Fonctions

1 Notion de fonction

Certaines suite d'instructions dépendant de paramètres sont utilisées plusieurs fois dans un programme. On peut isoler ces instructions répétées dans une fonction.

Une fonction (au sens informatique) en Python possède :

- **un nom** avec les mêmes règles que les noms des variables.
- **des paramètres (ou arguments)**. Ceux-ci sont placés entre parenthèses. Occasionnellement certaines fonctions n'ont pas de paramètre, la parenthèse est alors vide.
- **un corps** : c'est la séquence d'instructions qui est répétée lors de l'appel à la fonction.
- **un retour de résultat** : c'est le résultat du calcul lorsque la fonction est écrite pour en effectuer un. Si toutes les fonctions au sens mathématique ont un retour (qui correspond à l'image) ce n'est pas toujours le cas en informatique (notamment lorsque la fonction est utilisée pour produire un affichage)

Une fonction en Python est introduite par le mot clé **def** :

```
def nom_de_fonction(parametres) :
    corps de la fonction
    return x #Le retour (éventuel) de la fonction
```

On retrouve l'indentation habituelle dans Python : le corps de la fonction est indenté d'une tabulation et celui-ci s'arrête lorsque l'indentation s'arrête.

Il est à noter que le mot-clé return interrompt l'exécution de la séquence d'instructions d'une fonction. Ceci peut occasionnellement être pratique pour gérer des cas exceptionnels.

Les variables qui figurent comme paramètres dans la définition de la fonction sont appelés arguments formels de la fonction. Les paramètres passés à la fonction lors de son appel sont appelés arguments effectifs de la fonction.

2 Fonctions et variables

En Python, on distingue deux sortes de variables : les variables globales et les variables locales.

Les variables globales sont des variables qui sont stockées tout au long de l'exécution d'un programme. À l'inverse, les variables locales sont des variables qui ne sont stockées en mémoire que durant l'exécution d'une fonction et qui sont effacées après celle-ci.

Considérons par exemple :

```
def somme_des_n_entiers_puissance_i(n,i):
    somme=0
    for indice in range(n):
        somme = somme + indice**i
    return somme

a=input("Combien d'entiers ? ")
p=input("A quelle puissance ? ")
print(somme_des_n_entiers_puissance_i(a,p))
```

Dans ce programme, les variables a et p sont des variables globales. La variable somme est locale à la fonction somme_des_n_entiers_puissance_i. On dit que la variable somme a une portée limitée

au corps de la fonction somme_des_n_entiers_puissance_i. Les variables globales ont une portée qui s'étend sur l'ensemble du programme.

Par ailleurs, la fonction somme_des_n_entiers_puissance_i a deux arguments formels qui sont respectivement n et i et est appelée avec deux arguments effectifs qui sont a et p.

Si l'on souhaite modifier une variable lors de l'exécution d'une fonction, on utilisera le mot clé global :

```
def nom_de_fonction(parametres):
   global x # La variable x a une portée qui sera le programme
   Corps de la fonction
```

De façon générale on préférera utiliser des variables globales pour représenter les constantes d'un problème. Il est préférable de leur donner des noms explicites, là où les variables locales, qui sont le plus souvent temporaires pourront avoir des noms plus courts. On évitera aussi souvent que possible d'utiliser le mot-clé "global" en Python. D'une part les variables globales occupent de l'espace mémoire inutile et d'autre part la compréhension d'un programme est nettement plus complexe avec l'usage de la syntaxe "global".

Exercice 1 : Les différentes variables de ce programme sont-elle locales ou globales ?

```
def f(x):
   somme = 0
   for k in range(1,x+1):
      somme = somme + k
   return somme

N=int(input("Quelle est la longueur de votre tableau ? "))
for i in range(N):
   j=i*(i+1)/2
   print("La somme des "+str(i)+" premiers entiers est de : "+str(f(i))+" par calcul
   direct et est de : "+str(j)+" par la formule")
```

3 Fonctions et documentation de programmes

Lorsqu'on conçoit une fonction, il est naturel de se poser la question de sa réutilisation ultérieure éventuelle (que ce soit dans le cadre du projet sur lequel on travaille, dans le cadre d'un projet ultérieur, ou pour d'autres développeurs). Il convient dès lors :

- De donner des noms explicites aux fonctions que l'on code
- De spécifier les hypothèses faites sur les arguments formels, leur relation avec le résultat renvoyé et les effets éventuels de la fonction (affichage...)

Python propose un mécanisme pour documenter les fonctions, sous la forme d'un commentaire placé avant le corps de la fonction, dans une chaîne de caractères. Cette aide est accessible par la commande "help f".

Par exemple :

```
def Hypothenuse(x,y):
   """Calcule l'hypothenuse d'un triangle de petits cotes x et y (nombres)"""
   return sqrt(x**2+y**2)
```

4 Problèmes liés à l'usage des fonctions

4.1 Ordre d'évaluation

Dans certains cas, le résultat de l'appel à une fonction dépend de l'ordre dans lequel les arguments sont évalués lors de l'appel

Exercice 2 : Que renvoie ce programme suivant que les arguments de somme sont évalués de gauche à droite ou de droite à gauche ?

```
n=0
def g(x):
   global n
   n=n+1
   return x+N
def somme(x,y):
   return x+y
print(somme(n,g(1))
```

Si pour le moment en Python, l'évaluation se fait toujours de gauche à droite, il n'en est pas forcément de même dans les autres langages, et n'en sera pas forcément de même dans les versions ultérieures de Python. Ainsi, on a intérêt à éviter d'écrire des fonctions dont le résultat dépend de l'ordre de l'évaluation des arguments.

4.2 Passage par valeur

Considérons le programme suivant :

```
def carre(x):
   x = x**2
   return x

a=3
print(f(a))
print(a)
```

Ce programme affiche évidemment 9 sur la première ligne de l'écran.
Pourtant de manière plus surprenante c'est 3 qui est affiché sur la seconde ligne !

On part en effet de l'état :

a	3

Mais dès lors que l'on appelle f(a) on se retrouve dans l'état suivant :

a	3

x	3

où la nouvelle variable x a reçu la valeur de a.
Après l'instruction x=x**2, l'état du programme est :

a	3

x	9

La fonction f renvoie la valeur de x, soit 9, et l'état du programme devient immédiatement après :

a	3

La variable x a ainsi disparu. Elle se comporte donc exactement comme une variable locale : le paramètre formel a une durée de vie limitée à l'exécution de la fonction.

Ce mécanisme, dont il faudra tenir compte lorsqu'on cherchera notamment à modifier une liste (par exemple pour la trier) s'appelle le passage par valeur lors de l'appel de fonctions. En effet seule la valeur de l'argument effectif est passée à la fonction et non l'argument effectif lui-même.

Exercice 3 Qu'affiche le programme suivant ?

```
def echange(a,b):
   z=a
   a=b
   b=z

a=1
b=2
echange(a,b)
print(a)
```

5 Fonctions comme variables de première classe

Les fonctions sont considérées comme des variables à part entière en Python. On dit que ce sont des variables de première classe.

Les fonctions peuvent ainsi être passées en argument, renvoyées comme résultat ou encore stockées dans une variable.

Une application immédiate est la définition d'un opérateur mathématique par une fonction. Reprenons l'exemple du calcul approché de la valeur d'une intégrale par la méthode des rectangles à gauche vu en T.P. :

```
def Rectangle(a,b,N,f):
   valeur_somme = 0
   k=(float(b-a))/N
   for i in range(N):
      valeur_somme=valeur_somme + k*f(a+i*k)
   return valeur_somme
```

Pour calculer une valeur approchée de l'intégrale de $I = \int_0^1 \frac{dx}{1+x^2}$ (intégrale qui vaut en théorie le quart de π et nous a permis d'obtenir une valeur approchée de π on peut ainsi préalablement définir une fonction Fraction :

```
def Fraction(x):
   return 1./(1+x**2)
```

et obtenir par exemple une valeur approchée de I (avec 100 pas d'intégration) à l'aide de l'instruction Rectangle(0,1,100,Fraction).

Python offre une autre possibilité, qui est l'usage d'une fonction anonyme. En effet notre fonction Fraction est une simple expression. À l'aide de la syntaxe :

lambda x :e

où e est une expression dépendant éventuellement de la variable x, on peut réduire notre calcul d'intégrale à l'expression :

```
Rectangle(0,1,100,lambda x:1./(1+x**2))
```

Il est également possible d'écrire des fonctions qui retournent des fonctions :

Exercice 4 : Que renvoie la fonction suivante ?

```
def h(f,g):
   return lambda x:f(g(x))
```

6 Fonctions prédéfinies

Tous les langages de programmation répandus proposent des fonctions toutes faites pour la plupart des besoins courants, écrits par les concepteurs du langage ou par des utilisateurs au fil du temps puis intégrés dans les distributions du langage. Python n'échappe pas à cette règle. Ces différentes fonctions sont organisées en modules qui les regroupent autour de thématiques variées, comme des fonctions mathématiques, des fonctions de traitement d'image, de représentation exacte des fractions, la compression de fichiers, les protocoles réseau..

Parmi les modules les plus utiles dans ce cours, citons :
- **math** qui contient les fonctions usuelles en Analyse ;
- **random** qui sert à générer des nombres pseudo-aléatoires ;
- **time** qui permet de travailler avec l'horloge interne du système et donc de mesurer le temps écoulé lors de l'exécution de fonctions ;
- **numpy** qui fournit des outils pour le calcul scientifique, notamment sur la manipulation de matrices ;
- **scipy** qui s'appuie sur numpy et fournit des outils de calcul scientifique ;
- **matplotlib** qui est une bibliothèque de tracés graphiques.

Pour utiliser les fonctions d'un module, on commence par importer ce module une fois pour toutes, en début de session ou en début de programme. Par exemple :

```
import math
```

Ensuite, on peut utiliser dans la suite de la session ou du programme les fonctions de ce module en faisant précéder leur nom par celui du module. Par exemple :

```
math.log(3)
```

Une autre forme possible d'importation est d'utiliser :

```
from math import log
```

On peut alors utiliser directement la fonction log.

Enfin, nous importerons couramment des modules en leur donnant un nom abrégé, par exemple :

```
import matplotlib.pyplot as plt
```

On pourra alors utiliser la fonction `matplotlib.pyplot.plot` à l'aide de `plt.plot`
Ceci permet de lever les ambiguïtés sur un nom de fonction existant dans plusieurs modules. Par exemple les modules `math` et `numpy` disposent tous deux d'une fonction `cos`.

Lorsqu'on souhaite utiliser un très grand nombre de fonctions d'un module, il est possible d'importer toutes les fonctions du modules à l'aide de :

```
from module import *
```

Cette syntaxe, lourde pour la mémoire, est à éviter lorsqu'on ne souhaite utiliser qu'un nombre très limité de fonctions d'un module. Néanmoins, dans le cadre d'un projet un peu complexe, on sera amenés à regrouper nos fonctions en plusieurs modules. On importera alors le plus souvent toutes les fonctions des modules que l'on a créé précédemment.

Python propose plusieurs centaines de modules, certains comportant un très grand nombre de fonctions. Il est donc exclu de tous les connaître. Un programmeur qui veut être efficace à intérêt d'utiliser des fonctions de bibliothèques. Il doit donc savoir chercher la documentation qui lui est utile. Beaucoup de langages de programmation proposent une documentation officielle en ligne. Pour Python celle-ci peut être consultée sur : http ://docs.python.org/