T1** et **T2**, le script calcule et affiche le produit matriciel de deux tableaux.

4	x	2	+	3	x	18	+	8	x	9	+	2	x	3	->	140
---	---	---	---	---	---	----	---	---	---	---	---	---	---	---	----	-----

3	6	4	5
4	3	8	2
12	9	7	3

T1

x

4	2
12	18
24	9
7	3

T2

=&gt;

215	165
258	140
345	258

T3

# Chapitre 7 : les tableaux

## Exercice d'application 5

Ecrire un script python qui permet de créer deux tableaux **T1** de **shape (n, p)** et **T2** de **shape (p, m)** . Après le remplissage des deux tableaux **T1** et **T2**, le script calcule et affiche le produit matriciel de deux tableaux.

12	x	4	+	9	x	12	+	7	x	24	+	3	x	7	->	345
----	---	---	---	---	---	----	---	---	---	----	---	---	---	---	----	-----

3	6	4	5
4	3	8	2
12	9	7	3

T1

4	2
12	18
24	9
7	3

T2

=&gt;

215	165
258	140
345	258

T3

# Chapitre 7 : les tableaux

## Exercice d'application 5

Ecrire un script python qui permet de créer deux tableaux **T1** de **shape (n, p)** et **T2** de **shape (p, m)** . Après le remplissage des deux tableaux **T1** et **T2**, le script calcule et affiche le produit matriciel de deux tableaux.

12	x	2	+	9	x	18	+	7	x	9	+	3	x	3	->	258
----	---	---	---	---	---	----	---	---	---	---	---	---	---	---	----	-----

3	6	4	5
4	3	8	2
12	9	7	3

T1

x

4	2
12	18
24	9
7	3

T2

=&gt;

215	165
258	140
345	258

T3

# Chapitre 7 : les tableaux

## ***Calculer le produit matriciel***

- La méthode **dot()** permet de calculer le produit matriciel de deux tableaux.
- Le produit d'une matrice de **shape (n, m)** par une matrice de **shape (m, p)** donne une matrice de **shape (n, p)**

### **Exemple**

```
import numpy as np
T1=np.array([[3, 6, 4, 5],[4, 3, 8, 2], [12, 9, 7, 3]])
T2=np.array([[4, 2],[12,18], [24,9], [7, 3]])
T3=np.dot((T1,T2))
print(T3)
```

$$\begin{array}{|c|c|c|c|} \hline 3 & 6 & 4 & 5 \\ \hline 4 & 3 & 8 & 2 \\ \hline 12 & 9 & 7 & 3 \\ \hline \end{array}
 \quad \times \quad
 \begin{array}{|c|c|} \hline 4 & 2 \\ \hline 12 & 18 \\ \hline 24 & 9 \\ \hline 7 & 3 \\ \hline \end{array}
 \quad => \quad
 \begin{array}{|c|c|} \hline 215 & 165 \\ \hline 258 & 140 \\ \hline 345 & 258 \\ \hline \end{array}$$

T1                            T2                            T3

# Chapitre 7 : les tableaux

## **Exercice d'application 6**

Ecrire un script python qui permet de créer un tableau **T** de **shape (n, n)**. Après le remplissage du tableau **T** par **n x n** valeurs aléatoires de type **int**, le script calcule et affiche **son déterminant D**.

### Exemple

$$\begin{array}{c} T \\ \hline \end{array}
 \begin{array}{|c|c|c|} \hline 3 & 6 & 4 \\ \hline 4 & 3 & 8 \\ \hline 12 & 9 & 7 \\ \hline \end{array}
 \quad => \quad
 3 \times \begin{array}{|c|c|} \hline 3 & 8 \\ \hline 9 & 7 \\ \hline \end{array}
 - 6 \times \begin{array}{|c|c|} \hline 4 & 8 \\ \hline 12 & 7 \\ \hline \end{array}
 + 4 \times \begin{array}{|c|c|} \hline 4 & 3 \\ \hline 12 & 9 \\ \hline \end{array}$$

$$\begin{aligned}
 D &= 3 \times (3 \times 7 - 8 \times 9) - 6 \times (4 \times 7 - 12 \times 8) - 4 \times (4 \times 9 - 12 \times 3) \\
 &= 254.99999999999991
 \end{aligned}$$

# Chapitre 7 : les tableaux

## ***Calculer le déterminant d'une matrice***

- La méthode **det(...)** du module **numpy.linalg** permet de calculer le déterminant d'un tableau.

### Exemple

```
import numpy as np  
T=np.array([[3, 6, 4],[4, 3, 8], [12, 9, 7]])  
D=np.linalg.det(T)  
print(D)
```

T      
$$\begin{array}{|c|c|c|} \hline 3 & 6 & 4 \\ \hline 4 & 3 & 8 \\ \hline 12 & 9 & 7 \\ \hline \end{array}$$
      =>       $\text{Det } (T) = 254.99999999999991$

# Chapitre 7 : les tableaux

## Exercices : Série N7