Rappels de Python 2 : listes et fonctions

1 Listes

1.1 Création d'une liste

Quand on veut stocker plusieurs valeurs dans une variable, le plus courant en Python est d'utiliser une *liste*. Une liste est une séquence (suite) finie de valeurs séparées par des virgules, les crochets `[ ]` servent de délimiteurs :

```
li = [] # une liste vide
li = [1, 2, 3] # une liste qui contient trois éléments (ici trois entiers)
```

Remarque. Les commentaires

Il est fréquent dans un script d'ajouter des *commentaires* (du texte qui n'est pas du code, par exemple pour expliquer ce qu'on est en train de faire). Ces passages sont précédés du caractère dièse `#`, ils sont ignorés lors de l'exécution du code.

La fonction `len` renvoie la taille (le nombre d'éléments) de la liste :

```
print(len(li)) # on affiche la taille de li
```

On accède à un élément de liste par son *indice* écrit entre crochets. Attention, les indices commencent à zéro !

```
print(li[1]) # on affiche le DEUXIEME élément de li dans la console
li[2] = 0 # le TROISIEME élément de la liste est zéro
print(li) # on affiche à nouveau toute la liste
```

Exercice 1. Taper les lignes de code précédentes dans un script `listes.py` et exécuter.

Si la liste à créer contient beaucoup d'éléments, on peut la définir *en compréhension*, comme on définit les ensembles en compréhension en maths (par exemple, l'ensemble des 10 premiers entiers pairs peut se définir par $\{n \in \{0,1,2,...,9\} \mid n = 0 \,(\text{modulo}\, 2)\}$). Ce qu'on peut traduire en Python par :

```
li2 = [n for n in range(10) if n % 2 == 0]
```

Exercice 2. Dans le même script, définir et afficher une liste `li2` qui contient les 100 premiers entiers divisibles par 3.

Exercice 3. Dans le meme script, définir en compréhension une troisième liste `li3` qui contient les 30 premiers entiers impairs, en se basant cette fois sur la définition $n = 2p+1$. Afficher cette liste.

On peut également définir une liste en ajoutant ses éléments un par un grâce à la méthode `append`. Par exemple pour créer la liste des carrés des 10 premiers entiers :

```
li = [] # liste vide
for i in range(10): # on répète 10 fois avec i=0, 1, 2..., 9
    li.append(i*i) # on ajoute i^2 à la liste
```

Exercice 4. Écrire le code précédent dans le script et afficher la liste produite.

Exercice 5. Parcourir la liste précédente avec une boucle `for` pour ajouter 1 à chaque élément puis afficher à nouveau la liste.

Exercice 6. Importer les fonctions `sin` et `cos` du module `math`, puis créer une liste `li_sin2` des carrés des sinus des 10 premiers entiers, une liste `li_cos2` des carrés des cosinus des 10 premiers entiers. Parcourir ces deux listes pour produire la liste des carrés $\sin^2 + \cos^2$ pour les 10 premiers entiers.

Remarque. Pour importer les fonctions `sin` et `cos`, il suffit d'écrire en début de script :

```
from math import sin, cos
```

2 Fonctions

Les fonctions permettent de donner un nom à un bloc de code paramétré, ce qui évite par exemple de devoir copier-coller les lignes :

```
if a > b:
    print(a)
else:
    print(b)
```

À chaque fois qu'on veut déterminer le max de deux entiers. On définira donc une fonction `maxi` :

```
def maxi(a:int, b:int)->int:
    """
    Retourne le maximum des deux entiers a et b.
    """
    if a > b:
        return a
    else:
        return b
```

Qu'on appellera (= utilisera) à chaque fois qu'on en aura besoin. Autre avantage à utiliser la fonction `maxi` : à chaque appel, on n'affiche pas le résultat mais on le *retourne*, ce qui permet ensuite d'utiliser la valeur retournée au sein d'un calcul. Par exemple pour trouver le maximum de deux entiers, le multiplier par 2 et afficher le résultat :

```
n = 2
m = 3
print(2 * maxi(n, m))
```

La *documentation* de la fonction est constituée des types des arguments et de la valeur de retour (dans l'exemple précédent la fonction deux `int` (entiers) en argument et retourne un `int`) et du docstring (les lignes écrites entre les balises `"""` juste en dessous de la signature de la fonction). **Il faut toujours écrire cette documentation**.

Pour tous les exercices ci-dessous, écrire toutes les fonctions demandées dans un seul script `fonctions`.

Exercice 7. Écrire une fonction `carre(x)` qui renvoie la carré d'un nombre `x`. Tester la fonction pour `x = 3`.

Exercice 8. Écrire une fonction `distance` qui prend en argument les coordonnées de deux points du plan et qui renvoie la distance euclidienne qui les sépare.

Remarque. La distance entre les points $A(x_A, y_A)$ et $B(x_B, y_B)$ est $d_{\text{AB}} = \sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$. La fonction racine carrée s'importe par :

```
from math import sqrt
```

Exercice 9. Écrire une fonction `valAbs` qui renvoie la valeur absolue d'un nombre `x`.

Exercice 10. Écrire une fonction `dman` qui retourne la distance de Manhattan entre deux points A et B définie par $d_{\text{man}}(A, B) = |x_B - x_A| + |y_B - y_A|$ en se servant de la fonction `valAbs` précédente.

Exercice 11. Écrire une fonction `affiche_liste` qui parcourt un par un les éléments d'une liste pour les afficher sous la forme `indice : élément`.

Exercice 12. Écrire une fonction `zeros` qui prend un entier `n` en argument et qui renvoie une liste qui contient `n` fois l'entier 0.

Exercice 13. Écrire une fonction `zeros2D` qui prend deux entiers `n` et `m` en arguments et qui renvoie une liste de `n` listes qui contiennent toutes `m` fois l'entier 0.

Exemple. L'appel `zeros2D(2, 3)` devra retourner la liste `[[0,0,0], [0,0,0]]`.

Exercice 14. Écrire une fonction `reste` qui renvoie le reste de la division euclidienne d'un entier `a` par un entier `b`. Attention Seules les opérations d'addition et de soustraction sont permises ! Écrire de même une fonction `quotient`. Tester les deux fonctions avec `a=10` et `b=4`, puis avec `a=4 et b=10`.