Plan du chapitre

I Rappels sur les fonctions toutes faites

I.1 La fonction input

Déjà vue et revue, nous ne rappellerons pas sa syntaxe. Contentons-nous de rappeler qu'elle **renvoie** systématiquement une chaîne de caractères.

I.2 La fonction print

Vue et revue également, mais nous allons préciser sa syntaxe :

- possibilité de prendre plusieurs arguments.

```
>>> x=3
>>> print(x,'*2=',x*2)
3 *2= 6
```

- Possibilité de changer le séparateur entre les arguments, qui est par défaut un espace :

```
>>>print(1,1,sep="+")
1+1
```

- Possibilité de passer à la ligne suivante grâce à **'\n'** :

```
>>>print('PTSI','\n','Informatique')
'PTSI'
 'Informatique'
```

Attention danger !
La fonction **print** produit un affichage, mais ne renvoie aucun résultat.

```
>>> a=print(12)
12
>>> print(a)
None
```

I.3 Utilisation des modules

Sous Python, il existe assez peu de fonctions toutes faites d'accès direct. Nous avons donc deux possibilités pour obtenir d'autres fonctions :

- les créer nous-mêmes, ce qui sera l'objet de la deuxième partie ;

- les **importer** depuis un **module**.

Nous verrons dans quelques semaines des modules très importants pour le calcul et l'ingénierie numérique (**numpy**, **scipy**, **matplotlib**). Contentons-nous pour l'instant de rappeler la syntaxe générale pour importer un module, ou simplement une fonction d'un module.

- La première méthode consiste simplement à importer le module via la commande :

```
import module
```

Dans ce cas, toutes les fonctions de ce module, devront, pour être utilisées, être précédées du nom du module :

```
>>>module.fonction()
```

On dit que le module a un **espace des noms** différent de celui du programme principal.

Si le nom du module est trop long, on peut lui assigner un nouveau nom :

```
import module as mod
```

Cela se révélera très utile lorsque vous devrez importer **numpy**, **matplotlib.pyplot**, et **scipy.odeint** par exemple : s'il fallait réécrire les noms des modules à chaque fois, ce serait une perte de temps ! Par exemple avec la fonction racine carrée du module **math** :

```
>>> import math as ma
>>> ma.sqrt(16)
4.0
```

- Si l'on veut que les fonctions du module soient accessibles directement depuis l'espace des noms du programme, on écrira :

```
from module import *
```

Attention danger !
Cette méthode est déconseillée, car elle fait encourir un risque d'homonymie entre une fonction du module et une fonction crée par nos propres soins.

- On peut également, avec cette même syntaxe, importer simplement une seule fonction du module, toujours directement accessible depuis l'espace des noms du programme :

```
from module import fonction
```

Par exemple :

```
>>> from math import factorial
>>> factorial(4)
24
```

II Créer une nouvelle fonction

Depuis le début de l'année, à chaque fois que nous voulons tester un algorithme sur un paramètre, nous demandons à l'utilisateur d'entrer une valeur pour ce paramètre via la fonction **input**. C'est assez fastidieux : il est nettement plus simple de définir une **fonction**, qui prendra en argument(s) le ou les paramètre(s) en question. Ainsi, nous pourrons tester l'algorithme autant de fois que nous le souhaitons, plutôt que de devoir relancer à chaque fois le programme.

II.1 Syntaxe

Une fonction est définie par 4 éléments :

- un nom, qu'il vaut mieux choisir explicite et court ;
- ses **arguments** ;
- un bloc d'instructions, marqué par l'indentation et appelé **le corps de la fonction** ;
- et, éventuellement, **la valeur qu'elle renvoie**.

La syntaxe est la suivante :

```
def <nom_de_fonction>(<argument_1>,...,<argument_n>):
        <instr 1>
        ...
        <instr p>
        return <donnée à renvoyer>
```

Les mots-clés nouveaux sont donc **def** qui montre que l'on définit une fonction, et **return**, qui spécifie la valeur à renvoyer, si l'on souhaite en renvoyer une.

II.2 Quelques exemples

Exemple 1 : une première fonction qui produit un affichage mais qui ne renvoie rien.

Ecrivons une fonction **table** qui prend comme argument un entier naturel et qui affiche la table de cet entier naturel.

```
def table(n):
    for i in range(1,11):
        print(n,'*',i,'=',n*i)
```

L'appel de cette fonction est très simple :

```
>>> table(5)
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

Exemple 2 : une fonction qui ne produit pas d'affichage, mais qui renvoie une valeur.

Ecrivons maintenant une fonction **hypothenuse** qui prend comme arguments deux flottants a et b et qui renvoie la valeur de $\sqrt{a^2+b^2}$.

```
def hypothenuse(a,b):
    return (a**2+b**2)**0.5
```

```
>>> hypothenuse(8,6)
10.0
```

Attention danger !

Bien faire la différence entre **print** et **return**.

Rappelons encore une fois que la fonction **print** produit un affichage, mais ne renvoie aucune valeur. Considérons par exemple les deux fonctions suivantes :

```
def hypothenuse1(a,b):
    return (a**2+b**2)**0.5

def hypothenuse2(a,b):
    print((a**2+b**2)**0.5)
```

Voyons les différences lors de l'exécution de ces deux fonctions :

```
>>> x=hypothenuse1(8,6)
>>> print(x)
10
>>> y=hypothenuse2(8,6)
10.0
>>> print(y)
None
```

Savoir-faire

La plupart du temps, on écrit des fonctions qui produisent des valeurs, valeurs qui sont stockées dans des variables pour être utilisées dans la suite.

Une bonne pratique consiste donc à utiliser (quasiment) systématiquement **return** plutôt que **print** pour terminer l'exécution d'une fonction.

Attention danger !

Avec **return**, toute sortie est définitive !

```
def rechercheDouze(L):
    for i in range(len(L)):
        print(L[i])
        if L[i]==12:
            return i
```

```
>>> L=[5,2,12,6,-1,3,12]
>>> a=rechercheDouze(L)
5
2
12
>>> print(a)
2
```

II.3 Variables locales et variables globales

Définition

Une variable définie à l'intérieur d'une fonction est invisible à l'extérieur de cette fonction : elle n'est définie que pendant l'exécution de la fonction, puis revient indéfinie à la fin de cette exécution, la valeur qu'elle a prise pendant l'exécution étant oubliée : on dit pour cela qu'elle est une **variable locale**.

Par exemple :

- Le script

```
def f(x):
    A=2*x
    return A
```

- On l'exécute :

```
>>> f(5)
10
>>> A
NameError: name 'A' is not defined
```

Si une variable locale porte le même nom qu'une variable existant en dehors de la fonction, Python ne fait alors pas de confusion entre la variable locale et la variable extérieure qui n'est pas modifiée à l'issue de l'exécution de la fonction. Par exemple :

- Le script

```
A=12
def f(x):
    A=2*x
    return A
```

- On l'exécute :

```
>>> f(5)
10
>>> A
12
```

Définition

La notion de variable locale s'oppose à celle de variable globale : une **variable globale** est connue dans tout le programme.

On peut modifier ce comportement grâce à la construction **global**.

▷ Le script

```
A=12
def f(x):
    global A
    A=2*x
    return A
```

▷ On l'exécute :

```
>>> f(5)
10
>>> A
10
```

Attention danger !

De façon générale, une bonne pratique consiste à utiliser les variables globales pour représenter les *constantes* du problème, constantes qui ne doivent donc pas être modifiées lors de l'exécution du programme.

L'usage de la construction **global** est donc souvent le signe d'une programmation maladroite !

Les paramètres d'une fonction sont des grandeurs locales et peuvent donc porter le même nom qu'une variable extérieure sans risque de confusion.

▷ Le script

```
x=42

def f(x):
    y=2*x
    return y
```

▷ On l'exécute :

```
>>> f(10)
20
>>> x
42
```

Entrainement 1

Quelles sont les variables locales et globales de la fonction **f** ? Qu'affiche le programme suivant ?

```
def f():
    global a
    a=a+1
    c=2*a
    return a+b+c
a=3
b=4
c=5
print(f())
print(a)
print(b)
print(c)
```

a est une variable globale, tandis que **c** est une variable locale de **f**.

```
16
4
4
5
```

II.4 Aide d'une fonction

Lorsque l'on ne connait pas le fonctionnement d'une fonction, Python nous permet d'accéder à son **aide** grâce à la fonction **help()** :

```
>>> help(abs)
Help on built-in function abs in module builtins:

```

```
abs(...)
    abs(number) -> number

    Return the absolute value of the argument.
```

Lorsque vous créez une fonction originale, il peut également être très utile d'indiquer son fonctionnement, au cas où quelqu'un d'autre aurait besoin de l'utiliser, ou simplement pour vous permettre d'y revenir plus tard. C'est extrêmement simple : il suffit d'écrire l'aide sous la forme d'une chaîne de caractères au tout début de votre fonction.

▷ Le script

```
def maximum(a,b):
    "renvoie le max de a et b"
    if a<b:
        return b
    else:
        return a
```

▷ Appel de l'aide

```
>>> help(maximum)
Help on function maximum in module __main__:

maximum(a, b)
    renvoie le max de a et b
```

II.5 Quelques exercices

Entrainement 2

Ecrire une fonction qui prend en argument le rayon d'un disque et qui renvoie sa surface.

```python
import math as m
def surface(rayon):
    return m.pi*rayon**2
```

Entrainement 3

Ecrire une fonction qui prend en arguments le rayon et la hauteur d'un cylindre et qui renvoie son volume. On utilisera la fonction de l'exercice précédent.

```python
def volume(rayon,hauteur):
    return surface(rayon)*hauteur
```

Entrainement 4

Ecrire une fonction qui prend en arguments une chaîne de caractères et un caractère et qui renvoie le nombre de fois où le caractère est présent dans la chaîne.

```python
def comptage(phrase,lettre):
    N=0
    for x in phrase:
        if x==lettre:
            N+=1
    return N
```

```
>>> comptage("ptsi Dumont d'Urville","i")
2
```

Entrainement 5

Analyser la fonction suivante, qui prend comme argument une chaine de caractères, et proposer une explication à son fonctionnement.

```
def mystere(ch):
    ch2=''
    for i in range(len(ch)):
        ch2=ch2+ch[len(ch)-1-i]
    return ch2
```

La fonction **mystere** renvoie la chaîne de caractère inverse de **ch** : la variable locale **ch2** est initialisée à la chaîne vide, puis chacun des caractères de **ch** est concaténé au bout de **ch2**, en commençant par le dernier caractère de **ch**, et en terminant par le premier.

III Mécanismes avancés

III.1 Fonctions locales

Il arrive que l'utilisation d'une fonction soit limitée à la définition d'une autre fonction.

Supposons par exemple qu'on souhaite écrire une fonction **max3** calculant le maximum de 3 entiers. Pour cela, il est élégant de commencer par la définition d'une fonction **max2** calculant le maximum de 2 entiers, pour l'utiliser ensuite deux fois.

Cependant, on ne souhaite pas nécessairement rendre visible la fonction **max2**. On la définit donc localement à la fonction **max3**.

```
def max3(x,y,z):
    def max2(u,v):
        if u<v:
            return v
        else:
            return u
    A=max2(x,y)
    return max2(A,z)
```

```
>>> max3(1,8,6)
8
```

III.2 Fonctions comme valeurs de première classe

Définition

En Python, une fonction est une valeur comme une autre, c'est-à-dire qu'elle peut être passée en argument, renvoyée comme résultat ou encore stockée dans une variable. On dit pour cela que les fonctions sont des **valeurs de première classe**.

Exemple 3

Ecrire une fonction qui prend en arguments une fonction f et un entier naturel n et qui renvoie la somme $\sum_{k=0}^{n} f(k)$.

Tester avec la fonction $f : x \mapsto x^2$.

```
def sommeFonction(f,n):
    A=0
    for k in range(n+1):
        A+=f(k)
    return A

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

```
>>> sommeFonction(carre,10)
385
```

Exemple 4

Ecrire une fonction **derive** qui prend comme arguments une fonction f et un flottant ϵ et qui renvoie la fonction :

$$x \mapsto \frac{f(x+\epsilon)-f(x)}{\epsilon}.$$

Vérifier que pour $f : x \mapsto x^2$, **derive(f,0.001)** est une bonne approximation de $x \mapsto 2x$.

```
def derive(f,eps):
    def g(x):
        return (f(x+eps)-f(x))/eps
    return g

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

```

fp=derive(carre,0.001)
```

```
>>> fp(3)
6.000999999999479
>>> fp(100)
200.00100000106613
```