1 TD 3 : Fonctions, Dictionnaires

(correction page ??)

Abordé lors de cette séance	
programmation	fonctions, dictionnaires
algorithme	cryptage de Vigenère

En cas de problème avec les accents, il faut ajouter au début du programme #`coding : latin – 1`.

Première demi-heure : Fonctions .

Les fonctions sont des portions de programmes qui reproduisent les mêmes instructions. La fonction suivante calcule un polynôme de second degré $x^2 + x - 5$. A chaque fois qu'on appellera la fonction `polynome`, elle fera le même calcul sur des `x` différents. Cela évite principalement d'avoir à recopier les mêmes lignes à chaque fois qu'on en a besoin.

```
def polynome ( x ) :
    return x*x + x - 5
```

Ce qui suit le mot-clé `return` est le résultat de la fonction. Il y a deux catégories de fonctions : celles qui existent déjà et celles que vous écrirez. La fonction `cos` existe déjà : elle fait un calcul que vous n'avez pas à réécrire. La fonction `polynome` décrite plus haut n'existait pas avant de l'avoir définie. On distingue la définition d'une fonction :

```
def polynome ( x, coefficient ) :
    return sum ( [ x**i * c for i,c in enumerate(coefficient) ] )
```

De son utilisation ou appel :

```
y = polynome ( 1.2, [ 1, 2, -1] )  # calcul de -x^2 + 2x + 1 pour x = 1.2
```

On peut appeler une fonction depuis une autre fonction. Une fonction peut prendre autant de paramètres que l'on veut à condition qu'ils aient des noms différents. On peut aussi leur associer une **valeur par défaut** :

```
from math import log  # on importe une fonction existante
def log_base ( x, base = 10 ) :
    return log (x) / log(base)
```

On peut écrire soit :

```
y = log_base (1000)      # identique à y = log_base (1000, 10)
z = log_base (1000, 2)   # logarithme en base deux
```

Les fonctions doivent porter des noms différents. Dans le cas contraire, seule la dernière existe.

```
def polynome ( x ) :       # remplacée par la seconde fonction
    return x*x + x - 5
def polynome ( x, coefficient ) :
    return sum ( [ x**i * c for i,c in enumerate(coefficient) ] )
```

Exercice 1 :

Les fonctions `chr` et `ord` sont symétriques l'une de l'autre : elles convertissent un nombre en lettre et réciproquement.

```
print ( chr( 65 ), chr (97) )
print ( ord("B"), ord ("b") )
```

Le symbol % permet d'obtenir le reste d'une division entière. Il faut écrire une fonction qui retourne la lettre suivante dans l'ordre alphabétique. La lettre qui suit `z` est `a`.

```
def lettre_suivante(lettre) :
    # ....
    return ....
```

Important

1. Une variable créée à l'intérieur d'une fonction n'existe pas à l'extérieur : c'est une variable locale

   ```
   def calcul(x) :
       y = x**2
       z = x + y

   print (z)
   ```

   Les variables `y` ou `z` ne servent que d'intermédiaire de calcul. Le seul résultat qui sort de la fonction suit le mot-clé `return`. Le programme précédent produit donc l'erreur suivante :

   ```
   Traceback (most recent call last):
     File "e.py", line 6, in <module>
       print (z)
   NameError: name 'z' is not defined
   ```

2. Sans mot-clé `return`, le résultat est `None`.

   ```
   def calcul(x) :
       y = x**2
       z = x + y

   a = calcul(x)
   print (a)       # affiche None
   ```

3. Une fonction prend fin dès l'exécution du premier mot-clé `return` rencontré lors de l'exécution.

```
def valeur_absolue(x) :
    if x < 0 :
        return -x
    else :
        return x
```

4. Une fonction peut être récursive : elle s'appelle elle-même.

```
def recursive(x) :
    if x / 2 < 1 : return 1
    else : return recursive (x/2) + 1
```

Seconde demi-heure : Dictionnaires .

Une liste est un containeur qui contient d'autres valeurs indexées par des entiers. Un dictionnaire est un containeur qui contient d'autres valeurs indexées par presque n'importe quoi.

```
d = { }   # un dictionnaire vide
d = { 'a' : 'acronym', 'b': 'bizarre' }  # un dictionnaire qui associe 'acroym' à 'a' et 'bizarre' à 'b'.
z = d ['a']   # z reçoit la valeur associée à 'a' et stockée dans le dictionnaire d
```

Quelques fonctions utiles :

```
d = { 'a' : 'acronym', 'b': 'bizarre' }
l = len(d)    # retourne le nombre d'élément de d
b = 'a' in d  # b vaut True si une valeur est associée à 'a', on dit aussi que la clé 'a' est présente
x = d.get ('a', '')  # x vaut d['a'] si la clé 'a' existe, il vaut '' sinon
```

On utilise un dictionnaire pour compter les lettres d'un texte par exemple :

```
texte = "exemple de texte"
d = { }
for c in texte :
    d[c] = d.get(c,0) + 1
print (d)
# affiche {'a': 1, 'p': 1, 'e': 5, 'd': 1, ' ': 2, 'x': 2, 't': 2, 'm': 1, 'l': 1}
```

Les valeurs peuvent être n'importe quoi, y compris des listes ou des dictionnaires. Les clés doivent être des types immuables (nombre, chaînes de caractères, tuple incluant des types immuables). Si vous utilisez un autre type, cela déclenche une erreur :

```
f = [3,4]
d[f] = 0
```

Erreur :

```
Traceback (most recent call last):
  File "e.py", line 2, in <module>
    d[f] = 0
TypeError: unhashable type: 'list'
```

Lorsqu'on affecte une valeur à une clé, le dictionnaire crée la clé ou remplace la valeur précédente par la nouvelle :

```
d = { }
d['a'] = 0   # création d'une clé
d['a'] = 1   # remplacement d'une valeur
```

On peut aussi créer un dictionnaire de façon compacte :

```
d = { s:len(s) for s in ['un', 'deux', 'trois'] }
```

Exercice 2

On considère une liste de mots :

```
mots = ['eddard', 'catelyn', 'robb', 'sansa', 'arya', 'brandon',
        'rickon', 'theon', 'rorbert', 'cersei', 'tywin', 'jaime',
        'tyrion', 'shae', 'bronn', 'lancel', 'joffrey', 'sandor',
        'varys', 'renly', 'a' ]
```

Ecrire une fonction qui retourne les mots qui ont un 'y' en seconde position ?

```
def mots_lettre_position (liste, lettre, position) :
    # ....
    return [ .... ]
```

Troisième demi-heure : Utilisation d'un dictionnaire

Votre réponse contient vraisemblablement une boucle. En construisant un dictionnaire à partir de la liste précédente, il est possible d'enlever cette boucle (**exercice 4**)

```
def mots_lettre_position (dictionnaire_bien_choisi, lettre, position) :
    return dictionnaire_bien_choisi. get ( .... , [] )
```

Comment faire ?

Quatrième demi-heure : Crypter et décrypter selon Vigenère

Le code de César est une permutation de lettre ou un décalage. Toutes les lettres du message sont décalées d'un nombre fixe :

```
JENESUISPASCODE
MHQHVXLVSDVFRGH
```

Le code de Vigenère introduit un décalage qui dépend de la position de la lettre dans le message à coder. On choisit d'abord un mot qui servira de code DOP puis on le traduit en décalages : [3, 14, 15]. Pour coder, on décale la première lettre de 3, la seconde de 14, la troisième 15, la quatrième de 3 à nouveau... L'objectif de cette partie est d'écrire une fonction qui code le message (**exercice 5**) et de comprendre comment elle peut être légèrement modifiée pour décoder (**exercice 6**).

```
def code_vigenere ( message, cle) :
    # ....
    return message_code
```

Pour aller plus loin ou pour ceux qui ont fini plus tôt .

Pour l'exercice de la partie C, combien de mots sont stockés dans le dictionnaire bien choisi ?