4 - Fonctions

1. Définition d’une fonction
2. Appel d’une fonction
3. Variables locales et variables globales
4. Modules de fonctions

4.1 - Définition d'une fonction

- Les fonctions permettent de décomposer les programmes en sous-programmes et de réutiliser des morceaux de programmes.

- Une fonction est un programme Python défini à partir de **paramètres d'entrées** qui retourne éventuellement une **valeur de sortie**.

- La syntaxe de la définition d'une fonction Python est la suivante :

  def *<nom de la fonction>* ***(*** *<liste de paramètres>* ***) :***

  <bloc d'instructions>

- Une instruction ***return*** *<expression>* dans le bloc d'instructions définissant une fonction provoque la fin d'exécution de la fonction avec le retour de la valeur de l'expression qui suit.

4.1 - Définition d’une fonction

```
>>> def compter_lettre(lettre, texte) :
    n=0
    for c in texte :
        if c == lettre :
            n += 1
    return "nombre d'occurences de la lettre " + lettre + " : " + `n`

>>> print compter_lettre(‘e’, ‘je reviens’)
nombre d’occurrences de la lettre e : 3
>>>
```

4.2 - Appel d'une fonction

- Une fois qu'une fonction *f* a été définie, elle peut être utilisée dans une expression particulière qu'on nomme un **appel de fonction** et qui a la forme $f(v_1, v_2, \ldots, v_n)$, où $v_1, v_2, \ldots, v_n$ sont des expressions dont la valeur est transmise au paramètres. On parle d'un **appel de fonction par valeur** par opposition à un **appel par référence**.

- Python offre un mécanisme d'instanciation des paramètres par défaut. On peut écrire la liste des paramètres en entête d'une définition de fonction comme suit :

$$p_1, p_2, \ldots, p_k, p_{k+1} = expr_1, p_{k+2} = expr_2, \ldots, p_{k+n} = expr_n$$

Les k premiers paramètres doivent obligatoirement précisés à l'appel de fonction mais pas les n derniers. L'appel de fonction se fait donc avec *k* arguments au minimum et *k+n* arguments au maximum. Si un paramètre p_{k+i} n'est pas instancié explicitement, il prend la valeur par défaut de $expr_i$.

4.2 - Appel d'une fonction

```
>>>def pluriel(mot, famille = 'standard'):
    if famille == 'standard' :
        return mot + 's'
    if famille == 's':
        return mot
    if famille == 'oux' :
        return mot + 'x'
    if famille == 'al' :
        return mot[:-1] +'ux'
```

```
>>> print pluriel('maison')
'maisons'

>>> print pluriel('souris', 's')
'souris'

>>> print pluriel('chou', 'oux')
'choux'

>>> print pluriel('cheval', 'al')
'chevaux'

>>>
```

4.3 - Variables locales et variables globales

- Les variables qui sont introduites dans la définition d'une fonction peuvent être utilisées dans la suite de la définition mais pas à l'extérieur de la fonction. Ces variables sont dites **locales** par opposition aux **variables globales** qui sont introduites à l'extérieur de la définition d'une fonction et qui peuvent être utilisées à l'intérieur comme à l'extérieur de cette définition.

- Lorsque le même nom est utilisé pour introduire une variable locale et une variable globale, Python distingue bien deux variables différentes mais à l'intérieur de la définition de la fonction, c'est à la variable locale auquel le nom réfère.

4.3 - Variables locales et variables globales

```
>>> def f(x):
        y=2
        return x + y

>>> print f(3)
5
>>> print y

Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    print y
NameError: name 'y' is not defined
>>> u = 7
>>> def g(v):
        return u * v

>>> print g(2)
14
```

```
>>> def h(u):
        return u

>>> print h(3)
3
>>> print u
7
>>> def k(w) :
        u = 5
        return w+u

>>> print k(3)
8
>>> print u
7
>>>
```

4.4 - Modules de fonctions

- On peut ranger les définitions de fonctions se rapportant à une même application au sein d'un script commun baptisé **module**. Un module est sauvegardé sous forme d'un fichier dont le nom a la forme *<nom du module>.py*.

- Pour utiliser un module, il faut se servir de l'instruction *import <nom du module>.* L'exécution de cette instruction consiste à exécuter le script définissant le module (ce script peut contenir des instructions autres que des définitions de fonctions).

- Pour importer un module, Python a besoin de connaître le chemin qui permet d'accéder au fichier correspondant. Ce chemin doit apparaître dans la liste des chemins possibles stockés dans la variable *path* du module *sys.*

4.4 - Modules de fonctions

- Ensuite, pour utiliser les objets introduits dans le module, on les désigne par *<nom du module>.<nom de l'objet>.*

- Pour éviter de devoir préfixer les noms des objets par le nom du module, on peut pour l'importation utiliser l'instruction ***from*** *<nom du module>* ***import*** *<nom de l'objet>.*
  Pour importer tous les objets du module, on utilise l'instruction ***from*** *<nom du module>* ***import*** *

- Pour ranger les modules, on peut les grouper en paquets (packages) et réitérer cette opération de groupement au niveau des paquets eux-mêmes. On utilise encore l'instruction *import* pour importer des paquets ou des modules rangés dans des paquets. Il faut pour cela indiquer le chemin permettant d'accéder à l'objet à importer.

- Python fournit un ensemble de fonctions pré-définies mais aussi une bibliothèque standard de modules dont on a la description sur le Web à l'adresse : *http://docs.python.org/lib/lib.html*

4.4 - Modules de fonctions

```
# module concernant le cercle placé dans le fichier « /Users/perrier/Desktop/figures/cercle.py »
# « /Users/perrier/Desktop/figures » est un paquet de modules
pi = 3.14

def aire(rayon):
    return pi*rayon**2

def longueur(rayon):
    return 2*pi*rayon
```

```
>>> import sys
>>> sys.path.append('/Users/perrier/Desktop/')
>>> from figures import cercle
>>> r = 5
>>> print 'aire d\'un cercle de ', r, ' cm de rayon : ', cercle.aire(r), ' cm2'
aire d'un cercle de  5  cm de rayon :  78.5  cm2
>>> from figures.cercle import aire
>>> print 'aire d\'un cercle de ', r, ' cm de rayon : ', aire(r), ' cm2'
aire d'un cercle de  5  cm de rayon :  78.5  cm2
```

4.5 - Exercices

1. L'algorithme du Soundex est utilisé dans les bibliothèques pour représenter les noms de personnes. Il a l'avantage d'être peu sensible aux variations d'écriture des noms; par exemple, Jurafsky, Jarofsky, Jarovsky et Jarovski seront tous représentés par J612.

   Réaliser les différentes tâches suivantes qui composent l'algorithme dans l'ordre où elles sont décrites :

   a) Ecrire une fonction *soundex1* qui prenne un nom propre en entrée, conserve la première lettre du nom et supprime toutes les occurrences non initiales des lettres a, e, h, i, o, u, w, y.

   b) Ecrire une fonction *soundex2* qui prenne la sortie de *soundex1* en entrée et remplace les lettres autres que la première par des entiers selon les règles :

      b, f,p,v → 1 c,g,j,k,q,s,x,z → 2 d,t → 3 l → 4 m,n → 5 r → 6

4.5 - Exercices

c) Ecrire une fonction *soundex3* qui prenne la sortie de *soundex2* en entrée et remplace les suites de chiffres identiques par un chiffre unique (666 → 6)

d) Ecrire une fonction *soundex4* qui prenne la sortie de *soundex3* en entrée et enlève les chiffres au-delà des 3 premiers ou, s'il en manque pour aller jusqu'à 3, compléte par des zéros.

e) Ecrire une fonction *code_noms_propres* qui prenne en entrée un texte et retourne le même texte où les noms propres auront été codés selon l'algorithme du Soundex.

4.5 - Exercices

2. On considère des arbres binaires qui sont représentés comme des triplets (étiquette de la racine, sous-arbre gauche, sous-arbre droit). Les arbres vides sont considérés comme des arbres binaires particuliers représentés par ().
Par exemple, voici un arbre binaire :(2, (4, (), (1, (), ())), (4, (), ()))
Définir les fonctions suivantes qui manipulent de tels arbres et retournent :

   a) La hauteur d'un arbre.

   b) La liste des étiquettes des feuilles d'un arbre.

   c) La liste des étiquettes de tous les nœuds.

   d) Un arbre qui est l'arbre de départ où toutes les occurrences d'une étiquette ont été remplacés par une autre étiquette.

   e) L'arbre de départ retourné selon un axe vertical