TP 2 : La notion de fonction

Objectifs du TP. Écrire des fonctions !

1 Introduction à la notion de fonction (rappel)

En informatique, une fonction est proche d'une fonction en mathématiques : elle prend en paramètres des arguments et renvoie un résultat. La déclaration se fait avec le mot clef `def`, la structure générale étant la suivante :

```
def f(arguments):
   [instructions]
   return resultat
```

Par exemple la fonction `somme` qui suit, réalise la somme de deux entiers.

```
def somme(x,y):
   return x+y
```

On peut récupérer le résultat d'une fonction avec une instruction d'affectation, par exemple `s=somme(3,4)` affecte à `s` la valeur 7.

Dans une fonction en informatique, on peut utiliser des variables : sauf mention explicite du contraire, *toutes les affectations effectuées dans une fonction sont locales à la fonction*, c'est-à-dire que les variables introduites dans une fonction sont créées dans la fonction et détruites en sortie, indépendamment de l'existence d'une variable définie en dehors de la fonction et portant le même nom. On a vu en cours un script permettant de passer d'un entier à sa représentation en base $b \geq 2$. Voici une fonction prenant en entrée un entier N et un entier b, et renvoyant la liste des chiffres associés, en notation *little endian*.

```
def passage_base10_baseb(N,b):
    L=[]
    while N>0:
        L.append(N%b)
        N=N//b
    return L
```

Qu'importe s'il existe des variables `N`, `L` ou `b` définies en dehors de la fonction. Lorsque la fonction est appelée, par exemple avec `passage_base10_baseb(42,3)`, on réalise une association locale au corps de la fonction entre `N` et 42, ainsi que `b` et 3. Le corps de la fonction se déroule, puis lorsqu'on arrive en fin de fonction au niveau du `return`, avec `L` valant `[0, 2, 1, 1]`, cette liste prend la place de `passage_base10_baseb(42,3)` à l'endroit où la fonction a été appelée. Les variables `N`, `L` et `b` ne sont plus définies en dehors de la fonction (ou ont les valeurs qu'elles avaient avant l'appel à la fonction si elles étaient déja définies).

Notez qu'une fonction peut faire appel à d'autres fonctions, et agir sur l'environnement (par exemple afficher quelque chose à l'écran). L'instruction `return` n'est pas obligatoire, mais est impérative si on veut que la fonction « renvoie quelque chose ». Si `return` n'est pas utilisée, la fonction renvoie `None`, c'est-à-dire rien [1] : en particulier, on ne confondra pas `print`, qui affiche quelque chose à l'écran, et `return` qui permet de renvoyer quelque chose.

Une fonction peut très bien comporter plusieurs instructions `return`, mais dans ce cas, (au plus) une seule est exécutée. Par exemple :

```
def etape_syracuse(n):
    if n%2==0:
        return n//2
    return 3*n+1
```

Si n est pair, on sort de la fonction lorsqu'on rencontre `return n//2`, la fonction renvoie donc $n/2$. Dans le cas contraire, elle renvoie $3n+1$. Pour une telle fonction, c'est un peu plus clair d'utiliser un `else`, mais cela n'a rien d'obligatoire.

1. C'est un objet d'un type nouveau : `NoneType`. À vrai dire c'est le seul objet de ce type.

2 Premières fonctions.

Les exercices qui suivent demandent de manipuler instructions conditionnelles et boucles à l'intérieur de fonctions. Il est important de tester vos codes, dans la console Python par exemple. Vérifiez que vous obtenez comme le texte.

Exercice 1. Sans utiliser le module `math`, la fonction `abs`, la fonction `pow` ou encore l'opérateur `**` :

1. définir une fonction `absolue` prenant en entrée un entier et retournant sa valeur absolue (on ne demande pas de vérifier que le paramètre passé à la fonction est un entier).

```
>>> a=absolue(-4)
>>> b=absolue(2)
>>> a+b
6
```

2. définir une fonction `fact` prenant en entrée un entier positif et retournant sa factorielle définie par $n! = \prod_{i=1}^{n} i$.

```
>>> a=fact(4)
>>> a
24
```

Rappel : `range(k,m)` fournit les entiers de $[\![k, m-1]\!]$.

3. définir une fonction `puissance` prenant deux arguments x et n et retournant x^n. (On supposera que n est un entier positif, et on posera $x^0 = 1$ pour tout réel x).

```
>>> puissance(2,0)
1
>>> puissance(4,4)
256
```

Exercice 2. *Utilisation d'une fonction dans une autre fonction.*

1. Écrire une fonction `max2(a,b)` prenant en paramètre deux entiers et retournant le maximum des deux. *Remarque :* cette fonction existe déja en Python sous le nom `max`, on ne l'utilisera pas ici, on comparera simplement `a` et `b`.

2. En déduire une fonction `max3(a,b,c)` retournant le maximum de 3 entiers, et utilisant `max2`. *Remarque :* la fonction `max` de Python peut également être utilisée à la place de `max3`.

```
>>> max3(-2, -4, -3)
-2
```

3. Écrire une fonction `max_liste(L)` prenant en entrée une liste non vide et retournant son maximum, à l'aide de `max2`. *Remarque :* la fonction `max` de Python fonctionne également sur les listes.

```
>>> max_liste([1, 2, 8, 5, 4, 2])
8
```

Exercice 3. *Somme des chiffres d'un entier*. Écrire une fonction `sdc(n)` calculant la somme des chiffres de l'entier naturel n passé en paramètre. On utilisera des divisions euclidiennes par 10 pour obtenir les chiffres un par un.

```
>>> sdc(6416)
17
>>> sdc(sdc(sdc(4444**4444)))
7
```

3 Encore des listes

Rappels. On commence par un rappel du TP précédent, sous forme de démonstration Python (essayez si besoin).

```
>>> L=[1,5,7,8]
>>> L+[0,8,9]
[1, 5, 7, 8, 0, 8, 9]
>>> L.append(4)
>>> L
[1, 5, 7, 8, 4]
```

```
>>> L[0]
1
>>> L[3]
8
>>> list(range(1,8))
[1, 2, 3, 4, 5, 6, 7]
>>> list('truc')
['t', 'r', 'u', 'c']
>>> [x*x for x in range(1,5)]
[1, 4, 9, 16]
>>> [x*x for x in range(1,5) if x%2==0]
[4, 16]
>>> [0, 1]*3
[0, 1, 0, 1, 0, 1]
>>> [4]*15
[4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]
```

Slicing. Le slicing (« tranchage ») permet d'obtenir de nouvelles listes à partir d'une liste plus grande : `L[i:j]` est la liste constituée des éléments de `L` d'indice i inclus à j exclus. Cette liste est donc vide pour $i \geq j$:

```
>>> L = [1, 5, 7, 8, 4]
>>> L[0:2]
[1, 5]
>>> L[2:5]
[7, 8, 4]
>>> L[4:3]
[]
```

On n'est aussi pas obligé de mettre le premier (compris comme étant 0) ou le dernier (compris comme étant la longueur de la liste) :

```
>>> L[:2]
[1, 5]
>>> L[:]
[1, 5, 7, 8, 4]
```

Enfin, on peut spécifier un pas p, permettant de prendre seulement un élément sur p.

```
>>> L[0:5:2] #les éléments d'indice entre 0 inclus et 5 exclus, par pas de 2.
[1, 7, 4]
```

Ce pas peut-être négatif :

```
>>> L[3:0:-1] #de l'indice 3 inclus, à 0 exclus, par pas de -1.
[8, 7, 5]
```

C'est très pratique pour créer une nouvelle liste dont les éléments sont les mêmes, mais à l'envers :

```
>>> L[::-1]
[4, 8, 7, 5, 1]
```

Un avertissement pour finir :

```
>>> L
[1, 5, 7, 8, 4]
>>> T=L
>>> T[0]=0
>>> L
[0, 5, 7, 8, 4]
```

Attention, la ligne `T=L` ne crée pas de nouvelle liste, mais associe simplement le nom `T` à la même chose que `L` : les éléments en mémoire ne sont pas recopiés ! On a simplement copié la *référence* vers l'emplacement mémoire où sont stockés les éléments. Si l'on veut vraiment copier les éléments on utilisera l'une des syntaxes suivantes (ou une autre équivalente) :

```
>>> T=L[:]
>>> T=[x for x in L]
```

Les listes dans les fonctions. Attention, lorsqu'on passe une liste en paramètre d'une fonction, on passe la référence. Par conséquent, si on utilise `L[i]=x` dans une fonction dont `L` est un paramètre, le changement sera visible en dehors de la fonction puisque c'est la mémoire qui a été modifiée. Par exemple :

```
def fonction_stupide(L):
    L[0]=1
# Cette fonction ne renvoie rien, et modifie le
# premier élément de la liste passée en paramètre
```

```
>>> liste=[0,0]
>>> fonction_stupide(liste)
>>> liste
[1, 0]
```

Exercice 4. *Création de listes.*

1. Écrire une fonction `carres(n)` qui renvoie la liste des n premiers carrés (de 0^2 à $(n-1)^2$).
2. Écrire une fonction `non_div_3(n)` qui renvoie la liste des entiers de $[\![0, n-1]\!]$ qui ne sont pas divisibles par 3.
3. Écrire une fonction `addition(L1,L2)` qui prend en paramètres deux listes supposées de même taille, et qui renvoie la liste obtenue par addition terme à terme des éléments de `L1` et `L2`. Par exemple :

   ```
   >>> addition([0,2,5],[1,4,8])
   [1,6,13]
   ```

   Attention, votre fonction ne doit pas modifier les listes passées en paramètre, mais créer une nouvelle liste.
4. La liste ci-après représente une séquence d'ADN, donnée sous la forme d'une liste de chaînes de caractères.

   ```
   L=["A","C","G","T","T","A","G","C","T","A","A","C","G","G","A","T","C","C"]
   ```

   Les chaînes de caractères n'ont pas encore été vues, mais là on se contente simplement de chaînes d'un seul caractère. Une chaîne de caractères s'écrit entre guillemets, et on peut tester l'égalité de chaînes de caractères comme tous les types que l'on a vus jusqu'à présent. Écrire une fonction `complementaire(L)` prenant en entrée une telle liste et renvoyant la liste complémentaire. Elle s'obtient en remplaçant A par T, T par A, C par G et G par C. Attention : on ne modifiera pas la liste initiale, mais on créera une nouvelle liste. On suppose que la liste ne contient que les caractères `"A"`, `"C"`, `"G"` et `"T"`, et on pourra éventuellement écrire une fonction intermédiaire `comp_carac(c)` prenant en entrée un caractère et renvoyant un caractère.

Exercice 5. *Le juste prix !* On va créer un jeu du juste prix, aussi connu sous le nom de « C'est plus, c'est moins ». Le principe est le suivant : l'ordinateur choisi un nombre au hasard entre 1 et 1000. Vous essayez de deviner ce nombre, et pour ce faire, vous proposez des entiers au clavier. À chaque proposition, l'ordinateur vous dit si l'entier qu'il a choisi est plus grand ou plus petit que votre proposition. Le jeu s'arrête [2] lorsque vous donnez l'entier choisi, avec un message de félicitations.

1. Tout d'abord, votre programme doit choisir un entier au hasard. On va pour cela importer la fonction `randint` du module `random`. Le code suivant affiche à l'écran un nombre au hasard entre 1 et 1000.

   ```
   from random import randint
   print(randint(1,1000))
   ```

   Écrire une fonction `nombre_au_hasard()` (ne prenant pas d'argument) qui renvoie un entier aléatoire entre 1 et 1000.
2. Programmez maintenant le jeu proprement dit, sous la forme d'une fonction `juste_prix()` (ne prenant pas d'argument). Pour demander à l'utilisateur de donner un nombre, vous pouvez utiliser la fonction `input`. Cette fonction renvoie ce que l'utilisateur tape au clavier, sous forme de chaîne de caractères. Pour obtenir un entier, il faut convertir cette chaîne via la fonction `int()`. On peut placer un message comme paramètre à `input()`, sous forme de chaîne de caractères. Le code suivant convertit donc ce que l'utilisateur entre au clavier en entier et le stocke dans la variable `n`.

   ```
   n=int(input("Un nombre entre 1 et 1000 ? "))
   ```

   Voici un exemple de déroulement :

   ```
   >>> juste_prix()
   Un nombre entre 1 et 1000 ? 500
   C'est plus !
   Un nombre entre 1 et 1000 ? 750
   C'est plus !
   ```

2. On utilisera donc naturellement une boucle `while`.

```
Un nombre entre 1 et 1000 ? 875
C'est plus !
Un nombre entre 1 et 1000 ? 930
C'est plus !
Un nombre entre 1 et 1000 ? 965
C'est moins !
Un nombre entre 1 et 1000 ? 947
C'est moins !
Un nombre entre 1 et 1000 ? 938
C'est plus !
Un nombre entre 1 et 1000 ? 943
Bien joué !
```

3. Rajouter la fonctionnalité « Rejouer ? », une fois que l'utilisateur a trouvé le nombre. S'il répond « oui » ou « Oui », on recommencera, sinon on arrêtera.

4 Des exercices en plus

Exercice 6. *Un algorithme de tri.* On s'attaque maintenant à un problème un peu plus conséquent : le problème du tri d'une liste. On dit qu'une liste est triée si ses éléments sont dans l'ordre croissant. On donne ci-dessous en pseudo-code un algorithme permettant de trier une liste. Le but est d'écrire cet algorithme en Python. Vous pouvez

Algorithme 1 : Le tri par sélection du minimum

Entrée : une liste L
Effet : la liste est modifiée, pour être triée dans l'ordre croissant
pour i *variant entre 0 et longueur*$(L)-2$ *inclus* **faire**
  $m \leftarrow$ indice du plus petit élément de la liste situé entre l'indice i et la fin de la liste;
  si $m \neq i$ **alors**
    Échanger les éléments aux positions i et m de L.

générer facilement des listes aléatoires de tailles conséquentes avec la fonction `randint` du module `random` :

```
>>> from random import randint
>>> L=[randint(0,10) for i in range(20)]
>>> L
[1, 8, 6, 2, 5, 5, 0, 7, 10, 4, 7, 1, 5, 9, 1, 9, 5, 3, 5, 0]
```

L'appel `randint(a,b)` renvoie un entier aléatoire entre a et b (inclus).

Écrire une fonction `tri_selection(L)` permettant de trier la liste `L` :

— on repérera l'indice m avec une boucle `for`.

— pour réaliser l'échange de `L[i]` et `L[m]`, on utilisera une variable intermédiaire.

```
>>> L=[1, 8, 6, 2, 5, 5, 0, 7, 10, 4, 7, 1, 5, 9, 1, 9, 5, 3, 5, 0]
>>> tri_selection(L) #la fonction ne renvoie rien
>>> L
[0, 0, 1, 1, 1, 2, 3, 4, 5, 5, 5, 5, 5, 6, 7, 7, 8, 9, 9, 10]
```

Votre fonction ne renverra rien, mais modifiera la liste passée en paramètre. Remarque : la *méthode* `sort` de Python permet de trier une liste, elle s'utilise comme suit : `L.sort()`

Exercice 7. *Plus longue section croissante.* Écrire une fonction `plus_longue_section_croissante` ayant pour argument une liste d'entiers et qui renvoie la longueur d'une plus grande section croissante (c'est-à-dire d'éléments successifs) que l'on peut extraire de la liste ainsi qu'une telle section.

```
>>> L=[1,2,5,3,5,7,8,2,5,6]
>>> plus_longue_section_croissante(L)
(4, [3, 5, 7, 8])
```

Remarque : si `a` et `b` sont deux objets, `(a,b)` est le couple formé de ces deux objets, et `return (a,b)` permet de retourner un tel couple dans une fonction. Les parenthèses sont en fait facultatives et `return a,b` produit le même résultat. Les couples (et plus généralement les n-uplets) sont très similaires aux listes mais sont immuables : on ne peut modifier une composante ou rajouter un élément ; on doit créer un nouveau couple.

Exercice 8. *Plus longue sous-séquence croissante* **. Reprendre l'exercice 7 avec la *plus longue sous-séquence croissante*. Le principe est le même, mais on n'impose plus que les éléments soient contigus. On cherchera une solution la plus efficace possible.

```
>>> plssc([18, 4, 20, 10, 41, 11, 13, 2, 13, 38, 43, 14, 0, 21, 37, 31, 27, 16, 36, 36])
[4, 10, 11, 13, 13, 14, 21, 31, 36, 36]
```