Programmation Python

2022/2023

Prof : Amina BENGAG

Pr. BENGAG

Programmation Python

Chapitre 2 : Les éléments de base

Pr. Amina BENGAG 18

Les éléments de base

I. Structure d'un programme en Python

1. variable

- Une variable stocke une donnée et lui donne un nom spécifique.

Nom_variable = valeur

- Par exemple, La variable **spam** stocke maintenant le nombre 5:

```
spam = 5
```

- **Remarque** : ne spécifier pas le type (s'adapte automatiquement)

Pr. Amina BENGAG 21

1.1 Les types des variables :

1. **Entier (int) :**
   - ✓ il s'agit des variables destinées à contenir un nombre entier positif ou négatif. (sans la virgule)
2. **Reel (float):**
   - ✓ il s'agit des variables numériques qui ne sont pas des entiers, c'est à dire qui comportent des décimales et qui sont des valeurs approximatives. (**4,1** - **18** - **-14**)
3. **Chaines de Caractère (str):**
   - ✓ Les variables de type caractère contiennent des caractères alphabétiques ou numériques (de 0 à 9). (**'A'** , **'2'** , **'?'** …)
4. **Booléen:**
   - ✓ Contient soit la valeur 'True' ou 'False'

Pr. Amina BENGAG 22

Les éléments de base

Exercice :
Définissez la variable **my_var** égale à la valeur 10. Puis afficher le contenu de ce variable.

```
# Écrivez votre code ci-dessous!
my_var = 10
print " la valeur de mon variable =",
my_var
```

```
la valeur de mon variable =  10
```


Les éléments de base

I. Structure d'un programme en Python

1. Variable

a. Exemple

```
Num_etudiant = 52

Pi = 3.14

Note = 17.5

Prenom = 'Salma'

Resultat1 = True

Resultat2 = False
```


Les éléments de base

Quelque règles de nom de variable

- Le nom de la variable **ne doit pas commencer** par un **nombre**
- Éviter les **mots réservés** par Python (print, def, and, for …)
- Python est **sensible** à la **casse**

Correct	Incorrect	Raison
Varible	Nom de variable	Comporte des espaces
Non_de_variable	52 Non_de_variable	Commence par un chiffre
non_de_variable	h5@site.fr	Caractère spécial @
_variable_1	Non-de-variable	Signe – interdit
_2136457895421	**True**	Nom réservé (mot clé)

Les éléments de base

Quelque règles de nom de variable

2. Mots réservés

and	del	from	none	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	false	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Les éléments de base

2. Affectation

- L'instruction suivante est une affectation :

```
spam = 5
```

- On peut affecter à une variable le contenu d'une autre variable

```
spam = 5
ping = 9
spam = ping
```

- On peut affecter des valeurs à plusieurs variables sur une seule ligne

```
spam = spam + 1
```


Les éléments de base

2. Affectation

- Affectation multiple :

```
>>> x = y = 3
>>> x
3
>>> y
3
```

- *Affectations parallèles*

```
>>> a, b = 4, 8.33
>>> a
4
>>> b
8.33
```


Les éléments de base

2. Affectation

- Exemple d'affectation

```
1. my_int = 7
2. my_int = 3
3. print " la valeur de my_int", my_int
```

```
1. # Écrivez votre code ci-dessous!
2. A = 10
3. B = A + 3
4. A = 3
5. Print  " A = ", A
6. print " B = ", B
```

→

```
A = 3
B = 13
```


Les éléments de base

2. Affectation

- Exemple d'affectation

```
A = 5
B = 3
C = A + B
A = 3
B = C – 2
print  " A = ", A
print " B = ", B
print " C = ", C
```

→

```
A = 3
B = 6
C = 8
```

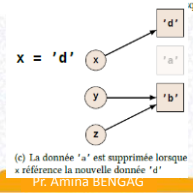
Les éléments de base

2. Affectation

- Variable, valeur & adresse

(a) Trois affectations

(b) La donnée 'c' est supprimée que z référence y



(c) La donnée 'a' est supprimée lorsque x référence la nouvelle donnée 'd'

Les éléments de base

2. Affectation

Exercice :
Écrire un programme permettant d'échanger les valeurs de deux variables A et B, et ce quel que soit leur contenu préalable.

```
A = 5
B = 6
C = A
A = B
B = C
Print " A = ", A
print " B = ", B
print " C = ", C
```

```
A = 6
B = 5
```


Les éléments de base

2. Affectation

Exercice :
Écrire un programme permettant d'échanger les valeurs de deux variables A et B, et ce quel que soit leur contenu préalable.

```
>>> a = 5
>>> b = 32
>>> a,b = b,a # permutation
>>> a
32
>>> b
5
>>>
```


Les éléments de base

3. Affichage

- ✓ Print() ⟺ Afficher la valeur d'une **expression** sur l'écran
- ✓ Une **expression** peut être :
  - ✓ Une **chaine de caractères** entre deux apostrophes ' '
  - ✓ Un **nombre**, une valeur d'un **variable**, un **résultat** d'une opération.

Les éléments de base

3. Affichage

✓ syntaxe

print (''Salam Alaykum'') ➔ afficher Salam Alaykum

print (n) ➔ afficher le **contenu** de la variable **n**

print (5) ➔ afficher le nombre **5**

print ('la valeur de n est : ' , n)
➔
Afficher :
- le message **la valeur de n est :**
- et le **contenu** de la variable **n**

Les éléments de base

3. Affichage

✓ Print() ⟺ Afficher une information

```
# print
print("Bonjour")
prenom = "Julien"

print("Bonjour", prenom, "comment tu vas")

# format
print("Bonjour {} comment tu vas".format(prenom))

# f-string
print(f"Bonjour {prenom} comment tu vas")
```


Les éléments de base

4. Lecture

✓ **Input** ⟺ Lire une valeur entrer au clavier et la stockée dans une variable

✓ **syntaxe**

Variable = input(''message facultatif'')

✓ **Remarque** : La fonction Input() retourne par défaut une valeur de type chaîne de caractères

Entier = int(input(''message facultatif''))

Reel = float(input(''message facultatif''))

Les éléments de base

4. Lecture

Exemple :

```
>>> # Test de la fonction input
>>> annee = input("Saisissez une année : ")
Saisissez une année : 2009
>>> print(annee)
'2009'
>>>
```

```
>>> type(annee)
<type 'str'>
```

```
>>> # On veut convertir la variable en un entier, on utilise
>>> # donc la fonction int qui prend en paramètre la variable
>>> # d'origine
>>> annee = int(annee)
>>> type(annee)
<type 'int'>
>>> print(annee)
2009
>>>
```

annee= int(input(''saisissez une année''))

Les éléments de base

5. Expressions

✓ Expression arithmétiques

Opérateur	Signification
+ , -	Addition , Soustraction
* , /	Multiplication , Division
//	Division entière
%	Reste de la division entière
**	Puissance

Les éléments de base

5. Expressions

✓ Les opérateurs arithmétiques composés

Opérateur	Opération normale	Opération composé
+=	X = X + Y	X += Y
-=	X = X − Y	X -= Y
*=	X = X * Y	X *= Y
/=	X = X / Y	X /= Y
%=	X = X % Y	X %= Y
//=	X = X // Y	X //= Y
**=	X = X % Y	X **= Y

Les éléments de base

5. Expressions

✓ Expressions de comparaisons

✓ Une expression de comparaison donne un résultat booléen (vrai ou faux). Les opérateurs de comparaison usuels sont : >, <, ==, >=, <=, !=.

✓ Exemple:

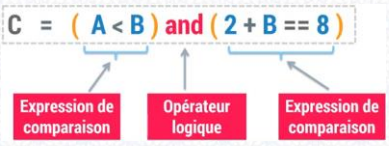
Les éléments de base

5. Expressions

✓ Expressions logiques

✓ Une expression logique est composé de
- ✓ Expressions de comparaisons
- ✓ Opérateur logiques (and, or, not)

✓ Exemple:



Programmation Python

Chapitre 3 : Structure de contrôle

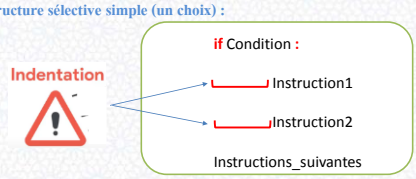
Structure de contrôle

I. La structure sélective :

La structure sélective est une structure dont les instructions sont exécutées selon les réponses des conditions.

1) La structure sélective simple (un choix) :

Syntaxe :



- Si la condition vaut Vrai alors le bloc d'instructions sera exécuté, sinon il sera ignoré.

Exemple : écrire un programme en python qui affiche le maximum de deux nombres réels.

Structure de contrôle

2) La structure alternative (deux choix) :

Syntaxe :

```
if Condition :
        Instructions1
else :
        Instructions2
```

- Si la condition vaut Vrai alors le bloc d'instructions1 sera exécuté, et le bloc d'instructions2 sera ignoré, sinon le bloc d'instructions2 sera exécuté et le bloc d'instructions1 sera ignoré.

- **Exemple :** Un programme qui demande un nombre entier à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif

Structure de contrôle

2) La structure alternative (deux choix) :

- **Exemple :** Un programme qui demande un nombre entier à l'utilisateur, et l'informe ensuite si ce nombre est positif ou négatif

```
print ( " Programme qui détermine la nature d'un nombre : " )
n = int ( input ( " Veuillez entrer un nombre : " ) )
if n > 0 :
        print ( " Ce nombre est positif " )
else :
        print ( " Ce nombre est négatif " )
```

Structure de contrôle

3) La structure à choix multiple:

Syntaxe :

```
if Condition1:
        Instructions1
elif condition2:
        Instructions2
else:
        Instructions3
Instructions_suivantes..
```

- **Exemple :** Un programme qui demande un nombre entier à l'utilisateur, et l'informe ensuite si ce nombre est positif, négatif ou nul

Structure de contrôle

3) La structure à choix multiple:

- **Exemple :** Un programme qui demande un nombre entier à l'utilisateur, et l'informe ensuite si ce nombre est positif, négatif ou nul

```
print ( " Programme qui détermine la nature d'un nombre : " )
n = int ( input ( " Veuillez entrer un nombre : " ) )
if n > 0 :
        print ( " Ce nombre est positif " )
elif n < 0 :
        print ( " Ce nombre est négatif " )
else :
        print ( " Ce nombre est nul " )
```


Structure de contrôle

3) La structure imbriquée :

Syntaxe :

```
if Condition1:
        Instructions1
else:
        if condition2 :
        Instructions2
        else:
        Instructions3
Instructions_suivantes...
```

- **Exemple :** Un programme qui demande un nombre entier à l'utilisateur, et l'informe ensuite si ce nombre est positif, négatif ou nul

Structure de contrôle

3) La structure imbriquée :

- **Exemple :** Un programme qui demande un nombre entier à l'utilisateur, et l'informe ensuite si ce nombre est positif, négatif ou nul

```
print ( " Programme qui détermine la nature d'un nombre : " )
n = int ( input ( " Veuillez entrer un nombre : " ) )
if n > 0 :
        print ( " Ce nombre est positif " )
else :
        if n < 0 :
                print ( " Ce nombre est négatif " )
        else :
                print ( " Ce nombre est nul " )
```

Structure de contrôle

4) Opérateur ternaire:

✓ **Permet de simplifier la syntaxe de la structure conditionnelle.**

Syntaxe :

```
Valeur_si_vrai if condition else valeur_si faux
```

Exemple :

```
A if A>B else B
```


Programmation Python

Chapitre 4: La structure répétitive (les itérations ou les boucles)

La structure répétitive

I. La structure répétitive :

Un programme a presque toujours pour rôle de répéter la même action un certain nombre de fois. Pour ce faire, on utilise une structure permettant de dire «Exécuter telles actions jusqu'à ce que telle condition soit remplie ».

En Python, nous allons voir deux types de boucles:

- **la boucle while ,**
- **la boucle for.**

La structure répétitive

1.1. La boucle while

Syntaxe :

```
while condition :
    Instructions 1
    Instructions 2
    ...
    Instructions n
```

Ce qui signifie → tant que la condition est vraie, on exécute les instructions.

Exercice : écrire un programme en python qui

- Demande à l'utilisateur de deviner un nombre secret entre 0-9
- Le nombre de tentative est 3

La structure répétitive

1.1. La boucle while

Solution :

```
secret = 5
tentative = 3

while tentative !=0:
    dev= int(input("entrer le nbr secret"))
    tentative = tentative - 1
    if dev == secret :
        print("bravo")
    else:
        print(f"il te reste {tentative} tentative")
```


La structure répétitive

1.1. La boucle while

Solution (le mot clé break):

```
secret = 5
tentative = 3

while tentative !=0:
    dev= int(input("entrer le nbr secret"))
    tentative = tentative - 1
    if dev == secret :
        print("bravo")
        break
    else:
        print(f"il te reste {tentative} tentative")
```


La structure répétitive

I. La structure répétitive :

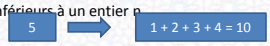
1.1. La boucle while

Exercice 1 :

- ✓ Écrire un programme en python qui demande à l'utilisateur de taper le nombre 26.
- ✓ Tant qu'il n'a pas tapé ce nombre, on lui redemande le nombre.
- ✓ Le programme ne pourra s'arrêter que si l'utilisateur tape le nombre 26.

Exercice 2 :

- ✓ Écrire un programme en python qui va permettre de calculer la somme des nombres entiers Inférieurs à un entier n



La structure répétitive

1.2. La Boucle for

- ✓ L'instruction **for** travaille sur des **séquences**. Elle permet d'executer une suite d'instructions un nombre de fois connu et fixé à l'avance.
- ✓ **Séquence** : une chaîne de caractère, liste, dictionnaire, tuples…

Syntaxe :

```
for element in sequence :

        liste d'Instructions
Instructions_suivantes
```

- **element** : une variable créer par **for**. Elle prend successivement chacune des valeurs figurant dans la **séquence** parcourue.

La structure répétitive

1.2. La Boucle for

Exemple :

```
mot = "python"
for i in mot :
    print(i)
```

```
p
y
t
h
o
n
>
```


La structure répétitive

1.2. La Boucle for

Exemple :

- *Demander une chaine de caractère*
- *Afficher la lettre si la lettre est une voyelle*
- *Afficher * si la lettre est une consonne*

```
chaine = input("entrer une chaine de caractère ")

for i in chaine :
    if i in 'aeuioyAEUIOY':
        print(i)
    else:
        print("*")
```


La structure répétitive

1.2. La Boucle for (en précisant le nombre de répétition)

la fonction range():

- range (N)

0	1	2	3	....	n-1

- Range (N,M)

n	N+1	N+2	N+3	...	M-1

- Range (N, M, pas)

N	N+pas	N+2pas	N+3pas	...

La structure répétitive

1.2. La Boucle for

Exemple :

Un programme qui affiche les nombre de 1 à 10.

```
for i in range(10):
    print(i+1)
```

La structure répétitive

1.2. La Boucle for

Exemple :

Un programme qui affiche la table de multiplication de 6.

```
for i in range(11):
    M = 6 * i
    print(f"6 * {i} = {M}")
```


La structure répétitive

1.3. Les boucles imbriquées

- L'imbrication de boucle permet d'écrire une boucle dans le bloc d'instructions d'une autre boucle
- **Syntaxe :**

```
for i in seqeunce1 :
    for j in seqeunce2 :
        Instruction_1
        Instruction_2
        ...
Instructions_suivantes
```


La structure répétitive

1.3. Les boucles imbriquées

Exemple :

Un programme qui affiche la table de multiplication de 1 à 10.

```
for i in range(1,11):
    for j in range(1,11):
        M = i * j
        print(f"{i} * {j} = {M}")
```


La structure répétitive

le mot clé continue

Permet d'ignorer l'itération actuelle de la boucle et de passer à l'itération suivante.

- **Exemple**

Ecrire un programme qui calcule la somme d'un maximum de 8 nombres entrés par l'utilisateur, si un nombre négatif est entré, la boucle ignore ce nombre.

La structure répétitive

le mot clé **continue**

solution

```
somme = 0

for i in range(1,9):
    n = int(input("entrer un nombre positive"))
    if n<0:
        continue
    somme = somme + n
print(f"la somme est : {somme}")
```


Programmation Python

Chapitre 5: Les fonctions

Les fonctions

I. **Fonctions**

1. **Définition**

- ✓ **Une suite d'instructions regroupées sous un non pour faire une action précise;**
- ✓ **Elle prend en entrée des paramètres (arguments) et retourne un résultat;**
- ✓ **Cette fonction pourra être appelé par son nom autant de fois que nécessaire**

2. **Syntaxe**

```
def nom_fonction(argument1, argument2,...):
    instruction1
    instruction2
    ...
    return resultat
```


Les fonctions

I. **Fonctions**

2. **Syntaxe**

```
def nom_fonction(argument1, argument2,...):
    instruction1
    instruction2
    ...
    return resultat
```

Avec retour de résultat et avec arguments

```
def nom_fonction():
    instruction1
    instruction2
    ...
    return resultat
```

Avec retour de résultat et pas d'arguments

```
def nom_fonction(argument1, argument2, ...):
    instruction1
    instruction2
    ...
```

Avec arguments et pas de retour

```
def nom_fonction():
    instruction1
    instruction2
    ...
```

Pas de retour et pas d'arguments

Les fonctions

I. Fonctions

3. Appel d'une fonction

```
...
valeur-retour = nom_fonction(argument1,argument2,...)
print(nom_fonction(argument1,argument2,...))
x = x + nom_fonction(argument1,argument2,...)
...
```

Avec retour de résultat et avec arguments

```
...
valeur-retour = nom_fonction()
print(nom_fonction())
x = x + nom_fonction()
...
```

Avec retour de résultat et pas d'arguments

```
...
nom_fonction(argument1, argument2)
...
```

Avec arguments et pas de retour

```
...
nom_fonction()
...
```

Pas de retour et pas d'arguments

Les fonctions

I. Fonctions

3. Exemple (une fonction qui calcule la somme)

```
def somme (a,b):
    c = a + b
    print (" a + b = ", c)
a = int (input (" Veuillez entrer la valeur de a : "))
b = int (input (" Veuillez entrer la valeur de b : "))
somme (a,b)
```

```
def somme (a,b):
    c = a + b
    return c
a = int (input (" Veuillez entrer la valeur de a : "))
b = int (input (" Veuillez entrer la valeur de b : "))
print (" a + b = ", somme (a,b))
```


Les fonctions

I. Fonctions

4. Variable globale et locale

```
def afficher():
    x = 5
    print(x)
afficher()
print(x)
```

#erreur car x est une variable locale

```
y = 10
def afficher():
    x = y
    print(x)
afficher()
print(y)
```

#affiche 10

#pas d'erreur y est une variable globale
#affiche 10

Les fonctions

I. Fonctions

4. Le mot clé global

```
y = 10
def afficher():
    global y
    y = 8
    print(y)
afficher()
print(y)
```

#affiche 8

#affiche 8

Les fonctions

I. Fonctions

4. Typages des arguments

On peut appeler une fonction en utilisant l'un des types suivants:

- Arguments requis
- Arguments par défaut
- Arguments avec étiquettes (mot-clé)

Les fonctions

I. Fonctions

4. Typages des arguments

On peut appeler une fonction en utilisant l'un des types suivants:

- Arguments par défaut

```
def soustraction( A , B = 0 ):
    C = A - B
    print ( " A – B = " , C )
A = float ( input ( " Saisir la valeur de A : " ) )
B = float ( input ( " Saisir la valeur de B : " ) )
soustraction( A )
```

```
def soustraction( A = 1 , B = 0 ):
    C = A - B
    print ( " A – B = " , C )
A = float ( input ( " Saisir la valeur de A : " ) )
B = float ( input ( " Saisir la valeur de B : " ) )
soustraction( )
```


Les fonctions

I. Fonctions

4. Typages des arguments

On peut appeler une fonction en utilisant l'un des types suivants:

- Arguments avec étiquettes (mot-clé)

```
def soustraction( A , B ):
    C = A - B
    print ( " A – B = " , C )
A = float ( i                A : " ) )
B = float ( i                B : " ) )
soustraction( A = 10 , B = 5 )
```


Les fonctions

I. Fonctions

5. lambda

✓ Python nous propose un autre moyen de créer des fonctions, des fonctions extrêmement courtes (limitées à une seule instruction).

✓ Syntaxe:

```
lambda arg1,arg2,... : instruction de retour
```

✓ Exemple :

```
f = lambda x : x * x
f(3)
>>> 9
```

Programmation Python

Chapitre 6: Modules et packages

Modules et packages

I. Modules

1. Définition

✓ Ce sont des programmes Python qui regroupent plusieurs **fonctions et variables** ayant un **rapport entre elles**.

✓ Exemple : Toute les **fonctions mathématiques** peuvent être placées dans un **module dédié aux math**.

2. Importation des modules

```
import nom_module
import nom_module as mod
```

3. Appeler une fonction du module

```
nom_module.nom_fonction()
```

4. Importer une fonction

```
from nom_module import nom_fonction
from nom_module import *
```


Modules et packages

I. Modules

1. Nos propres modules (exemples)

✓ Exemple : création d'une calculatrice

```
def somme( A , B ) :
    C = A + B
    print ( " A + B = " , C )
def soustraction( A , B ) :
    C = A - B
    print ( " A – B = " , C )
def multiplication( A , B ) :
    C = A * B
    print ( " A * B = " , C )
```

```
def division( A , B ) :
    if B != 0 :
        print ( " A / B = " , A / B )
    else :
        print ( " Impossible " )
somme( 50 , 10 )
soustraction( 15 , 35 )
multiplication( 7 , 3 )
division( 4 , 2 )
```


Modules et packages

I. Modules

1. Nos propres modules (exemples)

✓ Exemple : création d'une calculatrice (problème1)

- ✓ Somme
- ✓ Soustraction
- ✓ Multiplication
- ✓ Division
- ✓ Puissance
- ✓ Factoriel
- ✓ ……

→ Compliqué et long

Modules et packages

I. Modules

1. Nos propres modules (exemples)

- ✓ Exemple : création d'une calculatrice (problème2)
  - ✓ Programme1 (Somme, Soustraction)
  - ✓ Programme2 (Multiplication, Division, Puissance)
  - ✓ Programme3 (Factoriel, Somme, Soustraction, Puissance)
  - ✓ ……

Duplication

Modules et packages

I. Modules

1. Nos propres modules (exemple)

- ✓ Exemple : création d'une calculatrice (solution)
  - ✓ Créer deux fichier python :
    - ✓ Fonction.py : contient l'implémentation des fonctions
    - ✓ Calculatrice.py : utiliser pour appeler les fonctions

```
from fonctions import *
somme( 50 , 10 )
soustraction( 15 , 35 )
multiplication( 7 , 3 )
division( 4 , 2 )
```

Fonction.py

```
def somme( A , B ) :
    C = A + B
    print ( " A + B = " , C )
def soustraction( A , B ) :
    C = A - B
    print ( " A - B = " , C )
def multiplication( A , B ) :
    C = A * B
    print ( " A * B = " , C )
def division( A , B ) :
    if B != 0 :
        print ( " A / B = ", A / B )
    else :
        print ( " Impossible " )
```

Calculatrice.py

Modules et packages

II. Packages (paquets)

1. Définition

- ✓ C'est un ensemble de plusieurs **modules** regroupés dans le même **dossier**
- ✓ I**mporté** et utilisé de la même manière que les **modules**
- ✓ Doit contenir un fichier nommé **_ _init_ _.py** pour que python le considère comme un **paquet**. Ce fichier peut être **vide**.

Modules et packages

III. Écosystème (les modules disponibles)

1. Définition

- ✓ **Modules standards (déjà installé avec python)**
  - ✓ **math**
  - ✓ **random**
  - ✓ **Time**
  - ✓ **…**
- ✓ **Modules à télécharger**

Programmation Python

Chapitre 4: Les listes[,], les tuples(,), les dictionnaires{:}

Les listes

I. Les listes

- ✓ Les éléments de la liste sont placés entre deux **crochets []** et séparés par une **virgule**.
- ✓ Les éléments d'une **liste sont ordonnés**. Chaque **élément** a une valeur **d'index unique**. Un nouvel élément serait ajouté à la fin de la liste.
- ✓ N'a pas de taille fixe. Les éléments de la liste peuvent être **modifiés**. Nous pouvons **ajouter**, **modifier** ou **supprimer** des éléments d'une liste après sa création.
- ✓ Une liste peut contenir **différent types d'éléments** (chaine de caractères, entier, réel ou booléen)

Les listes

I. Les listes

- ✓ Exemple:
  - ✓ Fruit = ['pomme', 'banane', 'ananas', 'orange']

    0 1 2 3
  - ✓ Liste1 = ['pomme', '**ananas**', True, 10, 13, '**ananas**']

    0 1 2 3 4 5
  - ✓ Liste1 = ['pomme', True, 10, 13, 15, 'ananas']

    0 1 2 3 4 5

Les listes

I. Les listes

- ✓ Syntaxe :
  - ✓ Simple

    `nomListe = [ val1, val2, val3, val4, ...]`
  - ✓ Fonction liste()

    `nomListe = liste(( val1, val2, val3, val4, ...))`
  - ✓ Fonction rang()

    `nomListe = liste(range(début, fin, pas))`
  - ✓ Exemple `L1 = liste(range(5, 15))`

Les listes

I. Les listes

✓ Affichage et accès aux éléments d'une liste

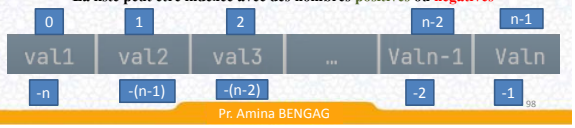
✓ Pour accéder ou afficher une liste

```
print (nom_liste)
```

✓ Pour accéder à un seul élément d'une liste, il faut préciser son index

```
nom_liste [indexe]  = valeur
print (nom_liste [index])
```

✓ La liste peut être indexée avec des nombres positives ou négatives



Les listes

I. Les listes

✓ Accès aux éléments d'une liste - Découpage

✓ Le technique de découpage permet d'extraire une plage spécifique d'élément d'une liste

```
nom_liste [indice_début, indice_fin, pas]
```

✓ Les nombres indice_début, indice_fin et pas son facultatifs

✓ L'indice_début est inclus et l'indice_fin est exclu du résultat

✓ Cette technique crée une nouvelle liste dont les éléments sont copiés de la liste d'origine.

Les listes

I. Les listes

✓ Accès aux éléments d'une liste - Découpage

✓ Exemple


```
# Affichage des quatre premiers éléments
print(L[0],L[1],L[2],L[3])
print(L[0:4])

# Affichage des éléments de la position 6
jusqu'à la fin
print(L[6],L[7],…,L[11],L[12])
print(L[6:13])
```


Les listes

I. Les listes

✓ Opération sur les listes

✓ L'opérateur + : utilisé pour la concaténation entre différentes listes

✓ Syntaxe :

```
L = L1 + L2
```

✓ L'opérateur * : utilisé pour multiplier une liste n fois

✓ Syntaxe :

```
L = L1 * 3
```

✓ Les opérateurs de comparaison (==, !=, <=,…) : utilisé pour comparer deux listes

✓ Syntaxe :

```
Résultat = L1 == L2
```

Les listes

I. Les listes

✓ Opération sur les listes

✓ L'affectation multiple : permet d'affecter des éléments d'une liste à plusieurs variables dans une seule ligne de code

✓ Syntaxe :

```
# Création de la liste
L = [ele1, ele2, ele3]

# Affectation des éléments aux variables
var1, var2, var3 = L
```

var1	ele1
var2	ele2
var3	ele3

L [n : m : pas] = [ele1, ele2, ele3,…]

Les listes

I. Les listes

✓ Opération sur les listes

✓ L'affectation multiple – Découpage

✓ Python permet également d'affecter un élément ou une liste d'éléments à une partie d'une liste

✓ Syntaxe

L [n : m : pas] = [ele1, ele2, ele3,…]

✓ Exemple

L1 :

'PYTHON'	20	'css'	14	'HTML'

L1 [: 5 : 3] = ['Java' , 12]

'JAVA'	20	'css'	12	'HTML'

Les listes

I. Les listes

✓ Méthodes et fonctions

Création	Ajout	Suppression	Opérations	Recherche	Duplication
List()	append()	remove()	len()	index()	copy()
	insert()	pop()	sum()		
	extend()	clear()	max()		
		del	min()		
			count()		
			reverse()		
			sort()		
			sorted()		

Les listes

I. Les listes

✓ Méthodes vs fonctions

✓ Une **fonction** est indépendante et appelée directement par son nom

✓ Syntaxe :

nomFonction (par1, par2, par3, …)

✓ Une méthode ne peut exister sans un objet qui l'appelle

✓ Syntaxe :

NomObjet.nomMéthode (par1, par2, par3, …)

Les listes

I. Les listes

- Fonction : list()
- Cette fonction permet de créer une liste des éléments

```
nomListe = list ( (ele1, ele2, ele3, ele4,...)
nomListe = liste( range(début, départ, pas ))
```

- Méthode : append()
- Cette fonction permet d'ajouter un élément à la fin d'une liste

```
nomListe.append( élément )
```

- Méthode : insert()
- Cette fonction permet d'ajouter un élément à un index spécifié dans une liste

```
nomListe.insert ( index, élément )
```


Les listes

I. Les listes

- Méthode : extend()
- Cette méthode permet d'ajouter les éléments d'une séquence (liste, tuple,…) à la fin d'une liste

```
nomListe.extend ( seq )
```

- Méthode : remove()
- Cette méthode supprime la 1ère occurrence de l'élément spécifié, si elle est présente. Sinon erreur.

```
nomListe.remove( élément )
```

- Méthode : pop()
- Cette méthode supprime un élément à un index spécifié et de le retourner comme valeur de retour. Si aucun index n'est spécifié, la méthode retourne et supprime le dernier élément

```
nomListe.pop( index )
```


Les listes

I. Les listes

- Méthode : clear()
- Cette méthode permet de supprimer tous les éléments d'une liste

```
nomListe.clear ()
```

- Le mot clé : del
- Utilisé pour supprimer un élément situé à l'index indiqué / supprimer une plage d'éléments d'une liste

```
del nomListe [ index ]
del nomListe [ plage ]
```


Les listes

I. Les listes

- Fonction : len()
- Cette fonction renvoi le nombre d'élément d'une liste

```
len( nomListe )
```

- Fonction : sum()
- Cette fonction renvoi la somme de tous élément d'une liste

```
sum( nomListe )
```

- Fonctions : max() et min()
- max : renvoi la plus grande valeur d'une liste

```
max ( nomListe )
```

- Min : renvoi la plus petite valeur d'une liste

```
min( nomListe )
```

Les listes

I. Les listes

✓ Méthode : count()

✓ Cette méthode permet de déterminer combien de fois un élément spécifié apparait dans la liste.

```
nomListe.count ( élément )
```

✓ Méthode : reverse()

✓ Cette méthode permet d'inverser l'ordre des éléments d'une liste

```
nomListe.reverse()
```


Les listes

I. Les listes

✓ Méthode : sort()

✓ Cette méthode permet de trier les éléments d'une liste.

```
nomListe.sort ( key = ..., reverse = ...)
```

✓ Key : un paramètre pour spécifier les critères de tri (optionnel)

✓ Reverse : si la valeur de ce paramètre est True, la liste sera triée par ordre décroissant (optionnel, par défaut : False)

Les listes

I. Les listes

✓ Méthode : sort()

✓ Exemple:

1. Créer la liste L1.

2. Trier les éléments de la liste par ordre croissant puis par ordre décroissant.

L1	'PYTHON'	'pHP'	'css'	C++	'HTML'

Les listes

I. Les listes

✓ Méthode : sort()

```
l1 = ['PYTHON', 'pHP', 'css', 'C++', 'html']

l1.sort()
print(l1)
#['C++', 'PYTHON', 'css', 'html', 'pHP']
l1.sort(reverse=True)
print(l1)
#['pHP', 'html', 'css', 'PYTHON', 'C++']
l1.sort(key = str.upper)
print(l1)
#['C++', 'css', 'html', 'pHP', 'PYTHON']
l1.sort(key = str.lower, reverse=True)
print(l1)
#['PYTHON', 'pHP', 'html', 'css', 'C++']
```

Les listes

I. Les listes

✓ Fonction : sorted()

✓ Renvoie la liste triée, sans affecter la liste d'origine.

```
L1 = sorted ( listeOrigine, key = ..., reverse = ... )
```

✓ Méthode : index()

✓ Cette méthode recherche la 1ère occurrence de l'élément spécifié et renvoie son index

```
nomListe.index( élément, début, fin )
```

✓ Début et fin : des paramètres optionnels

✓ Si élément n'appartient pas à la lise : erreur

Les listes

I. Les listes

✓ Méthode : copy()

✓ Renvoie une copie de la liste spécifiée

✓ Syntaxe

```
L1 = listeOrigin.copy()
```

✓ Cette méthode est utile pour conserver les éléments de la liste d'origine avant de modifier la liste.

```
l1 = [1, 2, 3]
l2 = [1, 2, 3]

l2 = l1.copy()
print(l1, l2)
l1.append(15)
print(f"l1= {l1}, l2 = {l2}")
#l1= [1, 2, 3, 15], l2 = [1, 2, 3]
```


Les listes

I. Les listes

✓ Parcourir une liste – for

✓ Syntaxe

```
for element in nomListe :
    #traitement de l'élément
```

✓ Exemple

L1

'Python'	'hTML'	'CSS'	c++	'PHP'

```
for valeur in L1 :
    print( valeur )
```

→

```
Python
hTML
CSS
C++
PHP
```


Les listes

I. Les listes

✓ Parcourir une liste – for avec range()

✓ Syntaxe

```
for index in range( len( nomListe )) :
    #traitement de l'index et de nomListe[index]
```

✓ Exemple

L1

'Python'	'hTML'	'CSS'	c++	'PHP'

```
for index in range(len( L1) ) :
    print( index, L1[index])
```

→

```
0 Python
1 hTML
2 CSS
3 C++
4 PHP
```

Les listes

I. Les listes

✓ **Parcourir une liste – for avec enumerate()**

✓ **Syntaxe**

```
for index, valeur in enumerate( nomListe ) :
  #traitement de l'index et de nomListe[index]
```

✓ **Exemple**

L1

'Python'	'hTML'	'CSS'	c++	'PHP'

```
for i, valeur in enumerate( L1 ) :
    print( f'le langage {i+1} est {valeur}' )
```

→

```
Le langage 1 est Python
Le langage 2 est hTML
Le langage 3 est  CSS
Le langage 4 est  C++
Le langage 5 est  PHP
```


Les listes

I. Les listes

✓ **Parcourir une liste – for avec zip()**

✓ **Syntaxe**

```
for L1_element, L2_element,... in zip ( L1, L2,... ) :
  #traitement L1_element, L2_element, ...
```

✓ **La boucle s'arrêtera au dernier élément de la liste ayant le plus petit nombre d'éléments.**

Les listes

I. Les listes imbriquées

✓ **Syntaxe**

```
nomListe = [ [a1, a2, ..., an], b, c, [d1, d2, d3, ..., dm], ...]
```

✓ **Exemple:**

L = [['HTML', 'CSS', 'PHP'], [True, False], [10, 12, 20, 19], 'ensah']

Les dictionnaires

II. Les dictionnaires

✓ **Se base sur un système de clé et de valeur, on va avoir pour chaque valeur stockée dans le dictionnaire une clé associée qui permet d'accéder à cette valeur**

✓ **Les clés sont uniques**

✓ **Une clé peut être : une chaine de caractères, un entier, un reel,...**

✓ **Syntaxe**

```
nomDict = { clé1 : valeur1, clé2 : valeur2, clé3 : valeur3, ...}
nomDict = dict ()
```

✓ **Exemple:**

D = {'prenom' : 'Med', 'nom' : 'rahmani', 'profession' : 'ingenieur'}

Les dictionnaires

II. Les dictionnaires

```
D = {"prenom" : "Med", "nom" : "rahmani", "profession" : "ingenieur"}
print(D)
#{'prenom': 'Med', 'nom': 'rahmani', 'profession': 'ingenieur'}

print(D["prenom"])
# Med

D["age"] = 28
print(D)
#{'prenom': 'Med', 'nom': 'rahmani', 'profession': 'ingenieur', 'age':
    28}

for key in D :
    print(f"{key} : {D[key]}")
#prenom : Med
#nom : rahmani
#profession : ingenieur
#age : 28
```


Les dictionnaires

II. Les dictionnaires

- ✓ **Méthodes keys(), values()**
  - ✓ **Renvoient respectivement les clés et les valeurs d'un dictionnaire**
  - ✓ **Exemple**

```
D = {"prenom" : "Med", "nom" : "rahmani", "profession" : "ingenieur"}
print(D.keys())
#dict_keys(['prenom', 'nom', 'profession'])

print(D.values())
#dict_values(['Med', 'rahmani', 'ingenieur'])
```


Les dictionnaires

II. Les dictionnaires

- ✓ **Méthode : items()**
  - ✓ **Renvoie un objet de type dict_items**
  - ✓ **N'est pas indexable, mais il est itérable**

```
print(D.items())
#dict_items([('prenom', 'Med'), ('nom', 'rahmani'), ('profession',
    'ingenieur')])

print(type(D.items()))
#<class 'dict_items'>

for k, v in D.items():
    print(k, v)
#prenom Med
#nom rahmani
#profession ingenieur
```


Les dictionnaires

II. Les dictionnaires

- ✓ **Méthodes get ()**
  - ✓ **Renvoie la valeur associée à une clé spécifiée,**
  - ✓ **Ne renvoie pas d'erreur si la clè n'existe pas**

```
D = {"prenom" : "Med", "nom" : "rahmani", "profession" : "ingenieur"}
print(D.get("prenom"))

#Med

print(D.get("age"))

#None

print(D.get("age","cette clé n'existe pas"))
#cette clé n'existe pas
```

Les dictionnaires

II. Les dictionnaires

- ✓ Une liste de dictionnaires
  - ✓ À partir de plusieurs dictionnaires qui possèdent les mêmes clés, on peut créer une liste de dictionnaire
  - ✓ Exemple

```
D1 = {"nom":"girafe","poids":1100, "taille":5.2}
D2 = {"nom":"lion","poids":800, "taille":1.2}

animaux = [D1, D2]

for D in animaux:
    print(D["nom"])
```


Les dictionnaires

II. Les dictionnaires

- ✓ La fonction dict()
  - ✓ Permet de convertir l'argument en dictionnaire.
  - ✓ L'argument doit être un objet séquentiel (expl : liste, tuple) contenant d'autres objets séquentiels de 2 éléments.
    - ✓ Exemple : une liste de liste de 2 éléments

```
Liste = [["nom","girafe"],["poids",1100]]
Dict = dict(Liste)
for k in Dict:
    print(k, Dict[k])
```


Les tuples

III. Les tuples

- ✓ Un tuple est une liste non-modifiable.
- ✓ On utilise les parenthèses () au lieu des crochets []
- ✓ Si on veut ajouter un élément, il faut créer un nouveau tuple
- ✓ Les opérateurs + et * fonctionnent comme pour les listes (concaténation et duplication)
- ✓ Les fonctions sum(), max(), min(), sorted()…

Les tuples

II. Les tuples

- ✓ Exemples:

```
>>> t = (1, 2, 3)
>>> t
(1, 2, 3)
>>> type(t)
<class 'tuple'>
>>> t[2]
3
>>> t[1:3]
(2, 3)
>>> t[0:2]
(1, 2)
>>> t[2] = 22
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t = t + (22,)
>>> t
(1, 2, 3, 22)
```

Programmation Python

Chapitre 5: Programmation Orientée Objet (2)

terminologie

I. Terminologie

- ✓ **Classe**
  - ✓ Équivalente à un nouveau type de données.
  - ✓ Exemple : list, tuple et str. On peut les manipuler en utilisant des méthodes conçues pour chaque classe
- ✓ **Objet**
  - ✓ C'est un exemple particulier d'une classe.
  - ✓ Les objets ont généralement deux sortes d'attributs :
    - ✓ Les données (attributs)
    - ✓ Les méthodes : pour manipuler les attributs d'une classe
    - ✓ Exemple :
      - ✓ [1, 2, 3] est une instance de la classe liste.
      - ✓ [1, 2, 3].sort() : appel de la méthode sort().

Objet = attributs + méthodes

Classes et instanciation d'objet

II. Classes et instanciation d'objet

- ✓ **L'instruction Classe**
  - ✓ Permet d'introduire la définition d'une nouvelle classe (c'est-à-dire d'un nouveau type de données)
  - ✓ Syntaxe

```
class Nom_classe:
    att1 = val1
    att2 = val2
    .
    .
```

  - ✓ Exemple

```
class Point:
    x = 0
    y = 0
    .
```


Classes et instanciation d'objet

II. Classes et instanciation d'objet

- ✓ **L'instanciation et ses attributs**
  - ✓ Les classes sont des fabriques d'objets : on construit d'abord l'usine avant de produire des objets !
  - ✓ On instancie un objet (c'est-à-dire qu'on le produit à partir de l'usine) en appelant le nom de sa classe comme s'il s'agissait d'une fonction.
  - ✓ Syntaxe

```
class Point :
    x = 0
    y = 0

# pt1 est un objet de la classe Point (une instance)
pt1 = Point()

#affichage des attributs de l'instance pt1
print(pt1.x)
print(pt1.y)
```

Classes et instanciation d'objet

III. Méthodes

- ✓ **Une méthode s'écrit comme une fonction du corps de la classe avec un premier paramètre self obligatoire**
  - ✓ **self représente l'objet sur lequel la méthode sera appliquée. Autrement dit self est la référence d'instance**
- ✓ **Syntaxe**

```
class Point :
    x = 0
    y = 0
 #methode affiche()
    def affiche(self) :
        print(f"{self.x}, {self.y}")
```


Classes et instanciation d'objet

III. Méthodes

- ✓ **L'instanciation et ses attributs**
- ✓ **Exemple**

```
class Point :
    x = 0
    y = 0
 #methode affiche()
    def affiche(self) :
        print(f"{self.x}, {self.y}")

# pt1 est un objet de la classe Point (une instance)
pt1 = Point()

pt1.x = 12
pt1.affiche()

#affichage des attributs de l'instance pt1
print(pt1.x)
print(pt1.y)
```


Méthodes spéciales

IV. Méthodes spéciales

- ✓ **Elles servent à :**
  - ✓ **Initialiser l'objet instancier (__init__);**
  - ✓ **Surcharger (Changer le comportement) ses operateurs (+, -, /, …);**
  - ✓ **Modifier l'affichage de l'objet (__str__) (__repr__);**
  - ✓ **Accéder aux attributs (__getattr__) (__setattr__);**
  - ✓ **Destruction de l'objet (__del__):**
  - ✓ **…**
- ✓ **Ces méthodes portent des noms prédéfinis, précédés et suivis de deux caractères de soulignement (__methodeSpeciale__)**

Méthodes spéciales

IV. Méthodes spéciales

1. L'initialisateur

- ✓ **Lors de l'instanciation d'un objet, la structure de base de l'objet est créée en mémoire, et la méthode __init__ est automatiquement appelée pour initialiser l'objet.**
- ✓ **C'est typiquement dans cette méthode spéciale que sont créés les attributs d'instance avec leur valeur initiale.**
- ✓ **Exemple**

```
class Vecteur2D :
    def __init__(self,x0,y0):
        self.x = x0
        self.y = y0

v1 = Vecteur2D(2, 5)
print(v1.x)
```

Méthodes spéciales

✓ **Attribut d'objet VS attribut de classe**

```
class Etudiant :
    nbr_etudiant = 0 # attribut de classe pour compter le nbr
        d'étudiants crées
    def __init__(self, nom, cne):
        #à chaque fois qu'on crée un objet, on incrémente le compteur
            nbr_etudiant
        Etudiant.nbr_etudiant +=1
        self.nom = nom
        self.cne = cne

et1 = Etudiant("Rahmani", 2554)
et2 = Etudiant("Alaoui", 2597)

print(f"etudiant1 : {et1.nom}, {et1.cne} \n etudiant2 {et2.nom}, {et1
    .cne} \n nbr = {Etudiant.nbr_etudiant}")
```

Attribut d'objet

etudiant1 : Rahmani, 2554
etudiant2 Alaoui, 2597
nbr = 2

Méthodes spéciales

IV. Méthodes spéciales

2. Surcharge des opérateurs

✓ **La surcharge permet à un opérateur de posséder un sens différent suivant le type de ses opérandes.**

✓ **Soient deux instances, obj1 et obj2, les méthodes spéciales suivantes permettent d'effectuer les opérations arithmétiques courantes :**

Nom	Méthode spéciale	Utilisation
opposé	__neg__	-obj1
addition	__add__	obj1 + obj2
soustraction	__sub__	obj1 - obj2
multiplication	__mul__	obj1 * obj2
division	__div__	obj1 / obj2
division entière	__floordiv__	obj1 // obj2

Méthodes spéciales

IV. Méthodes spéciales

2. Surcharge des opérateurs

✓ **Exemple:**

```
class Vecteur2D :
    def __init__(self,x0,y0):
        self.x = x0
        self.y = y0
    def __add__(self,seconde):
        return Vecteur2D(self.x+seconde.x, self.y+seconde.y)
    def __str__(self):
        return f"Vecteur2D({self.x}, {self.y})"

v1 = Vecteur2D(2, 5)
v2 = Vecteur2D(3.2, 11.6)
print(v1 + v2)
```

Vecteur2D(5.2, 16.6)

Méthodes spéciales

IV. Méthodes spéciales

3. Cas particulier

✓ **On peut utiliser un objet dans le constructeur d'une autre classe conteneur**

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

class Segment :
    """Classe conteneur utilisant la classe Point."""
    def __init__(self, x1, y1, x2, y2) :
        self.orig = Point(x1, y1)
        self.extrem = Point(x2, y2)
    def __str__(self) :
        return (f"Segment : [({ self.orig.px}, { self.orig.py}), ({
            self.extrem.px}, { self.extrem.py})]")

s = Segment(1.0, 2.0, 3.0, 4.0)
print(s) # Segment : [(1, 2), (3, 4)]
```

Héritage et Polymorphisme

V. Héritage et Polymorphisme

1. L'héritage

- ✓ C'est le mécanisme qui permet de se servir d'une classe préexistante pour en créer une nouvelle qui possédera des fonctionnalités supplémentaires ou différentes.
- ✓ Il permet de réutiliser une classe existante:
  - ✓ En l'adaptant: ajouter de nouveaux attributs et méthodes à la nouvelles classes ou adapter (personnaliser) les attributs et les méthodes de la classes existante.
  - ✓ En la factorisant: plusieurs classes peuvent partager les mêmes attributs et les mêmes méthodes.

Héritage et Polymorphisme

V. Héritage et Polymorphisme

1. L'héritage

```
class Mere :
    def __init__(self, attr1, attr2):
        self.attr1 = attr1
        self.attr2 = attr2
    .
    .
class Fille(Mere) :
    def __init__(self, attr1, attr2, attr3):
        super().__init__(attr1,attr2)
        self.attr3 = attr3
    .
    .
```


Héritage et Polymorphisme

V. Héritage et Polymorphisme

2. Le polymorphisme

- ✓ Le polymorphisme par dérivation est la faculté pour deux méthodes (ou plus) portant le même nom mais appartenant à des classes héritées distinctes d'effectuer un travail différent.
  - ✓ Cette propriété est acquise par la technique de la surcharge.
- ✓ La dérivation décrit la création de sous-classes par spécialisation. Elle repose sur la relation « est-un ».

Héritage et Polymorphisme

V. Héritage et Polymorphisme

1. Polymorphisme

```
class Quadrupede :
    def piedsAuContactDuSol ( self ):
        return 4

class QuadrupedeDebout ( Quadrupede ) :
    def piedsAuContactDuSol ( self ):
        return 2

chat = Quadrupede()
print(chat.piedsAuContactDuSol())

homme = QuadrupedeDebout()
print(homme.piedsAuContactDuSol())
```

Héritage et Polymorphisme

V. Héritage et Polymorphisme

2. Le polymorphisme

✓ **Exemple : un Carre « est-un » Rectangle particulier pour lequel on appelle l'initialisateur de la classe mère avec les paramètres longueur=cote et largeur=cote :**

Le constructeur de la classe mère

Le constructeur de la classe fille

```
class Rectangle:
    #deux attributs longueurs et largeur
    def __init__(self, longueur=30, largeur=15):
        self.L, self.l = longueur, largeur
        self.nom = "rectangle"
    def __str__(self):
        return f"nom : {self.nom}"

class Carre(Rectangle): # héritage simple
"""Sous-classe spécialisée de la super-classe Rectangle."""
    def __init__(self, cote=20):
    # appel au constructeur de la super-classe de Carre :
        super().__init__(cote, cote)
        self.nom = "carré" # surcharge d'attribut

r = Rectangle()
c = Carre()
print(r)
print(c)
```


Héritage et Polymorphisme

VI. L'encapsulation

✓ **L'encapsulation est un principe qui consiste à cacher ou protéger certaines données de notre objet.**

✓ **Pour JAVA ou C++, vous n'avez pas le droit de faire, depuis l'extérieur de la classe : mon_objet.mon_attribut**

✓ **En python, les attributs sont par défaut publique**

✓ **Pour créer un attribut privé : __nomAttributPrivé**

✓ **En python, on peut protéger les attributs importants d'un objet en contrôlant le résultat demandé**

Héritage et Polymorphisme

VI. L'encapsulation

```
class Personne:
    def __init__(self, nom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.__lieu_residence = "Paris" # Notez le souligné _ devant le nom
    def _get_lieu_residence(self):
        """Méthode qui sera appelée lors d'accée en lecture à l'attribut 'lieu_residence'"""
        print("On accède à l'attribut lieu_residence !")
        return self.__lieu_residence
    def _set_lieu_residence(self, nouvelle_residence):
        """Méthode appelée lors d'accée en écriture 'lieu_residence'"""
        print(f"Attention, il semble que {self.nom} déménage à {nouvelle_residence}")
        self.__lieu_residence = nouvelle_residence
    # On va dire à Python que notre attribut lieu_residence pointe vers une propriété
    lieu_residence = property(_get_lieu_residence,_set_lieu_residence)
Rahmani = Personne("Rahmani")
print(Rahmani.nom)
print(Rahmani.lieu_residence)# appel du get_lieu_residence
Rahmani.lieu_residence = "oujda" #apple du set_lieu_residence
```



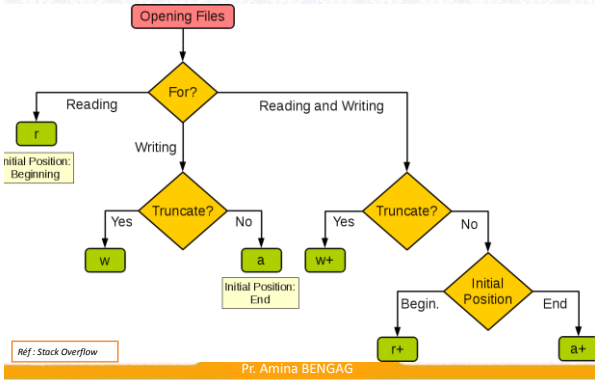
```
class Personne:
    def __init__(self, nom):
        """Constructeur de notre classe"""
        self.nom = nom
        self.__lieu_residence = "Paris" # Notez le souligné _ devant le nom

    def __getattr__(self, attribut):
        """sera appelé si ma classe n'a pas un attribut dont le nom est age"""
        print(f"Attention la classe n'a pas d'{attribut}")

    def __setattr__(self, attribut, val_att):
        """sera appelé si ma classe n'a pas un attribut dont le nom est age"""
        print(f"le setattr de la classe personne est appelé pour l'attribut {attribut}")
        # self.age  = val_att
        object.__setattr__(self, attribut, val_att)


    def get_lieu_residence(self):
        """Méthode qui sera appelée lors d'accée en lecture à l'attribut 'lieu_residence'"""
        print("On accède à l'attribut lieu_residence !")
        return self.__lieu_residence

    def set_lieu_residence(self, nouvelle_residence):
        """Méthode appelée lors d'accée en écriture 'lieu_residence'"""
        print(f"Attention, il semble que {self.nom} déménage à {nouvelle_residence}")
        self.__lieu_residence = nouvelle_residence
    # On va dire à Python que notre attribut lieu_residence pointe vers une propriété
    lieu_residence = property(get_lieu_residence,set_lieu_residence)
Rahmani = Personne("Rahmani")
print(Rahmani.nom)
print(Rahmani.lieu_residence)# appel du get_lieu_residence
Rahmani.lieu_residence = "oujda" #apple du set_lieu_residence
# Rahmani.age = 10
# print(Rahmani.age)
```

Programmation Python

Chapitre 6: Manipulation des fichiers

Manipulation des fichiers

I. Ouverture : Lecture / Écriture



Manipulation des fichiers

I. Ouverture : Lecture / Écriture

Mode	Read	Write	Create New File*	Truncate
r	Yes	No	No	No
w	No	Yes	Yes	Yes
a	No	Yes	Yes	No
r+	Yes	Yes	No	No
w+	Yes	Yes	Yes	Yes
a+	Yes	Yes	Yes	No

*Creates a new file if it doesn't exist.

Réf : Stack Overflow

Manipulation des fichiers

II. Ouverture

✓ **Syntaxe**

file = **open**([nom du fichier], [mode ouverture])

✓ **Le mode 'r'** : ouverture d'un fichier existant en lecture seule,

✓ **Le mode 'w'** : ouverture en écriture seule, écrasé s'il existe déjà et crée s'il n'existe pas,

✓ **Le mode 'a'** : ouverture et écriture en fin du fichier avec conservation du contenu existant

✓ **Le mode '+'** : ouverture en lecture et écriture

Manipulation des fichiers

II. Ouverture

1. Lecture

- ✓ Lecture totale avec la méthode read()
- ✓ Syntaxe

contenu = file.**read()**

- ✓ Exemple

```
f = open("myFile.txt", 'r')
contenu = f.read() # lecture du contenu
print(contenu) # impression du contenu
f.close() # fermeture du fichier
```


Manipulation des fichiers

II. Ouverture

1. Lecture

- ✓ Lecture partielle avec la méthode read(n)
- ✓ Syntaxe

contenu = file.**read(n)**

- ✓ Exemple

```
f = open("myFile.txt", 'r')
contenu = f.read(20) # lecture de 20 caractère du contenu du fichier
print(contenu) # impression du contenu
f.close() # fermeture du fichier
```


Manipulation des fichiers

II. Ouverture

1. Lecture

- ✓ Lecture séquentielle caractère par caractère en utilisant for avec la méthode read()
- ✓ Syntaxe

For c in file.**read()**

- ✓ La même opération peut être réalisée en utilisant la boucle while
- ✓ Exemple

```
f = open("myFile.txt", 'r')
s=""
for c in f.read():
    s = s + c
print(s)
```


Manipulation des fichiers

II. Ouverture

1. Lecture

- ✓ Lecture ligne par ligne avec la méthode readline()
- ✓ Syntaxe

file.**readline()**

- ✓ Exemple

```
f = open("myFile.txt", 'r')
print(f.readline()) # affiche la ligne n°1
print(f.readline()) # affiche la ligne n°2
```

Manipulation des fichiers

II. Ouverture

1. Lecture

✓ lecture de toutes les lignes en utilisant while avec readline()

✓ Exemple

```
f = open("myFile.txt", 'r')
s=""
while 1:
    ligne = f.readline()
    if(ligne == ""):
        break
    s = s + ligne
print(s) # impression de la totalité des lignes
```


Manipulation des fichiers

II. Ouverture

1. Lecture

✓ lecture ligne par ligne avec readlines()

✓ La méthode readlines() renvoie une liste dont les éléments sont les lignes du fichier

✓ Exemples

```
f = open("myFile.txt",'r')
content = f.readlines()
print(content[0]) # impression de la première ligne
print(content[1]) # impression de la deuxième ligne
```

```
f = open("myFile.txt",'r')
content = f.readlines()
for ligne in content:
    print(ligne)
```


Manipulation des fichiers

II. Ouverture

1. Lecture

✓ Lecture depuis une position donnée à l'aide de la méthode seek()

✓ Exemples

```
f = open("myFile.txt",'r')
f.seek(5) # sélection de la position 5
print(f.read()) #lire le fichier à partir de la 6 ème position
```


Manipulation des fichiers

II. Ouverture

2. Écriture

✓ Pour écrire dans un fichier existant, vous devez ajouter l'un des paramètres à la fonction open():

- ✓ "a" - Append - sera ajouté à la fin du fichier
- ✓ "w" - Write - écrasera tout contenu existant
- ✓ "r+" Lecture et écriture sans écraser le contenu existant

✓ Syntaxe

file.**write**(contenu)

Manipulation des fichiers

II. Ouverture

2. Écriture

✓ Exemple

```
# ouverture avec conservation du contenu existant
f = open ("myFile.txt", "a")
f.write ("Voici un contenu qui va s'ajouter au fichier sans écraser le contenu!")
f.close ()
# ouvrir et lire le fichier après l'ajout:
f = open ("myFile.txt", "r")
print (f.read())
```


Manipulation des fichiers

II. Ouverture

2. Écriture

✓ Exemple

```
# ouverture avec écrasement du contenu existant
f = open ("myFile.txt", "w")
f.write ("Désolé ! J'ai supprimé le contenu!")
f.close()
# ouvrir et lire le fichier après l'ajout:
f = open ("myFile.txt", "r")
print (f.read())
```


Manipulation des fichiers

II. Ouverture

2. Écriture

✓ Ajouter des lignes à un fichier en Python avec la méthode writelines (séquence)

✓ Exemple

```
f = open ("myFile.txt", "r+")
l = ["ligne1\n","ligne2\n,"ligne3\n"]
f.writelines(l)
f.close()
```