03 | Boucles et fonctions

I - Instructions de contrôle

I.1 Définition

Une instruction est une étape de programmation qui indique l'action à effectuer.
Un programme est une suite d'instruction.

I.2 Boucle if, instruction conditionnelle

```
>>> if nombre > 0 :                    # test
>>>     print ( '' nombre positif '')   # instruction
>>> elif nombre < 0 :
>>>     print('' nombre négatif '')
>>> else :
>>>     print ('' vaut 0 '')
```

I - Instructions de contrôle

I.3 Boucle for, instruction itérative

```
>>> for i in range (2, 10, 3) :                #début de la boucle for.
>>>       print (" le compteur vaut ", i )     # instruction
le compteur vaut 2
le compteur vaut 5
le compteur vaut 8
```

Remarque : range (i, j, k) est un compteur qui va de i (inclus) à j (exclu) avec un pas de k.

I - Instructions de contrôle

I.4 Boucle while, instruction itérative

La boucle while permet de répéter une instruction tant qu'une condition est vraie.

```
>>> nombre = 0                    # Définition des variables de test ou du compteur
>>> while nombre < 10 :           # condition d'arrêt
>>>     print (nombre)             # instruction
>>>     nombre + = 1               # incrémentation du compteur
```

Remarque : exécuter l'instruction break dans une boucle l'interrompt immédiatement. Si la condition est toujours vraie, la boucle ne se « termine » jamais… à éviter!

II - Les fonctions

II.1 Découpage en fonctions

Le découpage d'un script en plusieurs fonctions (modularité d'un programme) permet d'améliorer la **lisibilité** d'un programme ainsi que son **débogage**, et également de les **réutiliser** à plusieurs endroits.

II.2 Définition et appel

```
>>> def nom_de_la_fonction (argument 1, argument 2, ...)
>>>     ''' Documentation de la fonction '''        #permet de spécifier le rôle de la fonction
>>>     séquence d'instructions
>>>     return expression ou liste d'expression
```

II - Les fonctions

II.1 Découpage en fonctions

Le découpage d'un script en plusieurs fonctions (modularité d'un programme) permet d'améliorer la **lisibilité** d'un programme ainsi que son **débogage**, et également de les **réutiliser** à plusieurs endroits.

II.2 Définition et appel

```
>>> def nom_de_la_fonction (argument 1, argument 2, ...)
>>>     ''' Documentation de la fonction '''        #permet de spécifier le rôle de la fonction
>>>     séquence d'instructions
>>>     return expression ou liste d'expression
```

II - Les fonctions

II.2 Définition et appel

L'instruction return( ) provoque l'arrêt de la fonction.

II - Les fonctions

II.2 Définition et appel

L'instruction return( ) provoque l'arrêt de la fonction.

```
>>> def f1(x):   #Ne pas oubliez les deux points.
>>>       xcarre=x*x
>>>       return xcarree  #Notez bien l'indentation !
>>> print (f1 (5) )
25
```

II - Les fonctions

II.2 Définition et appel

Que fait cette fonction?

```
>>> def f1(N):
>>>     for i in range (N):
>>>         s=i
>>>     return s                    #return à la fin de (après) la boucle for
>>> print (f1 (5) )
4
```

II - Les fonctions

II.3 Modules

Ce sont des bibliothèques de fonctions déjà écrites. On retiendra les plus courantes :

- math # fonctions mathématiques usuelles
- numpy # fonctions avancées de manipulation des tableaux
- matplotlib # fonctions de tracé de courbe

```
>>> import math as mt          # on importe le module math que l'on renomme "mt"
>>> mt.sin (4)                 # appel de la fonction " sin " du module math.
```

Astuce : on peut aussi uiliser « from math import * »
L'étoile signifie que l'on importe tout… mais gare aux collisions

III - Ecriture d'un programme

III.1 Spécification et annotations d'un programme

La **spécification** permet d'informer les utilisateurs de la tache effectuée par la fonction, de préciser les contraintes imposées par les paramètres et ce qui peut être attendu des résultats. Elle peut aussi préciser les messages d'erreurs affichés en cas de mauvaise utilisation. Elle est inscrite au début du corps de la fonction entre **triples guillemets**.

```
>>> def nom_de_la_fonction(arguments):
>>>     """ informations concernant la fonction, non obligatoires
>>>         mais recommandées """
>>>     corps de la fonction
```

III - Ecriture d'un programme

III.1 Spécification et annotations d'un programme

Cette spécification peut être appelée et affichée à l'aide de la fonction **help( )** dans l'interpréteur :

```
>>> help(nom_de_la_fonction)
informations concernant la fonction, non obligatoires mais recommandées
```

```
>>> import math
>>> 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.
```

III - Ecriture d'un programme

III.1 Spécification et annotations d'un programme

A des fins de compréhension, Il est important d'**annoter** certaines lignes ou blocs d'instructions afin de préciser leur rôle. On utilise pour cela un commentaire qui est une ligne de texte précédée du signe **#**.
Tout ce qui est présent après ce signe n'est pas utilisé par l'interpréteur.

```
>>> def mafonction(...):
>>> # importations de modules et de fonctions
>>> ...
>>> # définitions de constantes et de variables
>>> ...
>>> # définitions de la fonction principale
>>> var2=np.cos(var1+np.pi/2)          # Cosinus de la première variable déphasé de pi/2
>>> ...
```

III - Ecriture d'un programme

III.2 Assertion

Une **assertion** est l'affirmation qu'une propriété est vraie. Elle est composée du mot **assert** suivi d'une expression dont la valeur est interprétée comme une valeur booléenne. Si l'expression a la valeur True, il ne se passe rien. Sinon, le programme est interrompu et un message d'erreur s'affiche : **AssertionError**.

```
>>> def inverse(x):
>>> """ x est un nombre non nul de type int ou float
>>> la fonction renvoie l'inverse de x
>>> """
>>> assert x !=0
>>> return 1/x
```

III - Ecriture d'un programme

III.2 Assertion

Exemples d'utilisation :

```
>>> inverse(-5)
-0.2
>>> inverse(0)
Traceback (most recent call last):
        File "<ipython-input-4-c65803e96c4a>", line 1, in <module>
                inverse(0)
        File "<ipython-input-3-a8cd3ed710f9>", line 2, in inverse
                assert x!=0
AssertionError
```

III - Ecriture d'un programme

III.3 Débogage d'un programme

Les types d'erreurs que vous rencontrerez le plus fréquemment cette année sont :

- SyntaxError : invalid syntax :
- IndexError : list index out of range :
- ZeroDivisionError
- NameError
- TypeError
- ValueError
- IOError

IV - Lecture et écriture de fichiers

Éditer un fichier :

Pour éditer un fichier en Python, on utilise la fonction **open()** qui prend deux arguments : le **chemin d'accès du fichier** et le **type d'ouverture**.

Chemin d'accès :

On distingue le chemin relatif et le chemin absolu :

- Le chemin relatif concerne le dossier dans lequel se trouve votre fichier exécutable Python : le dossier de travail (exemple: "fichier.txt").
- Le chemin absolu est celui de l'arborescence systématique de votre système de fichiers (exemple: "C://Users/Documents/IPT/fichier.txt").

Ouverture d'un fichier :

```
>>> objet = open ( ' fichier_test.txt ' , 'mode' )
```

IV - Lecture et écriture de fichiers

Ouverture d'un fichier :

Le second paramètre 'mode' permet d'indiquer la manière dont le fichier va être utilisé. On distingue quatre modes différents :

- 'r' pour **read**, le fichier est ouvert en lecture
- 'w' pour **write**, le fichier existant est écrasé et le nouveau contenu écrase le précédent
- ' a ' pour **append**, toutes les données sont ajoutées à la fin du fichier
- ' r+ ' pour **ouvrir le fichier en lecture et écriture**

IV - Lecture et écriture de fichiers

Lecture d'un fichier :

Un fichier texte est composé de lignes, chaque ligne étant une chaîne de caractères terminée par **'\n'** qui indique le saut de ligne. Si on veut stocker sur une même ligne du fichier plusieurs données de nature différente, on a besoin d'un délimiteur pour séparer les valeurs. Ce délimiteur peut être une tabulation ('\t') ou bien un point-virgule. Un fichier texte a un suffixe **" .txt"**.

Remarque : Il est important de noter que même si les données sont des nombres (entiers ou flottants), ils sont stockés en tant que chaîne de caractères.

Ecriture d'un fichier :

```
>>> objet = open ( ' fichier_test.txt ' , 'w' )
>>> objet.write ( ' l'informatique c'est cool ' )
```

IV - Lecture et écriture de fichiers

Fermeture d'un fichier :

il faut fermer un fichier avec Python après utilisation en utilisant la fonction **close**.

```
>>> objet.close ( )
```

Exemples:

```
1  def stocker_fichiers_valeurs(f , a , b , nom_fichier):
2  |       "échantillonnage de valeurs de la fonction f sur [a,b]. Les valeurs sont stockées dans le fichier \
3  |       texte nom_fichier, dans le même dossier que ce script. les valeurs flottantes sont converties en \
4  |       chaînes de caractères"
5  |       X = np.linspace(a,b,250)  # on crée les abscisses (250 points entre a et b)
6  |       Y = [f(x) for x in X]  # on calcule les images correspondantes
7  |       f = open(nom_fichier , 'w')
8  |       # on ouvre le fichier en écriture 'w' = write (si le fichier existait, il est écrasé, s'il n'existe pas, il est créé.
9  |       for i in range(len(X)): # autant d'itérations que de points
10 |       |       f.write(str(X[i]) + '\t' + str(Y[i]))  # coordonnées du i-ième point séparées par une tabulation
11 |       |       f.write('\n')  # saut de ligne, un point par ligne
12 |       f.close()  # on ferme le fichier
```

IV - Lecture et écriture de fichiers

Exemples:

```
1  def lecture_fichiers_valeurs_fn(nom_fichier):
2  |       f = open(nom_fichier , 'r') # fichier existant ouvert en lecture seule : 'r' = read
3  |       lignes = f.readlines()  # on lit toutes les lignes d'un coup.
4          la variable lignes est une liste dont chacune des composantes  est l'une des lignes du fichier.
5  |       n = len(lignes)
6  |       X, Y = [], []  # initialisation des abscisses et des ordonnées
7  |       for i in range(n):
8  |       |       coords = (lignes[i][:−1]).split('\t') # i-ième ligne de ligne = i-ième ligne du fichier
9  |       |       # On exclut le dernier caractère de la chaîne qui est le saut de ligne.
10 |       |        # on décompose la chaîne avec le séparateur '\t' : on récupère deux coordonnées.
11 |       |       X.append(float(coords[0]))  # abscisse sous forme d'un flottant
12 |       |       Y.append(float(coords[1]))
13 |       return X, Y  # on retourne la liste des abscisses et celle des ordonnées
```

V - Tracé graphique

Sous Python le module **matplotlib** possède une multitude de fonctionnalités de tracés.

```
>>> import matplotlib.pyplot as plt        # importation du module de tracé
>>> x = [i for i in range (-10, 11) ]      # création de la liste x
>>> y = [ i**2 for i in x ]                # création de la liste y tel que y = f(x) = x²
>>> plt.title ( 'Titre du graphe')
>>> plt.xlabel ( ' nom des abcisses')
>>> plt.plot ( x, y, label = ' x**2 ' )    # création de la courbe à partir des listes x et y
>>> plt.legend ( )
>>> plt.show ( )
```

```
>>> import matplotlib.pyplot as plt
>>> plt.plot (X,Y)          avec X et Y liste des abscisses et ordonnées de la courbe à tracer.
>>> plt.show()
```