École Nationale de Commerce et de Gestion de Tanger

Module : Algorithmes & Programmation python

Semestre 6

Pr. CHERRAT Loubna
l.cherrat@uae.ac.ma
Année universitaire: 2024/2025

Objectif du module

COURS+TD+TP

- Comprendre les concepts fondamentaux de l'algorithmique et de la programmation en Python.
- Apprendre à résoudre des problèmes en utilisant des algorithmes.
- Acquérir une maîtrise pratique de la programmation en Python pour la manipulation de données et la réalisation de projets informatiques.
- Comprendre les principes des structures de données et leur application dans la résolution de problèmes informatiques.

Plan 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 : Représentation graphique - ***TD6***
- Chapitre 7 : Programmation orienté objet (POO) – ***TD7***

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** est **une séquence d'actions** (**instructions**) permettant d'arriver**, en un temps fini** , à **un résultat déterminé** à partir d'une **situation donnée**.

❑ ***La conception d'un algorithme est caractérisée par 4 étapes :***

- *Comprendre la nature du problème posé.*
- *Préciser les données fournies (L'entrée des données).*
- *Préciser les résultats que l'on désire obtenir (La sortie des résultats).*
- *Déterminer le processus de transformation des données en résultats (Le traitement des données).*

Quelles sont les données dont on dispose?

Quelles sont les résultats que l'on doit obtenir?

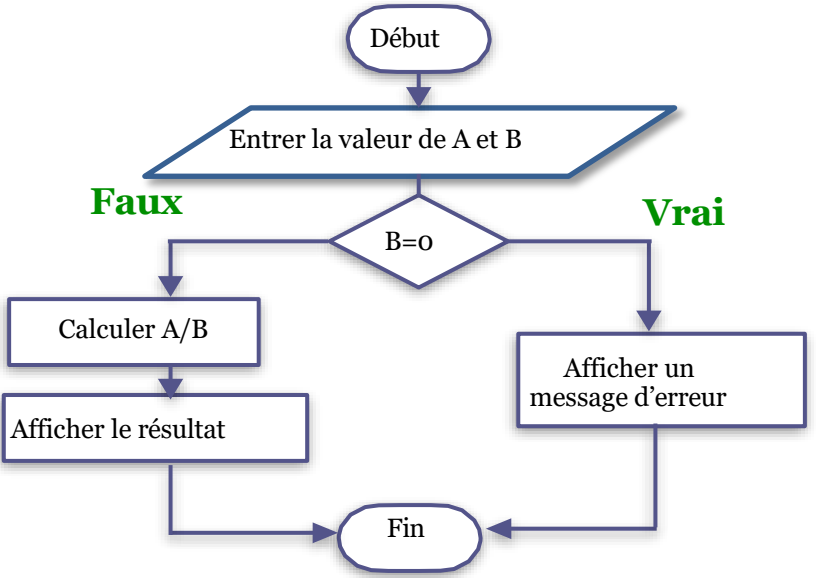
Comment obtenir ces résultats?

Généralités

Algorithme

Un algorithme peut être représenté sous forme d'un **organigramme** ou d'un **pseudocode.**

➢ **Exemple d'un organigramme**



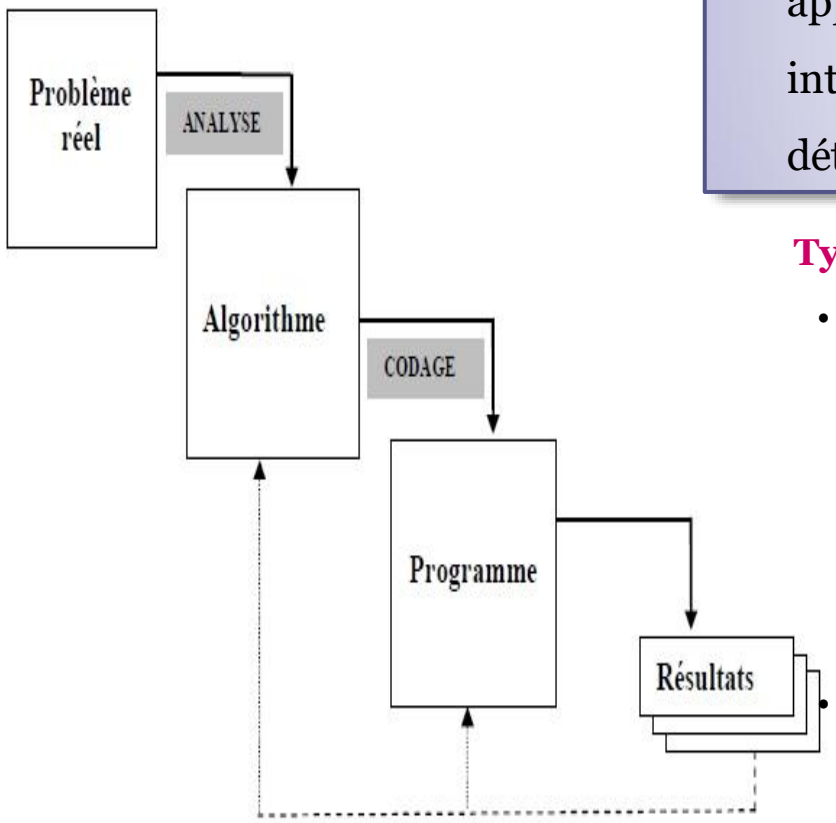
Nom	Symbole	Définition
Flèches		Elles indiquent le sens du traitement.
Début/Fin		Ce symbole indique le début ou la fin de l'organigramme
Entrés/Sorties		Ce symbole indique les données d'entrées et de sorties.
Boite de traitement		Elle indique un traitement spécifique qui peut être exécuté
Boite de décision (test)		Elle permet d'envoyer le traitement sur un chemin ou sur un autre, selon le résultat du test.

➢ **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



- Un **programme** est une suite d'instructions écrites dans un code bien précis appelé **langage de programmation** pouvant être compilés ou interprétés et exécutés par une machine 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.)*

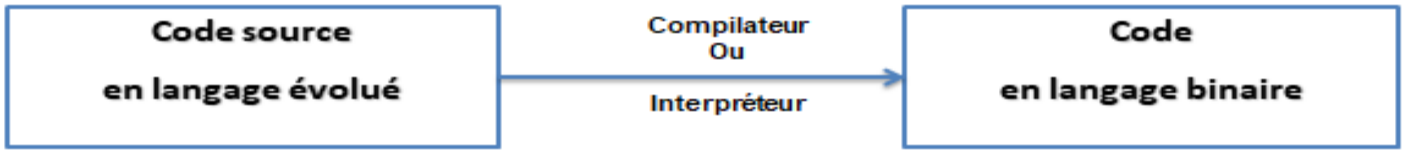
Généralités

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 langages 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 code 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

❑**Compilateur:** traduire le programme entier une fois pour toutes.

Ex : Langage C

Langage binaire 0 et 1

- Positif : plus rapide à l'exécution
- Négatif : il faut recompiler à chaque modification

Généralités

Interprétation

C'est l'analyse et l'exécution de chaque ligne du code source en quelques instructions du langage machine, qui sont ensuite directement exécutées au fur et à mesure sans la génération du code objet.

❑ **Interpréteur** : traduire au fur et à mesure les instructions du programme à chaque exécution.

Ex : Langage python

Positif : exécution instantanée appréciable pour les débutants.

Négatif : exécution lente par rapport à la compilation.

Généralités

Différence entre Interpréteur et Compilateur

Interpréteur	Compilateur
• Convertit le programme en prenant une seule ligne à la fois.	• Analyse l'ensemble du programme et le traduit dans son ensemble en code machine.
• L'analyse du code source prend moins de temps, mais le temps d'exécution global est plus lent.	• L'analyse du code source prend beaucoup de temps, mais le temps d'exécution global est comparativement plus rapide.
• Aucun code d'objet intermédiaire n'est généré, la mémoire est donc efficace.	• Génère du code d'objet intermédiaire qui nécessite donc davantage de mémoire.
• Continue de traduire le programme jusqu'à ce que la première erreur soit rencontrée. Par conséquent, le débogage est facile.	• Il génère le message d'erreur uniquement après avoir analysé l'ensemble du programme. Par conséquent, le débogage est relativement difficile.

Généralités

Type de programmation

Généralement, on distingue entre deux types de programmation :

Programmation procédurale et **la 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 :

Fortran, Cobol, Pascal, C, …

Généralités

Type de programmation

Généralement, on distingue entre deux types de programmation :

Programmation procédurale et **la 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++, Python, Ruby, Javascript, etc.

Généralités

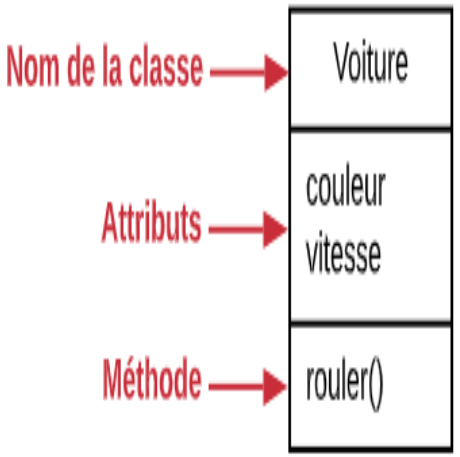
➢ Programmation orienté objet

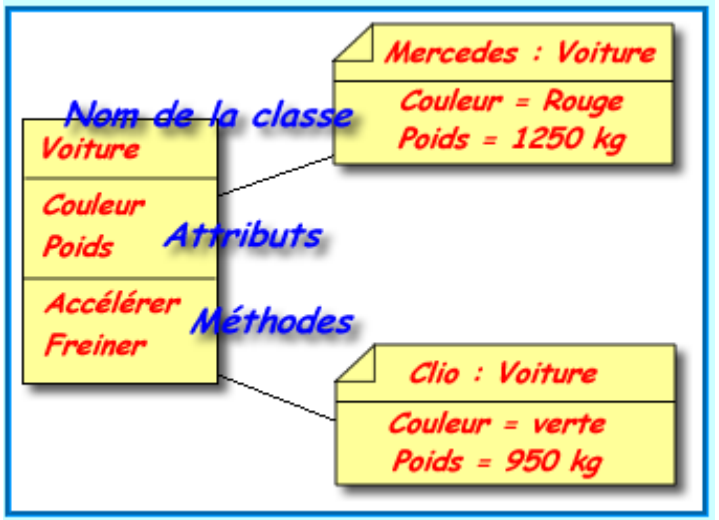
Classe

- Un modèle pour créer des objets. Elle définit les propriétés (**attributs**) et les comportements (**méthodes**) communs à tous les objets créés à partir de cette classe.

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.





Généralités

➢ Programmation orienté objet

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
```

Création d'un objet en python

```
# Création d'une instance de la classe Etudiant
etudiant1 = Etudiant("Ahmed", 20, 15)

# 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(18.0)
```

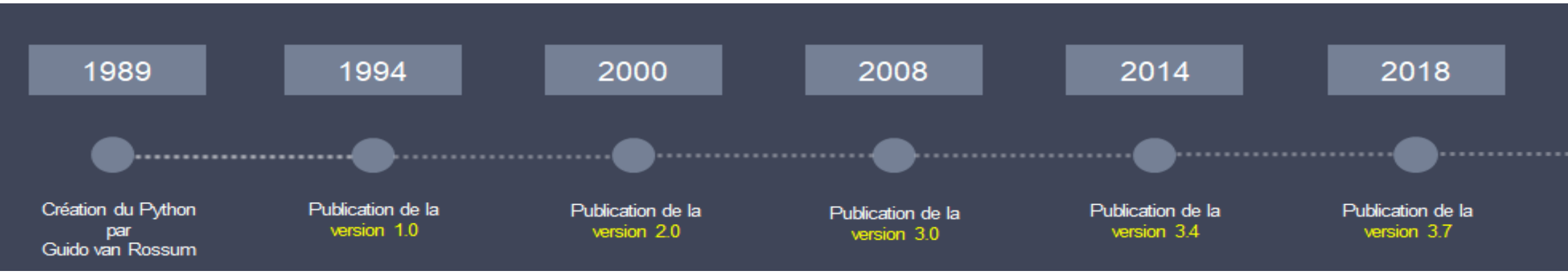
Généralités

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
- **2001 :** naissance de la PSF (**Python Software Fundation**)

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

- **2006 :** première sortie de **IPython**
- **Fin 2008 :** sorties 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

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
- De nombreux modules sont disponibles à partir de bibliothèques optimisées (souvent écrites en C ou C++)

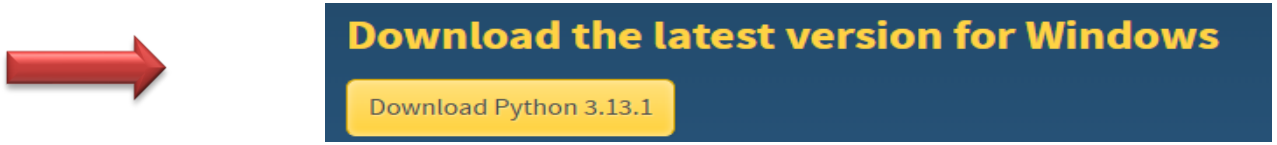
Orienté objet

- Modèle objet puissant mais pas obligatoire
- Structuration multifichier aisée des applications

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 : Interactive Development Environment** tels que :

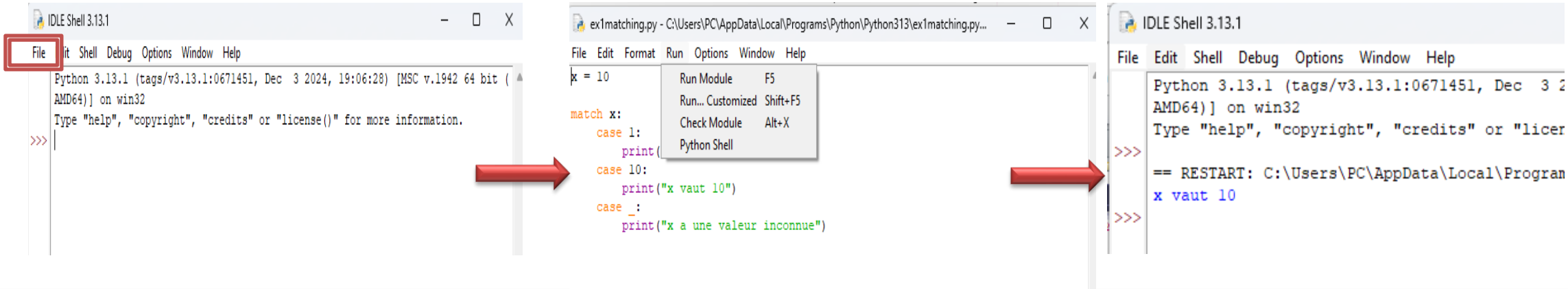
IDLE **Pyzo** **Pycharm**

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

Généralités

Processus d'exécution d'un programme en python dans IDLE

1. **Ouvrir IDLE.**
2. **Créer un nouveau fichier** avec **File > New File**.
3. **Écrire le code Python** dans la fenêtre qui apparaît.
4. **Sauvegarder le fichier** avec l'extension **.py.**
5. **Exécuter le script** en allant dans **Run > Run Module** ou en appuyant sur **F5**.
6. **Voir les résultats** dans la fenêtre **Shell**.

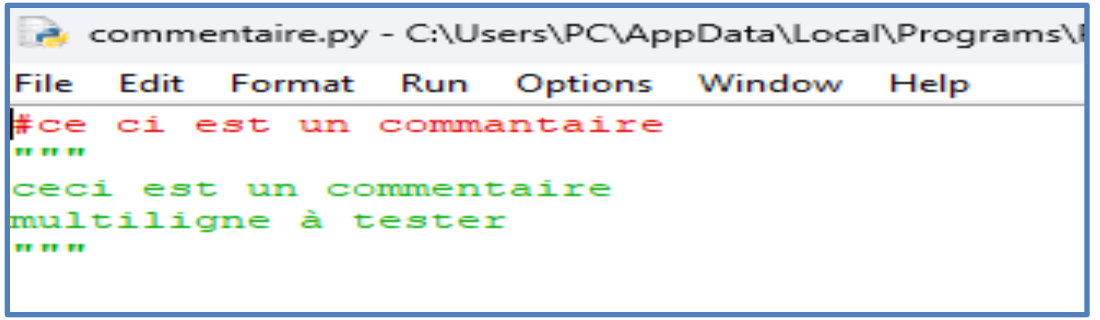


Chapitre 1 : Bases de la programmation Python

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 alternatives
  - 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

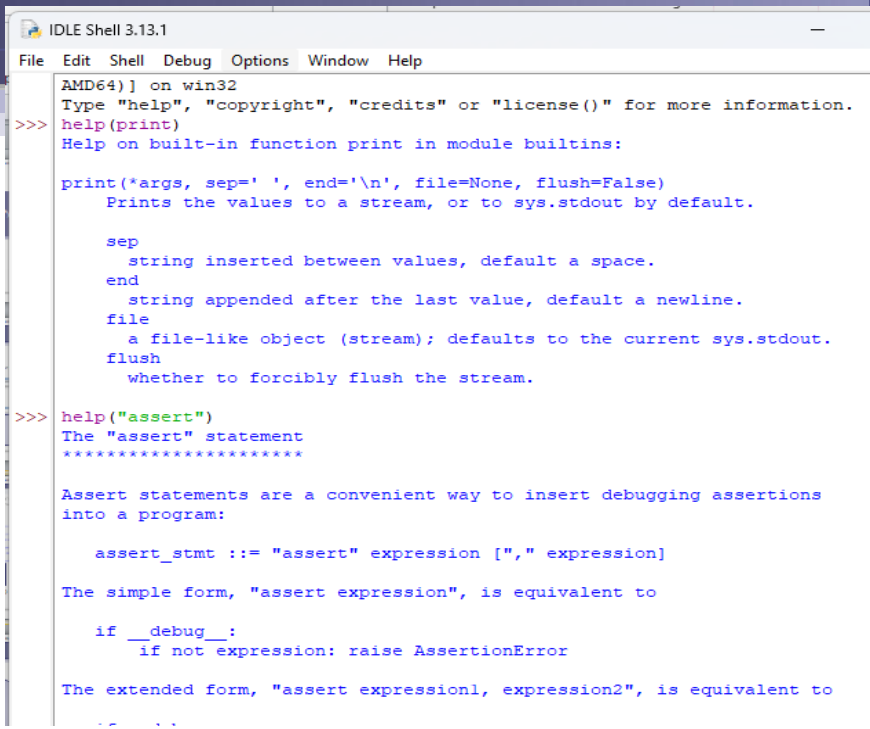
  ……

  …….dernière ligne de commentaire """

Chapitre 1 : instructions de base

Mots résérvés au langage python

False	class	from	or	None
global	pass.	True	def	if
and	del	return	break	for
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)

>>> est l'invite de commande de l'interpréteur Python (prompt en anglais)

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

✓ Python est sensible à la casse, ce qui signifie que les variables age, Age et AGE sont différentes.

➢ **Exemples : i, numéroProduit, coef_a, X1, age. . .**

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'>
```

```
>>> x=3.45
>>> type(x)
<class 'float'>
>>>
```

Chapitre 1 : Instructions de base

Types natifs des données

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

Chapitre 1 : instructions de base

Opérateurs arithmétiques sur les types numériques

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

Opérateurs logiques

and	ET logique
or	OU logique
not	Négation logique

Chapitre 1 : instructions de base

Opérations sur les chaînes de caractères

- Pour les chaînes de caractères, deux opérations sont possibles: l'addition et la multiplication

'Salut' + " Python" ⟶ 'Salut Python'

'Salut' * 3 ⟶ 'SalutSalutSalut'

- L'opérateur d'addition + concatène (assemble) deux chaînes de caractères.
- L'opérateur de multiplication * entre un nombre entier et une chaîne de caractères duplique (répète) plusieurs fois une chaîne de caractères.
- **'** et **"** sont équivalents pour définir des chaînes de caractères.
- Python offre également des **triple guillemets** (**'''** ou **"""**) pour les chaînes multilignes. Ces guillemets sont utiles pour les chaînes longues ou lorsque tu veux inclure des sauts de ligne.

Chapitre 1 : instructions de base

Comparaison des chaînes de caractères

- C'est l'ordre alphabétique qui est utilisé dans le cas où l'on compare deux lettres majuscules ou minuscules.
- les majuscules apparaissent avant les minuscules (en termes de codes Unicode).Par conséquent, les majuscules sont considérées comme plus petites que les minuscules.

Exemple:

- "a" < "b"
- 'M' < 'm'
- "Maman" < "Papa"
- "maman" > "Papa "
- ord('A') renvoie 65
- ord('a') renvoie 97
- chr(65) envoie 'A'
- len("Bonjour") renvoie 7

➢ **Fonctions utiles pour la comparaison des chaines de caractères :**

ord() : permet de connaître le code Unicode d'un caractère.

chr() :permet de connaître le caractère correspondant à un code Unicode.

Len(): renvoie le nombre de caractères que la chaine contient.

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

int(x)	Convertir x vers un nombre entier
float (x)	Convertir x vers un nombre réel
str(x)	Convertir x vers une chaine de caractères

➢ **Exemples :**

```
x = "42"
y = int(x)
-> y va contenir 42
```

```
x = 3.14
y = int(x)
-> y va contenir 3
```

```
x = "abc"
y = int(x)
 -> Cela lèvera une erreur
```

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 :

❑En algorithmique, l'affectation se note avec **le signe** ←

Var← e : attribue la valeur de e à la variable Var

- **e** peut être une valeur, une autre variable ou une expression
- l'affectation ne modifie que ce qui est à gauche de la flèche

❑ En python, l'affectation se note avec **le signe** **=**

- **Identificateur_variable = valeur simple**
- **Identificateur_variable = expression arithmétique**

➢ **Exemples :**

```
a=3
x=y
s=p*(r**2)
```

Chapitre 1 : instructions de base

Opérateurs d'affectation

- 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 :

```
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.

```
>>> x=2
>>> y=2.3
>>> x=y='abc'
>>> print(type(y))
<class 'str'>
>>>
```

Chapitre 1 : instructions de base

Opérateurs d'assignation

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

Exemple :

```
A=3  A+=2     #Ajouter 2 à A puis stocker la nouvelle valeur dans A → A va contenir 5
B=12 B//=5    # Diviser B sur 5 puis stocker la partie entière du résultat dans B → B va contenir 2
C=2  C**=3    # calculer C à la puissance 3 puis stocker le résultat dans C : → C va contenir 8
```

Chapitre 1 : instructions de base

➢ ***Essayez de prédire le résultat de chacune des instructions suivantes:***

- `(1+2)**3` → **27**
- `"Da" * 4` → **'DaDaDaDa'**
- `"Da" + 3` → **TypeError: can only concatenate str (not "int") to str**
- `("Pa"+"La") * 2` → **'PaLaPaLa'**
- `("Da"*4) / 2` → **TypeError: unsupported operand type(s) for /: 'str' and 'int'**
- `5 / 2` → **2.5**
- `5 // 2` → **2**
- `5 % 2` → **1**
- `str(4) * int("3")` → **'444'**
- `int("3") + float("3.2")` → **6.2**
- `str(3) * float("3.2")` → **TypeError: can't multiply sequence by non-int of type 'float'**
- `str(3/4) * 2` → **'0.750.75'**

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 } " )
```

Exemple

```
S=21
print ( " La somme est : ")
print (S)
print (2*S)
print("La somme est:",S)
print ( f " La somme est : {S} ")
```

Résultat d'exécution

NB:
f : Indique que la chaîne est une f-string

Chapitre 1 : instructions de base

Instruction d'écriture : print

Mot clé sep

```
>>> nom=" Mohamed"
>>> x=24
>>> print(nom,'a',x)
 Mohamed a 24
>>> print(nom,'a',x,sep='')
 Mohameda24
>>> print(nom,'a',x,sep='-')
 Mohamed-a-24
```

- **Séparateur par défaut (espace) :** les éléments sont séparés par un espace par défaut.
- Le mot-clé **sep** en Python est utilisé dans la fonction **print()** pour définir un séparateur personnalisé entre les éléments que tu veux afficher.

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 **chaine de caractères,** Pour faire des calculs sur la valeur saisie, il faut la convertir à **un entier** avec la fonction **int (nom_variable)** ou **un réel** avec la fonction **float(nom_variable).**

Syntaxe générale

iden_var = **input** ()

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

```
>>> x=input()
12
>>> print(x)
12
>>> type(x)
<class 'str'>
```

```
>>> x=int(input())
12
>>> type(x)
<class 'int'>
```

NB: L'argument (le texte) que tu mets entre les guillemets dans **input ("message")** est le message d'invite que l'utilisateur verra avant d'entrer sa réponse. Ce message est facultatif, mais il peut être utile pour guider l'utilisateur sur ce qu'il doit entrer.

```
>>> x=input("Quel est l'âge du condidat:")
Quel est l'âge du condidat:25
>>> print("l'âge du condidat est",x)
l'âge du condidat est 25
```

Chapitre 1 : instructions de base

Instruction de lecture des variables : input

- Problème de l'exemple précédent:

```
>>> type(x)
<class 'str'>
>>> x+=5
Traceback (most recent call last):
  File "<pyshell#48>", line 1, in <module>
    x+=5
TypeError: can only concatenate str (not "int") to str
```

- Solution:

```
>>> x=int(input("Quel est l'âge du condidat:"))
Quel est l'âge du condidat:25
>>> print("l'âge du condidat est",x)
l'âge du condidat est 25
```

```
>>> x+=5
>>> print(x)
30
```

Exemple

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

Résultat d'exécution

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
```

Condition

Instructions à exécuter si l'expression logique est vraie

Exemple

```
if  X > 0 :
      print ( " c'est un nombre  strictement positif ")
```

Chapitre 1 : structure conditionnelle

L'instruction : if … else

if expression logique **:** — Condition

Instruction 1
…
Instruction n — à exécuter si l'expression logique est vraie

else :

Instruction 1
…
Instruction m — à exécuter si l'expression logique est fausse

Exemple

```
if X > 0 :
    print (" c'est un nombre strictement positif")
else :
    print (" c'est un nombre négatif ou nul ")
```

NB:

Indentation: Espaces ou tabulations qui définissent quels morceaux du code appartiennent à quel bloc

Chapitre 1 : structure conditionnelle

L'instruction : if … elif … else

```
if condition1:
    # Code exécuté si condition1 est vraie
elif condition2:
    # Code exécuté si condition1 est fausse et condition2 est vraie
elif condition3:
    # Code exécuté si les conditions précédentes sont fausses et condition3 est vraie
…
elif condition4:
    # Code exécuté si les conditions précédentes sont fausses et condition4 est vraie
else:
    # Code exécuté si aucune des conditions précédentes n'est vraie
```

Chapitre 1 : structure conditionnelle

Exemple 1

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

Chapitre 1 : structure conditionnelle

Exemple 2: Ecrire un script Python qui affiche si vous êtes senior, adulte, adolescent ou enfant selon votre âge .

Avec : if elif

```
age = int(input("Entrez votre âge : "))
if age >= 60:
    print("Vous êtes senior.")
elif age >= 18:
    print("Vous êtes adulte.")
elif age >= 13:
    print("Vous êtes un adolescent.")
else:
    print("Vous êtes un enfant.")
```

Avec: imbrication de if else

```
age = int(input("Entrez votre âge : "))

if age >= 60:
    print("Vous êtes senior.")
else:
    if age >= 18:
        print("Vous êtes adulte.")
    else:
        if age >= 13:
            print("Vous êtes un adolescent.")
        else:
            print("Vous êtes un enfant.")
```

Chapitre 1 : structure conditionnelle

- La structure conditionnelle ***match … case*** **(Selon en algorithmique)** est appelée aussi ***à choix multiple*** ou ***sélective*** car elle sélectionne entre plusieurs choix à la fois, et non entre deux choix alternatifs (le cas de la structure IF).

L'instruction : match … case

match Expression :

case Valeur1 :

Instructions à exécuter si **Expression = Valeur1**

…

case ValeurN :

Instructions à exécuter si **Expression = ValeurN**

case other : (ou _)

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

Chapitre 1 : structure conditionnelle

Exemple

```
mois= int(input("Entrez le numéro du mois : "))

match mois:
    case 1: print("C'est Janvier")
    case 2: print("C'est Février")
    case 3:print("C'est Mars")
    case 4:print("C'est Avril")
    case 5:print("C'est Mai")
    case 6: print("C'est Juin")
    case 7:print("C'est Juillet")
    case 8: print("C'est Août")
    case 9:print("C'est Septembre")
    case 10:print("C'est Octobre")
    case 11:print("C'est Novembre")
    case 12: print("C'est Decembre")

    case _:  print("Erreur, Tapez un numéro entre 1 et 12")
```

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 mentions affichée est : **Bien**
- Si **12 <= moyenne <14**, la mentions affichée est **Assez Bien**
- Si **10 <= moyenne <12**, la mentions 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

Les boucles

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

- ***On distingue 2 types de boucles en langages de programmation :***

❑ ***Les boucles à compteur ou définie:***

- *On sait à l'avance combien de fois la boucle devra tourner et une variable (le compteur) compte les répétitions.*

Ex : ***la boucle Pour en algorithmique***

❑ ***Les boucles à événement ou indéfinie:***

- *On ne sait pas à l'avance le nombre de fois que la boucle sera exécutée.*

Ex : la boucle ***Tantque*** *et la boucle* ***Répéter …jusqu'a***

Chapitre 1 : structure itérative

- Deux boucles sont utilisées en python : La boucle **for** et la boucle **while**

La boucle for

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

} à exécuter pour chaque valeur du compteur

Exemple

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

Résultat d'exécution

Chapitre 1 : structure itérative

La boucle for

```
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

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 k in range (1, 10, 3) :
    print (k, end = "  ")
```
 # le résultat d'exécution est l'affichage des valeurs 1 4 7

Remarque :

L'instruction range fournit une liste de valeurs.

Exemples :

range(6)	représente la liste des valeurs : **0**,1, 2, 3, 4, 5
range (1,6)	représente la liste des valeurs : 1, 2, 3, 4, 5
range (1, 6,1)	représente la liste des valeurs : 1, 2, 3, 4, 5
range (1,6, 2)	représente la liste des valeurs : 1,3,5
range(6,5,-1)	représente la valeur suivante : 6

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

NB: En Python , on peut réaliser l'imbrication des boucles, qui consiste à placer une boucle à l'intérieur d'une autre boucle.

Exemple

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

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
```

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-à-d, 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 "Bonjour":
   if  val == "o":
      continue
    print(val)
```

Résultat d'exécution

```
B
n
j
u
r
```

Exemple avec la boucle while

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

Résultat d'exécution

```
6
4
3
2
1
```

Chapitre 1 : Bases de la programmation Python

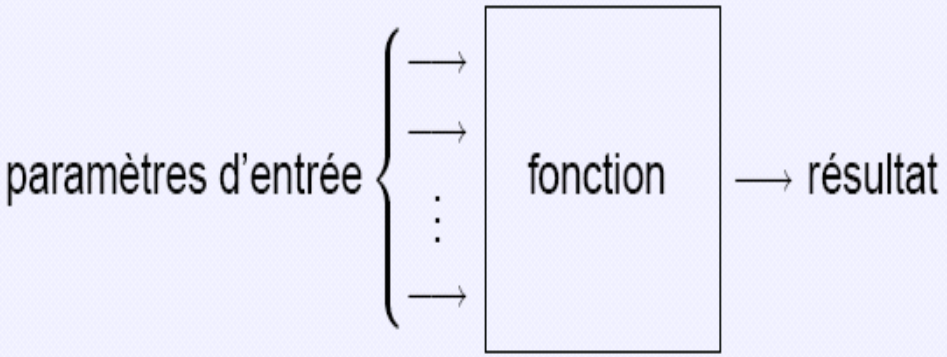
Exercices : Série N°1

Chapitre 2 : Fonctions & Modules

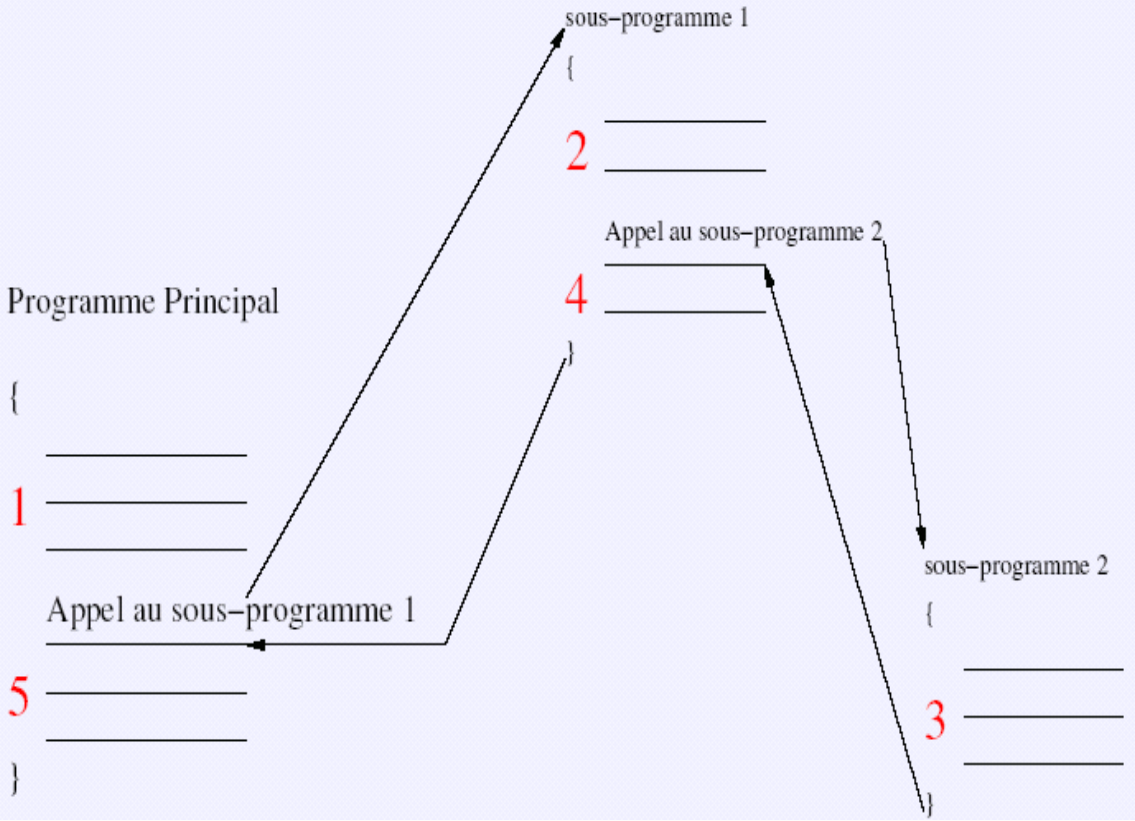
Chapitre 2 : Fonctions

■Certains problèmes conduisent à des programmes longs, difficiles à écrire et à comprendre. On les découpe en des parties appelées **sous-programmes.**

- Une fonction est un sous-programme qui retourne une valeur calculée en fonction des valeurs passées en entrée



- Une fonction prend zéro ou plusieurs paramètres et renvoie éventuellement un résultat
- Une fonction qui ne retourne pas de résultat est une procédure



Chapitre 2 : Fonctions

■***Les fonctions et les procédures sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs intérêts :***

- Permettent de **"factoriser" les programmes**, c.à.d. de mettre en commun les parties qui se répètent.
- Permettent une **structuration** et une **meilleure lisibilité** des programmes.
- **Facilitent la maintenance** du code (il suffit de modifier une seule fois).
- Ces procédures et fonctions peuvent éventuellement être **réutilisées** dans d'autres programmes.

Chapitre 2 : Fonctions

Définition

- Une fonction est un **bloc d'instructions** regroupées sous un **même nom** .
- Une fonction peut avoir des **paramètres/arguments**
- Les paramètres servent à échanger des données entre le programme principale (ou la fonction appelante) et la fonction appelée.
- 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 : 15
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

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)
```

paramètre formel n

paramètre effectif N

Résultat d'exécution

```
Entrer le nombre des valeurs à afficher : 15
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
```

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

```
entrer le premier nombre : 12
entrer le deuxième nombre : 45
la somme des deux valeurs est :  57
>>>
```

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

```
entrer le premier nombre : 6
entrer le deuxième nombre : 4
la somme des deux valeurs est :  10
>>>
```

Chapitre 2 : Fonctions

Exercice d'application

Écrire une fonction **volumeSphere** qui calcule le volume d'une sphère de rayon r.

- Tester la fonction volumeSphere par un appel dans le programme principal.

NB: Volume d'une sphère = 4/3 * π * (Rayon) ^{3}

Solution

```
def volumeSphere(r):
    pi = 3.14
    volume = (4/3) * pi * (r ** 3)
    return (volume)
# appeler la fonction
rayon = float(input("Saisir le rayon de la sphère : "))
print(f"Le volume de la sphère avec un rayon de {rayon} est {volumeSphere(rayon):.2f}:")
# OU
V=volumeSphere(rayon)
Print("Le volume de la sphère est:" , V:.2f)
```

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

Chapitre 2 : Fonctions

Les fonctions récursives :Exemple d'une fonction calculant la factorielle d'un entier

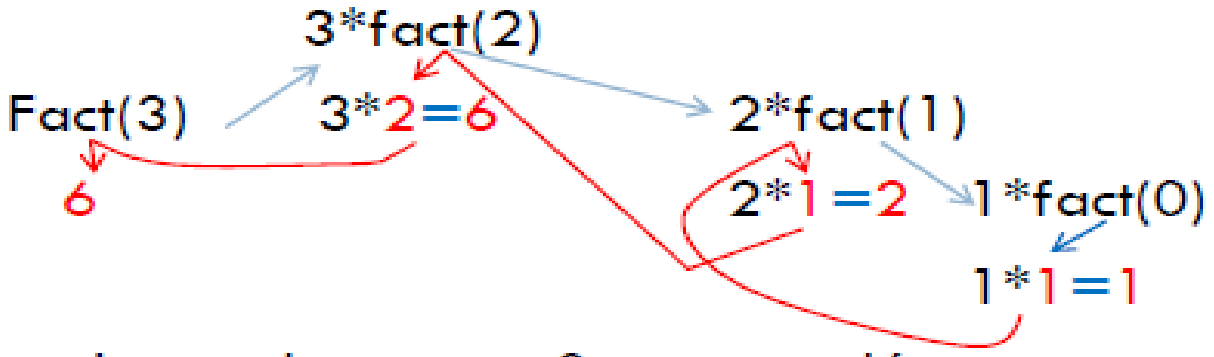
Méthode récursive

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

➢ **Appel de la fonction:**

```
x = int(input("Saisir un nombre : "))
print(f"La factorielle de {x} est {factorielle(x)}")
```

- Le déroulement de l'appel de fact(3):



- La condition n = 0 est appelée test d'arrêt de la récursivité.
- Il est impératif de prévoir un test d'arrêt dans une fonction récursive, sinon l'exécution ne s'arrêtera jamais.
- L'appel récursif est traité comme n'importe appel de fonction.

Chapitre 2 : Modules

Définition

- 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'organiser et de structurer le code ;
  - 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.
- Module **os** : pour interagir avec le système d'exploitation

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))
```

```
from math import sqrt   # Utilisation directe de sqrt
 print(sqrt(16))          # Affiche 4.0
```

Chapitre 2 : Modules

Exemple de création et d'importation d'un module

Module calcul.py

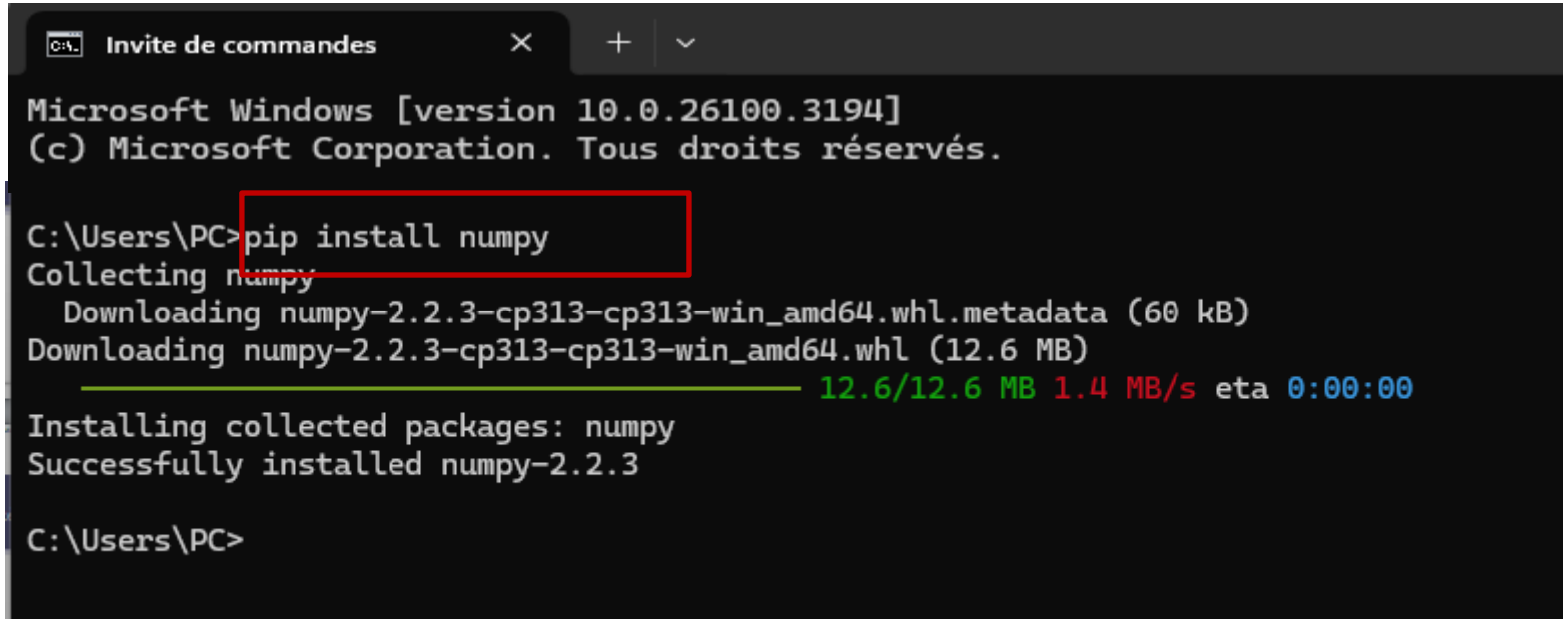
```
def addition (x, y) :
    return (x+y)

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 : Modules

- ***Exemple d' installation de modules non inclus par défaut dans Python via pip***



Chapitre 2 : Fonctions & Modules

Exercices : Série N2

Chapitre 3 : Chaines de caractères

Chapitre 3 : chaines de caractères

Définition

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

Syntaxe de définition des chaines de caractères

```
ch1 = " c'est une chaine de caractères "

ch2 = ' ceci est une chaine de caractères '

ch3 = ""   # déclaration d'une chaine vide

ch4 ="""chaine de caractères sur plusieurs lignes """   ou   ch5 = ''' haine de caractères sur plusieurs lignes '''

ch5 = r " c'est un texte1\n \t c'est un texte2 "  # ceci consiste à ignorer les caractères spéciaux \n \t
```

Chapitre 3 : chaines de caractères

Manipulation des chaines de caractères

- Les opérateurs mathématiques qui peuvent être appliqués sur une chaine de caractères sont :

 * et +

- Pour la comparaison des chaines de caractères, on utilise les opérateurs : ==, **!**= ,< , >

Exemple

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

Chapitre 3 : chaines de caractères

Manipulation des chaines de caractères

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

Caractère	s	a	l	u	t		l	e	s		p	r	o	g	r	a	m	m	e	u	r	s
Indice (positif)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Indice (négatif)	-22	-21	-20	-19	-18	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Exemple

```
ch3 = "salut les programmeurs"
print (ch3[1] ) # permet d'afficher le caractère 'a'
print (ch3[-5] ) # permet d'afficher le caractère 'm'
```

Chapitre 3 : chaines de caractères

Manipulation des chaines de caractères

Le **slicing** des chaînes en Python est une fonctionnalité puissante pour extraire, manipuler et transformer des sous-parties de chaînes de manière concise et efficace. Cela permet de travailler avec des chaînes de façon flexible sans avoir à recourir à des boucles complexes.

La syntaxe de base pour le **slicing** d'une chaine est : **chaine[start:stop:step]**

start : L'indice de début (inclus).

stop : L'indice de fin (exclus).

step : Le pas (pour choisir un intervalle entre les éléments).

- Les indices **négatifs** sont utilisés pour accéder aux éléments depuis la fin de la chaîne.
- Les chaînes sont **immuables** ; le slicing crée de nouvelles chaînes.

Exemple: ch3 = "salut les programmeurs"

```
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'
print (ch3[0:12:2]) # Découper de l'indice 0 à l'indice 12 avec un pas de 2: 'sltlsp'
print ( ch3 [:])  # permet d'afficher toute la chaîne  'salut les programmeurs'
```

Chapitre 3 : chaines de caractères

La fonction len (), bool ()

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

Exemple

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

Chapitre 3 : chaines de caractères

Méthodes associées aux chaines de caractères

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

Exemple

print(dir(str))

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

Chapitre 3 : chaines de caractères

Les méthodes startswith(), endswith()

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

Exemple

```
chaîne = "234567891"
if chaîne.startswith ("2345") :
    print("ok")        # permet d'afficher ok
print(chaîne.endswith ("91"))   # permet d'afficher True
print(chaîne.endswith ("2"))    # permet d'afficher False
```

Chapitre 3 : chaines de caractères

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

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

Exemple

string = "CE MATIN il fait beau"

```
print(string.upper()) #permet d'afficher 'CE MATIN IL FAIT BEAU '
print(string.lower()) #permet d'afficher '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'
print(string.title())) #permet d'afficher 'Ce Matin Il Fait Beau'
```

```
CE MATIN IL FAIT BEAU
ce matin il fait beau
ce matin IL FAIT BEAU
Ce matin il fait beau
Ce Matin Il Fait Beau
>>>
```

Chapitre 3 : chaines de caractères

Les méthodes : index (), rindex ()

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

Exemple

```
chaine1 = "Bonjour  tout  le monde"
print(chaine1.index("t")) # permet d'afficher  8
print(chaine1.index("u", 7, 19)) # permet d'afficher 10
print(chaine1.rindex("o") )# permet d'afficher 17
print(chaine1.rindex("o", 7, 19) )# permet d'afficher 17
print(chaine1.rindex("o", 5,9) )# permet d'afficher ValueError: substring not found
```

Voici comment les **index** sont attribués :

B	o	n	j	o	u	r		t	o	u	t		l	e		m	o	n	d	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Chapitre 3 : chaines de caractères

La méthode : count ()

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

Exemple

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

Voici comment les **index** sont attribués :

B	o	n	j	o	u	r		t	o	u	t		l	e		m	o	n	d	e
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

Chapitre 3 : chaines de caractères

Les méthodes : find (), rfind()

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

Exemple

```
chaine1 = "Salut tout le monde tout le monde"
chaine2 = "tout"
print(chaine1.find(chaine2)) # permet d'afficher 6
print(chaine1.rfind(chaine2)) # permet d'afficher 20
print(chaine1.find("toute")) # permet d'afficher -1
```

Chapitre 3 : chaines de caractères

La méthode : replace ()

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

Exemple

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

Chapitre 3 : chaines de caractères

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

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

Exemple

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

Chapitre 3 : chaines de caractères

Les méthodes : isalpha() ,isdigit(), isalnum (),isspace()

- **isalpha()** est une méthode de chaîne en Python qui permet de vérifier si tous les caractères d'une chaîne sont des lettres alphabétiques (A-Z et a-z).
- **Retourne True** si tous les caractères de la chaîne sont des lettres (sans chiffres, espaces ou symboles). **Retourne False** sinon.
- Fonctionne aussi pour les lettres accentuées et les caractères alphabétiques d'autres langues grâce au support Unicode de Python

Exemple

```
print("Bonjour".isalpha())        # True
print("Bonjour123".isalpha())     # False (contient des chiffres)
print("".isalpha())               # False (chaîne vide)
print("Salut!".isalpha())         # False (caractère spécial "!")
print("Café".isalpha())  # True (l'accent est accepté car c'est une lettre Unicode)
print("salut le monde".isalpha())   # False (car il y a des espaces)
```

Méthode	Vérifie quoi ?
`isalpha()`	Seulement des lettres
`isdigit()`	Seulement des chiffres
`isalnum()`	Lettres ou chiffres (alphanumérique)
`isspace()`	Seulement des espaces blancs

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

  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**.

Une structure de données sert à :

- Ranger des données (ex : des nombres, des textes…)
- Pouvoir **accéder facilement** à ces données
- Les **modifier** ou les **trier** rapidement
- Résoudre des problèmes plus efficacement

Chaque structure est adaptée à un besoin précis :

- Rapidité d'accès
- Pas de doublons
- Ordre important ou non
- Données modifiables ou figées

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 = [element1, element_2, ..., element_n]

Exemple

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

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 dupliquer la liste exactement **valeur_numérique** fois.

  nom_liste1= nom_liste * valeur_numérique

Exemple

```
Liste_1 = [13, 11]      #permet de définir une liste de chiffres
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

Appliquer des opérations sur les listes

- Avec l'opérateur **+=**, vous pouvez ajouter des éléments de manière assez simple.

x +=y ajoute tous les éléments de **y** à la fin de **x** , **mais sans créer une nouvelle liste**.

```
list = [1,2,3,"a"]
list += [43]

print(list)
>>>> [1,2,3,"a", 43]
```

```
list += "hello"
print(list)
>>>> [1, 2, 3, 'a', 43, 'h', 'e', 'l', 'l', 'o']
```

```
list += ["hello"]
print(list)

>>>> [1, 2, 3, 'a', 43, 'h', 'e', 'l', 'l', 'o', 'hello']
```

Chapitre 4 : les listes

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

Exemple

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

->*NB:*Quand tu mets 2, Python va **découper la chaîne en 3 morceaux max** (parce qu'il fait **2 coupures**).

Chapitre 4 : les listes

Autres méthodes de liste : join()

- La méthode **join()** permet de fusionner les éléments d'une **liste de chaînes de caractères** en une seule chaîne 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'indiçage **positif** (0, 1, 2, etc.)
  - L'indiçage **négatif** (-1, -2, -3, etc.)
  - La méthode de **slicing** (extraire une tranche d'éléments d'une liste)

Exemple

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

Chapitre 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

ou

for nom_variable **in range(len(**nom_lliste**)):**
    Instruction_1
    … Instruction_n

Exemple

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

Résultat d'exécution

```
Lundi
Mardi
Mercredi
Jeudi
```

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

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)

•enumerate(Liste) ➡commence à 0 par défaut.

•enumerate(Liste, start=1) ➡commence à 1 (ou n'importe quel autre nombre que tu veux).

Exemple

```
Liste_1 = ["Lundi", "Mardi", "Mercredi", "Jeudi"]
for i, x in enumerate(Liste_1) : print (i, x)
for i, x in enumerate(Liste_1 , start=1) : print (i, x)
```

Résultat d'exécution

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

```
1 Lundi
2 Mardi
3 Mercredi
4 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
print (L) # permet d'afficher []
```

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
print (L) # permet d'afficher : [1, 2, 3, 6, 7, 9]
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** .

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, 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 chaines 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() ) # permet d'afficher :[2, 6, 11, 18, 19, 25, 27, 43, 114]
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 occurence 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 d'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_m]]

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]
```

Liste[1] fait référence à la deuxième sous-liste de Liste, qui est [2, 'Mai', 2005].
Ensuite, [2]fait référence à l'élément à l'indice 2 de cette sous-liste, qui est l'année 2005.

crée une copie de la liste et accède à la deuxième sous-liste de cette copie.

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 :**

nouvelle_liste = [fonction(element) for element in 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]
print (liste_2) # permet d'afficher : [1, 4, 9, 16, 25]
```

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]
print (L2) # permet d'afficher : [2, 4, 6, 8, 10]
```

Chapitre 4 : les listes

Exercice d'application

Écrire un programme en Python qui effectue les opérations suivantes :

1.Créer une liste **ma_liste** contenant les nombres suivants : 10, 20, 30, 40, 50, 60, 70.

2.Ajouter le nombre 80 à la fin de la liste.

3.Remplacer le troisième élément par 35.

4.Supprimer le dernier élément de la liste.

5.Afficher la longueur de la liste (nombre d'éléments).

6.Afficher tous les éléments un par un à l'aide d'une boucle.

7. Vérifier si un nombre saisie par l'utilisateur est présent dans la liste.

8. Trouver et afficher le plus grand et le plus petit nombre de la liste.

9.Afficher la liste dont les éléments sont la moitié des valeurs de **ma_liste** et qui sont des multiples de 20.

Chapitre 4 : les listes

Solution

```
# 1. Créer une liste
ma_liste = [10, 20, 30, 40, 50, 60, 70]
# 1. Créer une liste autrement
ma_liste = list(range(10,71,10))
print(ma_liste)
# 1. Si je veux demander à l'utilisateur de saisir les éléments de la liste:
ma_liste = []
# Demander à l'utilisateur combien de nombres il veut ajouter
n = int(input("Combien de nombres voulez-vous ajouter à la liste ? "))
# Demander à l'utilisateur de saisir chaque nombre
for i in range(n):
    valeur = int(input(f"Saisissez le {i+1}-ème nombre: "))
    ma_liste.append(valeur)
print(ma_liste)
```

Chapitre 4 : les listes

Solution

```
# 2. Ajouter 80 à la fin de la liste
ma_liste.append(80)
print("Liste après ajout :", ma_liste)
# 3. Remplacer le troisième élément (30) par 35
ma_liste[2] = 35
print("Liste après modification :", ma_liste)
# 4. Supprimer le dernier élément
ma_liste.pop()
print("Liste après suppression :", ma_liste)
# 5. Afficher la longueur de la liste
print("Longueur de la liste :", len(ma_liste))
# 6. Afficher tous les éléments un par un
print("Éléments de la liste :")
for elem in ma_liste:
    print(elem)
```

```
# 7.  Vérifier si un nombre est présent dans la liste
nombre = int(input("Entrez un nombre pour vérifier s'il est dans la liste : "))
if nombre in ma_liste:
    print(f"{nombre} est présent dans la liste ")
else:
    print(f"{nombre} n'est pas présent dans la liste ")
#8. Trouver le plus grand et le plus petit élément
plus_grand = max(ma_liste)
plus_petit = min(ma_liste)
print("Plus grand élément :", plus_grand)
print("Plus petit élément :", plus_petit)
#9.
resultat = [x / 2 for x in ma_liste if x % 20 == 0]
print(resultat)
```

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 →(1, 2, 3, 4)
print(tuple_2)      # permet d'afficher : (1, 2, 3, 4)
tuple_3 = 1, 2, 3, 4 # permet de définir un tuple des entiers →(1, 2, 3, 4)
print(sum(tuple_3))     # permet d'afficher 10
tuple_4 = ("Lundi", 22, "Janvier", 2000) # permet de définir un tuple de différents types
print(tuple_4, tuple_4[-1])    # permet d'afficher : ('Lundi' 22, 'Janvier', 2000)  2000
tuple_5 =(34,)  # permet d'afficher :(34,)
```

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 ne modifient pas les éléments du tuple *(car un tuple est une séquence immuable) tel que* **len()**, **min() / max()**, **sum()**.
- 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   # permet d'afficher : ('a', 'b', 'a', 'b', 'a', 'b', 'a', 'b', 'c', 'd', 'e')
print(tuple3)
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 chaine de caractères vers le type **tuple**
- La syntaxe utilisée est la suivante :

nom_tuple= tuple (nom_chaine)

nom_tuple=tuple(nom_liste)

Exemple

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

Chapitre 4 : les tuples

Exemple

- On peut extraire les valeurs d'un tuple dans des variables :

```
coordonnees = (10, 20)
 x, y = coordonnees
print(x) # 10
print(y) # 20
```

- On peut aussi utiliser * pour stocker plusieurs éléments :

```
t = (1, 2, 3, 4, 5)
a, *b, c = t
print(a)  # 1
print(b)  # [2, 3, 4]
print(c)  # 5
```

➢ ***Différences entre liste et tuple***

Caractéristique	Liste (`list`)	Tuple (`tuple`)
Délimitation	`[]`	`()`
Modifiable	Oui (mutable)	Non (immuable)
Taille dynamique	Oui	Non
Performance	Plus lent	Plus rapide
Utilisation	Quand on a besoin de modifier les données	Pour des données fixes (ex: coordonnées GPS)

Chapitre 4 : le dictionnaire

Définition d'un dictionnaire

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

  nom_dictionnaire = { clef_1 : valeur, clef_2 : valeur, …, clef_n : valeur}
- Les clefs d'un dictionnaire sont **uniques** et **immuables** (ex : nombres, chaînes de caractères, tuples)..

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'}
dico = dict(nom="Amine", âge=16, classe="5ème") OU dico = dict("nom":"Amine", "âge":16, "classe":"5ème")
print(dico) # permet d'afficher : {'nom': 'Amine', 'âge': 16, 'classe': '5ème'}
```

Chapitre 4 : le dictionnaire

La méthode get()

- La méthode **get()** renvoie la valeur de **la clé** 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[2]) #permet d'afficher : deux, mais, si la clé n'existe pas, Python renvoie une erreur. Pour éviter cela, on peut utiliser .get()
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

Exemple

```
dico = dict(nom="Amine", âge=16, classe="5ème") OU dico = dict("nom":"Amine", "âge":16, "classe":"5ème")
print(dico) # permet d'afficher : {'nom': 'Amine', 'âge': 16, 'classe': '5ème'}
dico["moyenne"] = 15.5 #Ajouter une nouvelle clé-valeur
print(dico)
dico["âge"] = 17 #Modifier une valeur
print(dico)
del dico["moyenne"] #Supprimer des éléments
print(dico)
Sa-classe = dico.pop("classe") #Supprimer des éléments
print(Sa-classe)  # 5ème
print(dico)
print("nom" in dico)  # affiche True :il permet de vérifier si une clé existe
print("moyenne" in dico)  # False
dico.clear()        #Supprimer tous les éléments
print(dico)      # {}
```

```
{'nom': 'Amine', 'âge': 16, 'classe': '5ème'}
{'nom': 'Amine', 'âge': 16, 'classe': '5ème', 'moyenne': 15.5}
{'nom': 'Amine', 'âge': 17, 'classe': '5ème', 'moyenne': 15.5}
{'nom': 'Amine', 'âge': 17, 'classe': '5ème'}
5ème
{'nom': 'Amine', 'âge': 17}
{}
>>>
```

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_dictionaire = { clef_1 : liste_1, clef_2= liste_2,…., clef_n : liste_n}

Exemple

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

Chapitre 4 : le dictionnaire

Dictionnaire d'une structure de données

Exemple

dico_3= *{1: {'Hiver': ['Décembre', 'Janvier', 'Février'], 'Printemps': ['Mars', 'Avril', 'Mai']}, 2: {'Eté': ['Juin', 'Juillet', 'Aôut'], 'Automne': ['Septembre', 'Octobre', 'Novembre']}}*

dico_3[1] → Accède au premier dictionnaire (dico_1).

dico_3[1]["Hiver"] → Accède à la liste des mois d'hiver.

print(dico_3[1]["Hiver"]) →résultat: ['Décembre', 'Janvier', 'Février']

dico_3[1]["Hiver"][0] →Accéder au premier mois de l'hiver

print(dico_3[1]["Hiver"][0]) →affiche le premier élément de cette liste → résultat: Décembre

dico_3[2]["Eté"][-1] → Prend le dernier élément de la liste des mois d'été .

print(dico_3[2]["Eté"][-1]) → affiche: Août

Chapitre 4 : l'ensemble set

➢ *Une structure ensembliste set:*

Une structure **ensembliste** sert à **gérer des collections d'éléments uniques**, sans doublons. Elle permet aussi de faire des **opérations mathématiques d'ensemble**, comme l'union, l'intersection, etc.

- il n'y a pas **d'accès indexé** aux éléments, **ni d'ordre** entre ces derniers,
- On construit un ensemble par une expression de la forme : **set (Séquence)** où séquence est une donnée parcourable (liste, tuple, chaîne de caractères, range, etc.).

Exemple

```
animaux = {"chat", "chien", "lapin", "chat"}
print(animaux)  # {'chien', 'lapin', 'chat'} → le doublon "chat" est supprimé
```

Exemple

```
a =set() #définir un ensemble vide
print (a) # Le script affiche:{}
```

Chapitre 4 : l'ensemble set

➢ *Une structure ensembliste set:*

Une structure **ensembliste** sert à **gérer des collections d'éléments uniques**, sans doublons. Elle permet aussi de faire des **opérations mathématiques d'ensemble**, comme l'union, l'intersection, etc.

- il n'y a pas **d'accès indexé** aux éléments, **ni d'ordre** entre ces derniers,
- On construit un ensemble par une expression de la forme : **set (Séquence)** où séquence est une donnée parcourable (liste, tuple, chaîne de caractères, range, etc.).

Exemple

```
animaux = {"chat", "chien", "lapin", "chat"}
print(animaux)  # {'chien', 'lapin', 'chat'} → le doublon "chat" est supprimé
```

Exemple

```
a =set() #définir un ensemble vide
print (a) # Le script affiche:{}
```

Chapitre 4 : l'ensemble set

a = set(['ENCGT', 'S6','Prog Python ']) #définir un ensemble à partir d'une liste

print (a) #Affiche **{'ENCGT', 'S6','Prog Python'}**

a = set(range(10)) #définir un ensemble à partir de range

print (a) #Affiche **{0,1,2,3,4,5,6,7,8,9}**

➢ ***Opérations ensemblistes en Python :***

Opération	Symbole Python	Exemple
Union	`|`	ou .union()`
Intersection	`&` ou `.intersection()`	`A & B` → éléments dans A et B
Différence	`-`	`A - B` → éléments dans A mais pas B
Diff. symétrique	`^`	`A ^ B` → dans A ou B mais pas les deux

Exemple

```
A = {1, 2, 3}
B = {3, 4, 5}
print(A | B) ou  A.union(B)   # affiche{1, 2, 3, 4, 5}
print(A & B) ou A.intersection(B)  # affiche: {3}
print(A - B)  # Différence: {1, 2}
print(A ^ B)  # Différence symétrique: {1, 2, 4, 5}
```

Chapitre 4 : l'ensemble set

➢ *Opérations ensemblistes en Python :*

ensemble **.add(**élément**)**: ajout de l'élément indiqué à l'ensemble indiqué

ensemble **.remove(**élément**)**: suppression de l'élément indiqué de l'ensemble indiqué

ensemble1 **.issubset(**ensemble2**)**: Tous les éléments de l'ensemble 1 appartiennent à l'ensemble2?

Exemple

```
a ={'ENCG', 'S6','Prog Python'}
a.add('Tanger')
print(a) → permet d'afficher :{'S6', 'Prog Python', 'ENCG', 'Tanger'}
a.remove('Prog Python')
print(a) → permet d'afficher: {'Tanger', 'S6', 'ENCG'}
b={'Tanger', 'S6', 'ENCG'}
c= b.issubset(a)
print( c ) → permet d'afficher: True
```

Exercices : Série N3

Chaines de caractères et structures de données