Rappels sur les bases de la programmation en python

Table des matières

Introduction

On rappelle dans ce qui suit quelques éléments de programmation en python. L'objectif n'est pas d'apprendre par cœur tous les éléments rappelés, mais de vérifier leur compréhension dans le TP associé puis de l'utiliser comme aide-mémoire dans les TP suivants.
Nous utiliserons en TP :

— Capytale, un environnement dédié à l'enseignement qui permet de faire du python en ligne, en passant par Toutatice ;
— Spyder, un éditeur de texte adapté à la programmation en python.

Progresser en programmation demande un minimum de pratique. Pour cela, il serait utile d'installer python sur un ordinateur auquel vous avez accès. Vous pouvez par exemple installer la distribution thonny (`https://thonny.org/`), ou miniconda (`https://docs.anaconda.com/miniconda/`).
On peut également utiliser python en ligne : `https://www.online-python.com/`.

1 Premiers pas

1.1 Calculs

On adopte ci-dessous la présentation qui sera celle de Capytale : ce que l'on tape est écrit après une invite de commande `ENTREE[.]` ; ce que l'on obtient après `SORTIE[.]`. On peut par exemple utiliser la fenêtre python comme une calculatrice :

```
ENTREE[1] : 2 + 2
SORTIE[1]: 4
```

Les principales opérations de calcul (sur les flottants) sont les suivantes : `+` (addition), `-` (soustraction), `*` (multiplication), `/` (division), `**` (puissance, par exemple `2**3`).
Les nombres décimaux s'écrivent avec un point (exemple `1.45`) et la multiplication ne peut pas être sous-entendue (on écrit `2*(1.56+3)` et non `2(1.56+3)`).

1.2 Variables

Dès que le calcul est un peu long (ou naturellement lorsqu'on écrit un programme), on est amené à utiliser des variables.
L'affectation d'une valeur à une variable se fait par le signe `=` :

```
ENTREE[2] :  a = 5
```

Sur la ligne précédente, la variable `a` a été créée et la valeur `2` lui a été affectée (rien n'est affiché). Si ensuite on effectue :

```
ENTREE[3] :  b = a * 2
```

une nouvelle variable `b` est créée et la valeur `10` lui est affectée ; pour afficher la valeur contenue dans `b` :

```
ENTREE[4] : b
SORTIE[4]: 10
```

ou bien (mieux car plus explicite)

```
ENTREE[5] : print(b)
SORTIE[5] : 10
```

On peut modifier la valeur contenue dans `a` :

```
ENTREE[6] : a = a + b + 1
```

```
ENTREE[7] : print(a)
SORTIE[7] : 16
```

On peut effectuer plusieurs affectations simultanées :

```
ENTREE[8] : c , d = 3 , 4
```

2 Types des variables

En python, on n'a pas besoin de déclarer le type d'une variable pour la créer (comme on l'a vu sur l'exemple précédent). On peut demander le type d'une variable ; par exemple après les commandes précédentes :

```
ENTREE[9] : type(a)
SORTIE[9] : int
```

2.1 Entiers (`int`)

On utilise plutôt le terme "integer" et python utilise `int`. Les opérations spécifiques aux entiers sont celles de la division euclidienne ; pour obtenir le quotient et le reste :

```
ENTREE[10] : 7 // 2
SORTIE[10] : 3
```

```
ENTREE[11] : 7 % 2
SORTIE[11] : 1
```

2.2 Flottants (`float`)

C'est le type avec lequel sont faits les calculs numériques. On a déjà vu la syntaxe des principales opérations. On peut également utiliser l'écriture scientifique (par exemple `1.7e-5`) :

```
ENTREE[12] : 10 * (1.7e-5)
SORTIE[12] : 0.00017
```

Cela ne pose pas de problème avec python de mélanger des types dans une opération (si le calcul a un sens bien sûr) :

```
ENTREE[13] : 1.53 / 2
SORTIE[13] : 0.765
```

2.3 Booléens (`bool`)

Les booléens sont les valeurs `True` et `False` . Elles peuvent également être affectée à des variable directement :

```
ENTREE[1] : A = True
            type(A)
SORTIE[1] : bool
```

ou bien comme étant la réponse à une condition :

```
ENTREE[2] : B = (2>3)
            print(B)
SORTIE[2] : False
```

Remarque : on a réinitialisé la numérotation des entrées et sorties pour signifier que l'ensemble des variables créées auparavant ont été éffacées.
Les principaux opérateurs permettant d'obtenir ainsi un booléen sont : `>`, `<`, `>=`, `<=`, `==`, `!=` (différent). Attention à ne pas confondre `=` (affectation) et `==` (test d'égalité).
Les principales opérations sur ces booléens sont les suivantes (en reprenant les variables définies précédemment) :

```
ENTREE[3] : B or A
SORTIE[3] : True
```

```
ENTREE[4] : B and A
SORTIE[4] : False
```

```
ENTREE[5] : not B
SORTIE[5] : True
```

2.4 Listes

Ce paragraphe est assez long : les listes jouerons un rôle important dans la construction de nombreux programmes.

2.4.1 Définition et premières propriétés

Le type liste est un type dit composé (les listes contiennent des éléments ayant un autre type. Elles peuvent être déclarée de manière directe :

```
ENTREE[1] : L1 = [1,2,3,4]
            type(L1)
SORTIE[1] : list
```

Elles peuvent contenir des objets de types différents :

```
ENTREE[2] : L2 = [ 1 , True , 3 , 5.23 ]
```

Les principales commandes de manipulation des listes sont les suivantes :

Obtenir la longueur d'une liste :

```
ENTREE[3] : len(L1)
SORTIE[3] : 4
```

Obtenir un élément d'une liste :

```
ENTREE[4] : L1[2]
SORTIE[4] : 3
```

Les éléments d'une liste `L` sont numérotés à partir de `0` et jusqu'à `len(L)-1`.
La valeur obtenue peut être affectée à une variable.

Modifier un élément d'une liste :

```
ENTREE[5] : L1[1] = 0
            print(L1)
SORTIE[5] : [1, 0, 3, 4]
```

Concaténer deux listes :

```
ENTREE[6] : L1 + L2
SORTIE[6] : [1, 0, 3, 4, 1, True, 5.23]
```

La multiplication d'une liste par un entier est à comprendre à partir de cette notion de concaténation :

```
ENTREE[7] : 3 * L1
SORTIE[7] : [1, 0, 3, 4, 1, 0, 3, 4, 1, 0, 3, 4]
```

Ajouter/enlever un élément à la fin d'une liste :

```
ENTREE[8] : L2.append(2)
            print(L2)
SORTIE[8] : [1, True, 5.23, 2]
```

Attention : cette commande ne crée pas une nouvelle liste ; elle modifie `L2`.
L'opération contraire, enlever un élément à la fin d'une liste, s'obtient par

```
ENTREE[9] : L2.pop()
            print(L2)
SORTIE[9] : [1, True, 5.23]
```

Cette opération modifie `L2` et renvoie la valeur du dernier élément de la liste.

Copier une liste :

```
ENTREE[10] : L3 = L1.copy()
```

On aurait pu penser à utiliser une simple affectation : `L3 = L1` ; mais cette façon de procéder pose des difficultés que nous n'abordons pas ici.

Tester si un élément est dans une liste :

```
ENTREE[11] : L  = [1,2,3,2,4,2]
             2 in L
SORTIE[11] : True
```

Compter le nombre d'occurences d'un élément dans une liste :

```
ENTREE[12] : L.count(2)
SORTIE[12] : 3
```

Compter le nombre d'occurences d'un élément dans une liste :

```
ENTREE[12] : del L[3]
             print(L)
SORTIE[12] : [1,2,3,4,2]
```

2.4.2 Sous-listes

Étant donnée une liste `L`, pour m et n tels que $0 \leq m \leq n \leq len(L)$:
- `L[m:n]` est la sous-liste constituée des éléments de `L` dont les indices sont compris entre `m` et `n-1`.
  On retiendra qu'en python, dans différentes circonstances, l'indice de début est inclus et celui de fin est exclu.
- `L[m:]` est la sous-liste constituée des éléments de `L` dont les indices sont supérieurs à `m`.
- `L[:n]` est la sous-liste constituée des éléments de `L` dont les indices sont strictement inférieurs à `n`.
- `L[:]` est l'ensemble de la liste.

2.4.3 Définition en compréhension

On reprend pour liste `L1` la liste `[1, 0, 3, 4]`.
Une manière, souvent pratique, de créer une nouvelle liste est la déclaration dite "en compréhension" :

```
ENTREE[11] : [2 * x  for x in L1]
SORTIE[11] : [2, 0, 6, 8]
```

Dans ce cadre, la commande `range` permet d'obtenir des listes simples d'entiers. Par exemple :

```
ENTREE[.] : [i  for i in range(1,5)]
SORTIE[.] : [1, 2, 3, 4]
```

La commande `range` ne fournit pas directement une liste :

```
ENTREE[.] : R = range(5)
            type(R)
SORTIE[.] : range
```

Elle fournit une structure itérable qui sera également utilisée dans les boucles.
Voici quelques variantes : étant donnés des entiers m, n, p,
- `[i for i in range(m,n)]` est la liste des entiers entre `m` et `n-1` inclus ;
- `[i for i in range(n)]` est la liste des entiers entre `0` et `n-1` inclus ;
- `[i for i in range(m,n,p)]` est la liste des entiers entre `m` et `n-1` inclus, avec un pas `p`.

On notera à nouveau la convention : dans `range(m,n)`, `m` est inclus et `n` est exclu.

2.5 Chaînes de caractères

Les chaînes de caractères ("string") sont utilisées quand l'on veut afficher du texte ou traiter des données se présentant sous forme de texte.
Pour déclarer une chaîne de caractères :

```
ENTREE[1] : chaine1 = 'abc'
            print(type(chaine1))
SORTIE[1] : str
```

ou

```
ENTREE[2] chaine2 = "d'ef"
```

L'utilisation des chaînes de caractères présentent certaines similitudes avec les listes :

— pour concaténer :

```
ENTREE[3] : chaine1 + chaine2
SORTIE[3] : "abcd'ef"
```

— longueur :

```
ENTREE[4] : len(chaine1)
SORTIE[4] : 3
```

— pour obtenir un élément :

```
ENTREE[5] : chaine1[1]
SORTIE[5] : 'b'
```

L'extraction de sous-chaînes fonctionne également comme l'extraction de sous-listes.
Par contre, on ne peut affecter une nouvelle valeur à un élément d'une chaîne : la commande `chaine1[1] = 'g'` provoquerait une erreur. Cela est caractéristique du fait qu'une chaîne de caractères est un objet "non mutable".

3 Bases d'algorithmique

3.1 Instructions conditionnelles

Pour conditionner l'exécution d'une commande à une certaine condition on utilisera la structure suivante :

```
if condition1 :
    instruction1
    instruction1bis
    ... # il peut y avoir plusieurs instructions
elif condition2 :
    instruction2
... # il peut y avoir plusieurs elif
else :
    instructionFinale
```

Un point important à noter dans cette syntaxe est l'utilisation des `:` et du décalage (utiliser la touche de tabulation). C'est un point essentiel en python. Pour indiquer le début et la fin des instructions à exécuter si `condition1` est vérifiée, on décale ces instruction vers la droite (le retour à l'alignement initiale signifiant que cette suite d'instruction est terminée).
A noter également : les conditions doivent être des booléens ; on peut bien sûr combiner plusieurs booléens par les commandes `or`, `and` ...

3.2 Boucle for

Pour répéter une opération n fois on utilise la syntaxe

```
for i in range(n):
    instruction_i
```

L'opération en question peut être toujours la même ou dépendre de `i`.
Exemple : le programme suivant calcule et affiche la somme des 100 premiers entiers (100 inclus) :

```
S = 0 # on initialise la somme \'a 0
for i in range(101) :
    S = S + i
print("Somme des 100 premiers entiers : ", S)
```

À noter :

— on utilise `range(101)` car dans la syntaxe du `range` le dernier terme est exclu.

— Tout comme pour le `if`, c'est le décalage qui signale le début et la fin du bloc d'instructions à effectuer à chaque tour dans la boucle.

Les objets obtenus par la commande `range` ne sont pas les seuls qui peuvent servir de structure itérable pour une boucle `for` : on peut effectuer cela en particulier sur une liste ou une chaîne de caractères. Exemple : le programme suivant calcule la somme des éléments de la liste `L` donnée :

```
L = [1,3,6,4,7] #Entr\'ee du programme
S = 0 # on initialise la somme \`a 0
for x in L :
    S = S + x
print("Somme des \'el\'ements de L : ", S)
```

Exemple : le programme affiche les lettres de la chaîne de caractère `chaine` :

```
chaine = 'abcd'
for lettre in chaine :
    print(lettre)
```

3.3 Boucle while

Lorsque la fin d'une boucle n'est pas conditionnée à un nombre de tour fixé à l'avance mais à la réalisation d'une condition, on utilise une boucle "while" ("tant que").
Exemple le programme suivant détermine et affiche la valeur du plus petit entier n tel que la somme des n premiers entiers dépasse (strictement) 1000 :

```
n = 0
S = 0
while S <= 1000 :
    n = n + 1
    S = S + n
print(n)
```

4 Utilisation des fonctions

Pour introduire la syntaxe des fonctions, on part de l'exemple suivant : écrire une fonction `moyenne` qui prend en paramètre deux entiers et renvoie leur moyenne.

```
def moyenne(a,b):
    moy = (a+b)/2
    return moy
```

À noter :
- Le mot clef est `def`.
- L'utilisation des `:`.
- La délimitation du bloc d'instruction à nouveau à l'aide du décalage.
- Le mot clef indiquant la variable dont la fonction renvoie la valeur est `return`. L'utilisation de ce mot clef arrête l'exécution du bloc d'instruction.
- Dans l'absolu, une fonction peut ne pas se terminer par un `return`. Elle peut aussi afficher une valeur, un graphique, modifier une liste... Nous nous en tiendrons cependant pour l'instant à cette syntaxe.

Pour appeler la fonction et afficher un résultat, on procède ensuite de la manière suivante :

```
print(moyenne(2,3))
```

On peut bien sûr utiliser ce résultat pour autre chose qu'un affichage, par exemple l'affecter à une variable que l'on va réutiliser dans un programme.
Prenons un exemple : on a deux listes `L1` et `L2` de même longueur et on souhaite obtenir la liste des moyennes des termes `L1[i]` et `L2[i]` :

```
L1 = [7,5,9,8]
L2 = [5,4,8,6]
L_moy = []
for i in range(len(L1)):
    L_moy.append(moyenne(L1[i],L2[i]))
```

On peut même réutiliser une fonction dans une autre fonction. La fonction suivante par exemple prend en argument deux listes `L1` et `L2` de même longueur renvoie la liste des moyennes des termes `L1[i]` et `L2[i]` :

```
def liste_moy(L1,L2):
    L_moy = []
    for i in range(len(L1)):
        L_moy.append(moyenne(L1[i],L2[i]))
    return L_moy
```

C'est le principal intérêt des fonctions : elles peuvent être réutilisées et permettent ainsi de construire un programme informatique par étapes.
Un dernier point important à comprendre est le suivant : une variable définie dans une fonction est interne à cette fonction. Par exemple, la variable `moy` ne peut être appelée en dehors de la fonction `moyenne` définie ci-dessus. La suite d'instructions :

```
a = moyenne(2,3)
print(moy)
```

renvoie un message d'erreur signifiant que le nom `moy` ne correspond à aucune variable définie. Par contre, la variable `a` pourrait être réutilisée dans un autre programme.

5 Utilisation de bibliothèques

Certaines fonctionalités dont nous aurons besoin ne sont pas intégrées à la base de python. Il faut faire appel à des bibliothèques pour pouvoir les utiliser.

5.1 La bibliothèque `numpy`

C'est le cas par exemple des fonctions mathématiques usuelles, qui sont contenues (entre autres) dans le module `numpy`.
Pour l'utiliser, on aura le choix entre les syntaxe suivantes :

```
ENTREE[.] : from numpy import *
            print(exp(2)) # on n'a pas besoin du mot clef numpy
SORTIE[.] : 7.38905609893065
```

```
ENTREE[.] : import numpy
            print(numpy.exp(2))
SORTIE[.] : 7.38905609893065
```

```
ENTREE[.] : import numpy as np
            print(np.exp(2))
SORTIE[.] : 7.38905609893065
```

5.2 La bibliothèque `numpy.random`

La bibliothèque `numpy.random` permet d'obtenir des nombres aléatoires. Après import de la bibliothèque par la commande `import numpy.random as rd` :

— `rd.randint(a,b)` renvoie un entier aléatoire compris entre `a` (inclu) et `b` (exclu).
— `rd.random()` renvoie un flottant aléatoire compris entre 0 et 1 (et choisi de manière uniforme).

Nous verrons d'autres bibliothèques en cours d'année.