

Fonctions : Structurer un programme

1 Introduction

Les fonctions informatiques permettent de structurer un programme informatique compliqué en une architecture d'éléments plus simples.

1.1 Analogie avec les fonctions mathématiques

Soit $f : \mathbb{R} \rightarrow \mathbb{R}$



```
def square(x):
    y=x*x
    return(y)
```

1.2 Fonctions de service

Certains téléphones sont équipés de la fonction **réveil**. Elle déclenche une sonnerie si l'heure de l'horloge correspond à l'heure définie de réveil.



```
def reveil ( heure_reveil ):
    if heure_reveil ==horloge():
        sonner()
```

1.3 Usage d'une fonction

Le langage python inclut 69 fonctions de bases, en plus des opérateurs de calculs. Ces fonctions utilisent un certains nombres d'arguments et renvoient certains résultats. Pour connaître les arguments et le resultat, il suffit d'utiliser la fonction `help()`. Par exemple :



```
>>> help(abs)
Help on built-in function abs in module builtins:
abs(x, /)
    Return the absolute value of the argument.
>>> abs(-3.2)
3.2
```

Quand on veut utiliser une fonction, pour l'appeler, on a besoin des informations suivantes :

- son nom ;
- ses paramètres et leur type ;
- le type de la valeur de retour (s'il y en a une).

Ces informations sont communément appelées la spécification de la fonction.

2 Ecrire une fonction

Un programmeur peut aussi créer ses propres fonctions. Il faut prendre le soin de bien les spécifier pour qu'elles soient utilisables par d'autres personnes. L'écriture d'une fonction reprend les principes d'écriture d'un programme auxquels s'ajoutent des éléments complémentaires.

2.1 Contexte d'usage : interaction d'une fonction

2.2 Elements de rédaction d'une fonction

- Une fonction, sous python, est introduite par le mot clé **def**. Elle possède :
- un **nom** (mêmes règles que pour les noms de variables) ;
 - des **paramètres** ou **arguments**, placés entre parenthèses.
 - un **corps** : séquence d'instructions exécutée lors de l'**appel** de la fonction ;
 - un retour de résultats (optionnel) introduit par le mot-clé **return**.

```
python
def nom_de_la_fonction(arguments)
    corps_de_la_fonction
    return(valeurs)
```

Remarques :

- Les fonctions doivent être définies en début de programme ;
- Le corps de la fonction est indenté d'une tabulation ou 4 espaces ;
- La fin de la fonction est indiquée par l'indentation ou lorsque le mot-clé **return** est atteint.

2.2.1 Documentation d'une fonction

Outre le fait qu'il convient de donner un nom explicite à une fonction, il est vivement recommandé de la documenter, c'est-à-dire de spécifier les prérequis sur les arguments formels, leur relation avec le résultat renvoyé, les effets éventuels attendus de la fonction.

Ce commentaire doit être placé au début du corps de la fonction, sous forme d'une chaîne de caractère entourée de trois guillemets.

```
python
def Hypothenuse(x,y):
    """Calcule l'hypothénuse d'un triangle rectangle de petits cotes x et y (nombres)"""
    return((x**2+y**2)**0.5)
```

Le commentaire est alors accessible par la commande : **help**(nom de la fonction).

3 Usages des variables avec des fonctions

3.1 Exemple introductif

```
python
1  def somme_des_n_entiers_puissance_i (n, i):
2      somme=0
3      for indice in range(n):
4          somme = somme + indice**i
5      return somme
6
7  a=int(input("Combien d'entiers ? "))
8  p=int(input("A quelle puissance ? "))
9  S=somme_des_n_entiers_puissance_i(a,p)
0  print("la sommes des n premieres entiers à
la puissance i est : ",S)
```

3.2 Portée des variables

Dès lors qu'on utilise des fonctions, il devient nécessaire de distinguer deux sortes de variables :

- les **variables globales** ont une portée sur l'ensemble du programme : elle sont utilisables (appel et affectation) depuis le programme et les fonctions ;
- les **variables locales** qui ont une portée limitée au corps de la fonction : elles s'effacent de la mémoire à la fin de l'exécution de la fonction.

Conseil :

- Utiliser des noms de variables différents pour les variables locales et globales.

3.3 Transmission des paramètres d'une fonction

Lors de la définition d'une fonction, les variables qui figurent comme arguments sont appelées **paramètres formels**. Car, ils n'ont pas encore d'existence réelle dans la mémoire. Les paramètres transmis à la fonction lors de son appel sont appelés **paramètres effectifs**. Lors de l'appel d'une fonction :

1. les paramètres formels sont déclarés en tant que variables locales ;
2. ils prennent les valeurs de constantes ou de celles des variables globales précisé par les paramètres effectifs.

3.4 Application

On considère le programme suivant :

```
python
1  def carre(x):
2      y = x**2
3      return(y)
4  # début du programme
5  a=3
6  print(carre(a))
7  print(a)
```

Décrire l'évolution de l'état du programme.

3.5 Utiliser une variable globale

L'usage d'une variable globale dans une fonction est pleinement autorisée. Deux cas sont à distinguer :

Si la fonction ne fait qu'appeler la variable globale : la variable globale doit être définie avant l'appel de la fonction.

```
python
1 def test(x):
2     x=x+n
3     return(x)
4 n=10
5 print( test(1),n)
```

Remarque : ordre d'évaluation

Si la fonction doit modifier une variable globale : la variable doit être

- définie avant l'appel de la fonction ;
- déclarée dans la fonction comme étant globale, en utilisant le mot clé **global**.

```
python
1 def test(x):
2     global n
3     n=x+n
4     return(x)
5 n=10
6 print(n, test(1),n)
```

3.6 Applications

Décrire l'état du programme ci-dessous :

```
python
1 def f1 (arg1) :
2     x1 = arg1*a
3     return(x1)
4
5 def f2 (arg2) :
6     x2=arg2*a
7     x2=arg2*x1
8     return(x2)
9
10 a=14.
11 print (f1(3))
12 print (f2(3))
```

Qu'affiche le programme ci-dessous :

```
python
1 def F(x):
2     print("variable locale : ",x)
3     y=x**2
4     return(y)
5
6 x=2
7 print("variable globale : ",x)
8 print(F(3))
```

3.7 Fonction de plusieurs arguments, à plusieurs résultats

Une fonction peut nécessiter plusieurs arguments et retourner plusieurs valeurs :

```
python
1 def division_euclidienne (a,b):
2     return(a//b,a%b)
3 quotient, reste = division_euclidienne (13,3)
4 print(quotient)
5 print(reste)
```

4 Spécificités en Python

4.1 Fonctions prédéfinies

Python s'appuie sur 68 fonctions de bases. Pour résoudre des problèmes complexes, il est possible de s'appuyer sur d'autres fonctions contenues dans des modules qu'il faut importer au cas par cas. Parmi les modules les plus utiles dans ce cours, citons :

- **math** : contient les fonctions usuelles en analyse
- **random** : sert à générer des nombres pseudo-aléatoires
- **numpy** et **scipy** : fournit des outils variés pour le calcul scientifique
- **matplotlib** : permet le tracé de graphes

Il y a différentes possibilités pour charger un module – « Charger » prend ici le sens d'amener dans la mémoire active associée au programme ou à la console en cours d'utilisation. Pour bien faire, il place le chargement en début de programme.

4.1.1 Première méthode : le chargement direct

from module **import** *

Toutes les fonctions/méthodes du module sont chargées dans la mémoire active. Il est possible de les appeler directement.

On peut ne charger que certaines fonctions du module avec l'instruction :

from module **import** fonctions_a_importer

Pour plus de commodité et éviter les redondances de nom, il est possible de renommer les fonctions au moment de l'appel avec le mot clé `as` :

from module **import** fonction_a_importer **as** nouveau_nom_fonction

4.1.2 Deuxième méthode : le chargement indirect

import module

Toutes les fonctions/méthodes du module sont chargées dans la mémoire active, mais regroupé dans un « paquet » qui porte le nom du module. Une fois le module chargé, pour appeler une fonction en particulier du module, il faut alors utiliser la syntaxe :

module. fonction_en_particulier

Cette méthode permet d'assurer l'absence de redondance des noms utilisés.

4.1.3 Application : interpréter les chargements suivants :

from math **import** *

import numpy **as** np

import scipy **as** sp

import scipy.integrate **as** integr

4.2 Usage d'une valeur par défaut d'argument

Une fonction peut s'appuyer sur une valeur par défaut d'argument. L'argument est alors précisé ainsi : `argument=valeur`. Cela signifie que, si l'appel de la fonction ne précise pas de paramètres effectifs pour un argument, alors la valeur par défaut sera considérée.

```
python 1 def Hypothenuse(x=3,y=3):  
      2     return sqrt(x**2+y**2)  
      3  
      4 print(Hypothenuse())
```

4.3 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. C'est à dire que des fonctions peuvent ainsi être passées en argument, renvoyées comme résultat ou encore stockées dans une variable.

Pour définir une variable de type fonction, dans certains contexte il n'est pas possible d'utiliser le mot-clé **def**. On utilise alors le mot-clé **lambda** :

fonction=**lambda** x : expression

où **x** est l'argument de la fonction et **expression** est une expression de **x**.

Exemple : fonction composée $h = f \circ g$

```
python 1 def h(f,g)  
      2     return lambda x : f(g(x))
```