Organiser un programme :
Fonctions, modules, fichiers

1 Qu'est-ce qu'une fonction

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 $f : \mathbb{R} \to \mathbb{R}$

```python
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.

```python
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 :

```python
>>> 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. Elles forment la signature de la fonction, petit texte descriptif accessible en utilisant l'aide (`help`).

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

2.2.1 Les éléments du code

Une fonction, sous python, est introduite par le mot clé **def**. Elle possède :

- un **nom** (mêmes règles que pour les noms de variables) ;
- une **spécification**, petit texte placé entre """ décrivant la spécification ;
- 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`.

```
def nom_de_la_fonction(arguments)
    """Specification ..."""
    corps_de_la_fonction
    return valeurs
```

Remarques :

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

2.2.2 Specification : documentation d'une fonction

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

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

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

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

3 Usages des variables avec des fonctions

3.1 Exemple introductif

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

a=int(input("Combien d'entiers ? "))
p=int(input("A quelle puissance ? "))
S=somme_des_n_entiers_puissance_i(a,p)
print("la sommes des n premieres entiers àla 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 : elles sont utilisables (appel et affectation) depuis le programme et les fonctions ;
- les **variables locales** qui ont un 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'existance 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
def carre(x):
    y = x**2
    return y
# début du programme
a=3
print(carre(a))
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
def test(x):
    x=x+n
    return x
n=10
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
def test(x):
    global n
    n=x+n
    return x
n=10
print(n,test(1),n)
```

3.6 Applications

Décrire l'état du programme ci-dessous :

```python
def f1 (arg1) :
    x1 = arg1*a
    return x1

def f2 (arg2) :
    x2=arg2*a
    x2=arg2*x1
    return x2

a=14.
print(f1(3))
print(f2(3))
```

Qu'affiche le programme ci-dessous :

```python
def F(x):
    print("variable locale : ",x)
    y=x**2
    return y

x=2
print("variable globale : ",x)
print(F(3))
```

3.7 Fonction de plusieurs arguments, à plusieurs résultats

Une fonction peut nécessiter plusieurs arguments et peut retourner plusieurs valeurs :

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

4 Paradigmes

En informatique, il existe plusieurs façons d'envisager la programmation. Ces approches de programmations sont appelées « paradigme ».

- Programmation fonctionnelle : une fonction n'agit que sur ses variables locales.
- Programmation orientée objet : un objet agit sur un autre objet (exemple : un lave-vaisselle nettoie une assiette)

Python acceptant les deux approches, il en résulte un effet secondaire difficile à maitriser : l'effet de bord (voir cours dédié).

4.1 Python et la programmation orientée objet

La programmation orientée objet est une approche de programmation que nous ne chercherons pas à maîtriser à court terme. Elle présente néanmoins quelques avantages que nous utiliserons dès cette année.

Spécificité

En python, *tout est objet*. Et tout objet est caractérisé par des attribus et des méthodes.

- un attribut est une propriété particulière de l'objet ;
- Une méthode est assimilable à une fonction spécifique de l'objet.

- l'attribut peut être récupéré avec la syntaxe : objet.attribut
- la méthode peut être appelé avec la syntaxe : objet.méthode(argument)

```
>>> a=1.5+3.j
>>> a.real
1.5
>>> a.imag
3.0
>>> a.conjugate ()
(1.5-3j)
>>> a.conjugate
<built-in method conjugate of complex object at 0x02D68938>
```

5 Les modules

5.1 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 placer le chargement en début de programme.

5.1.1 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
```

5.1.2 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.

5.1.3 Application : interpréter les chargements suivants :

```
from math import *

import numpy as np

import scipy as sp

import scipy.integrate as integr
```

6 Données externes : les fichiers

Un fichier est une donnée conservée dans une mémoire de masse (disque dur). Nous ne manipulerons que les fichiers les plus simples : les fichiers texte, reconnaissable grace au suffise « `.txt` ».

6.1 Fichiers textes externes

6.1.1 Présentation

Un fichier externe est un fichier sauvegardé sur un support de masse. Il contient des données préenregistrées selon une convention (*format*). Sous Python, il faut travailler de la façon suivante :

1. On ouvre le fichier dans un premier temps : on précise le nom du fichier avec son chemin d'accès et le mode d'accès (lecture seule ou lecture & écriture) ;
2. On lit ou on écrit des données dans ce fichier ;
3. On ferme le fichier une fois qu'on a terminé de travailler avec.

6.1.2 Ouverture et fermeture d'un fichier

Ouvrir un fichier correspond à :
- définir en mémoire une variable correspondant à un lien vers le fichier ;
- signifier au système d'exploitation que le fichier est en cours d'utilisation.

```
mon_fichier=open(Chemin,par)
```

où :
- `Chemin` est le nom du fichier précédé de son chemin d'accès : `U:/Python/Projet1/fichier.txt`
- `par` spécifie le mode d'ouverture (`r` pour lecture seule, `w` pour lecture et écriture)

Enfin, pour fermer le fichier, la méthode est la suivante :

```
mon_fichier.close()
```

6.1.3 Lecture et écriture du contenu textuel

La lecture se fait par méthodes, qui dépendent évidemment du contenu. Dans le cas d'un fichier texte, nous n'en retiendrons que trois :

1. `mon_fichier.read()` : renvoie la chaîne de caractères contenue dans le fichier
2. `mon_fichier.readline()` : renvoie la première ligne contenue dans le fichier. Un nouvel appel de cette méthode renvoie la seconde ligne, etc.
3. `mon_fichier.readlines()` : renvoie une liste de chaînes de caractères où chaque élément est une ligne du contenu du fichier.

L'écriture se fait par la méthode suivante, qui remplace le contenu actuel du fichier par la chaine `s` :

```
mon_fichier.write(s)
```