langage

python

Table des matières

I - Présentation du langage Python

1. Langage python

Python est un **langage** informatique **interprété** à usage généraliste , simple à apprendre et à utiliser, qui est essentiellement utilisé pour :

- **coder des petits scripts ou programmes simples**
- piloter/orchestrer des appels vers des fonctions prédéfinies efficaces (quelquefois codées en langage "C" de bas niveau et rapide)
- **piloter/orchestrer des calculs scientifiques**

Le langage python peut également être utilisé dans d'autres domaines (contrôles d'affichages graphiques, accès à des bases de données ou des fichiers, sites web, ...) sans cependant se démarquer d'autres langages informatiques (également bien adaptés pour effectuer efficacement ces tâches) .

1.1. Principales caractéristiques du langage python

simple	code facilement compréhensible et apprentissage rapide
interprété (et pas compilé)	comme les langages "basic" , "javascript" , "perl" , "ruby" , ... *avantages* : **souplesse et résultats immédiats** *inconvénients* : ***exécution pas rapide*** (si 100% python)
à usage général	on trouve des bibliothèques de fonctions prédéfinies pour presque tous les usages (calculs , accès bases de données , affichages , ...)
multi-paradigme	code procédural et séquentiel pour scripts simples code en mode "orienté objet" pour applications élaborées ...
mature et bien implanté	python existe depuis plus de 25 ans et est beaucoup utilisé dans le domaine scientifique et dans le cadre de l'administration système (ex : linux, ...)
open-source	accès et usage libre et gratuit . Seules quelques extensions "clefs en main" sont quelquefois payantes .
multi-plateformes	étant interprété, le langage python peut facilement être utilisé sur tout type de plateformes (Windows , Linux , Mac , smartphones, ...)
syntaxe avec indentations	contrairement à beaucoup d'autres langages qui délimitent des blocs de code via { et } ou via "begin" et "end" , le langage python n'utilise pas de délimiteur mais a une structure de code contrôlée par des indentations (décalages par rapport de débuts des lignes)

Bien qu'intéressant sur bien des points, le langage Python n'est pas interprété par les navigateurs "web/internet" . Les navigateurs "IE , Firefox , Chrome, ..." ont historiquement fait le choix d'utiliser le langage interprété javascript (bien aussi et un peu plus rapide que python) .

1.2. Historique et évolution

Première version publique officielle : 0.9.0 en **1991** développé par "Guido van Rossum" (Pays bas , Amsterdam).

De 1995 à 1999 : évolution du langage aux états-unis (CNRI, ...) , **version 1.6 en 1999**

A partir de **2001** et la version 1.6.1 , l'évolution du langage python est contrôlée par la "***Python Software Foundation***".

Python a longtemps existé en version 2.x (durant la décennie 2000-2010) .

Les version 3.x (à partir de 2008/2010) sont sur certains petits points en rupture avec les version 2.x. Il est donc conseillé aujourd'hui de ne plus utiliser l'ancienne version 2 (dans la mesure du possible)

La version **3.7** de python (datant de ***2018***) est une bonne version récente et stable du langage python.

1.3. Distributions de "python"

En tant que langage interprété , python peut être pris en charge par plusieurs variantes du moteur d'interprétation.

L'implémentation de référence est **CPython** (interpréteur codé en langage C) .
Il existe aussi **Jython** (basé sur une machine virtuelle java) , ...

En outre, on trouve aujourd'hui certaines distributions packagées de python incluant "moteur d'interprétation + éditeurs + ensemble de bibliothèques + ...) .

Les principales distributions (basées sur CPython) sont :

- **WinPython**
- **Annaconda**
- ...

La plupart des distributions sont à usage scientifique et elles incluent les bibliothèques de calculs ***numPy*** et ***sciPy*** .

Autrement dit, pour installer python sur un ordinateur on peut installer que python ou bien toute une distribution telle qu'anaconda .

II - Prise en main de l'interpréteur Python

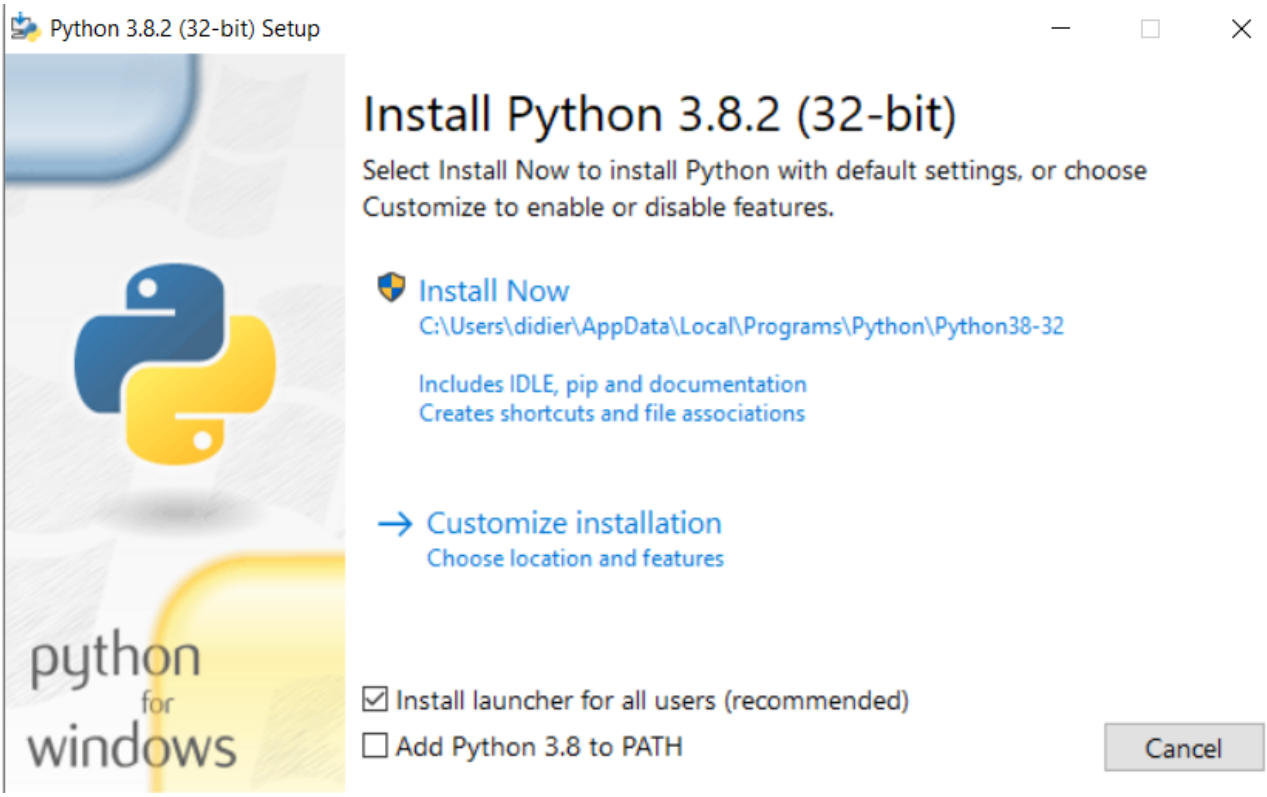
1. Installation et première utilisation de python

Avant de pouvoir utiliser le langage python il faut installer un des interpréteurs disponibles

1.1. Quelques installations possibles de python

Python est souvent installé d'office sur les distributions linux mais pas forcément dans une version récente . Par exemple sur **linux ubuntu 18.04** , le phython 3.6 pré-installé se lance avec la commande "**python3**" .

Pour installer python sur un ordinateur windows , on peut se connecter sur le site officiel **https://www.python.org/** et effectuer un téléchargement de l'installeur via l'url ***https://www.python.org/ftp/python/3.8.2/python-3.8.2.exe*** (environ 26 Mo) .



On peut éventuellement ajouter Python au PATH pour pouvoir ultérieurement le lancer facilement depuis une fenêtre CMD ou autre .

On peut également installer indirectement Python en installant toute une distribution telle qu'anaconda .

https://www.anaconda.com ,
https://repo.anaconda.com/archive/Anaconda3-2020.02-Windows-x86.exe ***(environ 478 Mo)***

Répertoire d'installation par défaut: *C:\ProgramData\Anaconda3*, éventuellement ajouté au PATH.

1.2. Prise en main de l'interpréteur python

Au sein d'une fenêtre de commande (CMD ou PowerShell ou shell linux ou ...) , la commande **python -V** permet d'afficher la version de l'interpréteur python (ex : 3.7.6) .

En lançant la commande "**python**" sans argument , on peut ainsi **lancer l'interpréteur python en mode interactif** . Celui ci nous invite alors à saisir des commandes après une invite (prompt) **">>>"** .
Toute "commande / ordre / expression" saisi(e) est alors immédiatement interprété(e) et le résultat s'affiche immédiatement .

```
python
Python 3.7.6 .....
>>> 2+3
5
>>> 4*6
24
>>>
```

1.3. Ecriture et lancement d'un programme python

Un fichier de code python a par convention l'extension "**.py**" et peut être saisi avec un très grand nombre d'éditeur (nodepad++, visual studio code , ...) .

Exemple: **hello.py**

```
a=2
b=3
c=a+b
print('Hello world')
print('c=',c)
```

Pour lancer ce script (petit programme) , on peut lancer la commande

python hello.py

Ce qui provoque l'affichage suivant :

Hello world

c= 5

Et pour automatiser un lancement depuis un double click via l'explorateur de fichiers de windows on peut éventuellement écrire un fichier de lancement tel que celui ci :

lancer_prog_python.bat

```
REM avec PATH contenant le répertoire d'installation de python
python hello.py
pause
```

1.4. Instructions élémentaires du langage python .

NB : En langage python les **commentaires** sont des **fin de lignes commençant par #**

p1.py

```
a=1  # nomVariable=valeur_a_affecter
print(a) # affiche la valeur de la variable a (ici 1)
a=2
print(a) # affiche la nouvelle valeur de la variable a (ici 2)
b=a*3+4  # variableResultat = expression d'un calcul
c=a+b
print("b=",b,"c=",c)  # affiche plusieurs choses en les séparant par des espaces
                      # affiche ici b= 10 c= 12
prenom = "alex"  # une chaine de caractères est délimitée en python par des " " ou des ' '
nom = 'Therieur'
nomComplet = prenom + ' ' + nom # concaténation (ajout bout à bout)
print(nomComplet) # affiche alex Therieur

# input('texte question') demande à saisir/renseigner une valeur
age = input ("quel est ton age ? ")
print ('age renseigné:' , age);  # affichera la valeur choisie/précisée .
```

Attention: *un + entre 2 chaînes de caractères (string) déclenche une concaténation (valeurs juxtaposées bout à bout) tandis qu'un + entre 2 nombres déclenche une addition*

*La fonction **float( ) converti une chaîne de caractères en une valeur numérique** (avec potentiellement une virgule notée "." en anglais) .*

Exemple :

```
a=input('a:') # exemple a: 2 et a vu comme '2'
b=input('b:') # exemple b: 3 et b vu comme '3'
c=a+b # '2' + '3' = '23'
print('c=a+b=',c) # affiche par exemple c=a+b= 23

a=float(input('a:')) # exemple a: 2 et a vu comme 2.0
b=float(input('b:')) # exemple b: 3 et b vu comme 3.0
c=a+b # 2.0 + 3.0 = 5.0 = 5
print('c=a+b=',c) # affiche par exemple c=a+b= 5
```

Quelques exercices :

Ecrire un petit script calculant la moyenne de 2 nombres avec des valeurs à saisir :

x=*4 ou 2 ou ...*

y=*8 ou 10 ou ...*

moyenne= *6.0 ou ...*

moyenne.py

Autres essais libres. c'est en forgeant que l'on devient forgeron.

1.5. Commentaires multi-lignes et séparateur d'instructions

```
x=2 ; y=3 ; z=x*y ; print("z=x*y=",z) #  ; est un séparateur d'instructions (sur même ligne)
'''
commentaire sur plusieurs lignes
délimité par triple simple quote .
'''
print('suite')
```

III - Syntaxes élémentaires , types , boucles

1. Types et syntaxes élémentaires de Python

1.1. Types de données en python 3

Comme beaucoup d'autres langages interprétés (tel que javascript) , le langage python a un ***typage dynamique*** :
Il n'est pas nécessaire de préciser le type d'une variable.
Une variable python a , à un instant donné, un type qui dépend de la valeur affectée .

Les principaux types élémentaires sont précisés dans le tableau ci-après

type	*caractéristiques*	*exemples*
int	nombre entier	**-12** **0** **234**
float	nombre à virgule flottante	**-12.5** **0.0** **234.78**
complex	nombre complexe (avec partie imaginaire)	**2+4j**
str	chaîne de caractères (string)	**'abc'** **"def"**
bool	booléen (True or False)	**True** **False**
list	liste (ou tableau redimensionnable) d'éléments	**["rouge" , "vert" , "bleu"]**

la fonction type() renvoie le type (à l'instant présent) d'une variable python .

```
a=128
type(a) # retourne int
b='bonjour'
type(a) # retourne str
```

Pour les nombres complexes , la lettre j a été choisie à la place de i car i est souvent utilisé comme indice pour les boucles.

1.2. Opérations sur chaînes de caractères (str/string)

```
nom="toto"
nom=nom.upper() # retourne la chaîne transformée avec des caractères majuscules
print(nom) # affiche TOTO
```

1.3. Bloc d'instructions et indentation

Dans beaucoup d'autres langages (ex : C/C++ , java , javascript, ...) , un bloc d'instruction est délimité par { et } (début et fin).

Le langage python a quant à lui choisi de délimiter un bloc d'instruction par un niveau d'indentation (un décalage homogène par rapport au début de ligne) : Toutes les instructions décalées via un même nombre d'espaces seront considérées comme appartenant à un même bloc . On conseil généralement 4 espaces pour différencier un sous-bloc de son bloc parent .

exemple en langage **C**	exemple en langage **python**

`int factorielle(int n)`
`{`
`if (n < 2) { return 1; }`
`else {`
`return n * factorielle(n - 1);`
`}`
`}`
	`def factorielle(n):`
	`if n < 2:`
	`return 1`
	`else:`
	`return n * factorielle(n - 1)`

NB :

- Python conseil l'utilisation de plusieurs espaces consécutifs (idéalement 4).
- Des tabulations peuvent éventuellement être utilisées à la place mais il faut choisir entre série d'espaces et tabulations : un mélange des 2 styles est normalement interdit par python3.

1.4. Tests et opérateurs logiques (comparaisons, ...)

```
if condition :
   bloc d'instruction déclenché si condition vérifiée
```

ou

```
if condition :
   bloc d'instruction déclenché si condition vérifiée
else :
   bloc d'instruction déclenché sinon (si condition non vérifiée)
```

Une condition (à vérifier) est généralement formulée comme un test de comparaison :

x == y	égal à
x != y	différent de
x > y	strictement supérieur à
x >= y	supérieur ou égal à
x < y	strictement inférieur à
x <= y	inférieur ou égal à

Exemples :

```
age=20
if age>=18:
    print('majeur pour age=',age)
    print('pas mineur')
print('suite dans tous les cas')

age=16
if age>=18:
    print('majeur pour age=',age)
else :
    print('mineur pour age=',age)
print('suite dans tous les cas')
```

==>

majeur pour age= 20
pas mineur
suite dans tous les cas
mineur pour age= 16
suite dans tous les cas

Une condition peut quelquefois être exprimée comme une combinaison logique de sous condition.

Le mot clef **and** correspond à un "et logique" : les 2 sous conditions doivent être vraies

Le mot clef **or** correspond à un "ou logique" : au moins une des 2 sous conditions doit être vraie

Exemples :

```
age=30
if (age>=18) and (age<=42) :
        print('age entre 18 et 42 ans')
print('suite dans tous les cas')

age=16
if (age<18) or (age>=65) :
        print('enfant ou bien personne agée')
print('suite dans tous les cas')
```

1.5. Listes et fonction len() pour connaître la taille/longueur

Un ***tableau redimensionnable*** (appelé **list** en python) est ***une collection de valeurs consécutives*** dont les **positions** (appelés **indices**) vont de **0 à n-1** .

La fonction prédéfinie **len**(liste) retourne la **taille (ou longueur) d'une liste ou d'un tableau**
La syntaxe *listeXy*[*positionN*] permet d'accéder directement à un élément de la liste .

Exemples :

```
listeDeCouleurs =        [ 'rouge' , 'vert' , 'bleu' , 'noir' , 'blanc' ]
# indices ou positions :      0        1       2       3       4
print('la taille de liste de couleurs est ' , len(listeDeCouleurs) ) # affiche 5
print('la première couleur est ' , listeDeCouleurs[0] ) # affiche rouge
print('la couleurs du milieu est ' , listeDeCouleurs[2] ) # affiche bleu
print('la dernière couleur est ' , listeDeCouleurs[4] ) # affiche blanc
print('en dernier ' , listeDeCouleurs[ len(listeDeCouleurs)-1 ] ) #  blanc
```

1.6. range() , séquence et généralités sur les boucles

Une **séquence** (appelée également ***tuple***) est une **succession de valeur séparée par des virgules** et est assez souvent générée par la fonction **range**()

La fonction **list**(1 , 2 , 3) **construit une liste** (ici **[1, 2, 3]**) **à partir d'une séquence .**

La fonction prédéfinie **range**(**posDebut,posArret,pas**) ***construit la séquence***
posDebut , posDebut+pas , posDebut+2*pas , ...
qui s'arrête lorsque ***posDebut+n*pas*** n'est plus strictement inférieur à ***posArret*** .

Au sein de la fonction range(*posDebut*,**posArret**,*pas*) , le paramètre facultatif *posDebut* a la valeur par défaut *0* s'il n'est pas précisé et le paramètre facultatif *pas* vaut 1 par défaut .

Concrètement :

- La fonction prédéfinie **range**(**n**) construit la séquence **0 , 1 , 2 , ... , n-1**
- **range(3,6)** construit la séquence **3 , ... , 6-1**
- **range(3,-1,-1)** construit la séquence inversée **3 , 2 , 1 , 0**

Exemples :

```
print('range(4)=de 0 à 3<4 =', list(range(4))) # affiche [ 0,1,2, 3 ]
print('range(3,6)=de 3 à 5<6 = ', list(range(3,6))) # affiche [ 3, 4 , 5 ]
print('range(4-1,0-1,-1)= séquence inversée = ', list(range(3,-1,-1))) # affiche [ 3, 2 , 1 , 0 ]
```

En programmation, une boucle permet d'exécuter plusieurs fois un même bloc d'instructions (avec souvent une petite variante) .

En python la boucle ***for*** signifie "***pour chaque éléments de ...***" ou bien "***pour chaque indice dans un certain intervalle***"
La boucle ***while*** signifie "***tant que la condition est vraie***"

1.7. Boucle for (pour chaque élement de ...)

```
#indicesDe0a3= 0,1,2,3
indicesDe0a3= range(4) # range(4) construit 0, 1, 2, 3
for i in indicesDe0a3 :
        print("i=",i)
print('suite apres la boucle')
```

affiche

```
i= 0
i= 1
i= 2
i= 3
suite apres la boucle
```

```
print('boucle de 2 à 8<9 par pas de 2 soit de 2 en 2 :');
for i in range(2,9,2) :
        print("i=",i)
print('suite apres la boucle')
```

==>

```
i= 2
i= 4
i= 6
i= 8
suite apres la boucle
```

```
joursDeLaSemaine = ['lundi','mardi','mercredi','jeudi','vendredi','samedi','dimanche']
# la liste (ou tableau) joursDeLaSemaine comporte 7 éléments dont les indices vont de 0 à 6
for jour in joursDeLaSemaine :
        print(jour,'est un element de joursDeLaSemaine')
print('suite apres la boucle')
```

==>

```
lundi est un element de joursDeLaSemaine
mardi est un element de joursDeLaSemaine
mercredi est un element de joursDeLaSemaine
jeudi est un element de joursDeLaSemaine
vendredi est un element de joursDeLaSemaine
samedi est un element de joursDeLaSemaine
dimanche est un element de joursDeLaSemaine
```

```
#nbJoursDansUneSemaine=7
nbJoursDansUneSemaine=len(joursDeLaSemaine) # la fonction len() retourne la taille/longueur
                                            #(length en englais)
for i in range(nbJoursDansUneSemaine) :
        print("i=",i,"jour=",joursDeLaSemaine[i])
print('suite apres la boucle')
```

==>

```
i= 0 jour= lundi
i= 1 jour= mardi
i= 2 jour= mercredi
i= 3 jour= jeudi
i= 4 jour= vendredi
i= 5 jour= samedi
i= 6 jour= dimanche
```

En exercices :

1. coder et exécuter avec python tous les exemples ci dessus (sans les commentaires et sans copier/coller) pour se familiariser avec la syntaxe .
2. calculer la somme et la moyenne de la liste [12 , 48 , 32 , 8 , 24]

1.8. Boucle while (tant que ...)

```
x=0
while x <= 5 :
        print ('x=',x,'x*x=',x*x)
        x=x+1
print('suite apres la boucle')
```

==>

```
x= 0 x*x= 0
x= 1 x*x= 1
x= 2 x*x= 4
x= 3 x*x= 9
x= 4 x*x= 16
x= 5 x*x= 25
suite après la boucle
```

Attention: de pas oublier x=x+1 sinon boucle infinie qui ne s'arrête jamais !!!

```
listeCouleurs = [ "rouge" , "noir" ]
listeCouleurs.append("vert") # ajoute l'élément "vert" à la liste
listeCouleurs.append("bleu") # ajoute l'élément "bleu" à la liste
print("listeCouleurs=",listeCouleurs); # ['rouge', 'noir', 'vert', 'bleu']
```

```
listeValeurs = []
valeurOuFin=input("val=")
while (valeurOuFin != 'fin' ) and (valeurOuFin != '' ) :
        nouvelleValeur = float(valeurOuFin)
        listeValeurs.append(nouvelleValeur)
        valeurOuFin=input("val=")
print('listeValeurs=',listeValeurs)
```

==>

```
val=4
val=6
val=8
val=
listeValeurs= [4.0, 6.0, 8.0]
```

En exercices :

3. coder et exécuter avec python tous les exemples ci dessus (sans les commentaires et sans copier/coller) pour se familiariser avec la syntaxe .
4. calculer la somme et la moyenne de la liste [12 , 48 , 32 , 8 , 24] sans utiliser la boucle for mais en utilisant la boucle while et en initialisant une variable i=0 avant la boucle .
5. transformer ["hiver" , "printemps" , "ete" , "automne"] en liste de valeurs en majuscules via une boucle au choix (for ou while) puis afficher toute la liste transformée .

IV - Structures de données (liste,dictionnaire,...)

1. Manipulation de listes en python

Le langage python gère des listes comme des tableaux redimensionnables .
Une liste python ressemble à un tableau javascript ou une ArrayList de java.
Au sein d'une liste python, les éléments sont ordonnés et les indices valides vont de 0 à n-1.

```
listeVide=[]

listeInitiale=[1,2,3]

liste=listeInitiale
liste.append(4) # ajoute un nouvel élément en fin de liste
print("liste=",liste) # affiche [1,2,3,4]

print("premier élément:",liste[0]) # affiche 1
liste[0]=1.1
print("premier élément modifié:",liste[0]) # affiche 1.1

#liste[4]=5  --> IndexError: list assignment index out of range

print("dernier élément:",liste[-1]) # affiche 4 (dernier élément)
```

```
liste2 = [ "a" , "b" , "c" ]
del liste2[1] # suppression de l'élément d'indice 1 (0,1,2)
print("liste2=",liste2) # affiche [ 'a' , 'c' ]

liste3 = [ "rouge" , "vert" , "bleu" ]
liste3.remove("vert") # suppression de l'élément dont la VALEUR est "vert"
print("liste3=",liste3) # affiche [ 'rouge' , 'bleu' ]
```

```
liste4=[1,2,3,4];
liste4.reverse() # inverse l'ordre des éléments de la liste
print("liste4=",liste4) # affiche [ 4,3,2,1 ]
```

```
nbElements=len(liste4)
print("longueur (nbElements) de liste4=",nbElements) # affiche 4
```

```
liste5=['a' , 'b' , 'a' , 'b' , 'c' , 'a' ]
nbOccurencesDeA = liste5.count('a')
print("nbOccurences de 'a' dans liste5=",nbOccurencesDeA) # affiche 3
```

```
indiceBdansListe5= liste5.index('b')
print("indiceBdansListe5",indiceBdansListe5) # 1 (premier trouvé)

indiceCdansListe5= liste5.index('c')
print("indiceCdansListe5",indiceCdansListe5) # 4
```

```
#liste5.index('e') --> ValueError: 'e' is not in list
```

```
liste = [ 'a' , 'b', 'c' ]

#boucle sur valeur des éléments:
for val in liste :
        print(val)
# a
# b
# c
```

```
#boucle sur indices et valeurs des éléments:
for tuple_indice_val in enumerate(liste) :
        print(tuple_indice_val , tuple_indice_val[0] , tuple_indice_val[1] )
# (0,'a') 0 a
# (1,'b') 1 b
# (2,'c') 2 c
```

```
liste6 = [ 2 , 4 , 6 ]
refListe = liste6 # refListe référence la même liste que celle référencée par liste6
refListe[0]=2.2 # même effet que liste6[0] = 2.2
print("liste6=",liste6) # affiche [ 2.2 , 4 , 6 ]

liste7 = [ 1 , 3 , 5]
liste8 = liste7.copy() # copie/duplication d'une liste
liste8[0]=1.1
print("liste7=",liste7) # affiche [1, 3 , 5]
print("liste8=",liste8) # affiche [1.1, 3 , 5 ]
```

```
maPile = [ 1 , 2 ,  3 , 4]
dernierElementRetire  = maPile.pop(); print(dernierElementRetire); # 4
dernierElementRetire  = maPile.pop(); print(dernierElementRetire); # 3
print(maPile); # [ 1, 2 ]
```

```
troisCouleurs="rouge;vert;bleu" # grande chaîne de caractères avec sous parties séparées par ";"
listeCouleurs = troisCouleurs.split(";")
print("listeCouleurs=",listeCouleurs) # ['rouge', 'vert', 'bleu']

mesCouleurs=";".join(listeCouleurs) # transforme liste en chaîne de caractères
print("mesCouleurs=",mesCouleurs) # affiche la chaîne rouge;vert;bleu
```

```
liste10=[ 'a' , 'b' , 'c' ]
if 'a' in liste10 :
        print('liste10 comporte a')
else :
        print('liste10 ne comporte pas a')

if 'd' in liste10 :
```

```
        print('liste10 comporte d')
else :
        print('liste10 ne comporte pas d')
```

2. Set (ensembles)

```
ensembleVide={}

#dans un ensemble, les éléments ne sont pas ordonnés (sans index stable)
#dans un ensemble, chaque élément est unique (sans duplication possible)

ensembleDeFruits={"apple", "banana", "cherry"}

print("ensembleDeFruits=",ensembleDeFruits) #  {'banana', 'cherry', 'apple'}
```

```
for f in ensembleDeFruits:
        print(f)
#banana
#cherry
#apple
```

```
#NB: une fois qu'un ensemble a été créé/initialisé , on ne peut pas modifier
# ses éléments mais on peut ajouter un nouvel élément via .add()
# ou bien de nouveaux éléments via .update()

ensembleDeFruits.add("orange") # ajout (sans notion d'ordre)
print(ensembleDeFruits) # {'apple', 'orange', 'banana', 'cherry'}

ensembleDeFruits={"apple", "banana", "cherry"}
ensembleDeFruits.update({"orange" , "peach"}) # ajout de plusieurs éléments
print(ensembleDeFruits) # {'apple', 'peach', 'banana', 'orange', 'cherry'}
```

```
ensembleDeFruits={"apple", "banana", "cherry"}
ensembleDeFruits.remove("banana") # supprime un élément s'il existe , erreur sinon
print(ensembleDeFruits)  # {'cherry', 'apple'}
ensembleDeFruits.discard("banana") # supprime un élément s'il existe toujours sans erreur

ensembleDeFruits.clear() # vide l'ensemble
print(ensembleDeFruits)  # {}
```

```
set1 = {"a", "b" , "c"}
set2 = {"d", "e", "f"}

set3 = set1.union(set2)
print(set3) # {'b', 'd', 'a', 'e', 'c', 'f'}

set4 = {"a", "b" , "c" , "d"}
set5 = {"c", "d" , "e" , "f"}

set6 = set4.intersection(set5)
```

```
print(set6) # {'d', 'c'}

#il existe également .isdisjoint() , .issubset() , .difference() , ...
```

3. Dictionnaires (associations)

```
dictionnaireVide={}
dictionnaire2 = dict()
```

Un dictionnaire python est une table d'association (Map) : (ensemble de couples (clef,valeur))
La syntaxe d'un dictionnaire python est très proche de JSON (javascript object notation)

```
dictionnaireCouleurs={
        "red":"#ff0000",
        "green":"#00ff00",
        "blue":"#0000ff"
}

rgbRed=dictionnaireCouleurs["red"];
print("rgbRed=",rgbRed); # #ff0000
dictionnaireCouleurs["red"]="#FF0000"; # change value
print(dictionnaireCouleurs["red"]); # #FF0000

rgbGreen=dictionnaireCouleurs.get("green");
print("rgbGreen=",rgbGreen); # #00ff00
```

```
dictionnaireCouleurs["black"]="#000000"; # add new pai (key ,value )
print(dictionnaireCouleurs["black"]); # #000000

dictionnaireCouleurs.update({"white":"#FFFFFF"}); # add new {key : value } pair
print(dictionnaireCouleurs["white"]); # #FFFFFF

dictionnaireCouleurs.update({"white":"#ffffff" , "yellow" : "#ffff00"});
                        # fusionner autre dictionnaire

print(dictionnaireCouleurs["yellow"]); # #ffff00
```

```
del dictionnaireCouleurs["white"]; #ou bien dictionnaireCouleurs.pop("white")
print(dictionnaireCouleurs); # affichage après suppression de l'association "white"
# {'red': '#FF0000', 'green': '#00ff00', 'blue': '#0000ff', 'black': '#000000', 'yellow': '#ffff00'}
```

```
dicoPers= { "nom" : "Bon" , "age" : 45 }
print("nom=" + dicoPers.get("nom")); # Bon
print("nom=" + dicoPers.get("prenom","prenomParDefaut")); # prenomParDefaut
```

```
dicoPers= { "nom" : "Bon" , "age" : 45 }
for key in dicoPers :
        print(key)
# nom
# age
```

ou bien

```
dicoPers= { "nom" : "Bon" , "age" : 45 }
for clef in dicoPers.keys() :
        print(clef)
# nom
# age
```

```
for val in dicoPers.values() :
        print(val)
# Bon
# 45
```

```
for clef,val in dicoPers.items() :
        print(clef,val)
# nom Bon
# age 45
```

```
dicoPers= { "nom" : "Bon" , "age" : 45 }
print("nbAssociations=" , len(dicoPers)) # 2

dicoDuplique=dicoPers.copy(); #ou bien dicoDuplique=dict(dicoPers);
dicoDuplique["age"]=30
print(dicoPers) # { "nom" : "Bon" , "age" : 45 }
print(dicoDuplique) # { "nom" : "Bon" , "age" : 30 }

dicoDuplique.clear() # vide le contenu du dictionnaire
```

```
if "nom" in dicoPers :
        print("dicoPers comporte la clef nom")
else :
        print("dicoPers ne comporte pas la clef nom")
```

```
#NB: au sein d'un dictionnaire une valeur peut être
#une liste ou un dictionnaire imbriqué:
dicoPers={
 "nom" : "Bon" ,
 "age" : 45 ,
 "fou" : False ,
 "adresse" : {
      "rue" : "12 rue elle",
               "codePostal" : "75008",
               "ville" : "Paris"
               } ,
  "sports" : [ "vélo" , "foot" ]
 }
print("dicoPers très proche du format JSON:" , dicoPers);
```

4. Manipulation de chaînes de caractères

```
s1="Hello"
s2=' World'
s3=s1+s2 # concaténation
print(s3) # Hello World
```

```
s4='''ile de
france'''
print(s4) # affiche une chaîne multi-lignes
```

```
s5="Bonjour"
premierCaractere=s5[0]
print(premierCaractere) # B

dernierCaractere=s5[-1]
print(dernierCaractere) # r
```

```
# [i,j] means .substring(i,j) included i and excluding j

troisPremiersCaracteres=s5[0:3] # s5[0:3] means s5[0:3[ !!!
print(troisPremiersCaracteres) # Bon

n=len(s5) # length of string (7)

troisDerniersCaracteres=s5[n-3:n] # s5[n-3:n] means s5[n-3:n[ !!!
print(troisDerniersCaracteres) # our
```

```
s6=" Ile De France "
s6Bis=s6.strip() #like trim of other language --> supprime espaces inutiles
                 #au début ou à la fin
print(s6Bis) # "Ile De France"
```

```
s7="Mont Saint Michel"
s7Maj = s7.upper(); print(s7Maj) # MONT SAINT MICHEL
s7Min = s7.lower(); print(s7Min) # mont saint michel

s7="Mont Saint Michel"
s7Bis=s7.replace(' ','-') # replace substring with another string
print(s7Bis) # Mont-Saint-Michel
```

```
s8="partie1;partie2;partie3"
listeParties=s8.split(';')
print(listeParties) # ['partie1', 'partie2', 'partie3']
```

```
s9="un deux trois"
if "deux" in s9 :
        print("s9 comporte deux")
else :
        print("s9 ne comporte pas deux")

#il existe aussi le test if "deux" not in s9
```

```
nom="toto"
age=30
taille=1.80
# .format remplace {0} , {1} , {2} , ... par les 1er,2eme,3eme arguments
description="{0} a {1} an(s) et mesure {2} m".format(nom,age,taille)
print(description) # toto a 30 an(s) et mesure 1.8 m
```

```
#caractères spéciaux : \n = new line , \t = tabulation , ...
#escape : \\ means \

s10="\tHello" ; print(s10); #      Hello
s11="surLigne1\nsurLigne2" ; print(s11);
# surLigne1
# surLigne2
```

```
s12="dupond";
s12Bis=s12.capitalize() # transforme première lettre en Majuscule
print(s12Bis); # Dupond
```

```
fileName="p2.py"
dotIndex = fileName.find(".")
# .index() retourne position de la chaine recherchée et erreur si pas trouvée
# .find() retourne position de la chaine recherchée et -1 si pas trouvée
print("position . =" , dotIndex) # 2
```

```
s13="phrase finissant par un point."
if s13.endswith(".") :
        print("s13 se termine par '.' ")
```

```
s14="123"
if s14.isdigit() :
        print("s14 ne comporte que des caractères numériques ")
```

5. Dates

```
import datetime

d = datetime.datetime.now()
print(d) # exemple: 2020-05-03 00:18:51.608375
```

--> étudier le module *datetime* pour approfondir si besoin

V - Fonctions, Modules, pip

1. Fonctions (python)

Un **programme** bien **structuré** est généralement constitué de **blocs de code réutilisables** appelés "**fonctions**" .

Au sein du langage python :

- la définition d'une fonction est introduite par le mot clef **def** .
- une fonction a toujours des **parenthèses** (au sein de la définition et des appels)
- une fonction peut avoir (ou pas) des paramètres et une valeur calculée et retournée via le mot clef **return** .

1.1. Fonctions élémentaires

```
#fonction basique sans paramètre retournant une valeur fixe:
def genererMessageSalutation() :
        message="bonjour"
        return message
```

```
#fonction/procédure basique ne retournant aucune valeur
#mais exécutant une action:
def saluer():
        salutation=genererMessageSalutation() #appel de sous-fonction
        print(salutation)
```

```
saluer() #l'appel de la fonction déclenche l'affichage de bonjour
```

1.2. Fonctions avec paramètres

```
#fonction multiplier avec 2 paramètres formels a et b
def multiplier(a,b):
        return a*b
```

```
x=4; y=5;
res = multiplier(x,y); # appel de la fonction multiplier en passant
                       # les valeurs des paramètres effectifs x et y
                       # lors de l'appel a est une copie de x
                       # et b est une copie de y

print("res=",res) # affiche res=20
```

2. Modules et packages (python)

Dès qu'un programme comporte beaucoup de lignes de code, il est conseillé de **répartir les instructions dans plusieurs fichiers complémentaires** .
Certains fichiers (utilisés par d'autres) constitueront ainsi des modules de code prêts à être réutilisés.

2.1. importations de fonctions

my_fct.py

```
def doubleDe(x):
        return 2*x

def moitieDe(x):
        return x/2
```

my_app.py

```
# importation de toutes les fonctions du fichier my_fct.py
# pour pouvoir les appeler depuis ce fichier my_app.py
from my_fct import *

x=12;
print(doubleDe(x)) # affiche 24
print(moitieDe(x)) # affiche 6
```

ou bien

my_app.py

```
import my_fct

# --> appels de my_fct.doubleDe(x) et my_fct.moitieDe(x)

x=12;
print(my_fct.doubleDe(x)) # affiche 24
print(my_fct.moitieDe(x)) # affiche 6
```

2.2. packages de fichiers réutilisables (modules/librairies)

On range souvent dans un ***répertoire*** (ex : ***my_util***) un paquet de fichiers "python" réutilisables :

Exemple :

my_util/op3.py

```
def tripleDe(x):
        return 3*x

def tiersDe(x):
        return x/3
```

my_util/op4.py

```
def quatreFois(x):
        return 4*x

def quartDe(x):
        return x/4
```

my_app.py

```
#from my_util.op3 import *
from my_util.op3 import tripleDe
from my_util.op4 import *


x=12 ;
print(tripleDe(x)) # affiche 36
print(quartDe(x)) # affiche 3.0
```

2.3. Principaux modules prédéfinis du langage python

modules	***fonctionnalités***
random	nombres aléatoires et autres
math	fonctions mathématiques (sin, cos, ...)
os	
time	
sys	

3. Calculs mathématiques élémentaires

Le module prédéfini ***math*** de python comporte quelques **fonctions mathématiques élémentaires** que l'on retrouve dans la plupart des autres langages de programmation (C, java, ...) .

3.1. Principales fonctions du Module math

fonction	fonctionnalité
math.**sin(x)**	sinus avec x en radians
math.**cos(x)**	cosinus avec x en radians
math.**tan(x)**	tangente avec x en radians
asin(x) , acos(x) , atan(x)	arc sinux , arc cosinus , arc tangante
math.**radians(x)**	convertit un angle de degrés en radians
math.**degrees(x)**	convertit un angle de radians en degrés
math.**sqrt(x)**	racine carrée
math.**pow(x,y)**	élève x à la puissance y
math.**exp**(x)	calcule **e** puissance x
math.**log**(x)	calcul ln(x)
math.**hypot**(x,y)	longueur de l'hypoténuse = sqrt(x*x+y*y)
math.**ceil**(x)	arrondi à l'entier au dessus : ceil(5.5) = 6
math.**floor**(x)	arrondi à l'entier en dessous : floor(5.5) = 5
math.**fabs**(x)	retourne la valeur absolue : fabs(-7) = 7
math.**gcd**(a,b)	retourne le pgcd : plus grand diviseur commun

Constantes : math.**pi** (3.141592....) et math.**e** (2.718281…)

Exemples :

```
from math import *
print("pi=",pi) # 3.141592653589793
print("e=",e) # 2.718281828459045
print("sin(pi/6)",sin(pi/6)) # 0.49999999999999
y=pow(2,3); print("2 puissance 3 = " , y); # 8
```

ou bien

```
import math
print("pi=",math.pi) # 3.141592653589793
print("e=",math.e) # 2.718281828459045
print("sin(pi/6)",math.sin(pi/6)) # 0.49999999999999
y=math.pow(2,3); print("2 puissance 3 = " , y); # 8
```

3.2. Exemple (résolution d'équation du second degré)

```
import math
#NB: math.sqrt() calcule la racine carrée (square root).

# resolution ax^2+bx+c=0
def resolEq2ndDegre(a,b,c):
        delta = b*b-4*a*c
        if delta==0 :
                x1=x2=-b/(2*a)
        if delta > 0 :
                x1=(-b-math.sqrt(delta))/(2*a)
                x2=(-b+math.sqrt(delta))/(2*a)
        if delta <0 :
                x1=(-b-1j*math.sqrt(-delta))/(2*a)
                x2=(-b+1j*math.sqrt(-delta))/(2*a)
        print("solutions pour equation ax^2+bx+c=0 avec a=",a, "b=",b , "c=" ,c );
        print("x1=",x1)
        print("x2=",x2)

resolEq2ndDegre(2,-9,-5); # x1=-0.5 et x2=5

resolEq2ndDegre(2,-1,-6); # x1=-1.5 et x2=2

resolEq2ndDegre(1,3,9/8); # x1=x2=4/4=0.75

resolEq2ndDegre(1,2,5); # x1=-1-2j et -1+2j avec j=i et j^2=i^2=-1
```

3.3. Module "random" pour nombres aléatoires

```
import random

x =random.random()       # Random float x, 0.0 <= x < 1.0
print(x) # x=0.3581652418510137 ou autre

x=random.uniform(1, 10)  # Random float x, 1.0 <= x < 10.0
print(x) # x=6.1800146073117523 ou autre

x=random.randint(1, 10)  # Integer from 1 to 10, endpoints included
print(x) # x=6 ou autre

import string
small_letters = string.ascii_lowercase # séquence des caractères de a à z
print(small_letters) # affiche abcdefghijklmnopqrstuvwxyz
c=random.choice( small_letters) # retourne un éléments de la séquence
                                # choisi aléatoirement : ici une lettre en a et z
print(c) # affiche c ou r ou autre

subList = random.sample([1, 2, 3, 4, 5],  3)  # Choose 3 elements
print(subList) # affiche [5, 2, 1] ou autre
```

4. Fonctions natives (sans import nécessaires)

...

5. PIP

...

VI - Gestion des exceptions / erreurs

1. Gestion des exceptions (python)

NB : En python comme dans la plupart des autres langages de programmation, une division entre 2 nombres entiers provoque une erreur/exception dans le cas d'une ***division par zéro*** .

Attention : python soulève l'exception ***ZeroDivisionError: float division by zero*** *lors d'une division entre "float" (x / 0.0) , là où d'autres langages tels que java ou c++ retourne "NaN" (not a number)*

1.1. plantage du programme sans traitement d'exception

exceptions.py

```
a=5
b=0
c=a/b
print(c)
```

==>

```
Traceback (most recent call last):
  File "exceptions.py", line 3, in <module>
    c=a/b
ZeroDivisionError: division by zero
```

1.2. avec traitement des exceptions (try / except)

```
a=6
#b=2
b=0
try :
        c=a/b
        print("res division=" , c)
except :
        print("attention: une erreur s'est produite !!!")

print("suite du programme qui ne s'est pas planté")
```

avec b=2 :

```
res division= 3.0
suite du programme qui ne s'est pas planté
```

avec b=0 :

```
attention: une erreur s'est produite !!!
```

```
suite du programme qui ne s'est pas planté
```

1.3. Lever une exception personnalisée (raise Exception)

```
def myDivision(x,y):
        if y==0 :
                raise Exception("division par zéro invalide")
                    else :
                return x/y
```

```
a=6
#b=2
b=0

try :
        c=myDivision(a,b);
        print("res myDivision=" , c)
except Exception as e :
   print("une erreur a eu lieu :" , e)  # affiche une erreur a eu lieu : division par zéro invalide
```

1.4. Syntaxe facultativement complète (try / except / finally)

....

VII - Gestion des fichiers (depuis python)

1. Gestion des fichiers en python

1.1. ouverture , lecture et écritures

```
'''
principaux modes d'ouverture:
        r :  read
        w : write / ré-écriture (écrasement)
        a : append (ajout à la fin)

le fichier est souvent créé en écriture s'il n'existe pas

modes secondaires d'ouverture :
        b : binaire (ex: images , videos, ...)
        t : texte
'''

f= open("data.txt","wt")
print("f=",f); # affiche le descripteur de fichier ouvert, par exemple :
#f= <_io.TextIOWrapper name='data.txt' mode='wt' encoding='cp1252'>
f.close() # fermeture du fichier
```

```
#ouvrir un nouveau fichier et écrire 2 lignes dedans:
f= open("data.txt","wt")
f.write("ligne1\n")
f.write("ligne2\n")
f.close();
```

```
#ré-ouvrir un fichier existant et ajouter 2 lignes dedans:
f= open("data.txt","at")
f.write("ligne3\n")
f.write("ligne4\n")
f.close();
```

```
#ré-ouvrir un fichier existant et charger son contenu d'un seul coup:
f= open("data.txt","rt")
toutLeContenu=f.read(); print(toutLeContenu);
f.close();
```

```
#ré-ouvrir un fichier existant et lire son contenu ligne par ligne via .readline() et while
f= open("data.txt","rt")
ligneLue="?"
while ligneLue :
        ligneLue=f.readline()
        if ligneLue.endswith("\n") :
```

```
                ligneLue=ligneLue[:-1]  # enlever le dernier caractère
        print(ligneLue);
f.close();
```

```
#ré-ouvrir un fichier existant et lire son contenu ligne par ligne via boucle for :
f= open("data.txt","rt")
for ligneLue in f :
        if ligneLue.endswith("\n") :
                ligneLue=ligneLue[:-1]
        print(ligneLue);
f.close();
```

1.2. avec fermeture automatique (with)

```
###### with keyword for automatic closing (as in try/except/FINALLY) ####

with open("data2.txt","wt") as f :
        f.write("ligne1\n")
        f.write("ligne2\n")
        # automatic f.close() even in case of exception
```

1.3. noms de fichiers, répertoires , chemins

...

2. Formats de fichiers et gestion en python

2.1. gestion du format json

JSON signifiant **JavaScript Object Notation** est un format de données très classique utilisé pour :
- **paramétrer des configurations**
- **structurer des données échangées** (documents , appels de WS REST, ...)

La structure et la syntaxe du format json est très proche de celle d'un dictionnaire python.

Principales équivalences :

Python	JSON
dict	object
list, tuple	array
str	string
int, long, float	number
True	true
False	false
None	null

La bibliothèque prédéfinie **json** (à importer via ***import json***) comporte essentiellement les méthodes :

- **dumps**() permettant de transformer des données python en chaîne de caractères json .
- **loads**() permettant d'analyser une chaîne de caractères json et d'effectuer une conversion en python

Exemple d'écriture au format json dans un fichier :

```
import json

#personne1 en tant que dictionnaire python :
personne1={
  "nom" : "Bon" ,
  "age" : 45 ,
  "fou" : False,
  "adresse" : {
        "rue" : "12 rue elle",
                    "codePostal" : "75008",
                    "ville" : "Paris"
                    } ,
   "sports" : [ "velo" , "foot" ]
 }

#p1AsJsonString = json.dumps(personne1);
p1AsJsonString = json.dumps(personne1,indent=4);
print("p1AsJsonString=",p1AsJsonString)
with open("p1.json","wt") as f :
        f.write(p1AsJsonString)
```

contenu du fichier ***p1.json*** généré :

```
{
   "nom": "Bon",
   "age": 45,
   "fou": false,
   "adresse": {
      "rue": "12 rue elle",
      "codePostal": "75008",
      "ville": "Paris"
   },
   "sports": [
      "velo",
      "foot"
   ]
}
```

Exemple de lecture d'un fichier au format json:

```
#relecture du fichier p1.json et extraction du contenu en données python:
import json

with open("p1.json","rt") as f :
        fileContentAsJsonString=f.read()
        pers = json.loads(fileContentAsJsonString)
        print("pers=",pers);
        print("type(pers)=",type(pers));
```

-->

```
pers= {'nom': 'Bon', 'age': 45, 'fou': False, 'adresse': {'rue': '12 rue elle', 'codePostal': '75008', 'ville': 'Paris'}, 'sports': ['velo', 'foot']}
type(pers)= <class 'dict'>
```

2.2. gestion du format xml

...

VIII - Python orienté objet

1. Programmation orientée objet en python

1.1. Classe et instances

En langage python, pas de mof clef **~~this~~** mais le mot **self** signifiant "objet courant de la classe" (qui sera spécifié depuis du code extérieur à la classe via un préfixe de type ***obj.***)

En langage python, la fonction constructeur (initialisant les valeurs interne de l'objet) a le nom spécial **__init__** .

Exemple :

objets.py

```
import math

########### code de la classe Cercle en python :

class Cercle() :

   #constructeur avec valeurs par défaut:
        def __init__(self,xc=0,yc=0,rayon=0) :
                self.xc=xc
                self.yc=yc
                self.rayon=rayon

        #méthode spéciale __str__ (équivalent à .toString() de java)
        #qui sera automatiquement appelée lors d'un print(cercle):
        def __str__(self) :
                return "Cercle(xc="+str(self.xc) +",yc="+str(self.yc)+",rayon="+str(self.rayon)+ ")"

        def perimetre(self) :
                return 2*math.pi*self.rayon

        def aire(self) :
                return math.pi*self.rayon*self.rayon

###### utilisation de la classe Cercle


c1=Cercle(); #instanciation (pas de mot clef new) mais nom de classe
                #vue comme fonction créant une nouvelle instance
c1.rayon=40;
print("rayon de c1=",c1.rayon) # rayon de c1= 40
print("perimetre de c1=",c1.perimetre()) # perimetre de c1= 251.32741228718345
print("surface de c1=",c1.aire()) # surface de c1= 5026.548245743669

c2=Cercle(40,60,20) # Cercle(xc,yc,rayon)
print("rayon de c2=",c2.rayon) # rayon de c2= 20
```

```
print("c2=" , c2) # équivalent à print("c2=" , str(c2))
# affiche c2= Cercle(xc=40,yc=60,rayon=20)
```

Récupération des valeurs de l'instance sous forme de dictionnaire python :

```
#suite de l'exemple précédent (où c2 est une instance de la classe Cercle)
print("type(c2)=",type(c2)) # <class '__main__.Cercle'>
c2AsDict = vars(c2) # converti un objet en un dictionnaire  (autre solution = c2.__dict__ )
print("c2AsDict=",c2AsDict) # {'xc': 40, 'yc': 60, 'rayon': 20}
print("type(c2AsDict)=",type(c2AsDict)) #  <class 'dict'>
```

ANNEXES

IX - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie et liens vers sites "internet"

2. TP