Découvrir …… **PYTHON** (*3.x*)

Ce cours s'adresse à un lecteur qui connait bien l'Algorithmique de base.

PARTIE I : ALGORITHMIQUE ET PYTHON

Cette première partie est uniquement pour présenter Python à travers une « traduction » de l'algorithmique dans ce langage, en précisant quelques-unes de ses particularités.

A. INTRODUCTION :

Algorithmique : « langage de programmation » proche du langage naturel, souple car il s'adresse à la machine d'exécution de ses instructions qui est l'homme

Langage de programmation : langage destiné à un ordinateur, très strict, ne disposant que de peu de mots dans son vocabulaire, ne supportant donc aucune ambiguïté.

Python est un langage de programmation, tout comme Fortran, Pascal, C, C++, Java etc..

B. PRESENTATION:

1. Généralités

Python:

- a été créé par Guido Van Rossum (Pays-Bas) en 1991, il y a donc presque 30 ans
- est un langage :
  - **multi-plateforme** (Windows – Unix – Linux – Mac Os – Android …..)
  - **interprété, partiellement compilé.**
  - qui peut **fonctionner en deux modes** :
    - **le mode interactif :** comme une calculatrice (très) intelligente, (très) puissante.
    - **le mode édition :** écriture de scripts (on dira plus simplement programme) plus longs avec un éditeur, enregistrés dans un fichier, puis exécutés par l'interpréteur.
  - **enrichi** par une quantité impressionnante de librairies, de modules, composés de fonctions « prêtes à l'emploi ».
  - qui **respecte la casse** (**Masse** est différent de **masse**).
  - de **programmation objet** : en Python, tout est objet.

En réalité, nous ne manipulerons que des objets Python, sans réellement le savoir. Le mot objet sera souvent utilisé dans ce sens.

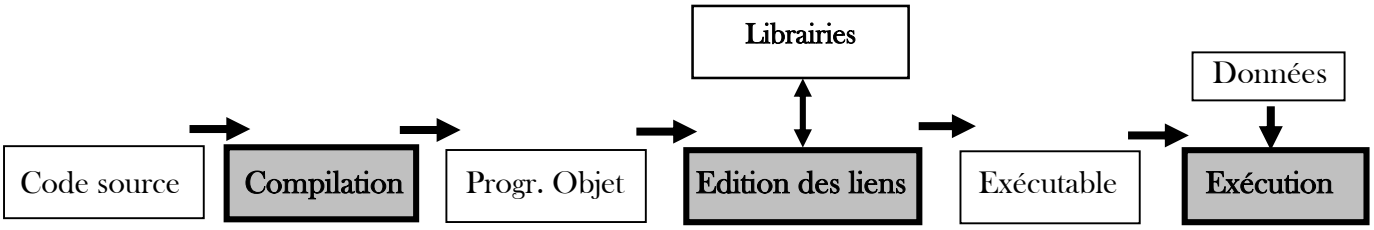
2. Interpréteur - Compilateur

A partir d'un Pseudo-code (un algorithme), traduit dans un langage de programmation donné, on obtient un **code source, destiné à être traité par un ordinateur.** Le code source peut être :

- exécuté directement par l'ordinateur, instruction par instruction. On dit alors que le langage utilisé est un langage **interprété**.
- « contrôlé d'abord dans son entièreté » (**compilé**) avant que l'ordinateur ne l'exécute. On parle alors de langage **compilé.**

Un langage interprété a besoin d'un **interpréteur, u**n langage compilé d'un **compilateur**.

a. Avec un compilateur (Fortran, Pascal, C, C++, etc…), on a le schéma suivant des différentes étapes de l'exécution d'un code source:



- on compile le code source (vérifications syntaxiques, lexicales et sémantiques). Le code source devient un programme objet.
- Lors de l'édition des liens, on vérifie que les ressources extérieures sollicitées sont bien accessibles.
- une fois ces vérifications satisfaites, le programme objet devient un exécutable, totalement écrit en langage machine (lisible directement par l'ordinateur)

Remarques: Un programme, une fois compilé, s'exécute très rapidement. Et l'exécutable peut être enregistré pour être exécuté plus tard.

b. Avec un interpréteur :

On exécute tous ces contrôles et transformations instructions par instructions.
Interpréter = Compiler et exécuter si possible, instruction par instruction.
Un point positif: dès qu'on décèle une erreur, l'exécution est stoppée, facilitant les corrections.

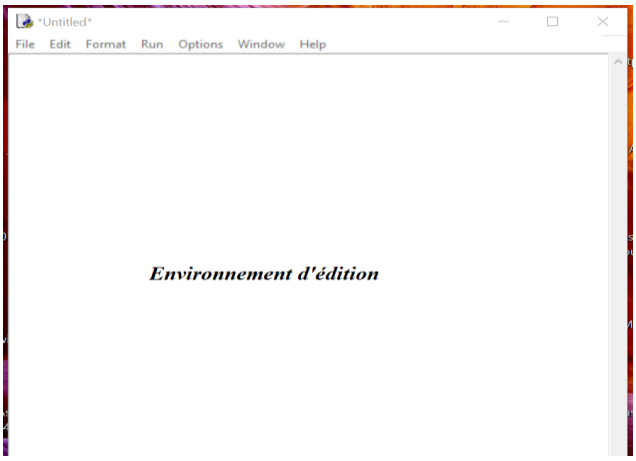
Python est plutôt interprété, mais il travaille avec un très grand nombre de librairies contenant des fonctions « prêtes à l'emploi ». Ce qui fait sa force.

C. INSTALLATION

On peut se procurer gratuitement de Python sur le site de Python lui-même :
www.python.org/downloads/ .
Toutes les versions depuis 2001 pour tous les systèmes d'exploitation y sont proposées.
La version utilisée pour les illustrations de ce document est Python 3.6.5, la dernière en date étant, au moment de la rédaction de ce document, la version 3.7.3

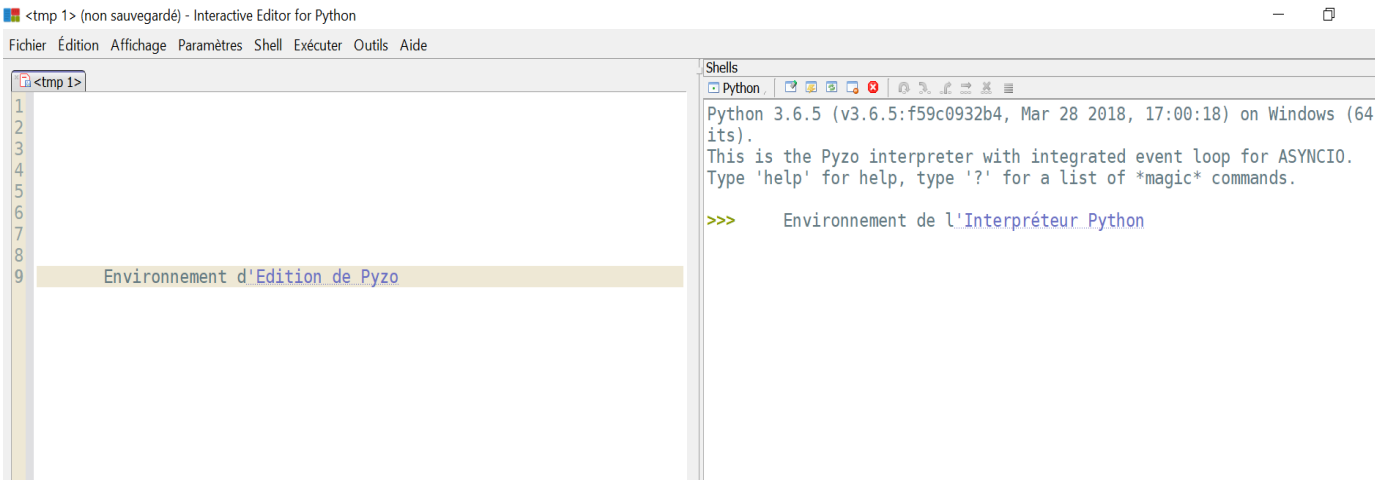
Python vient nativement avec **Idle**, un environnement complet de programmation comprenant un interpréteur et un environnement d'édition de scripts.

Une fois installé, on obtient les deux environnements ci-dessous, sous Idle:



Par défaut, on est sous l'interpréteur. Pour éditer un script, il faut ouvrir l'environnement d'édition en passant par l'onglet **File** qui propose d'ouvrir un fichier existant, ou de créer un nouveau fichier.
Un second environnement est utilisé dans ce document : Pyzo, à installer seulement une fois que Python est installé.

Page de travail sous Pyzo



D. MODE INTERACTIF – MODE EDITION

1. Le mode interactif

C'est le mode pour des calculs et tests rapides. Python, avec l'utilisation directe de l'interpréteur, se comporte comme une calculatrice (très) performante.

L'invite des commandes est >>>. Pour les valider, on tape la touche ↵ (entrer)

a. Quelques exemples :

```
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 2
900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()"
>>> 23+65
88
>>> a=87
>>> b=76
>>> a+b
163
>>>
```

```
>>> print("abcd",'  ','mnoq')
abcd    mnoq
>>> a="abcd"
>>> a
'abcd'
>>> a=a*3
>>> a
'abcdabcdabcd'
>>>
```

b. Remarques :

- _ (underscore) rappelle le dernier résultat calculé: _+'432' n'est autre que '56'+'432' qui est bien égal à '56432'. (à tester)
- L'interpréteur dispose de fonctions plus riches que les calculatrices usuelles.
- L'affichage du contenu de a en mode interactif est différent de celui en mode instruction (utilisation de print()).
- Taper un objet ou un nom d'objet, puis valider, affiche le contenu de celui-ci.
- Taper une commande à exécuter ne se traduit pas nécessairement par une réponse affichée.

  Ex : >>> c=87 action : affecter la valeur 87 à c. Aucun affichage.

  >>> x='cours' même remarque.

  >>> 6+9 6+9 est un objet « calculé » : la somme est calculée et l'objet résultat est affiché

  >>> a==c a==c est un objet. Sa valeur est affichée.

c. Une grande puissance de calcul:

```
>>> 3**100
515377520732011331036461129765621272702107522001
>>> 5**500
30549363634996046820519793932136176997894027405723266638936139092812916265247204577018572351080152282568751526935904671553178534278042839697351331142009178896307244205337728522220355888195318837008165086679301794879136633899370525163649789227021200352450820912190874482021196014946372110934030798550767828365183620409339937395998276770114898681640625
```

d. Possibilités dans ce mode de tester des scripts courts :

```
>>> for i in range(20):
...     if i%3==0:
...         if i!=0:print('*  ',end='')
...     elif i+1>=20:print(i)
...     else:print(i,end='  ')
...
1  2  *  4  5  *  7  8  *  10  11  *  13  14  *  16  17  *  19

>>>
```

e. Attention! On peut être face à certaines « bizarreries » de certains résultats: Certains calculs effectués par Python ne semblent pas si « exacts » que cela !

```
>>> 7*7/100
0.49
>>> 7/10*7/10
0.4899999999999994
>>> 0.7*0.7
0.4899999999999994

>>> 0.1*0.1
0.010000000000000002
>>> 0.1**2
0.010000000000000002
>>> 1/10*1/10
0.01

>>> 1/10*1/10*0.1
0.001
```

2. Le mode Edition

Dans ce mode, on écrit le script entier, que l'on peut enregistrer, tester, corriger, améliorer. L'exécution se fait avec l'interpréteur. Celle-ci est stoppée dès qu'une erreur est rencontrée, par compilation, ou par exécution.

Exemple 1: *Le script ci-dessus, enregistré **dans l'éditeur.***

```
25 for i in range(20):
26     if i%3==0:
27         if i!=0:print('*  ',end='')
28     elif i+1>=20: print('i)
29     else:print(i,end='  ')
```

*Exécution **dans l'interpréteur** et signalement d'une erreur.*

```
>>> (executing lines 25 to 29 of "Codage_base_b TP 02.py")
  File "E:\tests_py\Codage_base_b TP 02.py+24", line 4
    elif i+1>=20: print('i)
                          ^
SyntaxError: EOL while scanning string literal
```

Erreur signalée. Arrêt du programme

```
25 for i in range(20):
26     if i%3==0:
27         if i!=0:print('*  ',end='')
28     elif i+1>=20:print(i)
29     else:print(i,end='  ')
```

Script corrigé

```
>>> (executing lines 25 to 29 of "Codage_base_b TP 02.py")
1  2  *  4  5  *  7  8  *  10  11  *  13  14  *  16  17  *  19
```

Exemple 2 : *Script à exécuter **dans l'éditeur***

```
51 a,b=90,45
52 a=a*b
53 print('a=',a,'    b=',b)
54 a=a+b
55 print(c)
```

Exécution et détection d'erreur en cours d'exécution

```
>>> (executing lines 51 to 55 of "passage par
a= 4050     b= 45
Traceback (most recent call last):
  File "C:\Users\randr\Desktop\passage parame
<module>
    print(c)
NameError: name 'c' is not defined
```

Début d'exécution

Erreur signalée. Arrêt du programme.

E. PYTHON et ALGORITHMIQUE DE BASE : TYPES DE DONNEES

Les données manipulées par Python doivent être de types bien définis, ceux reconnus par lui.
Les types élémentaires habituels définis par l'Algorithmique, sont utilisés par Python : int (ENTIER), float (REEL), str (CHAINE DE CARACTERES), bool (BOOLEEN), les fichiers etc….
On peut ajouter le type complex (les nombres complexes), type élémentaire en Python.
D'autres types intégrés à Python, très utilisés, seront vus plus tard (list, tuple, dict, set, etc…..)

1. Manipulation de base : connaitre le type d'un objet utilisé et son adresse.

Commandes : **type(<objet>) :** retourne le type de cet objet.
id(<objet>) : retourne l'adresse de l'objet.

Exemple : type(5) retourne **<class 'int'>, qui indique le type int (ENTIER) de 5.**
id(a) retourne l'adresse de a

```
>>> x=9
>>> type(x)
<class 'int'>
>>>
```

x est un (objet) entier

```
>>> x=9.5
>>> type(x)
<class 'float'>
>>>
```

x est un (objet) réel

```
>>> a=9
>>> id(a)
1350724880
>>> s="merci"
>>> id(s)
3152616878744
>>>
```

2. Les types numériques:

a. Type int

Constantes : les entiers relatifs
Opérateurs : +,-,/ (division réelle),*, ** (exponentiation), % (modulo : reste d'une division entière), // (division entière), divmod (a,b) qui retourne un couple (q,r) formé par le quotient et le reste de la division entière de a par b.
Exemples :

```
>>>
>>> 34%3
1
>>> divmod(34,3)
(11, 1)
```

```
>>> 34//3
11
>>> 33/3
11.0
>>>
```

b. Type float

Constantes : les non entiers
Opérateurs : +,-,/ (division réelle),*, ** (exponentiation), // (division entière) , %, divmod().

```
>>> 32.0/5.0
6.4
>>> 32.0//5.0
6.0
>>> 32.0/5
6.4
```

```
>>> 32.0%5.0
2.0
>>> divmod(32.0,5.0)
(6.0, 2.0)
>>>
```

c. Type complex

Cela concerne les nombres complexes. Le nombre complexe i ($i^2 = -1$) est noté j.
Constantes : les nombres complexes a=u+jv, où u et v sont des réels, et j tel que $j^2=-1$.
Exemples :
x=3+j est un nombre complexe. On écrit aussi x=complex(3,1)

```
>>> x=3+1j
>>> type(x)
<class 'complex'>
>>>
```

5+2*j n'est pas un nombre complexe, de même 1+j.
Par contre 1+1j est un complexe.

Opérateurs : + , - , * , /, ** .
Exemple 1 : a=1-2j. b*a=(4+3j)*(1-2j)=10-5j
Exemple 2: b= 4+3j
b.real = 4 (partie réelle) ; b. imag=3 (partie imaginaire) ;
b.conjugate() = 4 – 3j (complexe conjugué : une méthode agissant sur l'objet b)
abs(b)=5 (module)

d. Le type bool:

Constantes : **True, False** (attention à l'écriture : T et F sont en majuscule)
Opérateurs : **not, and, or** (**XOR** n'existe pas)
Exemple : x un booléen
Exercice : Donner l'ir en Python et compléter le tableau ci-dessous

```
>>> x=True
>>> type(x)
<class 'bool'>
```

Table des vérités :

P	Q	Non P	P et Q	P ou Q
V	V	F	V	V
V	F	F	F	V
F	V	V	F	V
F	F	V	F	F

Remarque : **1 (ou tout entier non nul) est True et 0 est False. Par défaut, True est 1**

Exemple :

```
>>> a=8
>>> a+(5>3)
9
```

Explication : (5>3) est un booléen et il est True (=1)
a+(5>3) = a+1 = 8+1 = 9
a+True = 9
int(True) = 1

3. Le type str

En Python, **il n'y a pas de type caractère**. **Tout est chaine de caractères.**
Syntaxe : ''******* '' ou '********' ou " " " ******** " " " ou ''' ******* ''':
Exemples : '5' et ''5'' représentent la même chaine, de même ''merci'' et 'merci'.
" " " Ceci est un chaine
qui tient sur plus d'une ligne
" " "

```
>>> x="c'est le matin"
>>> type(x)
<class 'str'>
```

Opérateurs : concaténation (+), extraction de sous-chaines (**slicing**) – longueur (**len()**), multiplication par un entier positif etc…. que l'on verra plus loin
Exemple : x est une chaine de caractères :

4. Le type None : absence de valeur : **None Type** : une seule constante **None**

5. Le transtypage (ou cast) :

Exemple :
Utiliser **momentanément (** ou **définitivement)** une donnée d'un type donné (ex **int**) comme une donnée d'un autre type (ex type **float** ou type **str**):

Exemple a=8 , de type int b=float(a), c = str(a) :
float(a) utilise a comme un réel, et on affecte le résultat à b. b est de type float, mais a garde son type int. De même pour c=str(a).
Par exemple, ‘’mardi ’’+c donne la chaine ‘’ mardi 8’’ :
L’opérateur de transtypage est : <type>(objet à convertir)

F. PYTHON et ALGORITHMIQUE DE BASE : LES INSTRUCTIONS DE BASE

1. Structure globale d’un programme Python

Comme on l’a vu en Algorithmique, un programme est constitué d’un programme principal et de sous-programmes.
La programmation sous Python est fortement modulaire. Tout est fonction en Python, et le programme principal ne sert qu’à lancer des fonctions.
Dans cette partie, on se contentera de traduire un algorithme avec les mots de Python,

2. Les commentaires :

Toute ligne **commençant par #** est une ligne de commentaire
Tout groupe de lignes commençant par ‘’ ‘’ ‘’ et se terminant par ‘’ ‘’ ‘’ est un commentaire sur plusieurs lignes
Exemple :
ceci est un commentaire, qui se termine par un retour chariot.

```
>>> a=2
>>> a
2
>>> a=25
>>> a
25
>>> # a=36
>>> a
25
```

a=25 a été pris en compte.

a=36 n’a pas été pris en compte.

3. Affectation-Référencement.

a. Syntaxe d’une affectation: le signe =

a=8 # on « affecte » à a la valeur 8 qui est de type int
le nom a doit être à gauche et il reçoit. Ce que l’on reçoit est à droite (8).
5=y n’a pas de sens.

Les règles d’affectation vues en Algorithmique sont celles qui sont appliquées.

b. Ce qui se passe réellement

En Algorithmique, une variable, avant d’être utilisée, doit avoir été déclarée. Cela consiste à RESERVER un emplacement – mémoire, qui a sa propre adresse, à qui on donne le nom prévu de la variable, et qui est conditionné pour recevoir un type de données bien précis.
On a l’habitude d’assimiler une variable à une boîte située à une adresse donnée en mémoire, à qui on donne un nom, et pour qui on précise un type de données qu’elle peut contenir.
Exemple : VARIABLE ***note*** : REEL
a: ENTIER
On réserve deux emplacements en mémoire : ***note*** d’adresse F003 prévue pour ne contenir que des réels, et ***a*** d’adresse D56A prévu pour ne recevoir que des chaines de caractères.

Cette manière d'attribuer un type (définitif) à un identificateur est qualifiée de typage statique (« une fois fixé, on ne peut plus rien changer »).

En Python, en général il n'y a pas de déclaration explicite de variables. **Une variable doit être initialisée avant utilisation, et c'est lors de cette initialisation que son type est donné**.
Une variable est uniquement un identificateur d'objet, qui référence cet objet, et c'est l'objet référencé qui lui donne son type. Elle peut alors référencer plus tard un autre objet, du même type ou d'un autre type.

a=8 puis a='chaine' est une séquence non contradictoire

En Python, une affectation est un référencement d'objet par un identificateur, qui s'accompagne d'un typage et d'une attribution d'adresse.
Ce typage de variable pratiqué par Python est appelé typage dynamique, en opposition au typage statique vu plus haut.

c. Affectations multiples

- Les affectations multiples sont possibles :
  Exemple :
  a=8 ; b=9 ; c=12 peut être écrit : a,b,c=8,9,12
- Le script :
  a,b=8,9
  c=a ;a=b ;b=c équivaut à a,b=b,a
  (échange des valeurs de a et de b)

4. Les « Entrées/Sorties » (E/S ou I/O)

a. Entrées : Tout ce qui est saisi au clavier est chaine de caractères.

Commande utilisée : input()
Syntaxe : x= input(" message ") pour saisir x qui est de type str.
« message » est là pour préciser à l'utilisateur ce qui est attendu à la saisie. Sinon, input() attend une saisie au clavier sans avertir.

Exercice
Exemple : saisie de la valeur de a. Tester les commandes suivantes :

```
>>> a=input()
>>> a=input(" valeur pour a ? ")
>>> a=input(" a ?") puis >>> a=int(a)
>>> a=int(input(" a ?")) .
```

b. Sorties :

Commande utilisée : print()
Syntaxe : **sortie simple** : **print(a, b, c, ……..) :** a, b et c sont affichés l'un après l'autre, avec un espace de séparation automatique.

Exemples :

```
>>> a=9
>>> b="abcd"
>>> print(a,b)
9 abcd
```

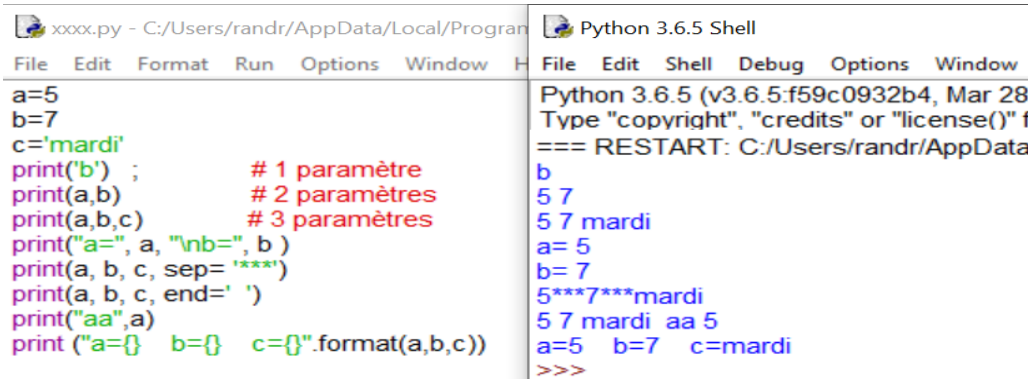
```
>>> a=9
>>> b="abcd"
>>> print("a=",a," b=",b)
a= 9  b= abcd
```

sortie « formatée », utilisant la « méthode » format():
print(canevas.format(a,b,c)),
où canevas est une chaine du type " ……. {} ……..{} …….{} … " (avec autant d'accolades {} que d'objets à afficher)

```
>>> a=9
>>> b="abcd"
>>> print("a={}      b={}".format(a,b))
a=9      b=abcd
```

Ici, le canevas est la chaine de caractères : '' a={} b={} ''

Exemples



Exercice

Commenter les cinq dernières sorties ci-dessus.

5. Les instructions conditionnelles

a. Opérateurs de comparaison, d'appartenance :

== (égal) ; != (différent) ; > ; >= ; < ; <= opérateurs usuels

is (vérifie si ce sont réellement les mêmes objets, deux objets identiques : même contenu et même adresse)

is not (objet non identique à)

in et **not in** qui vérifie si un objet apparait ou pas dans un 'ensemble' donné d'objets.

Exemples :

```
1  a=9
2  c=a
3  b=7
4  print('**apres initialisation de a, b et c**')
5  print('id(a)= ',id(a))
6  print('id(c)= ',id(c))
7  print('id(b)= ',id(b))
8  print('c is a?  ',c is a)
9  print('c==a? ', c==a)
10 b=b+c
11 d=float(2*a)
12 e=18.0
13 print('**apres affectations de a, b et c **')
14 print('id(b=b+c)= ',id(b))
15 print('id(d=float(2*a))= ',id(d))
16 print('id(e)= ',id(e))
17 print('d is e?  ',d is e)
18 print('d==e? ', d==e)
```

```
>>> (executing lines 1 to 18 of "<tmp 1>")
**apres initialisation de a, b et c**
id(a)=  1868721424
id(c)=  1868721424
id(b)=  1868721360
c is a?   True
c==a?  True
**apres affectations de a, b et c **
id(b=b+c)=  1868721648
id(d=float(2*a))=  2908901089568
id(e)=  2908901089616
d is e?   False
d==e?  True
```

- a et c sont identiques (mêmes adresses , mêmes valeurs)
- d et e ont la même valeur mais n'ont pas les mêmes adresses

Par exemple : a= ''EZ''

```
>>> a="EZ"
>>> b=6
>>> c=a
>>> d=b
```

```
>>> id(a)
3152616788296
>>> id(c)
3152616788296
>>> id(b)
1350724784
>>> id(d)
1350724784
>>>
```

b. syntaxe du if classique:

if *condition* **:** **# : obligatoire (annonce un bloc d'instructions)**

'' '' '' tout ce qui est prévu pour être exécuté dans le cas où la condition est satisfaite doit être **indenté** (décalé de (4) espaces (convention généralement adoptée) par rapport à la colonne du i du if.

'' '' ''

else : # encore : à ne pas oublier. else est **facultatif**

……..

Le ‘ :’ annonce d’un bloc, est équivalent à un DEBUT ….. FIN, ou à un POUR …. FINPOUR, ou à un TANTQUE….FINTQUE, en Algorithmique.

Conséquence : Les colonnes des débuts d’instruction sont très importantes car doivent tenir compte de l’indentation.
Exemple 1 :

```
15 a=8
16 if a>4:
17     print('oui')  # indentation correcte
18     j=0
19   b=11      # mauvaise colonne pour b
20 c=90.9
```

```
>>> (executing lines 15 to 20 of "comparaison python _ C.py")
  File "C:\Users\randr\Videos\comparaison python _ C.py+14", line 5
    b=11
    ^
IndentationError: unindent does not match any outer indentation level
```

Exemple 2

```
a=45
b=int(input('donner un entier '))
if b<=a:
   print('il est plus petit ou égal')
   c=0
else: print('il est plus grand')
```

Pour if , le bloc contient 2 instructions. On est obligé d’indenter
Pour else, il n’y a qu’une instruction. On peut rester sur la même ligne.

Principe :
Sauf si le bloc se réduit à une seule instruction, l’indentation est obligatoire.
Sortir d’un bloc (bloc if ou bloc else) revient à se positionner à la colonne du i du if concerné ou à celle du e du else concerné.

Exercice : Soit le code suivant :

```
if 2 :
    print(‘oui’) #indentation
else :
    print(‘condition non ok ‘) #indentation
    print(‘’non’’)
```

```
>>> if 2:
...     print("OUI")
... else:
...     print("NON")
...
OUI
```

La condition est satisfaite (2 est vrai car 2 est non nul). On affiche donc OUI

c. Instructions conditionnelles imbriquées

```
if condition1 :
        ……
else :
        if condition2 :
        ……
        else :
        ……
```

qui peut aussi être écrit :

```
if condition1 :
        ……
elif condition2 :
        ……
else :
        ……
```

Remarque. Quand on utilise des **elif**, il doit y avoir un unique **else**, exécuté lorsqu'aucune des conditions de ces elif ne sont satisfaites.

Exemple

```
97  a=int(input('donner un age'))
98  if a<=12: print('enfant')
99  elif a<=17: print('adolescent')
100 elif a<=60: print('adulte')
101 else: print('vieux')
102
```

Lors d'une condition comportant des inégalités séparées par OU ou ET, toutes les inégalités ne sont pas nécessairement testées.
Voici quelques situations :

```
>>> 3>2, 9==9.,0==0j, 'ADE'>'B'
(True, True, True, False)
```
9==9. est vrai car 9 est converti en float avant comparaison

```
>>> 1/4>1/2 and 4>3 and 2>1/0
False
```
1/4>1/2 est FAUX, donc les autres inégalités ne sont plus vérifiées la division par zéro n'a pas été constatée

```
>>> 1/4<1/2 and 4>3 and 2>1/0
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ZeroDivisionError: division by zero
```
1/4<1/2 est VRAI, de même 4>3 est VRAI, donc la division par zéro a été constatée

```
>>> 1/4<1/2 or 4>3 or 2>1/0
True
```
Ici c'est un OU, comme 1/4<1/2, aucun autre test n'a été effectué. La division par zéro n'a pas été décelée.

Remarque : L'instruction SUIVANT choix en Algorithmique **n'existe pas** en Python**.**

6. Les boucles

a. La boucle TantQue :

Syntaxe : **while** *condition* **: # indentation obligatoire si plus d'une instruction**

……

ligne sortie de la boucle (colonne du w de while)

```
>>> a=2

>>> while a<10:
...     a=int(input('redonner a>=10:    '))
...
redonner a>=10:    9
redonner a>=10:    7
redonner a>=10:    11

>>>
```

On sort de la boucle lorsque la condition cesse d'être vérifiée.

b. La boucle Pour

Ce type de boucle est d'une conception et d'une utilisation plus larges que celles que l'on voit ici en première apparition, où on ne fait que traduire l'instruction algorithmique, avec un entier i compteur de boucle :

Syntaxe : for *i* in range(*fin* **) :** # indentation obligatoire si plus d'une instruction.

……

Exemple :

```
>>> for i in range(6):
...     print(i)
...
0
1
2
3
4
5
```

ATTENTION : i in range(6) signifie : i dans l'intervalle des entiers de 0 à 6-1=5. **range(n)** est un objet qui accepte d'être parcouru « d'un bout à l'autre, le dernier étant exclu ».
Si on veut aller de 0 à 6, il faudrait écrire : for i in range(7)

Syntaxe complete: for i in range(k, m, h)

- S'il y a 3 arguments k, m et h, k est l'indice de début, m l'indice de fin (exclu), et h le pas.
- S'il y en a deux, le premier est l'indice du début, le second l'indice de fin (exclu)
- S'il n'y a qu'un seul, c'est l'indice de fin (exclu). 0 est l'indice de début.

```
>>> for i in range(2,20,3):
...     print(i,end=' ')
...
2 5 8 11 14 17
```

c. Sortie « avant l'heure » d'une boucle:

On peut sortir « avant l'heure » d'une boucle **for** ou **while** en utilisant le mot-clé **break.**
L'instruction **continue** permet de passer directement au tour suivant dans une boucle.

7. Les exceptions

Le couple d'instructions **try - except**
Prenons le script suivant qui saisit et affiche une série de 10 entiers.
Si au cours de la saisie, on s'est trompé et on a saisi une chaine de caractères, cela provoque une erreur qui a comme effet l'arrêt (inattendu) du programme.

```
>>>
>>> for i in range(10):
        x=int(input('donner un entier x:    '))
        print(x)

donner un entier x:    14
14
donner un entier x:    25
25
donner un entier x:    4
4
donner un entier x:    h
Traceback (most recent call last):
  File "<pyshell#8>", line 2, in <module>
    x=int(input('donner un entier x:    '))
ValueError: invalid literal for int() with base 10: 'h'
>>>
```

On dit **qu'il lève une exception** et affiche un message d'erreur.
Cette erreur est signalée ici à la dernière ligne : ValueError (erreur de valeur). Et la ligne où cette erreur est détectée est rappelée à la ligne précédente : x=int(……..)

Regardons maintenant le script :

```
i=0
while i<5:
    i=i+1
    try:
        x=int(input('donner un entier n:    '))
        print(x)
    except ValueError:
        print('*****************   Attention  erreur de saisie')
        i=i-1
```

Il donne ceci :

```
 RESTART: C:/Users/randriflo/AppData/Local/Programs/
donner un entier n:    5
5
donner un entier n:    8
8
donner un entier n:    9
9
donner un entier n:    p
*****************   Attention  erreur de saisie
donner un entier n:    5
5
donner un entier n:    1
1
```

Fonctionnement :
Le bloc d'instructions situé entre **try** et **except** est exécuté.
Si tout se passe bien, **except** est ignoré
En cas de saisie qui provoque une erreur dans le système dont la valeur est celle indiquée **par except**, ici **ValueError**, la partie **except** est exécutée, puis la boucle continue.
Si ce n'est pas le type d'erreur attendu, on essaie de voir la présence d'un **try** extérieur.
Ce script résout donc le problème d'erreur de type lors d'une saisie au clavier, qui provoque une erreur.

Quelques types d'erreurs : **KeyError, IndexError**, **ImportError**, **IOError**, **TypeError, NameError**, **SyntaxError** ….

8. Exemples de traduction :

Exemple - Exercice 1.
Soit l'algorithme suivant qui saisit un réel R qui doit être positif ou nul, et calcule et affiche l'aire du cercle de rayon R. On a pris PI = 3,14

```
PROGRAM aire
VARIABLE  R, aire : REEL
DEBUT
        AFFICHER(‘’ Donner le rayon’’) ; LIRE(R)
        TANTQUE R<0 FAIRE
                AFFICHER(‘’ Redonner le rayon R>=0 ’’) ; LIRE®
        FINTQUE
        aire ← 3,14*R*R
        AFFICHER(‘’ L’aire est :  ‘’, aire)
FIN.
```

Traduire cet algorithme en Python et le tester.

Exemple – Exercice 2 :

1. Ecrire un algorithme qui saisit deux valeurs des variables réelles x et y, les affiche sous le format ‘’ x contient : ‘’ et ‘’ y contient : ‘’, et échange le contenu de ces variables puis les affiche à nouveau.
2. Traduire cet algorithme en Python et le tester.
3. Utiliser une caractéristique du langage Python pour écrire un programme plus court.

Exemple - Exercice 3 : Traduire en Python l'algorithme ci-dessous :

```
PROGRAM  moyenne
VARIABLE       i, saisie, som, n : ENTIER
```

```
DEBUT
        AFFICHER(‘’ Donner n’’) ; LIRE(n)
        TANTQUE (n < 1 ) FAIRE
                AFFICHER(‘’Redonner n’’) ; LIRE(n)
        FINTQUE
        som ← 0
        POUR i ← 1 A n FAIRE
                AFFICHER(‘’ Donner un entier ’’) ; LIRE(saisie) ; som ← som+saisie
        FINPOUR
        AFFICHER(‘’moyenne=’’,som/n)
FIN.
```

Exemple - Exercice 4

a. Reprendre l’algorithme, le transformer pour qu’il détermine en même temps le plus petit entier saisi ainsi que le dernier numéro sous lequel il a été saisi.
b. Gérer les éventuelles exceptions dues à une saisie d’une valeur de saisie qui n’est pas un entier à l’aide d’un try – except.

Exemple - Exercice 5

Ecrire un script Python qui saisit un entier n qui doit être supérieur à 10, saisit une série de n réels strictement positifs et affiche à la fin la plus grande valeur saisie avec son numéro de saisie.

Exemple - Exercice 6

Ecrire une script Python qui saisit un entier et cherche à savoir s’il est premier.

G. FONCTIONS, PROCEDURES, METHODES.

On ne va voir ici qu’une simple introduction des fonctions et procédures, en comparaison avec ce qui a été vu en Algorithmique.

Pour l’instant, il faut comprendre par méthode une fonction ou une procédure au sens de l’algorithmique.

Pour :

- une meilleure organisation,
- une possibilité d’une réutilisation,
- une amélioration de la maintenance,
- rendre le programme principal le plus simple possible,

on utilise la technique de **programmation modulaire**, une programmation se basant le plus possible sur des modules, ou bibliothèques.

Un module est un regroupement de plusieurs fonctions ou procédures ou méthodes conçues séparément et qui peuvent être réutilisées. Il peut aussi contenir des paramètres spécifiques à ce module.

Pour la suite, le terme **fonction** signifiera fonction, procédure, ou méthode.

1. Généralités

Au sens algorithmique, une fonction ou procédure est caractérisée par :

- son nom et ses paramètres éventuels
- l’espace mémoire qui lui est réservé (géré par le système d’exploitation).
- ce qu’elle retourne (ou pas)

Exemples :
sauter(k) : une procédure dont le rôle est sauter k lignes à l'affichage. Elle a comme paramètre le nombre k, et elle ne retourne rien. On écrira en Algorithmique :

```
PROCEDURE  sauter(k : ENTIER)
        VARIABLE i : ENTIER
        DEBUT
            POUR i ALLANT DE 1 A k FAIRE
                AFFICHER(" ")
            FINPOUR
        FIN sauter
```

afficher() : procédure écrite pour afficher le mot ''merci''. Sans paramètre et ne retourne rien.

```
PROCEDURE  afficher()
        DEBUT
            AFFICHER (''merci'')
        FIN afficher
```

somme(a, b, c) : fonction qui calcule et retourne la somme de trois entiers a, b et c.

```
FONCTION  somme(a,b,c : REEL) :REEL
        DEBUT
            Somme=a+b+c
        FIN somme
```

2. Fonction en Python

En Python, on appelle **fonction** une fonction ou une procédure au sens algorithmique.
Les résultats à retourner au programme appelant le sont avec la commande **return**
Une fonction peut : - ne rien retourner,
- retourner un ou plusieurs résultats.

Syntaxe :

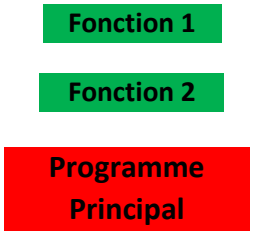
```
def  <nom_fonction>(<paramètres éventuels>) : # : provoquant une indentation
    ……..
    return a ( ou return a,b…..) ( facultatif )
```

Remarques :

Tout script en Python devrait être une fonction

a. **def** est un mot clé de déclaration d'une fonction.
b. L'en-tête ne prévoit **aucune déclaration** de type des paramètres et doit se terminer par un **:**
c. L'instruction **return** provoque une sortie de la fonction et transmet éventuellement un ou des résultats au programme appelant.
   Remarque : même s'il y a plusieurs instructions return dans une même fonction, une seule d'entre elles sera utilisée à chaque fois.

3. Nouvelle structure d'un programme Python



Les fonctions peuvent s'appeler entre elles et elles peuvent toutes être appelées par le programme principal. **On écrit d'abord toutes les fonctions, avant le programme principal**

4. Appel d'une fonction (au sein du programme principal)

On appelle une fonction par son nom en précisant les paramètres éventuels de l'appel.
Exemple : La fonction calc() suivante est à 3 paramètres : a, b et c

```
def calc(a,b,c) : return a+b*c
```

A l'appel somme(x,y,z), a prend la valeur de x, b celle de y et c celle de z, **dans cet ordre**.
On peut aussi effectuer l'appel somme(c=z, a=x, b=y), et là, l'ordre n'est pas important.

```
104 # fonction somme
105 def calc(a,b,c):
106     return a+b*c
107
108 #programme principal
109 x=int(input('donner a:    '))
110 y=int(input('donner b:    '))
111 z=int(input('donner c:    '))
112 print(calc(x,y,z))
113 print(calc(c=z, a=x, b=y))
```

```
>>> (executing lines 104
donner a:    12
donner b:    10
donner c:    20
212
212
```

5. Passage des paramètres.

En algorithmique, on a vu qu'il y a deux types de passages des paramètres :

- **le passage par valeur** : le sous-programme travaille avec des copies des paramètres. Tout éventuel changement effectué sur un paramètre à l'intérieur de la fonction n'affecte pas la valeur de ce paramètre pour le programme appelant.
- **le passage par adresse** : le sous-programme travaille directement à l'adresse du paramètre concerné. Tout changement de valeur de ce paramètre à l'intérieur du sous-programme est effectif à l'extérieur de celui-ci.

En règle générale, en Python, les paramètres sont référencés à des noms locaux de la fonction. Il n'y a donc pas de copie de données comme en C, lors de passage des paramètres par valeurs. Il s'agit uniquement de référencement. Prenons quelques exemples :

```
1  def essai01(a,b):
2      a=a+2
3      b=b*2
4      return a+b
5
6  a=3
7  b=5
8  res=essai01(a,b)
9  print('a_apres=',a,'  b_apres=',b)
10 print('resultat= ',res)
```

```
>>> (executing lines 1 to 10 of "<tmp 2>")
a_apres= 3   b_apres= 5
resultat=  15
```

```
1  def essai02(a,b):
2      a=a+'cours'
3      print('a_int=',a,'  b_int= ',b)
4      return a+b
5
6  a='le '
7  b=" d'info"
8  print('a_avant=',a,'  b_avant=',b)
9  res=essai02(a,b)
10 print('a_apres=',a,'  b_apres=',b)
11 print('resultat= ',res)
```

```
>>> (executing lines 1 to 11 of "<tmp 2>")
a_avant= le    b_avant=  d'info
a_int= le cours    b_int=   d'info
a_apres= le    b_apres=  d'info
resultat=  le cours d'info
```

Dans les deux cas, le passage se fait par valeur. **Mais ce n'est pas toujours le cas. On verra des situations pour lesquelles le passage se fait par adresse. Ceci est lié à la nature des données**
Les paramètres de type int, float, bool, complex, str sont passés par valeur.

6. Variables locales, variables globales pour une fonction

a. Définitions :

Variable **locale** pour une fonction : variable initialisée à l'intérieur de cette fonction, dont la durée de vie est celle de l'exécution de la fonction, et elle n'est pas visible hors de celle-ci.

Exemple :

Une erreur survient à l'exécution du script ci-dessous, à la dernière instruction.

```
29 def essai(m,p):
30     k=444
31     y=0.9
32     m=m+a+k
33     p=2*p+b+y
34     return m,p
35 a,b=1,1
36 d,g=7,9
37 print(essai(d,g))
38 print('d=',d,  ' *** ','g=',g)
39 print('k=',k,  ' *** ','y=',y)
40
41  k et y sont des variables locales à la fonction. Elles ne sont pas visibles en
42  dehors de la fonction. C'st ce qui a provoqué l'erreur.
```

m et p sont des paramètres
k et y sont des variables locales

a,b sont des variables du programme appelant (ici le programme principal). Elles sont vues par la fonction. Ce sont des variables globales

Une variable déclarée à l'intérieur d'une fonction peut être visible à l'extérieur de celle-ci en la rendant globale. On utilise pour cela l'instruction **global.**

```
17 def  essai03(a,b):
18     global s
19     print('s int=',s)
20     t=s+a+b
21     s=2*s
22     return t
23 s=9
24 d=essai03(5,6)
25 print('valeur retournée=',d)
26 print ('s_apres=',s)
```

```
>>> (executing lines 17 to 26 of "<tm
s int= 9
valeur retournée= 20
s_apres= 18
```

Exemple

t est une variable locale. a et b sont les paramètres de la fonction, et s est une variable locale, rendue globale. **s est vue par la fonction (s est connue à l'intérieur de la fonction) , et vue par le programme appelant (la valeur de s qui a changé est reconnue de l'extérieur par le programme appelant.)**

Exercice : Quels sont les affichages provoqués par le script ci-dessous ? Justifier.

```
42 def essai9(a,b):
43     global x,y
44     y=9
45     x=x+5
46     print('y int=',y,'    x int=',x)
47     m=y+a+b
48     return m
49 x=10
50 print(' la fonction retourne: ',essai9(5,6))
51 y=y+10
52 print('y ext=',y,'    x ext=',x )
53 print(' la fonction retourne: ',essai9(10,60))
54 y=y+10
55 print('y ext=',y,'    x ext=',x)
56 print('m=',m)
```

7. Fonction déclarée à l'intérieur d'une fonction :

Ceci est possible en Python. Mais il faut bien faire attention aux visibilités des variables

Exemple 1 : Tout se passe correctement.

```
def fonc1(a,b,c):
    a=a+b+c
    def fonc2(u,v):
        return 2*u+3*v
    b=fonc2(a,c)
    print('int a=',a,  '  b= ',b)
    return a,b
print(fonc1(1,2,3))
```

```
= RESTART: C:/Users/randriflo/
py =
int a= 6    b=  21
(6, 21)
>>>
```

Exemple 2 : Tout se passe correctement.

```
def fonc1(a,b):
    print('on entre fonc1')
    a=a+b
    c=300
    def fonc2(m,n):
        print('on entre das fonc2')
        t=m+n+c
        #c=900
        print('t_int_fonc2=',t)
        return t
    return a+fonc2(b,c)
p,q=7,9
print('fonc1=',fonc1(p,q))
```

```
================== RESTART: C:/
on entre fonc1
on entre das fonc2
t_int_fonc2= 609
fonc1= 625
>>>
```

Exemple 3 : Erreur constatée car c, initialisée à 300, est réinitialisée à l'intérieur de fonc2(). fonc2() y voit une utilisation de c avant initialisation. Cela provoque une erreur.

```
def fonc1(a,b):
    print('on entre fonc1')
    a=a+b
    c=300
    def fonc2(m,n):
        print('on entre das fonc2')
        t=m+n+c
        c=900
        print('t_int_fonc2=',t)
        return t
    return a+fonc2(b,c)
p,q=7,9
print('fonc1=',fonc1(p,q))
```

```
================== RESTART: C:/Users/randr/Desktop/dseser.py =====
on entre fonc1
on entre das fonc2
Traceback (most recent call last):
  File "C:/Users/randr/Desktop/dseser.py", line 13, in <module>
    print('fonc1=',fonc1(p,q))
  File "C:/Users/randr/Desktop/dseser.py", line 11, in fonc1
    return a+fonc2(b,c)
  File "C:/Users/randr/Desktop/dseser.py", line 7, in fonc2
    t=m+n+c
UnboundLocalError: local variable 'c' referenced before assignmer
```

8. Fonction avec des paramètres par défaut

Par exemple, soit la fonction suivante : def calc(a, b, c=12, e=25) : ……

c et e sont **des paramètres avec des valeurs par défaut** : c=12 et e=25

L'appel calc(5,8) équivaut à calc(5,8,12,25)

L'appel calc(5,8,e=87) équivaut à calc(5,8,12,87)

L'appel calc(5,8,e=87,c=69) équivaut à calc(5,8,69,87)

L'appel calc(5) est interdit.

Remarque : La déclaration **def calc(a, b, c=14, k)** est interdite.

Dès qu'un paramètre est déclaré avec une valeur par défaut, tous les autres paramètres qui sont déclarés après ce paramètre doivent aussi avoir des valeurs par défaut.

9. Fonctions récursives

Une fonction telle qu'on l'a vue jusque là, une fois appelée, retourne des résultats si elle en a à retourner, en utilisant return. Un return marque donc la session de la fonction.

Définition :

Une fonction récursive est une fonction capable de s'appeler elle-même.

Python accepte la récursivité au sein ses fonctions.

Exemples :

(i) Calcul de factoriel

Version récursive

Elle utilise la relation récursive :

(n+1) ! = (n+1) n !

```
33 def fact_rec(n):
34     if n<2:return 1
35     else: return n*fact_rec(n-1)
36 #prog principal
37 n=int(input('donner une entier n:  '))
38 print(str(n)+'!=',fact_rec(n))
```

Version itérative

Cette version calcule directement le

```
def fact_iter(n) :
        fact=1
        if n>1:
                for i in range(2,n+1) :
                        fact=fact*i
        return fact
```

Une fonction récursive doit avoir un point d'arrêt : ici, dès qu'on a à calculer 1! ou 0!. Puis, on remonte la pile pour obtenir le résultat final.
Pour bien comprendre le mécanisme, il suffit d'exécuter la version suivante, où on a inséré des marquages d'étapes :

```
1  def  fact_rec(n):
2      global i,p
3      if n<2 :
4          if i>1:
5              print('on entre avec n='+str(n)+' on trouve 1 puis on
   remonte pour calculer '+str(p)+'!')
6          else:print('on entre avec n=',n,' on trouve 1')
7          return 1
8      else :
9          print('on est entré avec n='+str(n)+' on calcule
   '+str(n)+'*'+str(n-1)+'!')
10         i=i+1
11         return n*fact_rec(n-1)
12
13 n=int(input('donner n: '))
14 i=1
15 p=n
16 print(str(p)+'!=',fact_rec(n))
```

Essayer de bien comprendre tout ce qui est écrit dans cette fonction.
Pourquoi p et i ont-elles été prises globales? Justifier les instructions print() utilisées.

(ii) Suite de Fibonacci:
Considérons par exemple la suite (U_n) : $U_0=U_1=1$; $U_n = 3*U_{n-1} + U_{n-2}$ pour n>=2 :

```
def fibo(n) :
        if n<2 : return 1
        else : return 3*fibo(n-1)+fibo(n-2)    ( à tester )
```

10. Fonctions lambda en Python

a. Fonctions lambda : fonction simple qui se définit à l'aide d'une seule instruction.
Rappel syntaxes : <nom>=lambda <liste des variables> : <expression>
Exemples :

(i) f=lambda x : x**2+1 définit la fonction $x \longrightarrow x^2+1$

(ii) g=lambda x,y : 2x+3y définit la fonction $(x,y) \longrightarrow 2x+3y$

(iii) h=lambda x : (2*x, 3*x**2-1) définit la fonction $x \longrightarrow (2x,\ 3x^2-1)$

```
>>> f=lambda x:x**2+1

>>> f(1)
2
```

```
>>> g=lambda x,y: 2*x+3*y

>>> g(1,2)
8
```

```
>>> h=lambda x:(2*x,3*x**2-1)

>>> h(3)
(6, 26)
```

b. Exemples - Exercices:

(i) Donner la structure générale d'un code comprenant le programme principal et au moins une fonction en Python

(ii) Qu'affichent les programmes suivants :

Prog 1 :

```
x=0
def  f(x) :
    return x*x+2
print('f(1)=',f(1))
print('f(x+1)=',f(x+1))
print('x=',x)
```

Prog 2 :

```
def  dessin(n):
    i=1
    while i<=n:
        print(i*'*')
        i=i+1
dessin(4)
dessin(6)
```

(iii) Réécrire la fonction f ci-dessus comme une fonction lambda.

(iv) Que calcule la fonction lambda fifi : **fifi=lambda x : 1 if x<2 else 3*fifi(x-1)+fifi(x-2)?**

(v) Que calcule la fonction lambda fact : **fact=lambda x : 1 if x<2 else x*fact(x-1) ?**

11. Fonctions et noms de variables en Python

Soient les scripts :

```
lambda=7
print('lambda=',lambda)
```

On signale une erreur de syntaxe. Le mot lambda est un **mot-clé et son utilisation est protégée**.

Liste des mots réservés:

and	**assert**	**break**	**class**	**continue**	**def**	**del**	**elif**	**else**	**except**
exec	**finally**	**for**	**from**	**global**	**if**	**import**	**in**	**is**	**lambda**
not	**or**	**pass**	**print**	**raise**	**return**	**try**	**while**	**Yield**	

```
26 a=[6,8,9,12]
27 n=len(a)
28 print('n=',n)
29 len=18
30 a=2*a
31 m=len(a)
```

```
>>> (executing lines 25 to 32 of "Codage_base_b TP 02
n= 4
Traceback (most recent call last):
  File "E:\tests_py\Codage_base_b TP 02.py", line 31,
    m=len(a)
TypeError: 'int' object is not callable
```

Pour le second script, len() est une fonction de Python, chargée avec Python, mais **len n'est pas un mot clé protégé**. Son utilisation comme identificateur n'est pas directement interdite. Tant qu'on ne cherche pas à utiliser la fonction len(), tout se passera normalement.

Par contre, dès qu'on essaie de récupérer la fonction len(), son accès devient impossible. Elle est masquée par l'identificateur len. Ce qui a provoqué l'erreur à la ligne 31.

Conclusion : Eviter d'utiliser les noms de fonction comme noms de variables !

Liste (non exhaustive) de quelques fonctions intégrées à Python:

abs	**bin**	**bool**	**chr**	**complex**	**divmod**	**eval**	**float**	**help**	**hex**
input	**int**	**len**	**max**	**min**	**ord**	**pow**	**print**	**range**	**repr**
round	**sorted**	**str**	**sum**	**type**	……	…….	…….	……..	……..

Pour une liste plus complète, aller à *http://docs.python.org/3.3/library/library/functions.html*

H. EXERCICES

Exercice 1

(U_n) est la suite numérique définie par $U_0=1$ et $U_{n+1}=3U_n-2n+7$ pour tout n, et (V_n) la suite définie par $V_n=3^n$ pour tout n. $Q_n=U_n/V_n$ définit pour tout n la suite (Q_n).

1. Ecrire un script qui saisit un entier n qui doit être positif ou nul et calcule et affiche le terme d'indice n de la suite (U_n)
2. Ecrire un script qui détermine le plus petit entier $n_0>3$ vérifiant $|Q_n-4| < 10^{-6}$.

Exercice 2

a. Ecrire un script Python qui effectue une série de saisies de réels qui doivent être strictement positifs (la série est arrêtée dès qu'on a saisi un réel négatif ou nul), et détermine dès la fin de saisie le plus petit des réels positifs saisis, ainsi que son numéro de saisie, calcule et affiche leur moyenne. On n'utilisera pas de liste.

b. Réécrire ce script en supposant que l'arrêt de la saisie est effectif seulement si la saisie d'un réel négatif ou nul est volontaire.

Exercice 3

Ecrire une fonction qui a comme paramètre un entier naturel n non nul, qui effectue une saisie de n réels, détermine et retourne maxi, le plus grand des saisies, indmaxi, son indice de saisie, mini, le plus petit des saisies, et indmini, son indice de saisie.

Exercice 4

1. Ecrire un script qui saisit un entier m devant être supérieur à 10, puis détermine m entiers choisis de manière aléatoire entre 1 et 500, et ne retient que ceux qui sont compris entre 20 et 50, puis affiche à la fin l'efficacité e de la saisie.
   Le choix d'un nombre aléatoire compris entre 1 et 500 est effectué à l'aide de la fonction randint() du module random.
   L'efficacité e de l'opérateur est calculée par la formule e=m/p*100, où p est le nombre de fois où on a sollicité la fonction randint(1,500) pour obtenir les m entiers compris entre 20 et 50.
2. Transformer ce script en une fonction sans paramètre nommée **partie()**, qui retourne la valeur e
3. On organise un jeu sur la base de cette expérience.
   Le jeu est composé de 5 parties consécutives.
   Une partie est une détermination de m=20 entiers compris entre 20 et 50 selon le procédé de la fonction ci-dessus. Une partie est sanctionnée par un score selon la règle suivante :
   - Si e < 10%, le score est 0. Si e>=90%, le score est 20.
   - Si 10% <= e < 30%, le score obtenu est 5
   - Si 30% <= e < 50%, le score est de 10
   - Si 50% <= e < 70%, le score est de 15
   - Si 70% <= e < 90%, le score est de 17

   Le score du joueur est la somme des scores obtenus aux 5 parties.
   a. Transformer la fonction partie() ci-dessus en une fonction **score()** qui retourne le score total obtenu à la fin de chaque partie.
   b. Ecrire un script qui utilise la fonction score() ci-dessus, et qui fait le bilan (sur 100) du jeu d'un joueur donné.

PARTIE II : LES MODULES SOUS PYTHON

Python s'appuie sur des quantités impressionnantes de modules pour fonctionner. Ces modules appartiennent à des librairies, natives (livrées avec Python) ou viennent d'ailleurs, ou tout simplement créés par vous même.
Avant d'utiliser un module, il faut l'importer, ou importer la fonction à utiliser de ce module. Sinon, cela provoque une erreur
Exemple :
Soit le script :

```
def rayon(aire):
    if aire<0:return False
    else:
        rayon=sqrt(aire/3.14)

aire=float(input('donner une aire:  '))
print(rayon(aire))
```

Pour aire=-5 :

```
>>> (executing lines 41 to 47 of "e
py")
donner une aire:  -5
False
```

Pour aire=18 :

```
>>> (executing lines 41 to 47 of "exercice 6
py")
donner une aire:  18
Traceback (most recent call last):
  File "E:\PYTHON\PYTHON___MOI\exercice 6 Gr
, line 47, in <module>
    print(rayon(aire))
  File "E:\PYTHON\PYTHON___MOI\exercice 6 Gr
, line 44, in rayon
    rayon=sqrt(aire/3.14)
NameError: name 'sqrt' is not defined
```

sqrt n'est pas reconnu par Python

La fonction sqrt() n'est pas automatiquement chargée avec Python. Il fallait importer le module qui la contient.

A. GENERALITES

1. Syntaxe : import <module>

Exemples : import ***os*** importe le module du système d'exploitation.
import ***sys*** importe le module système
import ***math*** import le module math (ématiques)

Une fois un module importé, on peut utiliser les fonctions qu'il contient, se renseigner sur ce qu'elles font.

Remarques :
La syntaxe ci-dessus permet d'importer le module entier. C'est le module entier , « le sac » contenant les fonctions, qui est importé. On n'a pas accès directement aux fonctions de ce module. Par exemple, si on veut utiliser la fonction f1() du module mod1, après avoir importé mod1, on saisit la commande mod1.f1().
Pour avoir accès directement à la fonction f1 de mod1 on peut :

- saisir la commande : **from** mod1 **import** f1.
  Dans ce cas, on n'importe que la fonction f1(), et elle s'utilise en tapant directement f1().
- saisir la commande **from** mod1 **import *** qui importe toutes les fonctions du module mod1 directement, dont f1. On peut alors taper directement f1() pour utiliser la fonction f1().

2. Alias – Importation multiple

a. Alias :

On peut donner un alias à un module importé (par exemple parce que son nom est long)
L'importation se fait alors comme suit (par exemple le module random):

```
import random as x
a=x.randint(2,80)   au lieu de a=random.randint(2,80)
```

b. Importation multiple.

On peut :

- importer plus d'un module à la fois : Exemple : import os, random
- importer plus d'une fonction à la fois : Exemple : from math import sqrt, log

3. Quelques modules standards en Python

- math (les fonctions mathématiques usuelles ……)
- sys (passage d'arguments – les entrées – sorties …..)
- os (communication avec le système d'exploitation ….)
- random (les nombres aléatoires ….)
- time (accès à l'heure de l'ordinateur – fonctions gérant le temps ….)
- calendar
- Tkinter (création d'objets graphiques ……)
- turtle (tracés de figures géométriques …..)
- doctest (permet l'écriture de la documentation et en même temps de faires des tests ….)
- decimal
- itertools
- etc……

4. Accès au contenu d'un module : commandes dir(), help().

On importe le module (module x par exemple)
On peut voir son contenu avec la commande dir(x),
Si f est une composante quelconque de x, on utilise la fonction help() pour se renseigner sur f : on exécute help(x.f).

Par exemple :
On a importé le module ***math***.

- **dir(*math*)** liste le contenu du module (ici math.)

  Remarque : Certaines méthodes, encadrées par __ sont des méthodes dites spéciales.
  Par exemple, dans le module math ci-dessus, la méthode __spec__.
  Celles qui nous intéressent en ce moment sont celles non encadrées par __

- **help*(math.floor)*** donne des informations précises sur cette fonction floor() du module math: arguments attendus et valeurs retournées.

B. CREATION D'UN MODULE

Lors de la réalisation d'un projet, on aura forcément à écrire plusieurs fonctions qu'on voudra regrouper en un « seul endroit » de votre disque dur, qu'on pourra importer et utiliser quand on le désire, sans qu'on ait à les réécrire. Il s'agit ici de créer un module.

Que doit contenir une fonction appartenant à un module donné ?
Prenons le cas de la fonction log() du module math.
La commande **help(math.log)** affiche :

```
>>> help(math.log)
Help on built-in function log in module math:

log(...)
    log(x[, base])

    Return the logarithm of x to the given base.
    If the base not specified, returns the natural logarithm (base e) of x.
```

La commande help() donne des informations sur cette fonction, informations données par la fonction elle-même. Il ne suffit donc pas d'écrire le code de la fonction, il faut aussi inclure des informations renseignant sur son utilisation.
Chaque fonction contient habituellement :

- une partie documentation décrivant ce qu'elle fait,
- le corps (le code) de la fonction elle-même.
- Une partie (qui s'ajoutera comme on le verra plus tard) concernant ce qui permet de tester son bon fonctionnement.

On se contentera dans un premier temps de créer des modules contenant des fonctions et leurs parties documentation.

1. Exemple de création effective d'un module.

Créons un module qui va contenir l'unique fonction valeur absolue :

a. Code de la fonction :

Soit la fonction valabs() dont le script est :

```
def valabs(x) :
    if x>=0 : return x
    else : return –x
```

b. Création du module :

On l'enregistre comme un fichier d'extension .py

Enregistrons-le sous le nom: **f_valabs_seul.py**, **dans le répertoire de travail.**

⇨⇨ Pour connaitre ce répertoire, on importe le module os et on exécute la commande os.getcwd() (pour get current working directory).
Vérifier que getcwd() est bien une méthode du module os.

Le module f_valabs_seul est ainsi créé, contenant la fonction valabs().

c. Exploitation :

On peut déjà essayer de l'exploiter.

(i) Importation du module crée : **import** f_valabs_seul.
(ii) Tapant **dir(f_valabs_seul),** on verra la fonction apparaître dans la liste résultat.
(iii) On peut alors utiliser help(f_valabs_seul.valabs) pour savoir ce qu'est cette fonction.

Résultat : help(f_valabs_seul.valabs) ne donne pas d'information, à part que c'est une fonction….

```
>>> import f_valabs_seul
>>> help(f_valabs_seul.valabs)
Help on function valabs in module f_valabs_seul:

valabs(x)
```

d. Incorporation de la documentation :

Insérons dans cette fonction du texte expliquant ce qu'elle est censée faire (docstring) :
« *Cette fonction retourne la valeur absolue de x, x étant un int ou un float* »

On a alors ceci :

```
def valabs(x):
    """
    cette fonction retourne la valeur absolue de x
    x étant un int ou un float
    """
    if x>=0:return x
    else:return -x
```

Enregistrons cette nouvelle version dans le fichier f_valabs.py, constituant un nouveau module (remplaçant le module f_valabs_seul.py).
On travaillera dorénavant avec ce module f_valabs.py

Importons le module f_valabs, et exécutons la commande : help(f_valabs.valabs).

```
>>> help(f_valabs.valabs)
Help on function valabs in module f_valabs:

valabs(x)
    cette fonction retourne la valeur absolue de x
    x étant un int ou un float
```

Cette fois, des informations sur la fonction sont disponibles.

e. Ajout d'une autre fonction.

Soit la deuxième fonction pythagore(), qui a trois arguments réels positifs x, y et z et qui vérifie si ces trois nombres satisfont la relation de Pythagore.
Elle peut être définie comme suit (on a déjà inclus la documentation):

```
def pythagore(x,y,z) :
    " " "
    Retourne les  x, y, z  et True si les réels x, y, z vérifient la relation de
    Pythagore : x**2=y**2+z**2 ou y**2=z**2+x**2 ou z**2=x**2+y**2,
    et les réels x, y, z et False sinon
    " " "
    ok=False
    if x**2==y**2+z**2 or y**2==z**2+x**2 or z**2==x**2+y**2:
        ok=True
    return ok
```

Cette fonction est écrite dans f_valabs.py, à la suite de valabs() .
On enregistre à nouveau le fichier f_valabs.py dans le répertoire courant.
On dispose maintenant de deux fonctions dans le module f_valabs.
On réactualise f_valabs au niveau de l'importation.

```
============================== RESTART: Shell ==============================
>>> import f_valabs
>>> dir(f_valabs)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
                        '__package__', '__spec__', 'pythagore', 'valabs']
>>> help(f_valabs.valabs)
Help on function valabs in module f_valabs:

valabs(x)
    calcule la valeur absolue du réel x

>>> help(f_valabs.pythagore)
Help on function pythagore in module f_valabs:

pythagore(x, y, z)
    retourne les réels x,y,z et True si les réels x, y,z
    vérifient Pythagore: x**2=y**2+z**2 ou y**2=z**2+x**2
    ou z**2=x**2+y**2, et les réels x,y,z et False sinon

>>>
```

valabs et pythagore sont bien présentes dans f_valabs

précise ce que fait valabs

précise ce que fait pythagore

2. Exercices :

Exercice 1

a. Documentez-vous sur quelques-uns des modules utilisés par Python, et testez quelques-unes de leurs fonctions : (math – cmath – random – os – sys – os.path – tkinter – turtle – array – etc……)

b. Créer un module contenant les deux fonctions suivantes :

(i) : plusgrand qui renvoie le plus grand de deux entiers a et b

(ii) : est_premier qui renvoie True si le nombre entier est premier, False sinon

Tester le module que vous avez créé en modifiant une fonction.

Exercice 2 TP

Il s'agit pour ce TP :

- de créer un module complet comprenant 3 fonctions avec leurs docstring.
- de contrôler leur bon fonctionnement.

Nom du module : **nombres**

Fonctions du module :

(i) : **puiss(x,n)** : élever x à une puissance n donnée sans utiliser l'opération **

(ii) : **dix_et_base(b,n)** :
b st un entier compris entre 2 et 16
si n est entier, il s'agit de coder l'entier n écrit en base 10, dans une base b (2<=b<=16). Le résultat est une chaine de caractères composée des digits issus du codage.
si n est une chaine de caractères, il s'agit de décoder en base 10 le nombre dont le codage en base b est cette chaine.

(iii) : **parfait(n)** : reconnaitre si un entier est parfait ou pas (égal à la somme de ses diviseurs propres)

1. Ecrire un script pour la fonction puiss(x, n).

Si x n'est pas un numérique (int ou float ou complex), retourner False, sinon, retourne x**n

2. Script du codage en base b : fonction **dix_et_base(b,n) :**

Cas 1 : n est un entier.

Commencer par une base b inférieure ou égale à 10. Puis, si c'est correct, étendre au cas où b peut être un entier compris entre 2 et 16.

Pour une base b >10, les digits sont : ‘a’ pour 10, ‘b’ pour 11, ‘c’ pour 12, ‘d’ pour 13, ‘e’ pour 14, ‘f’ pour 15.
Exemple : en base b=14, le codage de 35050 est :’cab8’. Le résultat est une chaine de caractères traduisant le codage.

Cas 2 : n est une chaine de caractères :

Vérifier la validité du codage donné en fonction de la base b. Ex ‘8af’ n’est pas conforme à un codage en base 14. Dans ce cas, retourner False.
Si c’est valide, retourner le nombre en base 10 correspondant.

3. Script de la fonction **parfait(n) :** Fonction retournant le nombre et la liste de ses diviseurs si le nombre est parfait, et le nombre et False si non.

PARTIE III : LES TYPES PROPRES A PYTHON

A : Présentation

1. Les types des objets : type mutable – type non mutable

On a déjà vu quelques types intégrés : int, float, str , bool, None. On va revenir sur le type str et on va découvrir d'autres types, dont certains sont largement utilisés en Python:

- le type tuple (tuple),
- le type list (liste),
- le type str (chaine de caractères),
- les fichiers
- le type dict (dictionnaire),

Ces types sont regroupés dans deux grandes familles : les **types mutable (modifiables)** et les **types non mutables (non modifiables).**

> ***La valeur d'une variable de type mutable peut être changée en gardant la même adresse (ces variables sont donc gérées par référence)***
> ***Pour changer la valeur d'une variable de type non modifiable, on recrée une nouvelle instance (nouvel objet) et on l'associe au nom de la variable par affectation. L'adresse de la nouvelle instance aura alors changé.***

Les types int, float, bool, complex, str sont non mutables
A chacun de ces types d'objet sont associées des méthodes (fonctions qui permettent de les manipuler)

Exemple:
On rappelle que la fonction id() donne l'adresse d'une variable ou d'un objet.

```
>>> a=9
>>> id(a)
1665625360
```
id(a)=id(b)

```
>>> b=9
>>> id(b)
1665625360
```

```
>>> a=10
>>> id(a)
1665625392
```

```
>>> a=a+b
>>> id(a)
1665625680
```

```
>>> a=a-9
>>> id(a)
1665625392
```
a a retrouvé son adresse

Le type int est non modifiable. La variable a change d'adresse lorsqu'elle change de valeur (lorsqu'elle référence un autre objet).

2. Objet séquence

Une séquence est une qualité que peut avoir certains objets en Python. Ce n'est pas un type d'objet, mais plutôt une caractéristique, une qualité d'objet.

Caractéristiques d'objets séquence :

Notons a un objet séquence.

- a est formé d'un nombre fini d'éléments auxquels on peut accéder par indice.
- la fonction len() retourne la longueur de l'objet séquence a, c'est-à-dire le nombre de ses éléments.
- Le premier indice est 0 et le dernier indice est len(a)-1.
  a[0] désigne le premier élément, l'élément qui se trouve à la position 1, a[2] est l'élément se trouvant en 3ème position dans a, le dernier élément de a est a[len(a)-1].
  Remarque : Ce dernier élément, a[len(a)-1], est aussi noté a[-1], l'avant-dernier est noté a[-2], ……., le premier est noté –(len(a)).

- **L'opérateur d'appartenance s'applique à une séquence : elt in a** est True si elt est dans a, et False sinon.
- On peut effectuer un **slicing** (découpage) d'un objet séquence avec la syntaxe a[i :j] : a[i :j] est la partie extraite de l'objet a contenant les éléments de la position (i+1) (d'indice i), à la position j (d'indice j-1), l'élément a[j] n'en faisant pas partie.
- On peut parcourir tout objet de type séquence. On peut pour cela utiliser les indices, ou une variable de parcours qui est du type de l'élément de l'objet.
  Exemple : a='maintenant'. len(a)=10 ;
  for i in range(10) : print(a[i]) parcourt a et affiche ses éléments un à un dans l'ordre d'apparition dans a.
  for i in a : print(i) fait exactement la même chose, mais ici i est un élément de a.

B. Le type str (chaine de caractères)

1. Définition

Les objets de type str sont des chaines de caractères. Leurs éléments sont donc des caractères, ceux regroupés dans le tableau ASCII.

```
>>> a='erreur'

>>> type(a)
<class 'str'>
```

Pour une chaine de caractères, **un ordre définitif de placement des caractères est défini**. Pour un caractère donné donc, s'il n'est pas à l'extrémité de la chaine, on sait qui est avant lui (son prédécesseur) , qui est après lui (son successeur). Le premier caractère d'une chaine n'a pas de prédécesseur et le dernier n'a pas de successeur.

2. Syntaxe : * désignera un caractère quelconque.

La syntaxe est :
'*****' ou ''****'', pour une chaine sur une même ligne. Exemple : 'jour', ''mardi''.
''' ***** ''' ou '' '' '' ******* '' '' '' pour une chaine sur plusieurs lignes
Quand utiliser '**' et quand utiliser ''**'' ?
Le problème se pose lorsqu'une chaine contient ' ou '':
Principe :
Si l'un fait partie de la chaine, c'est l'autre qui peut être utilisé, ou alors, on utilise un \.

Exemples :
Pour le mot aujourd'hui, on peut écrire '' aujourd'hui'' ou 'aujourd\'hui'
Pour le mot abcde''fgh, on peut écrire 'abcde''fgh' ou ''abcde\''fgh''

3. Caractéristique

Un objet de type str est un **objet séquence.**
Le type str est NON MUTABLE (non modifiable)

Encodage
La manipulation de chaines de caractères dépend de l'encodage utilisé, qu'il faut préciser si on n'utilise pas celui utilisé par défaut par Python. **L'encodage précise les caractères reconnus.**
Exemple : l'encodage utf8 :
 # -*- coding : utf-8 -*- **à toujours placer en en-tête de script**

4. Opérations sur un objet de type str

Chaine vide : ‘’

Appartenance (ou pas) : in , not in:

a et b sont deux chaines**. b in a** retourne True si la chaine b est **contenue** dans la chaine a

Exemple :

a=’maintenant’ b=’a’, c=’ai’, d= ‘’aj’’, h=’ne’

b in a retourne True

c in a retourne True

‘f’ in a retourne False

‘f’ not in a retourne True

d in a retourne False

h in a retourne False

Concaténation: + Exemple: b+c=’aai’

Duplication : n*<chaine> retourne la juxtaposition, n fois, de la chaine.

Exemple : 5*h donne ‘nenenenene’

Longueur: len(< chaine>): retourne le nombre de caractères de la chaine.

Minimum, maximum d’une chaine : **min**(<chaine>), **max**(<chaine>).

Retourne le plus petit élément et le plus grand élément (au sens ASCII) de la chaine.

Exemple : a=’musique’

min(a) retourne ‘e’ ; max(a) retourne ‘u’

Accès à une sous-chaine (slicing):

<chaine>[ind_init : ind_fin] où ind_init est inclus et ind_fin est exclus.

<chaine>[i] donne le caractère d’indice i de <chaine>, donc (i+1) ème caractère, pour i allant de 0 à len(<chaine>)-1

<chaine>[-i], pour i allant de 1 à len(<chaine>) donne les caractères de <chaine> à partir de la fin.

Exemple :

Considérons la chaine a= ‘abfdcsert?mt’

Les caractères	a	b	f	d	c	s	e	r	t	?	m	t
indices positifs	0	1	2	3	4	5	6	7	8	9	10	11
indices négatifs	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

a[2] est le caractère d’indice 2, donc ‘f’.

a[1 :4] est la sous-chaine ‘bfd’

a[:5] est la sous-chaine ‘abfdc’

a [2 :] est la sous-chaine ‘’fdcsert?mt’’

a[:] est une copie de a (shallow copy)

a[2 :2] est la sous-chaine vide

a[2 :3] est la sous-chaine ‘f’

Exercice :

Afficher l’adresse de a

b=a[:]. Afficher l’adresse de b

c=a. Afficher l’adresse de c

Que remarque-t-on ? Justifier.

Tentative de modification d’une sous-chaine :

Exemple : a=’journee’

a[2] est le caractère ’u’. L’opération a[2]=’4’ tente de mettre 4 à la place de ‘u’. Elle provoque une erreur.

```
>>> a[2]='4'  # tentative de remplacer i par 4
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

L'erreur provient de ce que le type str **n'est pas mutable** (**non modifiable**)
Si on veut changer a[2] en '4', il faut créer une autre chaine de caractères et utiliser par exemple un **slicing.** (à faire en exercice)

5. Les méthodes :

Ce sont :

```
>>> (executing lines 109 to 116 of "exos_01_master1.py")
capitalize  casefold
 center  count  encode  endswith  expandtabs  find  format
 format_map  index  isalnum  isalpha  isdecimal  isdigit  isidentifier
 islower  isnumeric  isprintable  isspace  istitle  isupper  join
 ljust  lower  lstrip  maketrans  partition  replace  rfind
 rindex  rjust  rpartition  rsplit  rstrip  split  splitlines
 startswith  strip  swapcase  title  translate  upper  zfill
>>>
```

En règle générale, la syntaxe pour l'une quelconque de ces méthodes est :

<chaine>.<methode(<….>)

Quelques méthodes :

méthode title() :

description de la méthode title() : help(str.title) retourne :

```
>>> help(str.title)
Help on method_descriptor:

title(...)
    S.title() -> str

    Return a titlecased version of S, i.e. words start with title case
    characters, all remaining cased characters have lower case.
```

méthode index()

Afficher la description de cette méthode avec help()
Selon help() **:**

- **syntaxe : <chaine>.index(<sous-chaine>)**
- retourne l'indice du premier caractère de la première occurrence de cette sous-chaine, si celle-ci est contenue dans <chaine>.
  Si la sous-chaine est formée d'un unique caractère, elle retourne l'indice de la première occurrence de ce caractère dans la chaine.

Exemple 1 : a=''aujourd'houi''
Taper les commandes :
a. index('u') , a. index('ou') , a. index('auh') , a.index('ou',6), a. index('u',5).
Commenter.

méthode upper() : met les caractères de <chaine> en majuscule

méthode count() : retourne le nombre d'occurrences d'une sous-chaine dans une chaine.

C : Le type tuple

1. **Définition** : Ensemble d'objets dans une parenthèse, **de tout type,** séparées par des virgules.
   Type : tuple

   Exemple : (5, '5', 8, 'meme', 9, ('e',0,67),7)

```
>>> a=(5, '5', 8, 'meme', 9, ('e',0,67),7)

>>> type(a)
<class 'tuple'>
```

   Création d'un tuple non vide : a=5,8,3,'5',9,2+3j donne le tuple (5,8,3,'5',9,2+3j)

2. **Propriétés :**
   - **Un objet de type tuple est un objet séquence**
   - **type non mutable**
   - **peut être imbriqué**
     Exemple : b=9,5,'oui',(8,9,10),10,14 donne à l'affichage : (9,5,'oui',(8,9,10),10,14)
   - **tuple vide :** (). Exemple : d=()
   - **tuple à un élément :** on fait suivre l'élément par une virgule.
     Exemple t=8, ce qui, à l'affichage donne le tuple (8,)
   - **longueur d'un tuple** : obtenue avec la fonction **len() :** Exemple : a=(7,8,45) len(a)=3
   - **Découpage** et **recopie, et récupération d'une partie** du tuple **(slicing):**
     a[0]=7. 0 est un index, l'index de 7 dans le tuple.

```
>>> a=8,7,14,5,9,14,25,'m',5,'987'

>>> a
(8, 7, 14, 5, 9, 14, 25, 'm', 5, '987')   affichage
>>> len(a)   longueur de a
10
>>> a[ :]
(8, 7, 14, 5, 9, 14, 25, 'm', 5, '987') recopiede a
>>> a[2:5]
(14, 5, 9)        3ème élément inclus au 6ème exclu
>>> a=b
>>> a is b
True        b est identique à a: même valeur et même adresse
```

   a[-1]='987'=a[len(a)-1]
   a[-2]=a[len(a)-2]=5

	a=(8,	7,	14,	5,	9,	14,	25,	'm',	5,	'987')
indices :	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
	-10	*-9*	*-8*	*-7*	*-6*	*-5*	*-4*	*-3*	*-2*	*-1*

   Opérations permises :
   Somme : (1,3,2)+(6,7,8,9)=(1,3,2,6,7,8,9)
   Produit par un entier : 2*(1,2,3) = (1,2,3,1,2,3)
   L'opération : 2+(1,2) provoque une erreur

3. **Méthodes :**
   Les méthodes où ne figure pas un double underscore (__) sont: **index(), count()**
   Ecrire (en exercice) un script qui n'affiche que ces méthodes.

   - **index(x) :** donne l'indice de l'élément x du tuple.
   - **count(x) :** donne le nombre d'occurrences de x dans le tuple.

D : Le type list

1. **Définition** : **Ensemble de valeurs (de tout type) entre deux crochets, séparées par des virgules. Le type est list**

Exemple

```
>>> m=[5,65.8,'4','??',[6,8,12],True]

>>> type(m)
<class 'list'>
```

Remarque : On peut rapprocher le type **list**, dans sa structure, au type enregistrement habituel rencontré en Algorithmique. Mais il offre beaucoup plus de possibilités.

Exemple :

seconde=[['Paul',15,[14,16,18]], ['Traore',17,[13,11,14]], ['Arnaud',16,[4,10,12]]] : liste de données constituées du nom de l'élève, de son âge et de ses trois notes selon le tableau :

Paul	15	14 – 16 – 18
Traore	17	13 – 11 - 14
Arnaud	16	4 – 10 - 12

2. **Propriétés :**
   - un objet de type list **est une séquence.** On peut donc utiliser **in, not in**
   - la **liste vide** est**: []**
   - le type list est **mutable**
   - il peut être **imbriqué**
   - **la longueur d'une liste** est obtenue avec la fonction **len()**
   - **découpage, recopie, et récupération d'une partie (slicing)** d'une liste sont possibles:

```
>>> l=[7,8,'874',[6,'d',9,'hj'],2,6]
>>> len(l)
6
>>> l[1]=888
>>> l
[7, 888, '874', [6, 'd', 9, 'hj'], 2, 6]   modification réussie
>>> l[ :]
[7, 888, '874', [6, 'd', 9, 'hj'], 2, 6]   recopie
>>> l[-1]
6
>>> id(l)
216735674952
>>> l[-3]='+'          l'adresse ne change pas après modification
>>> id(l)
216735674952
>>> p=l          >>> l is p   l et p sont identiques
                 True
```

Remarque :

- b=a[:] est une copie de la liste a. a==b est True (même contenu), mais a is b est False.
- b est une shallow copy de a.
- La commande dir() **retourne un objet de type list** où figurent toutes les méthodes et attributs de l'objet donné en paramètre.

Exemple :

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin', 'as
inh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'er
fc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', '
hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'l
og1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', '
tau', 'trunc']
```

3. Méthodes :

```
81 m=[]
82 for i in dir(list):
83     if '__' in i: continue
84     else:m.append(i)
85 for i in range(len(m)):
86     if i%10==0:
87         print('\n',m[i],end='   ')
88     else: print(m[i],end='   ')
```

Ce script donne la liste des méthodes non spéciales du type list:

```
append  clear  copy  count  extend  index  insert  pop  remove  reverse
sort
```

Ces méthodes ne créent pas de nouvelles instances car une liste est mutable (modifiable). x désigne un élément d'une liste et a et L sont deux listes.

- **append(x)** : ajoute un élément nouveau à la fin de la liste :
- **clear() :** rend la liste vide
- **copy() :** crée une autre liste.
- **count(x) :** renvoie le nombre d'occurrences de x dans la liste
- **extend(L),** où L est une autre liste
- **index(x) :** renvoie l'indice de la première occurrence de x
- **insert(i,x)** : insère l'élément x à l'indice i
- **pop() :** supprime le dernier élément de la liste et affiche ce qui a été supprimé.
- **remove(x)** : retire le premier élément de valeur x de la liste
- **reverse() :** inverse l'ordre des éléments
- **sort() :** trie les éléments

```
Type "copyright", "credits" or "license()" for more information.
>>> a=[7,8,'874','+',2,6]
>>> type(a)
<class 'list'>
>>> a.append('oui')
>>> a.insert(0,'debut')
>>> a
['debut', 7, 8, '874', '+', 2, 6, 'oui']
>>> a.append(7)
>>> a
['debut', 7, 8, '874', '+', 2, 6, 'oui', 7]
>>> a.remove(7)
>>> a
['debut', 8, '874', '+', 2, 6, 'oui', 7]
>>> a.reverse()
>>> a
[7, 'oui', 6, 2, '+', '874', 8, 'debut']
>>> a.pop()
'debut'
>>> a
[7, 'oui', 6, 2, '+', '874', 8]
>>> b=a
>>> a.clear()
```

On peut ajouter à ces méthodes :

- la commande **del** :
  - **del a**[2] supprime l'élément d'indice 2 de la liste a.
  - **del a**[2:5] supprime les éléments à partir de l'indice 2 jusqu'à l'indice 4.
  - **del a** : supprime la liste a.

- les opérations arithmétiques avec les listes : somme, produit par un entier.

```
Type "copyright", "credits" or "license()" for more information.
>>> a=[6,'4',['az',7,True],7]
>>> b=[6,9,12.0]
>>> 2*b
[6, 9, 12.0, 6, 9, 12.0]
>>> a+b
[6, '4', ['az', 7, True], 7, 6, 9, 12.0]
>>> a-b
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    a-b
TypeError: unsupported operand type(s) for -: 'list' and 'list'
```

```
>>> a=[4,5,6,[7,8,'4'],'jour',"demain",4]
>>> a[1]
>>> a[3]
>>> a[2]='56hT'
>>> a
>>> a[2:2]
>>> a[2:3]
>>> a[:4]
>>> a[2:]
>>> c=a[:]
>>> c
>>> c is a
>>> c==a
>>> del(c)
>>> c
```

E. Compléments sur les types list, str et tuple:

Soient la liste a : a=[‘4’,’5’,’a’, ‘’azerty’’,’0’] formée de chaines de caractères,
la chaine b : b=’azerty?456T’
le tuple c=(5,’5’,’python’,[4,8,6])

1. Deux méthodes supplémentaires :

a. Méthode join() :

La méthode **join()** transforme une liste a de chaines de caractères en une chaine de caractères, utilisant éventuellement un séparateur entre les chaines :

Syntaxe : **‘<**>’.join(<liste>)** , où <**> est un séparateur des éléments de la liste

Exemple : ‘’.join(a) transforme la liste a=[‘4’,’5’,’a’, ‘’azerty’’,’0’] en la chaine de caractères : ‘45aazerty0’

‘*’.join(a) transforme la liste a en la chaine ‘4*5*a*azerty*0’

b. Méthoder split() :

help(str.split) retourne :

```
>>> help(str.split)
Help on method_descriptor:

split(...)
    S.split(sep=None, maxsplit=-1) -> list of strings

    Return a list of the words in S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator and empty strings are
    removed from the result.
```

2. **Retour sur le transtypage : La commande :**
   - list(b) crée la liste physique [‘a’,’z’,’e’,’r’,’t’,’y’,’ ?’,’4’,’5’,’6’,’T’]
   - tuple(a) crée un tuple formé des éléments de la liste a : (‘4’,’5’,’a’,’’azerty’’,’0’)
   - list(c) crée une liste à partir des éléments du tuple c : [5,’5’,’python’,[4,8,6]]
   - dir() retourne un objet de type list.
   - tuple(b) crée le tuple (‘a’,’z’,’e’,’r’,’t’,’y’,’ ?’,’4’,’5’,’6’,’T’)

F : Les fichiers

***** **Faire attention à l’encodage utilisé.**

***** **Pour retrouver votre fichier, il faut importer le module os et exécuter la commande os.getcwd() qui va vous indiquer le répertoire courant où travaille Python. (cwd : current working directory). Sinon, utiliser la commande os.chdir() du module os pour vous placer dans le répertoire de votre fichier.**

1. **Définition :**
   fichier : collection d’informations stockées dans une mémoire de masse, de type **file.**
   Le type fichier est **itérable** (voir paragraphe suivant)

   Types de fichiers :
   Les fichiers se distinguent par :
   a. L’organisation adoptée :
      structurée (fichier d’enregistrement) ou non structurée (fichier texte)
   b. Le mode d’accès :
      séquentiel, ou indicé (comme un tableau)

   Le type est **itérable**

2. **Les fichiers textes, en séquentiel:**
   Opérations avec les fichiers texte:

   Ouverture :
   fich=open(‘ couleurs.txt’,’w’) : ouvre le fichier ‘couleur.txt’ en écriture. Crée donc le fichier sous ce nom ; et s’il existe déjà, il sera écrasé .
   fich est du type fichier
   fich=open(‘ couleurs.txt’,’r’) : ouvre le fichier ‘couleur.txt’ en lecture. Le fichier doit exister.
   fich=open(‘ couleurs.txt’,’a’) : ouvre le fichier ‘couleur.txt’ en mode ajout. Le fichier doit exister.

   Fermeture :
   fich.close() : ferme le fichier

   Lecture et écriture en bloc :
   a=fich.read() : lit tout le contenu du fichier en bloc et le met dans a.
   ***** **a est du type chaine de caractères**
   print(a) : affiche le contenu
   len(a) : donne la taille du fichier : le nombre de caractères

   Opérations à l’intérieur du fichier :
   Ecriture :
   fich.write(…….) argument : une seule chaine de caractères

fich.writelines(…….) argument : un fichier
Lecture :
fich.readlines() lit tout le fichier et renvoie un résultat qui est de type list, ligne par ligne
fich.readline() lit le fichier ligne par ligne, et renvoie donc une chaine de caractères

3. **Les fichiers textes, en mode binaire:** (à voir)

Exemple :
Le script ci-dessous

```
22  fich=open('text4.txt','w')              #ouverture en écriture
23  fich.write('abcdefgh\nijklmnopq')       #création du fichier text4*;txt
24  fich.close()                            #fermeture
25  fich=open('text4.txt','r')              #ouverture pour vérifier le contenu ( ici inutile car
26  a=fich.read()                           #il affiche automtiquement ce qui a été créé )
27  print(a,'\n   taille=',len(a))
28  fich.close()
29  fich=open('text4.txt','a')              #ouverture en mode ajout
30  fich.write('\nrstuvwxyz\nabcdefghijkl') #l'ajout
31  fich.close()
32  fich=open('text4.txt','r')
33  a=fich.readlines()
34  print("*****   sortie d'un readlines   *********\n") # ******
35  print(a)
36  fich.close()
```

donne

```
>>> (executing lines 22 to 36 of "01_enlever doublons_calc_interm_master1.py")
abcdefgh
ijklmnopq
   taille= 18
*****   sortie d'un readlines   *********

['abcdefgh\n', 'ijklmnopq\n', 'rstuvwxyz\n', 'abcdefghijkl']

>>>
```

G : Retour au passage des paramètres d'une fonction

On a vu que les paramètres de type int, float, bool, complex et str sont passés par valeur. Ces types sont **non mutables.**
Considérons une fonction dont un des paramètres est de type mutable, de type list par exemple, et un autre non mutable (une chaine de caractères par exemple).

```
21  def toto(x,y):
22      x.append('interieur')
23      y='intérieur'
24      return x,y
25  a=[4,9]
26  b='exterieur'
27  print('a avant= ',a)
28  print('b avant= ',b)
29  print('ce qui est retourné: ',toto(a,b))
30  print('a après= ',a)
31  print('b apres=',b)
```

qui donne comme sortie :

```
>>> (executing lines 21 to 31 of "<tmp 2>")
a avant=  [4, 9]
b avant=  exterieur
ce qui est retourné:  ([4, 9, 'interieur'], 'intérieur')
a après=  [4, 9, 'interieur']
b apres= exterieur
```

Commentaires :
Le premier argument, x, à l'exécution, est ici une liste, objet modifiable. Le second, y, est une chaine de caractères, objet non modifiable.
Les deux subissent une modification à l'intérieur de la fonction.
Après l'exécution de la fonction, le changement subi par le second paramètre n'est pas effectif en dehors de la fonction (passage par valeur donc), alors que le changement a été effectif en dehors de la fonction pour le premier argument (le passage est par adresse).
D'où le résultat suivant :

Le passage d'un paramètre non mutable se fait par valeur, celui d'un paramètre mutable se fait par adresse.

PARTIES IV : COMPLEMENTS : retour à la boucle for

1. Objets itérables:

Les deux syntaxes de la boucle for que l'on a vues sont:

a. for i in range(n), où n est un entier,
b. for i in a, où a est un objet séquence

range(n) est l'intervalle d'entiers de 0 à n(n exclu) de n éléments.
Pour la syntaxe a.

Le corps de la boucle est exécuté pour les valeurs de i allant de 0 à n-1.
On passe donc automatiquement de 0 à 1, puis de 1 au suivant, etc…

Pour la syntaxe b.

Le corps de la boucle est exécuté en parcourant l'objet a à partir de son premier élément, jusqu'au dernier.

Ces modes de fonctionnement sont ceux d'un ensemble plus général d'objets appelés **objets itérables,** car qui possèdent une méthode **next(),** qui permet de passer d'un élément à un suivant, de manière automatique.

Quelques objets itérables:

a. range(n) (et plus généralement range(i,n,k)) est un objet itérable.
range(n) a pour éléments les entiers consécutifs de 0 à n-1
range(i,n,k) a pour éléments les entiers consécutifs de i à n-1, avec un pas de k

Exemples : list(range(5)) donne [0,1,2,3,4]
list(range(1,11,2)) donne [1,3(=1+2),5,7,9]

b. un objet de type str , de type tuple , ou de type list, est itérable.
On peut le parcourir du premier élément caractère au dernier.

c. Un fichier séquentiel est itérable.

2. Syntaxe générale d'une boucle for :

```
for <element> in <iterateur> :
        ……
else :
        ……
```

Exemples:

a. a=range(6):
```
for i in range(6) :
        print(i,end=' ') # affiche   0 1 2 3 4 5
```
b. a=''eml?/gtf''
```
for k in a :
        print(k,end=' ') # affiche  e m l ? / g t f
```
c. a=('4',(6,9),'eml','?')
```
for j in a :
        print(j,end=' ') #  affiche 4 (6,9) eml ?
```
d. a=[7,9,(4,'4',0),12,[4,9]]
```
for m in a :
        print(m,end=' ') #  affiche 7 9 (4,4,0) 12 [4,9]
```

3. Listes en compréhension

a étant un objet itérateur, la traduction en phrase de l'instruction : **for i in a :**
est : **pour tout i appartenant à a** (sous-entendu qu'on va parcourir avec i les éléments de a)
Considérons le script :

```
01      a,b=[-2,5,7],[1,2,3,4]
02      c,d=[],0
03      for i in a :
04              for j in b :
05                      c.append(i*j)
06                      d=d+i*j
07      print(''c= {}  et  somme= {}''.format(c,d)
```

La partie correspondant aux lignes 02 à 06 de ce script peut être traduite à l'aide de la phrase: « créer la liste c formée par les **i×j pour tout i dans a et tout j dans b** et calculer la somme des éléments de c».
Ceci, en Python, s'écrit **en compréhension**, comme suit :

```
c=[i*j for i in a for j in b]    # écriture en comprehension
d=sum(c)
```

ou mieux, l'unique instruction: d=sum([i*j for i in a for j in b]).

4. Série d'exercices

Exercice 1 : Schéma de Horner :

$P(x)=a_nx^n+a_{n-1}x^{n-1}+.....+a_1x+a_0$ peut être écrit sous la forme :

$P(x)=(((((a_n)x+a_{n-1})x+a_{n-2})x+.....)x+a_1)x+a_0$. Calculer P(x) avec le schéma de Horner, c'est le calculer en prenant P(x) écrit avec la seconde forme.
Ecrire une fonction qui pour une liste donnée coefficients d'un polynôme, et pour une valeur de x donnée, calcule P(x).

Exercice 2 : Produit scalaire

Soit deux vecteurs X et Y de IR^n . Ecrire un script qui calcule le produit scalaire de ces deux vecteurs, d'abord sans chercher à optimiser les instructions, puis en essayant d'optimiser (en évitant par exemple d'utiliser des indices). X et Y sont deux listes de réels. (penser à une liste en compréhension)

File Edit Format Run Options Window Help

```
X,Y=[1,2,3,4,5],[6,7,8,9,10]
print(sum([x*y for x,y in zip(X,Y)]))
```

Exercice 3 : Produit matriciel

La liste X=[1,1,1] désigne le vecteur colonne $\begin{pmatrix}1\\1\\1\end{pmatrix}$ et la liste A=[[1,2,3],[5,3,7],[0,1,2]] représente la matrice $\begin{pmatrix}1&2&3\\5&3&7\\0&1&2\end{pmatrix}$.

1. Ecrire un script qui saisit la matrice A ci-dessus et affiche A sous forme « matricielle »
2. Ecrire une boucle qui calcule le produit matrice-vecteur Y = A.X et le stocke dans une liste Y.
3. Ecrire la même boucle en utilisant les list compréhension. Ecrire ce produit matriciel sans utiliser aucun indice.