Chapitre 3

Fonctions

1 - Introduction

La programmation impérative est le style de programmation de C, le programme est constitué d'un ensemble de fonctions qui s'appellent les unes les autres. Les fonctions sont appelées avec des arguments et renvoient des valeurs. En python, comme en C, toutes les fonctions renvoient des valeurs (il n'existe pas de procédures). En python les arguments d'une fonction sont passés par références (et non par valeurs comme en C par exemple).

La notion de fonction est très importante en informatique. Elles permettent en effet de découper un problème en sous-problèmes, qui peuvent eux-mêmes être décomposés en sous-tâches. Cette conception des programmes permet une meilleure lisibilité du code. De plus, les fonctions sont réutilisables à plusieurs endroits du programme voire dans plusieurs programmes différents pour peu qu'on ait pris la peine de les rassembler au sein de modules.

Nous verrons dans un premier temps les fonctions prédéfinies et dans un second nous traiterons des fonctions originales, c'est à dire celles que vous développerez vous-même afin de traiter le problème particulier que vous avez à résoudre.

2 - Fonctions prédéfinies

Ce sont les fonctions de la bibliothèque standard de python. Comme nous l'avons déjà dit cette bibliothèque est composée de très nombreux modules.

Nous avons déjà vu la fonction `raw-input` qui permet de saisir une valeur rentrée par l'utilisateur, de même il existe la fonction `print` qui permet d'afficher à l'écran la valeur d'une expression. Nous avons également rencontré les fonctions `len`, `min`, `max`. Toutes ces fonctions sont des fonctions intégrées du langage python.

On peut également importer des fonctions de modules plus spécifiques de la bibliothèque standard, nous avons vu par exemple lors des exercices la fonction `sqrt` du module `math`.

Importer un module de fonctions

- Importer le contenu d'un module

```
import math
b = math.sqrt(a)
```

- Importer une partie d'un module

```
from math import sqrt
```

- Importer tout un module

```
from math import *
```

Pratique déconseillée – sauf exceptions – car rempli l'espace des noms car tous les identifiants du module sont chargés (exceptés ceux qui commencent par le caractère _).

Lister le contenu d'un module

La fonction standard `dir()` permet de lister le contenu d'un module (ou de tout autre objet)

```
>>>dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs',
'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10',
'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan',
'tanh']
```

La fonction `help`() est la fonction d'aide en ligne de python.
exemples : (une fois que module `math` a été importé…)

```
>>>help(math)
>>>help(math.ceil)
```

3 - Fonctions originales

Les fonctions et les classes d'objets sont différentes structures de sous-programmes qui ont été imaginées par les concepteurs de langage de haut niveau pour découper un problème en sous-tâches plus élémentaires.

3.1 Syntaxe

La syntaxe python pour la définition d'une fonction est la suivante :

```
def nomDeLaFonction(liste des paramètres formels):
     bloc d'instructions
     return valeur_resultat
```

Bien sûr il faut que les fonctions aient été définies avant d'être appelées et en python la fonction principale __main__ se trouve en fin de programme dans le corps principal du programme (pas besoin de la nommer). *Un fichier source python se lit donc de bas en haut*. En bas, on a l'ordre dans lequel les fonctions sont successivement appelées et en haut, leur définitions.

Exemples :

```
>>>def ajoute(a,b):    # a,b paramètres formels (définition)
...    return a+b
>>>ajoute(2,3)      # 2 et 3 paramètres réels (appel fonction)
5                  # résultat affiché(interpréteur interactif)

>>>def table7():         # exemple fonction sans paramètres
...    for i in range(1,11):
...        print '7 * ',i,' = ',7*i
>>>table(7)
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70

>>>def table(n):
...    for i in xrange(1,11):
...     print n,'*',i,'=',n*i
>>>table(7)
>>>table(4)
```

3.2 Paramètres par défaut

En python, il est possible de définir des valeurs par défaut pour les arguments d'une fonction. On obtient ainsi une fonction qui peut être appelée avec une partie seulement des arguments.
Il ne faut pas utiliser une séquence modifiable comme paramètre par défaut (nous allons voir pourquoi ultérieurement).

Exemple :

```
>>>def politesse(nom,civilite = 'Monsieur') :
...    print 'Veuillez agreer,'civilite,nom,'mes salutations\
...    distinguees'
>>>politesse('Durant')
```

```
Veuillez agreer Monsieur Durant mes salutations distinguees.
>>>politesse('Dupont','Mademoiselle')
Veuillez agreer Mademoiselle Dupont mes salutations
distinguees.
```

Dans la plupart des langages de programmation les arguments fournis lors de l'appel d'une fonction (paramètres réels) doivent être dans le même ordre que celui des paramètres formels correspondant à la définition de la fonction.
En python lorsque les paramètres ont reçu des valeurs par défaut (selon la syntaxe indiquée ci-dessus), ils ont chacun une étiquette qui va permettre de fournir les paramètres réels dans n'importe quel ordre lors de l'appel de la fonction (ce qui est bien pratique en pratique…).

Exemple :

```
>>>politesse(civilite = 'Mademoiselle',nom = 'Dupont')
Veuillez agreer Mademoiselle Dupont mes salutations
distinguees.
```

Pourquoi il ne faut pas utiliser une séquence modifiable comme paramètre par défaut d'une fonction...

Attention, la valeur du paramètre par défaut n'est évaluée qu'une fois. Ceci a son importance quand une séquence modifiable est utilisée comme paramètres par défaut car cela peut engendrer des résultats non attendus. Donc **retenir comme règle qu'il ne faut pas passer une séquence modifiable en argument par défaut d'une fonction**. Ou alors le faire comme dans la seconde version de fonctionTruc (exemple ci-dessous).

Exemple :

```
>>>def fonctionTruc(a,l=[]):
...     l.append(a)
...     return l
...
>>> print fonctionTruc(1)
[1]
>>> print fonctionTruc(2)
[1, 2]
>>> print fonctionTruc(3)
[1, 2, 3]
```

En fait les appels successifs partagent tous la même liste `l` qui accumule les résultats des appels successifs. Si vous voulez créer une liste à chaque appel il faut procéder de la manière suivante :

```
>>>def fonctionTruc(a,l=None]):
...     if l == None:
...         l = []
...     l.append(a)
...     return l
>>> print fonctionTruc(1)
[1]
>>> print fonctionTruc(2)
[2]
>>> print fonctionTruc(3)
[3]
```

3.3 Typage des arguments

En python le typage des variables est dynamique, ceci est également vrai lors du passage des arguments entre fonction appelante et fonction appelée.
Exemple :

```
>>> def begaiement(arg) :
...     print arg,arg,arg
...
>>> begaiement('zut')
zut zut zut
>>> begaiement(5)
5 5 5
>>> begaiement((4,8))
(4,8) (4,8) (4,8)
>>> begaiement(6**2)
36 36 36
```

3.4 Variables locales, globales

- Variables locales

Lorsque les variables sont définies à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On parle de *variables locales* à la fonction.

Exemple :

```
>>> del table(base, deb=1,fin=10):
...    for n in xrange(deb,fin+1):
...         print base,'*',n,'=',base*n
```

Les variables `n`,`deb`,`fin` et `base` sont des variables locales.
Chaque fois que python appelle la fonction `table()` il réserve pour elle dans la mémoire de l'ordinateur un nouvel espace des noms. Les contenus des variables `n`,`deb`,`fin` et `base` sont stockés dans cet espace des noms qui est inaccessible depuis l'extérieur de la fonction.

Si on essaie par exemple d'afficher la valeur de `base` à l'extérieur de la fonction, l'interpréteur génère un message d'erreur.

```
>>> base
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'base' is not defined
```

- Variables globales

Les variables définies à l'extérieur d'une fonction sont des *variables globales*. En cas de conflit entre une variable locale et une variable locale (même nom pour les 2 variables) c'est la variable locale qui a la priorité et qui masque donc la variable globale.

Exemple :

```
>>> a = 10
>>> def function(a,b):
...     a = b*b
...     print a,b
>>> function(2,3)
>>> print a
9 3  # a vaut 9 en local à la fonction function()
10   # a vaut 2 en global après la fonction function()
```

Il y a possibilité de déclarer comme globale une variable définie à l'intérieur d'une fonction, pour cela il faut utiliser le mot clé global avant le nom de la variable lors de l'affectation.
Exemple :

```
>>> def compte_appels():
...     global cpt
...     cpt += 1
...     print cpt
>>> cpt = 0
>>> for i in xrange(0,5)
...     compte_appels()
1
2
3
4
5
```

```
>>> def compte_appels():
...     cpt += 1
...     print cpt
>>> cpt = 0
>>> for i in xrange(0,5)
...     compte_appels()

UnboundLocalError: local
variable 'cpt' referenced
before assignment
```

Chapitre 4

Structures de données

1 - Retour sur les données de base

Voici quelques unes des spécificités de python concernant les instructions de base que nous n'avons pas encore eu l'occasion d'aborder.

- L'affectation multiple

Python permet d'affecter leur valeurs a plusieurs variables à la fois

```
>>> a,b,c,d = 3,5,7,9
```

Ceci permet par exemple de revisiter l'algorithme d'échange des valeurs de 2 variables :

```
>>> a,b = b,a
```

- Les comparaisons multiples

Python considère une expression du type a < x < b comme valide.
Exemple :

```
>>> def f(x) :
...     if 5 <= x <= 10 :
...         print x,'appartient a l'intervalle [5,10]'
...
>>> f(5)
5 appartient a l'intervalle [5,10]
>>> f(4.99)
```

- Le typage dynamique et les longs de précision infinie

```
>>> a,b=1,1
>>> for c in xrange (1,50):
...     print c,' : ',b,type(b)
```

```
...     a,b = b, a+b
....
44 : 1134903170 <type 'int'>
45 : 1836311903 <type 'int'>
46 : 2971215073 <type 'long'>
47 : 4807526976 <type 'long'>
48 : 7778742049 <type 'long'>
....
```

Python a converti automatiquement (en dynamique) `b` en long pour pouvoir encoder les valeurs au-delà du maximum représentable par un entier de 32 bits. Le type `long` permet l'encodage de valeurs entières avec une précision quasi-infinie. Une valeur ainsi définie peut en effet avoir un nombre de chiffres significatifs quelconque, ce nombre n'étant limité que par la taille de la mémoire disponible sur la machine.
Vous pouvez vous amuser à écrire une fonction factorielle et tester les limites de votre ordinateur.

2 - Retour sur les chaînes de caractères

2.1 Les chaînes sont des séquences non modifiables

Les chaînes sont des séquences ; c'est à dire des ensembles de caractères qu'on peut parcourir à l'aide d'index du début à la fin. Ce sont des ensembles ordonnés dans le sens ou le caractère str[1] suit le caractère str[0] et précède le caractère str[2].
Les chaînes de caractères sont des séquences modifiables ; c'est à dire qu'elles ne peuvent pas se trouver à gauche du symbole de l'affectation.
Exemple :

```
>>> txt='bonjour Toto'
>>> txt[0]='B'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

Comment faut-il faire ?
>>> txt =
Par contre nous avons vu des opérateurs * et + qui semblent permettre la modification des chaînes. L'opérateur * est l'opérateur de répétition, l'opérateur + est l'opérateur de la concaténation. Le fait que les mêmes opérateurs fonctionnent différemment selon le contexte (adition si opérandes numériques, concaténation si opérandes de type `<str>`) s'appelle surcharge des opérateurs. Ce mécanisme n'est pas autorisé dans tous les langages.
Exemple :

```
>>> m = 'zut' * 4
>>> m
'zutzutzutzut'
>>> m += 'Toto'
m
'zutzutzutzutToto'
```

Contrairement aux apparences python ne modifie pas `m` mais réserve en mémoire un nouvel espace pour une chaîne et fait pointer l'identifiant `m` sur ce nouvel espace mémoire !

2.2 Comment copier une chaîne en python ?

```
>>> txt = 'Bonjour Toto'
>>> s = txt
```

Cette suite d'instructions ne fait pas une copie de la chaîne `txt` mais fait pointer l'identifiant `s` sur la même zone mémoire que l'identifiant `txt`. Pour copier une chaîne il faut procéder comme suit :

```
>>> s = txt[1:]
```

Python crée une seconde zone mémoire et il fait pointer l'identifiant `s` sur cette zone.

2.3 Les chaînes sont des objets

En python tout est objet même si pour l'instant nous avons très peu évoqué ce point puisque nous l'utilisons pour l'instant uniquement pour faire de la programmation impérative. Les chaînes sont donc des objets non modifiables sur lesquels peuvent s'appliquer un certain nombre de méthodes (http://docs.python.org/lib/string-methods.html). Voici une liste non exhaustive des fonctions intégrées de traitement de chaînes de caractères en python.

méthode	*description*
`capitalize()`	Retourne une chaîne dont 1ière lettre en majuscule
`lower()`	Retourne une chaîne en minuscules
`upper()`	Retourne une chaîne en majuscules
`isalpha()`	Retourne vrai si les caractères d'une chaîne sont alphabétiques
`isalnum()`	Retourne vrai si tous les caractères sont alphanumériques
`isdigit()`	Retourne vrai si les caractères d'une chaîne sont numériques
`isspace()`	Retourne vrai s'il y a uniquement des espaces dans la chaîne
`islower()`	Retourne vrai si tous les caractères sont en minuscules
`isupper()`	Retourne vrai si tous les caractères sont en majuscules
`count(ss)`	Retourne le nombre d'occurrence de sous-chaîne `ss` dans `s`
`endswith(ss)`	Retourne vrai si `s` se termine par sous-chaîne `ss`
`startwith()`	Retourne vrai si `s` commence par sous-chaîne `ss`
`find()` et `rfind()`	Retourne le plus petit (plus grand) index où `ss` apparaît dans `s`
`index()` et `rindex()`	*Idem* `find` sauf que retourne `ErrorValue` si `ss` n'est pas dans `s`
`join(seq)`	Retourne une chaîne qui est la concaténation des chaînes dans `seq`
`replace(old,new)`	Remplace les occurrences de `old` par `new`
`split(sep)` et `rsplit()`	Retourne la liste des mots de chaîne séparés par `sep`
`splitlines()`	Retourne la liste des lignes d'une chaîne
`strip()`	Retourne une chaîne sans les espaces en début et fin de chaîne

Exemples :

```
>>> a = "Universite d'Evry"
>>> b = a.replace('Evry','Antibes')
>>> b
"Universite d'Antibes"
>>> a = 'pomme,poire,peche,abricot'
```

```
>>> l = a.split(',')     # separateur(s) de mots
>>> l
[pomme,poire,peche,abricot]
```

2.4 Formatage de chaînes

Il est possible de créer une chaîne en formatant son contenu au moyen de l'opérateur %. La syntaxe de formatage est identique à celle de la fonction printf de la librairie standard du C.
Exemple :

```
>>> coul ='verte'
>>> temp =25 + 5.2
>>> print "la couleur %s correspond a une temperature < %.2f
degres celsius" % (coul,temp)
la couleur verte correspond a une temperature < 30.20 degres
celsius
```

Rappel des formats de la fonction printf du C

format	*rendu*
%c	Caractère
%s	Chaîne
%f	Réel
%e	Réel (notation exponentielle)
%d (%x %o)	Entier en décimal (hexadécimal ou octal)
%%	Le caractère %
%.2f	Réel avec 2 chiffres après la virgule
%5.2f	Réel avec 3 chiffres avant la virgule et 2 chiffres après (si moins de 5 digits avant la virgule alors justification à droite : 45.2)
%-5.2f	Idem mais justification à gauche : 45.2)
%10d	Entier justifié à droite :" 1"
%010d	Idem + complété avec 0 : "000000001"
%-10d	Entier justifié à gauche : "1 "
%30s	Chaîne justifiée à droite " Quelle belle journee"
%-30s	Chaîne justifiée à gauche "Quelle belle journee "
%15.12s	Chaîne avec 15 digits et précision de 12, justifiée à droite " Quelle belle"
%-15.12s	Idem mais avec justification gauche : "Quelle belle "

Exemple :

```
>>> for montant in [1500,1232.50,-70.12,884.2343] :
...  print 'francs : %.2f  euros %.2f' % (montant,
montant/6.55957)
francs : 1500.00  euros : 228.67
francs : 1232.50  euros : 187.89
francs : -70.12  euros : -10.69
francs : 884.23  euros : 134.80
```

3 - Retour sur les listes

3.1 Parcours à l'aide des compréhensions de listes (comprehension list)

Nous avons déjà vu comme parcourir une liste à l'aide la boucle for.
Rappel :

```
>>> l = []
>>> dir(l)      # liste des méthodes attachées aux listes
>>> l2 = []     # liste des méthodes ne commençant pas par __
for elt in dir(l) :
...  if elt[:2] != '__' :
...        l2.append(elt)
>>> l2
['append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

Il existe une autre manière de faire, appelée compréhension de liste (*comprehension list* en anglais) qui est très utilisée en python et généralement préféré par l'usage. Les compréhensions de liste offrent une syntaxe concise pour construire des listes.
Exemple :

```
l2 = [elt for elt in dir(l) if not elt.startswith('__')]
```

La syntaxe générale des compréhensions de listes est la suivante :

```
newlist =  [f(elt) for elt in seq if predicate(elt)]
```

3.2 Les listes sont des séquences modifiables

Contrairement aux chaînes de caractères les listes sont des séquences modifiables, du coup un élément de la liste peut se retrouver à gauche de l'opérateur d'affectation.
Exemple :

```
>>> l  = [1,2,3,4,5]
>>> l[0] = 10       # autorisé
>>> l
[10,2,3,4,5]        # la liste l a été modifiée
```

Attention à ne pas modifier la liste sur laquelle vous itérez !
Ceci peut avoir des effets imprévus. Par exemple :

```
>>>l = ['ab','cd','ef','gh','ij','kl']
>>>for elt in l :
...  print elt,
...  l.remove(elt)
...  print l
ab ['cd','ef','gh','ij','kl']
ef ['cd','gh','ij','kl']
ij ['cd','gh','kl']
```

Pour éviter cela, il faut itérer sur une copie de la liste (crée grâce à [:]), cette copie peut-être anonyme comme dans l'exemple ci-dessous.

```
>>>l = [‘ab’,’cd’,’ef’,’gh’,’ij’,’kl’]
>>>for elt in l[:]:
...  print elt,
...  l.remove(elt)
...  print l
ab ['cd,', 'ef', 'gh', 'ij', 'kl']
cd, ['ef', 'gh', 'ij', 'kl']
ef ['gh', 'ij', 'kl']
gh ['ij', 'kl']
ij ['kl']
kl []
```

3.3 Les listes sont des objets

On peut donc leur appliquer un certain nombre de méthodes. Pour savoir lesquelles il suffit d’appeler la fonction dir (exemple en haut de la page) et/ou help la fonction d’intérêt.

```
>>>dir(l)
>>>help (l.extend)
extend(...)
    L.extend(iterable)  --  extend  list  by  appending  elements
from the iterable
```

Méthodes	*Description*
`append(elt)`	Ajoute `elt` à la liste
`count()`	Compte les éléments de la liste
`extend(seq)`	Etend la liste avec les éléments de la sequence
`index(elt)`	Renvoie l’index de l’élément `elt`
`insert(pos,elt)`	Insére `elt` immédiatement avant l’index `pos`
`pop()`	Supprime le dernier objet après l’avoir retourné
`pop(index)`	Retourne l’élément à la position `index` et le supprime
`remove(elt)`	Supprime l’élément ayant la valeur `elt`
`reverse()  ***`	Renverse la liste – modifie l’objet sur lequel elle est appelée
`sort()  ***`	Ordonne la liste - modifie l’objet sur lequel elle est appelée
`del nombre[1:3]`	Enlève les éléments d’index 1 à 3 (3 non inclus)

4 - Les dictionnaires

Les types composites abordés jusqu’à présent (chaînes, listes et tuples) étaient des séquences, c’est à dire des suites ordonnées d’éléments auxquels on pouvait accéder à l’aide d’index (entiers) si on connaissant leur position dans la séquences (opération de *slicing* ou tranches).

Les dictionnaires constituent un autre type composite qui ressemblent aux listes dans la mesure où ils sont modifiables mais qui ne sont pas des séquences. *Les éléments rangés dans un dictionnaire ne sont pas dans un ordre immuable*. On accède à n’importe lequel d’entre eux grâce à une *clé* qui à la différence des listes peut être autre chose qu’un entier (en particulier des chaînes de caractères). Les clés peuvent être numériques, alphabétiques voire composites (sous certaines conditions).

4.1 Création d'un dictionnaire

Les dictionnaires sont notés entre accolades, comme un ensemble de couples clé/valeur {1 :'a', 2 :'b', 3 :'c'}. On rappelle qu'il n'y a pas de notion d'ordre dans un dictionnaire.
Création d'un dictionnaire :

```
>>> notes={}
>>> notes['Toto'] = 10   #clé = 'Toto' et valeur = 10
>>> notes['Titi'] = 12
>>> notes['Tata'] = 8
>>> notes
{'Toto' :10, 'Titi' :12, 'Tata' :8}
>>> print notes['Titi']
12
```

4.2 Opérations sur les dictionnaires

Opérations	*Descriptions*
`len(a)`	Nombre de valeurs dans le dictionnaire `a`
`a[k]`	Renvoie la valeur associée à la clé `k`
`a[k] = y`	Associe la valeur `y` à la clé `k`
`del a[k]`	Supprime la valeur `a[k]`
`k in a`	Vrai s'il existe une valeur associée à la clé `k`
`k not in a`	Vrai s'il n'y a pas de valeur associée à la clé `k`

4.3 Les dictionnaires sont des objets

Méthodes	*Descriptions*
`keys()`	Retourne la liste des clés du dictionnaire
`values()`	Retourne la liste des valeurs du dictionnaire
`items()`	Retourne la liste des couples (clé,valeur)
`has_key(k)`	Retourne vrai s'il existe une valeur associée à la clé `k`
`get(k[,d])`	Retourne la valeur associée à la clé `k` ou `d` s'il n'a pas de valeur associée (`d` vaut `None` par défaut)
`setdefault(k[,d])`	Retourne `a.get(k,d)` et fait `a[k]=d` si `k` n'est pas dans le dictionnaire `a`
`clear()`	Vide le dictionnaire
`copy()`	Retourne une copie du dictionnaire
`update(dict)`	Met à jour le dictionnaire à partir d'un autre dictionnaire

Remarquez que pour ajouter une nouvelle valeur il suffit de créer un nouveau couple (cle,valeur).

```
>>> fruits = {'pomme' :'verte', 'banane' :'jaune' }
>>> fruits['cerise']='rouge'
>>> fruits.keys()
['pomme','banane','cerise']
```

Exercices du TD2