3

Les fonctions

Les fonctions prédéfinies

L'un des concepts les plus importants en programmation est celui de fonction. Les fonctions permettent en effet de décomposer un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés en fragments plus petits, et ainsi de suite. D'autre part, les fonctions sont réutilisables : si nous disposons d'une fonction capable de calculer une racine carrée, par exemple, nous pouvons l'utiliser un peu partout dans nos programmes sans avoir à la réécrire à chaque fois.

Vous avez déjà rencontré d'autres fonctions intégrées au langage lui-même, « **print** », « **input** », « **len** », etc.

Importer un module de fonctions

Les fonctions intégrées au langage sont relativement peu nombreuses : ce sont seulement celles qui sont susceptibles d'être utilisées très fréquemment. Les autres sont regroupées dans des fichiers séparés que l'on appelle des modules.

Il existe un grand nombre de modules préprogrammés qui sont fournis d'office avec Python. Vous pouvez en trouver d'autres chez divers fournisseurs. Souvent on essaie de regrouper dans un même module des ensembles de fonctions apparentées, que l'on appelle des bibliothèques.

Le module « **math** », contient les définitions de nombreuses fonctions mathématiques. Pour pouvoir utiliser ces fonctions, il vous suffit d'incorporer la ligne suivante au début de votre script :

```
from math import [* | listeObjets]
```

Cette ligne indique à Python qu'il lui faut inclure dans le programme courant toutes les fonctions du module « **math** », lequel contient une bibliothèque de fonctions mathématiques préprogrammées.

```
>>> from math import sqrt
>>> nombre = 121
>>> print("racine carrée de", nombre, "=", sqrt(nombre))
racine carrée de 121 = 11.0
>>> angle = pi/6 # soit 30°
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'pi' is not defined

>>> from math import *
>>> nombre,angle = 121,pi/6 # soit 30°
>>> # (la bibliothèque math inclut aussi la définition de pi)
>>> print("racine carrée de", nombre, "=", sqrt(nombre))
racine carrée de 121 = 11.0
>>> print("sinus de", angle, "radians", "=", sin(angle))
sinus de 0.5235987755982988 radians = 0.49999999999999994
```

Les modules internes

Python dispose de nombreux modules préinstallés. Cette liste est trop longue pour figurer dans ce document, elle est aussi susceptible de s'allonger au fur et à mesure du développement du langage Python. Voici une liste des modules les plus utilisés.

module	signification
asyncio	Thread, socket, protocol.
calendar	Gérer les calendriers, les dates.
cgi	Utilisé dans les scripts CGI (programmation Internet)
cmath	Fonctions mathématiques complexes.
copy	Copies d'instances de classes.
csv	Gestion des fichiers au format CSV.
datetime	Calculs sur les dates et heures.
gc	Gestion du garbage collector.
getopt	Lire les options des paramètres passés en arguments d'un programme Python.
glob	Chercher des fichiers.
hashlib	Fonctions de cryptage.
htmllib	Lire le format HTML.
math	Fonctions mathématiques standard telles que cos, sin, exp, log...
os	Fonctions systèmes dont certaines fonctions permettant de gérer les fichiers
os.path	Manipulations de noms de fichiers
pathlib	Manipulation de chemins.
pickle	Sérialisation d'objets, la sérialisation consiste à convertir des données structurées de façon complexe en une structure linéaire facilement enregistrable dans un fichier.
profile	Etudier le temps passé dans les fonctions d'un programme.
random	Génération de nombres aléatoires.
re	Expressions régulières.
shutil	Copie de fichiers.
sqlite3	Accès aux fonctionnalités du gestionnaire de base de données SQLite3.
string	Manipulations des chaînes de caractères.
sys	Fonctions systèmes, fonctions liées au langage Python.
threading	Utilisation de threads.
time	Accès à l'heure, l'heure système, l'heure d'une fichier.
tkinter	Interface graphique.
unittest	Tests unitaires (ou comment améliorer la fiabilité d'un programme).
urllib	Pour lire le contenu de page HTML sans utiliser un navigateur.
xml.dom	Lecture du format XML.
xml.sax	Lecture du format XML.
zipfile	Lecture de fichiers ZIP.

Vous pouvez installer d'autres packages à l'aide de la commande :

```
pip install nomPackage
```

```
C:\Users\Razvan BIZOÏ>pip install pandas
Requirement already satisfied: pandas in c:\programdata\anaconda3\lib\site-packages
Requirement already satisfied: python-dateutil>=2 in c:\programdata\anaconda3\lib\site-packages (from pandas)
```

```
Requirement already satisfied: pytz>=2011k in
c:\programdata\anaconda3\lib\site-packages (from pandas)
Requirement already satisfied: numpy>=1.7.0 in
c:\programdata\anaconda3\lib\site-packages (from pandas)
Requirement already satisfied: six>=1.5 in
c:\programdata\anaconda3\lib\site-packages (from python-dateutil>=2->pandas)
```

Le programme suivant par exemple utilise les modules « **random** » et « **math** » pour estimer le nombre pi.

```
>>> import random
>>> import math
>>> somme = 0
>>> nb   =  1000000
>>> for i in range (0,nb) :
        # nombre aléatoire entre [0,1]
        x = random.random()
        y = random.random()
        r = math.sqrt(x*x + y*y) #racine carrée
        if r <= 1:
             somme += 1
>>> print("estimation ", 4 * float (somme) / nb)
estimation  3.140048
>>> print("PI = ", math.pi)
PI =  3.141592653589793
```

Le programme suivant calcule l'intégrale de Monte Carlo de la fonction $f(x) = \sqrt{x}$ qui consiste à tirer des nombres aléatoires dans l'intervalle $[a, b]$ puis à faire la moyenne des $\sqrt{x}$ obtenu.

```
>>> import random
>>> import math
>>> def integrale_monte_carlo(a, b, f, n):
        somme = 0.0
        for i in range(0, n):
             x = random.random() * (b-a) + a
             y = f(x)
             somme += f(x)
        return somme / n

>>> def racine(x):
        return math.sqrt(x)

>>> print(integrale_monte_carlo(0, 1, racine, 100000))
0.6676537057136576
```

La définition d'une fonction

Les scripts que vous avez écrits jusqu'à présent étaient à chaque fois très courts, car leur objectif était seulement de vous faire assimiler les premiers éléments du langage. Lorsque vous commencerez à développer de véritables projets, vous serez confrontés à des problèmes souvent complexes, et les lignes de programme vont commencer à s'accumuler...

L'approche efficace d'un problème complexe consiste souvent à le décomposer en plusieurs sous-problèmes plus simples qui seront étudiés séparément (ces sous-problèmes peuvent éventuellement être eux-mêmes décomposés à leur tour, et ainsi de suite). Or il est important que cette décomposition soit représentée fidèlement dans les algorithmes pour que ceux-ci restent clairs.

Les fonctions et les classes d'objets sont différentes structures de sous-programmes qui ont été imaginées par les concepteurs des langages de haut niveau afin de résoudre les difficultés évoquées ci-dessus. Nous allons commencer par décrire ici la définition de fonctions sous Python. Les objets et les classes seront examinés plus loin.

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

```
def nomDeLaFonction(liste de paramètres):
    ...
    bloc d,instructions
    ...
```

- Vous pouvez choisir n'importe quel nom pour la fonction que vous créez, à l'exception des mots réservés du langage, et à la condition de n'utiliser aucun caractère spécial ou accentué, le caractère souligné « **_** » est permis. Comme c'est le cas pour les noms de variables, il vous est conseillé d'utiliser surtout des lettres minuscules, notamment au début du nom, les noms commençant par une majuscule seront réservés aux classes.
- Comme les instructions « **if** » et « **while** » que vous connaissez déjà, l'instruction « **def** » est une instruction composée. La ligne contenant cette instruction se termine obligatoirement par un double point, lequel introduit un bloc d'instructions que vous ne devez pas oublier d'indenter.
- La liste de paramètres spécifie quelles informations il faudra fournir en guise d'arguments lorsque l'on voudra utiliser cette fonction, les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments.
- Une fonction s'utilise pratiquement comme une instruction quelconque. Dans le corps d'un programme, un appel de fonction est constitué du nom de la fonction suivi de parenthèses.

Si c'est nécessaire, on place dans ces parenthèses le ou les arguments que l'on souhaite transmettre à la fonction. Il faudra en principe fournir un argument pour chacun des paramètres spécifiés dans la définition de la fonction, encore qu'il soit possible de définir pour ces paramètres des valeurs par défaut.

La fonction sans paramètres

Le mode interactif de Python est en effet idéal pour effectuer des petits tests comme ceux qui suivent. C'est une facilité que n'offrent pas tous les langages de programmation !

```
>>> def table7():
        t = list()
        n = 1
        while n < 11:
            print(n * 7, end=' ')
            t.append(n * 7)
            n = n + 1
        print('\n')
        return t

>>> t = table7()
7 14 21 28 35 42 49 56 63 70
>>> print(t)
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```

Nous pouvons maintenant réutiliser cette fonction à plusieurs reprises, autant de fois que nous le souhaitons.

```
>>> def table7triple():
        t = list()
        print('La table par 7 en triple exemplaire :')
        t += table7()
        t += table7()
        t += table7()
        return t

>>> table7triple()
La table par 7 en triple exemplaire :
7 14 21 28 35 42 49 56 63 70

7 14 21 28 35 42 49 56 63 70

7 14 21 28 35 42 49 56 63 70

[7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 7, 14, 21, 28, 35, 42, 49,
56, 63, 70, 7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```

Une première fonction peut donc appeler une deuxième fonction, qui elle-même en appelle une troisième, etc.

- Créer une nouvelle fonction vous offre l'opportunité de donner un nom à tout un ensemble d'instructions. De cette manière, vous pouvez simplifier le corps principal d'un programme, en dissimulant un algorithme secondaire complexe sous une commande unique, à laquelle vous pouvez donner un nom très explicite, en français si vous voulez.
- Créer une nouvelle fonction peut servir à raccourcir un programme, par élimination des portions de code qui se répètent.

La fonction avec paramètres

Une fonction est une partie d'un programme qui fonctionne indépendamment du reste du programme. Elle reçoit une liste de paramètres et retourne un résultat. Le corps de la fonction désigne toute instruction du programme qui est exécutée si la fonction est appelée.

```
def fonction_nom (par_1, ..., par_n) :
    instruction_1
    ...
    instruction_n
    return res_1, ..., res_n
```

Dans la définition d'une telle fonction, il faut prévoir une ou plusieurs variables pour recevoir les arguments transmis. Une telle variable particulière s'appelle un paramètre. On lui choisit un nom en respectant les mêmes règles de syntaxe que d'habitude, et on place ce nom entre les parenthèses qui accompagnent la définition de la fonction.

```
x_1, ..., x_n = fonction_nom(val_1, val_2, ..., val_n)
```

Lorsque nous appellerons cette fonction, nous devrons bien évidemment pouvoir lui indiquer l'information que nous voulons transmettre à la fonction au moment même où nous l'appelons s'appelle un argument.

```
>>> import math
>>> def coordonnees_polaires (x,y):
        rho   = math.sqrt(x*x+y*y)
        theta = math.atan2 (y,x)
        return rho, theta

>>> def affichage (x,y):
        r,t = coordonnees_polaires(x,y)
        print ("cartésien (%f,%f) --> polaire (%f,%f degrés)" \
            % (x,y,r,math.degrees(t)))

>>> affichage (1,1)
cartésien (1.000000,1.000000) --> polaire (1.414214,45.000000 degrés)
>>> affichage (0.5,1)
cartésien (0.500000,1.000000) --> polaire (1.118034,63.434949 degrés)
>>> affichage (-0.5,1)
cartésien (-0.500000,1.000000) --> polaire (1.118034,116.565051 degrés)
>>> affichage (-0.5,-1)
cartésien (-0.500000,-1.000000) --> polaire (1.118034,-116.565051 degrés)
>>> affichage (0.5,-1)
cartésien (0.500000,-1.000000) --> polaire (1.118034,-63.434949 degrés)
```

Les paramètres avec des valeurs par défaut

Lorsqu'une fonction est souvent appelée avec les mêmes valeurs pour ses paramètres, il est possible de spécifier pour ceux-ci une valeur par défaut.

```
def fonction_nom (param_1, \
      param_2 = valeur_2, ..., param_n = valeur_n) :
```

Où fonction_nom est le nom de la fonction. param_1 à param_n sont les noms des paramètres, valeur_2 à valeur_n sont les valeurs par défaut des paramètres param_2 à param_n. La seule contrainte lors de cette définition est que si une valeur par défaut est spécifiée pour un paramètre, alors tous ceux qui suivent devront eux aussi avoir une valeur par défaut.

```
>>> def commander_carte_orange (nom, prenom, \
           paiement = "carte", nombre = 1, zone = 2) :
        print("nom : ",nom)
        print("prénom : ",prenom)
        print("paiement : ", paiement)
        print("nombre : ", nombre)
        print("zone :", zone)

>>> commander_carte_orange("BIZOÏ", "Razvan", "chèque")
nom :  BIZOÏ
prénom :  Razvan
paiement :  chèque
nombre :  1
zone : 2
```

L'ordre des paramètres

Lors de l'appel, le nom des paramètres n'intervient plus, supposant que chaque paramètre reçoit pour valeur celle qui a la même position que lui lors de l'appel à la fonction. Il est toutefois possible de changer cet ordre en précisant quel paramètre doit recevoir quelle valeur.

```
x_1, ..., x_n = fonction_nom (param_1 = valeur_1 \
                    , ..., param_n = valeur_n)
```

```
>>> commander_carte_orange(prenom="Razvan",nom="BIZOÏ")
nom :  BIZOÏ
prénom :  Razvan
paiement :  carte
nombre :  1
zone : 2
```

Cette possibilité est intéressante surtout lorsqu'il y a de nombreux paramètres par défaut et que seule la valeur d'un des derniers paramètres doit être changée.

Atelier 3 - Exercice 1.1- Exercice 1.2- Exercice 1.3- Exercice 1.4

Les paramètres modifiables

Les paramètres de types immuables et modifiables se comportent de manières différentes à l'intérieur d'une fonction. Ces paramètres sont manipulés dans le corps de la fonction, voire modifiés parfois. Selon le type du paramètre, ces modifications ont des répercussions à l'extérieur de la fonction.

Les types immuables ne peuvent être modifiés et cela reste vrai. Lorsqu'une fonction accepte un paramètre de type immuable, elle ne reçoit qu'une copie de sa valeur. Elle peut donc modifier ce paramètre sans que la variable ou la valeur utilisée lors de l'appel de la fonction n'en soit affectée. On appelle ceci un passage de paramètre par valeur. A l'opposé, toute modification d'une variable d'un type modifiable à l'intérieur d'une fonction est répercutée à la variable qui a été utilisée lors de l'appel de cette fonction. On appelle ce second type de passage un passage par adresse.

```
>>> def somme_n_premier_terme(n,liste):
        somme = 0
        for i in liste:
            somme += i
            n -= 1
            # modification de n (type immuable)
            if n <= 0: break
        liste[0] =  0
        # modification de liste (type modifiable)
        return somme

>>> l = [1,2,3,4]
>>> nb = 3
>>> print("avant la fonction ",nb,l)
avant la fonction  3 [1, 2, 3, 4]
>>> s = somme_n_premier_terme (nb,l)
>>> print("après la fonction ",nb,l)
après la fonction  3 [0, 2, 3, 4]
>>> print("somme : ", s)
somme :  6
```

La liste est modifiée à l'intérieur de la fonction. La variable nb est d'un type immuable. Sa valeur a été recopiée dans le paramètre n de la fonction. Toute modification de n à l'intérieur de cette fonction n'a aucune répercussion à l'extérieur de la fonction.

Dans l'exemple précédent, il faut faire distinguer le fait que la liste passée en paramètre ne soit que modifiée et non changée.

```
>>> def fonction (liste):
        liste = list()


>>> liste = [0,1,2]
>>> print(liste)
[0, 1, 2]
>>> fonction (liste)
>>> print(liste)
[0, 1, 2]
```

Il faut considérer dans ce programme que la fonction reçoit un paramètre appelé liste mais utilise tout de suite cet identificateur pour l'associer à un contenu différent.

L'identificateur liste est en quelque sorte passé du statut de paramètre à celui de variable locale.

```
>>> def fonction (liste):
        del liste

>>> print(liste)
[0, 1, 2]
>>> fonction (liste)
>>> print(liste)
[0, 1, 2]
```

Le programme qui suit permet cette fois-ci de vider la liste passée en paramètre à la fonction. La seule instruction de cette fonction modifie vraiment le contenu désigné par l'identificateur liste et cela se vérifie après l'exécution de cette fonction.

```
>>> def fonction (liste):
        del liste[0:len(liste)]

>>> print(liste)
[0, 1, 2]
>>> fonction (liste)
>>> print(liste)
[]
```

La fonction récursive

Une fonction récursive est une fonction qui s'appelle elle-même. La fonction récursive la plus fréquemment citée en exemple est la fonction factorielle. Celle-ci met en évidence les deux composantes d'une fonction récursive, la récursion proprement dite et la condition d'arrêt.

```
>>> def factorielle(n):
        if n == 0 : return 1
        else : return n * factorielle(n-1)
```

La dernière ligne de la fonction factorielle est la récursion tandis que la précédente est la condition d'arrêt, sans laquelle la fonction ne cesserait de s'appeler, empêchant le programme de terminer son exécution. Si celle-ci est mal spécifiée ou absente, l'interpréteur Python affiche une suite ininterrompue de messages.

```
>>> factorielle(9)
362880
>>> factorielle(99)
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
```

Python n'autorise pas plus de 1000 appels récursifs : factorielle(999) provoque nécessairement une erreur d'exécution même si la condition d'arrêt est bien spécifiée.

```
>>> factorielle(999)
Traceback (most recent call last):
  File "<pyshell#283>", line 1, in <module>
    factorielle(999)
  File "<pyshell#281>", line 3, in factorielle
    else : return n * factorielle(n-1)
  File "<pyshell#281>", line 3, in factorielle
    else : return n * factorielle(n-1)
  File "<pyshell#281>", line 3, in factorielle
    else : return n * factorielle(n-1)
  [Previous line repeated 989 more times]
  File "<pyshell#281>", line 2, in factorielle
    if n == 0 : return 1
RecursionError: maximum recursion depth exceeded in comparison
```

La liste des messages d'erreurs est aussi longue qu'il y a eu d'appels à la fonction récursive. Dans ce cas, il faut transformer cette fonction en une fonction non récursive équivalente, ce qui est toujours possible.

```
>>> def factorielle_non_recursive (n) :
        r = 1
        for i in range (2, n+1) :
            r *= i
        return r

>>> factorielle_non_recursive(999)
402387260077093773543702433923003985719374864210714632543799910429938512398629020592044208486969404800479988610197196058631666872994808558901323829669944590997424504087073759918823627727188732519779505950995276120874975462497043601418278094646496291056393887437886487337113...
```

Les variables locales et globales

Lorsque nous définissons des variables à l'intérieur du corps d'une fonction, ces variables ne sont accessibles qu'à la fonction elle-même. On dit que ces variables sont des variables locales à la fonction. Les contenus des variables locales sont stockés dans cet espace de noms qui est inaccessible depuis l'extérieur de la fonction.

Les variables définies à l'extérieur d'une fonction sont des variables globales. Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction ne peut pas le modifier.

```
>>> def mask():
        p = 20
        print(p, q)

>>> p, q = 15, 38
>>> mask()
20 38
>>> print(p, q)
15 38
>>> def mask():
        p = 20
        print(p, q)
        q = p                                    <--------------------

>>> mask()
Traceback (most recent call last):
  File "<pyshell#249>", line 1, in <module>
    mask()
  File "<pyshell#248>", line 3, in mask
    print(p, q)
UnboundLocalError: local variable 'q' referenced before assignment
```

Nous commençons par définir une fonction très simple (qui n'utilise d'ailleurs aucun paramètre). À l'intérieur de cette fonction, une variable « **p** » est définie, avec 20 comme valeur initiale. Cette variable « **p** » qui est définie à l'intérieur d'une fonction sera donc une variable locale.

Une fois la définition de la fonction terminée, nous revenons au niveau principal pour y définir les deux variables « **p** » et « **q** » auxquelles nous attribuons les contenus 15 et 38. Ces deux variables définies au niveau principal seront donc des variables globales.

Ainsi le même nom de variable « **p** » a été utilisé ici à deux reprises, pour définir deux variables différentes : l'une est globale et l'autre est locale. On peut constater dans la suite de l'exercice que ces deux variables sont bel et bien des variables distinctes, indépendantes, obéissant à une règle de priorité qui veut qu'à l'intérieur d'une fonction, ce sont les variables définies localement qui ont la priorité.

Cela signifie en effet que vous pourrez toujours utiliser une infinité de fonctions sans vous préoccuper le moins du monde des noms de variables qui y sont utilisées : ces variables ne pourront en effet jamais interférer avec celles que vous aurez vous-même définies par ailleurs.

Il est toutefois possible de définir une fonction qui soit capable de modifier une variable globale. Pour atteindre ce résultat, il vous suffira d'utiliser l'instruction

« **global** ». Cette instruction permet d'indiquer, à l'intérieur de la définition d'une fonction, quelles sont les variables à traiter globalement.

```
>>> def monter():
        global a
        a = a+1
        print(a)

>>> monter()
Traceback (most recent call last):
  File "<pyshell#252>", line 1, in <module>
    monter()
  File "<pyshell#251>", line 3, in monter
    a = a+1
TypeError: can only concatenate list (not "int") to list
>>> a = 15
>>> monter()
16
>>> monter()
17
>>> print(a)
17
```

L'aide sur la fonction

Le langage Python propose une fonction help qui retourne pour chaque fonction un commentaire ou mode d'emploi qui indique comment se servir de cette fonction.

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

round(...)
    round(number[, ndigits]) -> number

    Round a number to a given precision in decimal digits (default 0
digits).
    This returns an int when called with one argument, otherwise the
    same type as the number. ndigits may be negative.
```

Lorsqu'on utilise cette fonction help sur une fonction, le message affiché n'est pas des plus explicites. Pour changer ce message, il suffit d'ajouter en première ligne du code de la fonction une chaîne de caractères.

```
>>> def coordonnees_polaires (x,y):
        """convertit des coordonnées cartésiennes \n\
        en coordonnées polaires (x,y) --> (pho,theta)"""
        rho   = math.sqrt(x*x+y*y)
        theta = math.atan2 (y,x)
        return rho, theta

>>> help(coordonnees_polaires)
Help on function coordonnees_polaires in module __main__:

coordonnees_polaires(x, y)
    convertit des coordonnées cartésiennes
    en coordonnées polaires (x,y) --> (pho,theta)
```

Il est conseillé d'écrire ce commentaire pour toute nouvelle fonction avant même que son corps ne soit écrit. L'expérience montre qu'on oublie souvent de l'écrire après.

Atelier 3 - Exercice 1.5- Exercice 1.6- Exercice 1.7- Exercice 1.8

L'écriture simplifiée des fonctions

Lorsque le code d'une fonction tient en une ligne et est le résultat d'une expression, il est possible de condenser son écriture à l'aide du mot-clé « **lambda** ». Cette syntaxe est issue de langages fonctionnels comme le Lisp.

```
nom_fonction = lambda param_1, ..., param_n : expression
```

L'exemple suivant utilise cette écriture pour définir la fonction min retournant le plus petit entre deux nombres positifs.

```
>>> min = lambda x,y : (abs (x+y) - abs (x-y)) / 2
>>> print(min(1,2))
1.0
>>> print(min(5,4))
4.0
>>> def  min(x,y):
        return (abs (x+y) - abs (x-y))/2

>>> print(min(1,2))
1.0
>>> print(min(5,4))
4.0
```

La fonction lambda considère le contexte de fonction qui la contient comme son contexte. Il est possible de créer des fonctions lambda mais celle-ci utiliseront le contexte dans l'état où il est au moment de son exécution et non au moment de sa création.

```
>>> fs = []
>>> for a in range (0,5) :
...     f = lambda x : x + a
...     fs.append(f)
...     print(a)
0
1
2
3
4
>>> print(a)
4
>>> for f in fs :
...     print('a = ', a,' lambda = ',f(1))
a =  4  lambda =  5
a =  4  lambda =  5
a =  4  lambda =  5
a =  4  lambda =  5
a =  4  lambda =  5
```

Pour que le programme affiche les entiers de 1 à 5, il faut préciser à la fonction lambda une variable y égale à a au moment de la création de la fonction et qui sera intégrée au contexte de la fonction lambda.

```
>>> fs = []
>>> for a in range (0,5) :
...     f = lambda x,y=a : x + y
...     fs.append(f)
```

```
...
>>> for f in fs :
...     print('a = ', a,' lambda = ',f(1))
...
a =  4  lambda =  1
a =  4  lambda =  2
a =  4  lambda =  3
a =  4  lambda =  4
a =  4  lambda =  5
```

La fonction map

La fonction « **map** » renvoie une liste correspondant à l'ensemble des éléments de la séquence.

```
map : map(fonction, séquence[, séquence...]) -> liste
```

Avant d'être inséré dans la liste, chaque élément est passé à la fonction fournie. Cette dernière doit donc être de la forme :

```
fonction(element) -> element
```

Lorsque plusieurs séquences sont fournies, la fonction reçoit une liste d'arguments correspondants à un élément de chaque séquence. Si les séquences ne sont pas de la même longueur, elles sont complétées avec des éléments à la valeur « **None** ».

La fonction peut être définie à « **None** », et dans ce cas tous les éléments des séquences fournies sont conservés.

```
>>> a = lambda x: x**2
>>> b = list(map(a, (2,3,4)))
>>> b
[4, 9, 16]
>>> list(map(pow, (7, 5, 3), (2, 3, -2)))
[49.0, 125.0, 0.1111111111111111]
>>> a, c = [1,2,3,4,5], [2, -3, 5,7, -0.5]
>>> mul = lambda x, y: x*y
>>> d = sum(map(mul, a,c))
>>> d
36.5

>>> sq = lambda x: x**2
>>> cu = lambda y: y**3
>>> fc = (sq,cu)
>>> #Retour carré & cube de r - utilisation de 'map'
>>> def vv(r): return list(map(lambda z: z(r), fc))
>>> res1 = vv(5)
>>> res1
[25, 125]

>>> def meva(dd):
        'Avec dd comme sequence de nombres,\n\
        retrouvez leur moyenne et variance'
        bb = len(dd)
        med = sum(dd)/bb
        vr = sum(list(map(sq,dd)))/bb - med**2
        return med, vr

>>> seq = [0.0, 1.0, 2.0, 3.0, 4.0, 5.0, \
       6.0, 7.0, 8.0, 9.0, 10.0, 11.0, \
       12.0, 13.0, 14.0]
>>> res1 = meva(seq)
>>> res1
(7.0, 18.66666666666667)
```

La fonction filter

La fonction « **filter** » renvoie une liste correspondant à l'ensemble des éléments de la séquence pour lesquels la fonction fournie retourne vrai.

```
filter(fonction, séquence[, séquence...]) -> liste
```

Avant d'être inséré dans la liste, chaque élément est passé à la fonction fournie. Cette dernière doit donc être de la forme :

```
fonction(element) -> booléen
```

```
>>> symbols = '&#$€£§ç'
>>> code_car = [ord(s) for s in symbols if ord(s) > 160]
>>> code_car
[8364, 163, 167, 231]
>>>
>>> code_car = list(filter(lambda c: c > 160, map(ord, symbols)))
>>> code_car
[8364, 163, 167, 231]

>>> def factorial(n):
...     '''returns n!'''
...     return 1 if n < 2 else n * factorial(n-1)
>>> factorial(42)
1405006117752879898543142606244511569936384000000000
>>> map(factorial, range(11))
<map object at 0x0000023169057CC0>
>>> list(map(fact, range(11)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
>>> list(map(factorial, filter(lambda n: n % 2, range(6))))
[1, 6, 120]
```

Les iterators

À chaque fois qu'un objet est utilisé dans une boucle **for**, l'interpréteur génère en interne un « **iterator** » avec lequel il travaille. Un « **iterator** » est un objet qui contient une méthode « **next** » qui est appelée à chaque cycle et qui renvoie la séquence, élément par élément. Lorsqu'il n'y a plus d'éléments, l'iterator déclenche une exception de type « **StopIteration** ».

Les objets « **iterator** » peuvent être créés par le biais de la primitive « **iter** » qui prend en paramètre tout objet compatible avec les itérations.

```
>>> ma_chaine = "-test-"
>>> iterateur_de_ma_chaine = iter ( ma_chaine )
>>> iterateur_de_ma_chaine
<str_iterator object at 0x0000023169079080>
>>> next ( iterateur_de_ma_chaine )
'-'
>>> next ( iterateur_de_ma_chaine )
't'
>>> next ( iterateur_de_ma_chaine )
'e'
>>> next ( iterateur_de_ma_chaine )
's'
>>> next ( iterateur_de_ma_chaine )
't'
>>> next ( iterateur_de_ma_chaine )
'-'
>>> next ( iterateur_de_ma_chaine )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

>>> for i in ma_chaine : print(i)
...
-
t
e
s
t
-
```

Il est possible d'utiliser une notation abrégée pour créer un generator, à l'aide d'une generator expression.

```
>>> genexp = (i for i in range(5) if i % 2 == 0)
>>> next(genexp)
0
>>> next(genexp)
2
>>> next(genexp)
4
>>> next(genexp)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Les fonctions générateur

Il est possible d'utiliser une notation abrégée pour créer un générateur, à l'aide d'une générateur expression.

```
>>> genexp = (i for i in range(5) if i % 2 == 0)
>>> next(genexp)
0
>>> next(genexp)
2
>>> next(genexp)
4
>>> next(genexp)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

La création d'un « **iterator** » par le biais d'un générateur se résume à l'écriture d'une fonction qui parcourt les éléments de la séquence. Au lieu de retourner ces éléments par la directive return, la fonction doit faire appel à la directive « **yield** », qui sert à définir un point de sauvegarde.

Le mot-clé « **yield** » est un peu à part. Utilisé à l'intérieur d'une fonction, il permet d'interrompre le cours de son exécution à un endroit précis de sorte qu'au prochain appel de cette fonction, celle-ci reprendra le cours de son exécution exactement au même endroit avec des variables locales inchangées. Le mot-clé return ne doit pas être utilisé. Ces fonctions ou générateurs sont utilisées en couple avec le mot-clé « **for** » pour simuler un ensemble.

```
>>> def fonction_yield(n):
        i = 0
        while i < n-1:
            print("yield 1 ", i)
            yield i
            i = i+1
        print("yield 2 ", i)
        yield i

>>> for a in fonction_yield(2):
        print(a)

yield 1  0
0
yield 2  1
1
>>> for a in fonction_yield(3):
        print(a)

yield 1  0
0
yield 1  1
1
yield 2  2
2
```

Le programme affiche tous les entiers compris entre 0 et 2 inclus ainsi que le texte « **`yield 1`** » ou « **`yield 2`** » selon l'instruction « **`yield`** » qui a retourné le résultat. Lorsque la fonction a finalement terminé son exécution, le prochain appel agit comme si c'était la première fois qu'on l'appelait.

La compilation dynamique

Identificateur appelable

La fonction « **callable** » retourne un booléen permettant de savoir si un identificateur est une fonction, de savoir par conséquent si tel identificateur est appelable comme une fonction.

```
>>> x = 5
>>> def y() :
        return  None

>>> print(callable(x))
False
>>> #affiche  False  car  x  est  une  variable
>>> print(callable(y))
True
>>> #affiche True  car  y est une  fonction
```

eval

La fonction évalue toute chaîne de caractères contenant une expression écrite avec la syntaxe du langage python. Cette expression peut utiliser toute variable ou toute fonction accessible au moment où est appelée la fonction « **eval** ».

```
>>> x = 3
>>> y = 4
>>> def carre(x) : return x**2

>>> print(eval("carre(x)+carre(y)+2*x*y"))
49
>>> print(carre(x+y))
49
```

Si l'expression envoyée à la fonction « **eval** » inclut une variable non définie, l'interpréteur python génère une erreur.

compile, exec

Plus complète que la fonction « **eval** », la fonction compile permet d'ajouter une ou plusieurs fonctions au programme, celle-ci étant définie par une chaîne de caractères. Le code est d'abord compilé **« compile »** puis incorporé au programme « **exec** ».

```
>>> import math
>>> str = """def coordonnees_polaires(x,y):
        rho     = math.sqrt(x*x+y*y)
        theta = math.atan2 (y,x)
        return rho,  theta"""
>>> # fonction définie par une chaîne de caractères
```

La fonction compile prend en fait trois arguments. Le premier est la chaîne de caractères contenant le code à compiler. Le second paramètre contient un nom de fichier dans lequel seront placées les erreurs de compilation. Le troisième paramètre est une chaîne de caractères à choisir parmi **« exec »** ou « **eval** ». Selon ce

choix, ce sera la fonction « **exec** » ou « **eval** » qui devra être utilisée pour agréger le résultat de la fonction compile au programme.

```
>>> obj  =  compile(str,"","exec")  #fonction  compilée
>>> exec(obj) # fonction incorporée au programme
>>> print(coordonnees_polaires(1,1))
(1.4142135623730951, 0.7853981633974483)
```

L'exemple suivant donne un exemple d'utilisation de la fonction compile avec la fonction « **eval** ».

```
>>> import math
>>> str =  """math.sqrt(x*x+y*y)"""
>>> obj  =  compile(str,"","eval")
>>> x = 1
>>> x,y = 1,2
>>> print(eval(obj))
2.23606797749979
```

Atelier 3

Exercice n° 1

1. Définissez une fonction **`compteCar(ca,ch)`** qui renvoie le nombre de fois que l'on rencontre le caractère **`ca`** dans la chaîne de caractères **`ch`**.
2. Définissez une fonction **`indexMax(liste)`** qui renvoie l'index de l'élément ayant la valeur la plus élevée dans la liste transmise en argument.
3. Définissez une fonction **`nomMois(n)`** qui renvoie le nom du énième mois de l'année.
4. Définissez une fonction **`inverse(ch)`** qui permette d'inverser les l'ordre des caractères d'une chaîne quelconque. La chaîne inversée sera renvoyée au programme appelant.
5. Définissez une fonction **`compteMots(ph)`** qui renvoie le nombre de mots contenus dans la phrase **`ph`**. On considère comme mots les ensembles de caractères inclus entre des espaces.
6. Définissez une fonction **`changeCar(ch,ca1,ca2,debut,fin)`** qui remplace tous les caractères **`ca1`** par des caractères **`ca2`** dans la chaîne de caractères **`ch`**, à partir de l'indice **`debut`** et jusqu'à l'indice **`fin`**, ces deux derniers arguments pouvant être omis et dans ce cas la chaîne est traitée d'une extrémité à l'autre.
7. Définissez une fonction **`eleMax(liste,debut,fin)`** qui renvoie l'élément ayant la plus grande valeur dans la liste transmise. Les deux arguments **`debut`** et **`fin`** indiqueront les indices entre lesquels doit s'exercer la recherche, et chacun d'eux pourra être omis comme dans l'exercice précédent.
8. Définissez une fonction **`volBoite(x1,x2,x3)`** qui renvoie le volume d'une boîte parallélépipédique dont on fournit les trois dimensions **`x1`**, **`x2`**, **`x3`** en arguments.