3 | Boucles et fonctions

1 Instructions de contrôle

1.1 Définition

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

1.2 Boucle if, instruction conditionnelle

Syntaxe d'un test avec if :

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

Attention, l'indentation (c'est-à-dire le décalage) après les deux points est obligatoire en Python et permet de définir la séquence d'instructions qui sera effectuée selon la valeur des expressions. Toutes les instructions qui sont au même niveau font partie de la même séquence d'instructions.

Remarque : Le elif et le else ne sont pas nécessaire si l'on ne souhaite pas effectuer d'instruction en dehors du if.

1.3 Boucle for, instruction itérative

Syntaxe de la boucle for :

```
>>> 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. Les arguments i,j et k doivent ABSOLUMENT être des entiers, sinon on utilisera np.arange.

Attention, si vous devez modifier la séquence que vous êtes en train de parcourir dans la boucle, il est conseillé de faire une copie en premier sous peine de bogues difficiles à appréhender (risque de séquence infinie d'instructions). Par exemple la boucle suivante ne s'arrête jamais :

```
>>> for i in liste [ : ]:          # boucle for dans une copie de liste
>>>     liste.append ( i )         # ajout de i à la fin de la liste
```

Une boucle for peut aussi permettre de créer une liste en compréhension :

```
>>> [ x for x in range (25) if x**2 <= 25 ]
[0, 1, 2, 3, 4, 5]
```

1.4 Boucle while, instruction itérative

La boucle while est une boucle conditionnelle qui permet de répéter une instruction tant qu'une condition est vraie, le nombre d'itération n'est donc pas connu à l'avance.

Syntaxe de la boucle while :

```
>>> 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. A éviter cependant car cela implique souvent que l'on n'a pas su gérer un cas d'arrêt correctement.
Si la condition d'arrêt est mal choisie, incorrecte ou jamais vérifiée, la boucle tourne indéfiniment (on peut cliquer sur le carré rouge au dessus de la console sous spyder pour stopper l'execution). Il faudra donc toujours vérifier que la condition de la boucle while évolue afin de s'assurer que le programme se termine (cf. chapitre suivant).

2 Les fonctions

2.1 Découpage en fonctions

Lorsqu'un programme contient des parties de code similaires, ces parties doivent être placées dans des fonctions. Ce découpage d'un script en plusieurs fonctions (**modularité** d'un programme) permet d'**améliorer la lisibilité** d'un programme ainsi que son débogage. Cette modularité permet également de réutiliser une fonction plusieurs fois sans avoir à la modifier à plusieurs endroits si cela s'avérait nécessaire par la suite. Vous prendrez l'habitude d'entrer désormais tous vos scripts sous forme de fonctions.

2.2 Définition et appel

Syntaxe générale de définition d'une fonction :

```
>>> def nom_de_la_fonction (argument 1, argument 2, …)
>>>     ‘’’ Documentation de la fonction ‘’’      #permet de spécifier le rôle de la fonction
>>>     séquence d'instructions #On construit ce que la fonction doit renvoyer ou faire
>>>     return expression ou liste d'expression
```

Remarques :

- On appelle la fonction avec l'instruction nom_de_la_fonction(arguments). Un des points qui peut être long à comprendre est que l'argument ne doit pas être un cas particulier lors de l'écriture de la fonction, et sera matérialisé lorsque l'on appellera la fonction.

- Certaines fonctions ne contiennent pas de paramètres. Les instructions sont simplement exécutées.
- Le corps d'une fonction peut contenir d'autres fonctions. En revanche, tout ce qui est défini dans le corps de la fonction n'existe que dans ce corps (espace local). Les variables définies dans la fonction n'ont aucune existence en dehors de cette fonction (espace global) (exception pour les listes). C'est l'analogue des variables muettes en mathématiques.
- L'instruction return( ) provoque l'arrêt de la fonction. Toutes les instructions écrites après sont ignorées. Il est donc possible de la remplacer par print( ) si la fonction n'est destinée qu'à réaliser un affichage.

L'exemple suivant, simplissime, définit une fonction comme celle que l'on voit en maths.

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

Attention, les fonctions en informatique ne se résument pas à celles que l'on voit en maths, et peuvent effectuer des actions très diverses.

2.3 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

Syntaxe d'appel d'un module :

```
>>> 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.
```

3 Ecriture d'un programme

3.1 Spécification et annotations d'un programme

La spécification permet d'informer les utilisateurs de la tâche 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 résumée dans la « docstring », inscrite au début du corps de la fonction.

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

Le texte entre triples guillemets fournit une description de l'entrée, du traitement et du résultat. 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
```

Ou encore :

```
>>> 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.
```

Par ailleurs, un programme doit pouvoir être lu et relu facilement par toutes personnes intervenant dessus. Il est ainsi 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. Mais ça, vous le saviez déjà.

3.2 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. Cette stratégie permet d'éviter de faire des choses interdites (problème de typage, problème d'ensemble de définition…).

```
>>> 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
```

Exemples d'utilisation de cette fonction :

```
>>> 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
```

3.3 Débogage d'un programme

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

- SyntaxError : invalid syntax : l'interpréteur ne comprend pas les mots clés. Souvent dû à une faute de frappe, d'un ' : ', d'un ' *' ou d'une ' (' oublié . Il faut parfois chercher l'erreur dans les lignes qui précèdent
- IndexError : list index out of range : l'interpréteur cherche dans une liste (ou un tableau) un élément inexistant (hors des limites). N'oubliez pas que l'indice commence à 0.
- ZeroDivisionError : division par zéro
- NameError : l'instruction fait intervenir un nom (ou une variable) non défini(e)
- TypeError : l'instruction cherche à réaliser une action impossible pour le type de variable spécifié (par exemple diviser un nombre par une chaîne de caractère)
- ValueError : l'entrée utilisateur est invalide (ou absente)
- IOError (erreur input/output) : erreur de communication avec un module extérieur (fichier, port externe, etc …)

4 Lecture et écriture de fichiers

Il est possible d'accéder sous Python à des fichiers autres que des bibliothèques ou des scripts extérieurs. On peut par exemple interagir avec du contenu multimédia, un programme exécutable ou un fichier texte. Seul ce dernier cas sera exploré cette année.

Un fichier texte est un fichier qui peut être ouvert avec un éditeur de texte. Il existe une syntaxe simple en Python pour travailler avec ces fichiers. Les données stockées dans un fichier sont converties en chaînes de caractères et les méthodes existantes sur le type str permettent de traiter les données.

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. C'est le dossier de travail : si on ne précise rien de plus (exemple: "fichier.txt"), Python va opérer sur "fichier.txt" dans le même dossier que le programme lui-même.
- Le chemin absolu est celui de l'arborescence systématique de votre système de fichiers (exemple: "C://Users/Documents/IPT/fichier.txt"). Le chemin absolu permet d'accéder et d'opérer sur n'importe quel fichier présent dans la mémoire.

Ouverture d'un fichier :
La fonction open(nom_du_fichier) seule ne réalise rien de visible. Il est nécessaire d'affecter à une variable le résultat de cette ouverture.

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

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

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".

```
>>> objet = open ( ‘ fichier_test.txt ‘ , ‘r’ )
>>> objet.readline ( )          # lecture de la première ligne du fichier
>>> objet.readlines ( )         # lecture de toutes les lignes
```

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. Pour écrire un nombre dans un fichier, il faudra donc le convertir au préalable en chaîne (str(nombre)) et quand on lit un tel nombre, on récupère une chaîne de caractères qu'il faut convertir au préalable en flottant (float(chaîne)) ou en entier int(chaîne) si on veut faire des calculs avec.

Ecriture dans un fichier :

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

Fermeture d’un fichier :
Comme tout élément ouvert, il faut fermer un fichier avec Python après utilisation. Si on ne le fait pas, certains problèmes peuvent survenir. On utilise la fonction close.

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

On donne ci-dessous deux fonctions :

La première fonction prend en argument une fonction et des bornes a et b et qui calcule les coordonnées d'un échantillon de points de la courbe de la fonction sur [a,b]. Les données sont alors écrites dans un fichier texte dont le nom est passé en paramètre; on doit donc ouvrir ce fichier texte pour pouvoir y écrire.

La seconde fonction réalise l'opération inverse : elle ouvre le fichier existant, y lit les coordonnées des points et trace la courbe correspondante.

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

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

5 Tracé graphique

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

Syntaxe générale :

```
>>> 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 ( )
```

Le minimum requis pour tracer une courbe est la suite d'instruction suivante :

```
>>> 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()
```

Remarque : X et Y sont des listes qui correspondent aux abscisses et ordonnées de la courbe à tracer elles sont donc forcément de même taille (len(X) = len(Y)).

Table des matières