Premiers scripts, ou comment conserver nos programmes :

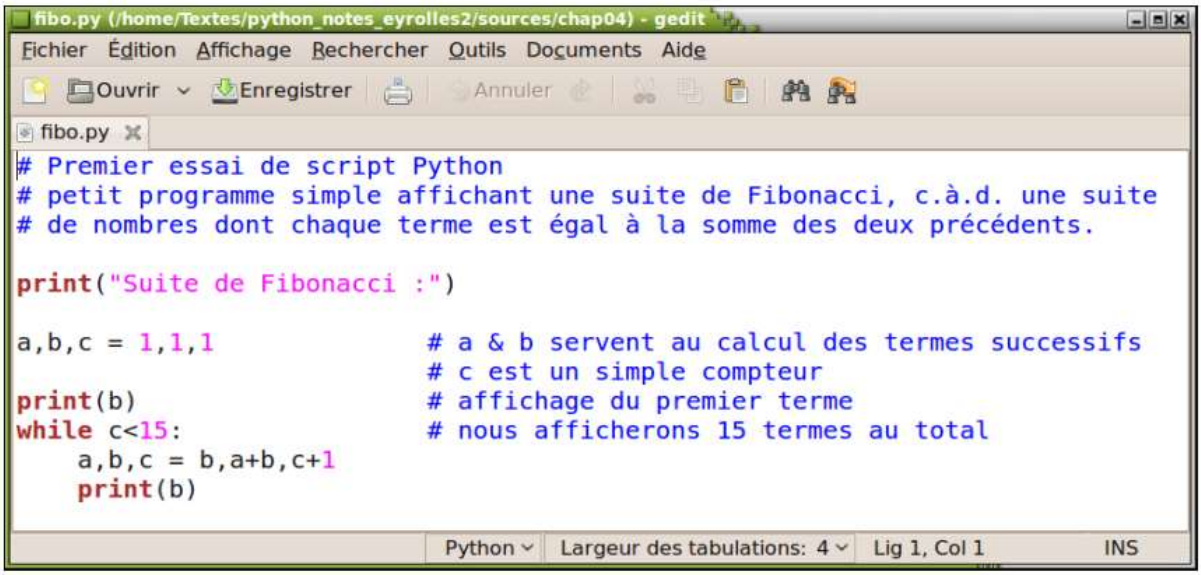
Jusqu'à présent, vous avez toujours utilisé Python en mode interactif (c'est-à-dire que vous avez à chaque fois entré les commandes directement dans l'interpréteur, sans les sauvegarder au préalable dans un fichier). Cela vous a permis d'apprendre très rapidement les bases du langage, par expérimentation directe. Cette façon de faire présente toutefois un gros inconvénient : toutes les séquences d'instructions que vous avez écrites disparaissent irrémédiablement dès que vous fermez l'interpréteur. Avant de poursuivre plus avant votre étude, il est donc temps que vous appreniez à sauvegarder vos programmes dans des fichiers, sur disque dur ou clef USB, de manière à pouvoir les retravailler par étapes successives, les transférer sur d'autres machines, etc.

Pour ce faire, vous allez désormais rédiger vos séquences d'instructions dans un *éditeur de texte* quelconque (par exemple *Kate*, *Gedit*, *Geany*... sous *Linux*, *Wordpad, Geany, Komodo editor...* sous *Windows*, ou encore l'éditeur incorporé dans l'interface de développement *IDLE* qui fait partie de la distribution de Python pour Windows). Ainsi vous écrirez *un script*, que vous pourrez ensuite sauvegarder, modifier, copier, etc. comme n'importe quel autre texte traité par ordinateur.

Par la suite, lorsque vous voudrez tester l'exécution de votre programme, il vous suffira de lancer l'interpréteur Python en lui fournissant (comme argument) le nom du fichier qui contient le script. Par exemple, si vous avez placé un script dans un fichier nommé « MonScript », il suffira d'entrer la commande suivante dans une fenêtre de terminal pour que ce script s'exécute :

```
python3 MonScript
```

Pour faire mieux encore, veillez à choisir pour votre fichier un nom qui se termine par l'extension .**py** Si vous respectez cette convention, vous pourrez aussi lancer l'exécution du script, simplement en cliquant sur son nom ou sur l'icône correspondante dans le gestionnaire de fichiers (c'est-à-dire l'explorateur, sous *Windows*, ou bien *Nautilus*, *Konqueror*... sous *Linux*).
Ces gestionnaires graphiques « savent » en effet qu'ils doivent lancer l'interpréteur Python chaque fois que leur utilisateur essaye d'ouvrir un fichier dont le nom se termine par **.py** (cela suppose bien entendu qu'ils aient été correctement configurés). La même convention permet en outre aux éditeurs « intelligents » de reconnaître automatiquement les scripts Python, et d'adapter leur coloration syntaxique en conséquence.
La figure ci-après illustre l'utilisation de l'éditeur *Gedit* sous *Linux (Ubuntu)* pour écrire un script :



Il serait parfaitement possible d'utiliser un système de traitement de texte, à la condition d'effectuer la sauvegarde sous un format « texte pur » (sans balises de mise en page). Il est cependant préférable d'utiliser un véritable éditeur « intelligent » tel que *Gedit, Geany*, ou *IDLE*, muni d'une fonction de coloration syntaxique pour Python, qui vous aide à éviter les fautes de syntaxe. Avec *IDLE*, suivez le menu : File → New window (ou tapez <Ctrl-N>) pour ouvrir une nouvelle fenêtre dans laquelle vous écrirez votre script. Pour l'exécuter, il vous suffira (après sauvegarde), de suivre le menu : Edit → Run script (ou de taper <Ctrl-F5>). Si l'interpréteur Python 3 a été installé sur votre machine comme interpréteur Python par défaut, vous devriez pouvoir aussi entrer tout simplement : **python MonScript** . Mais attention : si plusieurs versions de Python sont présentes, il se peut que cette commande active plutôt une version antérieure (Python 2.x).

On peut insérer des commentaires quelconques à peu près n'importe où dans un script. Il suffit de les faire précéder d'un caractère #. Lorsqu'il rencontre ce caractère, l'interpréteur Python ignore tout ce qui suit, jusqu'à la fin de la ligne courante.

Comprenez bien qu'il est important d'inclure des commentaires *au fur et à mesure de l'avancement de votre travail de programmation*. N'attendez pas que votre script soit terminé pour les ajouter « après coup ». Vous vous rendrez progressivement compte qu'un programmeur passe *énormément de temps* à relire son propre code (pour le modifier, y rechercher des erreurs, etc.). Cette relecture sera grandement facilitée si le code comporte de nombreuses explications et remarques.
Ouvrez donc un éditeur de texte, et rédigez le script ci-dessous :

```
# Premier essai de script Python
# petit programme simple affichant une suite de Fibonacci, c.à.d. une suite
# de nombres dont chaque terme est égal à la somme des deux précédents.

a, b, c = 1, 1, 1           # a & b servent au calcul des termes successifs
                            # c est un simple compteur
print(b)                    # affichage du premier terme
while c<15:                 # nous afficherons 15 termes au total
    a, b, c = b, a+b, c+1
    print(b)
```

Afin de vous montrer tout de suite le bon exemple, nous commençons ce script par trois lignes de commentaires, qui contiennent une courte description de la fonctionnalité du programme. Prenez l'habitude de faire de même dans vos propres scripts.

Certaines lignes de code sont également documentées. Si vous procédez comme nous l'avons fait, c'est-à-dire en insérant des commentaires à la droite des instructions correspondantes, veillez à les écarter suffisamment de celles-ci, afin de ne pas gêner leur lisibilité.

La fonction input() :

La plupart des scripts élaborés nécessitent à un moment ou l'autre une intervention de l'utilisateur (entrée d'un paramètre, clic de souris sur un bouton, etc.). Dans un script en mode texte (comme ceux que nous avons créés jusqu'à présent), la méthode la plus simple consiste à employer la fonction intégrée **input()**. Cette fonction provoque une interruption dans le programme courant. L'utilisateur est invité à entrer des caractères au clavier et à terminer avec <*Enter*>. Lorsque cette touche est enfoncée, l'exécution du programme se poursuit, et la fonction fournit en retour une chaîne de caractères correspondant à ce que l'utilisateur a saisi. Cette chaîne peut alors être assignée à une variable quelconque, convertie, etc.
On peut invoquer la fonction **input()** en laissant les parenthèses vides. On peut aussi y placer en argument un message explicatif destiné à l'utilisateur. Exemple :

```
prenom = input("Entrez votre prénom : ")
print("Bonjour,", prenom)
```

ou encore :

```
print("Veuillez entrer un nombre positif quelconque : ", end=" ")
ch = input()
nn = int(ch)              # conversion de la chaîne en un nombre entier
print("Le carré de", nn, "vaut", nn**2)
```

Importer un module de fonctions :

Vous avez déjà rencontré d'autres *fonctions intégrées* au langage lui-même, comme la fonction **len()**, par exemple, qui permet de connaître la longueur d'une chaîne de caractères. Il va de soi cependant qu'il n'est pas possible d'intégrer toutes les fonctions imaginables dans le corps standard de Python, car il en existe virtuellement une infinité: vous apprendrez d'ailleurs très bientôt comment en créer vous-même de nouvelles. 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*.

> *Les modules sont des fichiers qui regroupent des ensembles de fonctions*[26].

Vous verrez plus loin combien il est commode de découper un programme important en plusieurs fichiers de taille modeste pour en faciliter la maintenance. Une application Python typique sera alors constituée d'un programme principal, accompagné de un ou plusieurs modules contenant chacun les définitions d'un certain nombre de fonctions accessoires.
Il existe un grand nombre de modules pré-programmés 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*, par exemple, contient les définitions de nombreuses fonctions mathématiques telles que *sinus*, *cosinus*, *tangente*, *racine carrée*, etc. Pour pouvoir utiliser ces fonctions, il vous suffit d'incorporer la ligne suivante au début de votre script:

```
from math import *
```

Cette ligne indique à Python qu'il lui faut inclure dans le programme courant *toutes* les fonctions (c'est là la signification du symbole « joker » *) du module *math*, lequel contient une bibliothèque de fonctions mathématiques pré-programmées.
Dans le corps du script lui-même, vous écrirez par exemple :
racine = sqrt(nombre) pour assigner à la variable **racine** la racine carrée de **nombre**,
sinusx = sin(angle) pour assigner à la variable **sinusx** le sinus de **angle** (en radians !), etc. Exemple :

```
# Démo : utilisation des fonctions du module <math>

from math import *

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

L'exécution de ce script provoque l'affichage suivant :

```
racine carrée de 121 = 11.0
sinus de 0.523598775598 radians = 0.5
```

Ce court exemple illustre déjà fort bien quelques caractéristiques importantes des fonctions :

• une fonction apparaît sous la forme d'*un nom quelconque associé à des parenthèses*
exemple : **sqrt()**

• dans les parenthèses, on *transmet* à la fonction un ou plusieurs *arguments*
exemple : **sqrt(121)**
• la fonction fournit une *valeur de retour* (on dira aussi qu'elle « retourne », ou mieux, qu'elle « renvoie » une valeur)
exemple : **11.0**

Nous allons développer tout ceci dans les pages suivantes. Veuillez noter au passage que les fonctions mathématiques utilisées ici ne représentent qu'un tout premier exemple. Un simple coup d'œil dans la documentation des bibliothèques Python vous permettra de constater que de très nombreuses fonctions sont d'ores et déjà disponibles pour réaliser une multitude de tâches, y compris des algorithmes mathématiques très complexes (Python est couramment utilisé dans les universités pour la résolution de problèmes scientifiques de haut niveau). Il est donc hors de question de fournir ici une liste détaillée. Une telle liste est aisément accessible dans le système d'aide de Python :
Documentation HTML → Python documentation → Modules index → math
Au chapitre suivant, nous apprendrons comment créer nous-mêmes de nouvelles fonctions.

Module NumPy :

Chargement du module :

On charge le module NumPy avec la commande :

```
1  import numpy
```

Par convention, on utilise np comme nom raccourci pour NumPy :

```
1  import numpy as np
```

Objets de type array :

Les objets de type array correspondent à des tableaux à une ou plusieurs dimensions et permettent d'effectuer du calcul vectoriel. La fonction array() convertit un conteneur (comme une liste ou un tuple) en un objet de type array
Voici un exemple de conversion d'une liste à une dimension en objet *array* :

```
1  import numpy as np
2  a = [1, 2, 3]
3  np.array(a)
```

Sur un modèle similaire à la fonction range(), la fonction arange() permet de construire un *array* à une dimension :

```
1  np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Comme avec range(), on peut spécifier en argument une borne de début, une borne de fin et un pas :

```
1  np.arange(10, 0, -1)
```

```
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
```

on peut effectuer des opérations vectorielles **élément par élément** sur ce type d'objet, ce qui est bien commode lorsqu'on analyse de grandes quantités de données. Regardez ces exemples :

```
1  v = np.arange(4)
2  v
```

```
array([0, 1, 2, 3])
```

On ajoute 1 à **chacun** des éléments de l'*array* v :

```
1  v + 1
```

```
array([1, 2, 3, 4])
```

On multiplie par 2 **chacun** des éléments de l'*array* v :

```
1  v * 2
```

```
array([0, 2, 4, 6])
```

Il est aussi possible de multiplier deux *arrays* entre eux. Le résultat correspond alors à la multiplication **élément par élément** des deux *arrays* initiaux :

```
1  v * v
```

```
array([0, 1, 4, 9])
```

Array et dimensions :

Il est aussi possible de construire des objets arrays à deux dimensions, il sufft de passer en argument une liste de listes à la fonction array() :

```
1  w = np.array([[1, 2], [3, 4], [5, 6]])
2  w
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Voici quelques attributs intéressants pour décrire un objet array :

```
1  v = np.arange(4)
2  v
```

```
array([0, 1, 2, 3])
```

```
1  w = np.array([[1, 2], [3, 4], [5, 6]])
2  w
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

L'attribut .ndim renvoie le nombre de dimensions de l'*array*. Par exemple, 1 pour un vecteur et 2 pour une matrice :

```
1  v.ndim
```

```
1
```

```
1  w.ndim
```

```
2
```

L'attribut .shape renvoie les dimensions sous forme d'un tuple. Dans le cas d'une matrice (*array* à deux dimensions), la première valeur du tuple correspond au nombre de lignes et la seconde au nombre de colonnes.

```
1  v.shape
```

```
(4,)
```

```
1  w.shape
```

```
(3, 2)
```

Enfin, l'attribut .size renvoie le nombre total d'éléments contenus dans l'array :

```
1  v.size
```

```
4
```

```
1  w.size
```

```
6
```

```
1  a2 = np.ones((3, 3))
2  a2
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Méthodes de calcul sur les arrays et l'argument axis :

Chaque *array NumPy* possède une multitude de méthodes. Nombre d'entre elles permettent de faire des calculs de base comme .mean() pour la moyenne, .sum() pour la somme, .std() pour l'écart-type, .max() pour extraire le maximum, .min() pour extraire le minimum, etc. La liste exhaustive est disponible en ligne 3. Par défaut, chacune de ces méthodes effectuera l'opération sur l'*array* entier, quelle que soit sa dimensionnalité. Par exemple :

```
array([[2, 7],
       [6, 4],
       [0, 3],
       [1, 5]])
```

```
1  a.max()
```

```
7
```

La méthode .max() a bien renvoyé la valeur maximale . Un argument *très* utile existant dans toutes ces méthodes est axis. Pour un *array* 2D, axis=0 signifie qu'on fera l'opération le long de l'axe 0, à savoir les lignes. C'est-à-dire que l'opération se fait en variant les lignes. On récupère ainsi une valeur par colonne :

```
1  a.max(axis=0)
```

```
array([6, 7])
```

Dans l'*array* 1D récupéré, le premier élément vaut 6 (maximum de la 1ère colonne) et le second vaut 7 (maximum de la seconde colonne).

Avec axis=1, on fait une opération similaire, mais en faisant varier les colonnes. On récupère ainsi une valeur par ligne :

```
1  a.max(axis=1)
```

```
array([7, 6, 3, 5])
```

Indices :

Pour récupérer un ou plusieurs élément(s) d'un objet *array*, vous pouvez utiliser les indices, de la même manière qu'avec les listes :

```
1  a = np.arange(10)
2  a
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
1  a[1]
```

```
1
```

L'utilisation des tranches est aussi possible :

```
1  a[5:]
```

```
array([5, 6, 7, 8, 9])
```

Ainsi que les pas :

```
1  a[::2]
```

```
array([0, 2, 4, 6, 8])
```

Dans le cas d'un objet *array* à deux dimensions, vous pouvez récupérer une ligne complète (d'indice *i*), une colonne complète (d'indice *j*) ou bien un seul élément.

```
1  a = np.array([[1, 2], [3, 4]])
2  a
```

```
array([[1, 2],
       [3, 4]])
```

```
1  a[:,0]
```

```
array([1, 3])
```

```
1  a[0,:]
```

```
array([1, 2])
```

La syntaxe a[i,:] renvoie la ligne d'indice i, et a[:,j] renvoie la colonne d'indice j. Les tranches sont aussi utilisables sur un *array* à deux dimensions.

```
a[1, 1]
```

```
4
```

La syntaxe a[i, j] renvoie l'élément à la ligne d'indice i et à la colonne d'indice j. Notez que *NumPy* suit la convention mathématiques des matrices 4, à savoir, qu'**on définit toujours un élément par sa ligne puis par sa colonne**. En mathématiques, l'élément $a_{i\,j}$ d'une matrice A se trouve à la i^{me} ligne et à la j^{me} colonne :

Copie d'arrays :

```
a = np.arange(5)
a
```

```
array([0, 1, 2, 3, 4])
```

```
b = a
b[2] = -300
b
```

```
array([   0,    1, -300,    3,    4])
```

Construction automatique de matrices :

Il est parfois pénible de construire une matrice (*array* à deux dimensions) à l'aide d'une liste de listes. Le module *NumPy* possède quelques fonctions pratiques pour initialiser des matrices. Par exemple, Les fonctions zeros() et ones () construisent des objets *array* contenant des 0 ou des 1. Il sufft de leur passer en argument un tuple indiquant les dimensions voulues :

```
np.zeros((2, 3))
```

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

```
np.ones((3, 3))
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Enfin, si vous voulez construire une matrice avec autre chose que des 0 ou des 1, vous avez à votre disposition la fonction full() :

```
np.full((2, 3), 7, int)
```

```
array([[7, 7, 7],
       [7, 7, 7]])
```

Un peu d'algèbre linéaire :

Après avoir manipulé les objets *array* comme des vecteurs et des matrices, voici quelques fonctions pour faire de l'algèbre linéaire.

La fonction transpose() renvoie la transposée d'un *array*. Par exemple, pour une matrice:

```
1  a = np.resize(np.arange(1, 10), (3, 3))
2  a
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
1  np.transpose(a)
```

```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

Tout objet *array* possède un attribut .T qui contient la transposée, il est ainsi possible d'utiliser cette notation objet plus compacte :

```
1  a.T
```

```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

La fonction dot() permet de multiplier deux matrices :

```
1  a = np.resize(np.arange(4), (2, 2))
2  a
```

```
array([[0, 1],
       [2, 3]])
```

```
1  np.dot(a, a)
```

```
array([[ 2,  3],
       [ 6, 11]])
```

```
1  a * a
```

```
array([[0, 1],
       [4, 9]])
```

Notez bien que dot(a, a) renvoie le **produit matriciel** entre deux matrices, alors que l'opération a * a renvoie le produit **élément par élément**.

Pour toutes les opérations suivantes, nous utiliserons des fonctions du sous-module *linalg* de *NumPy*. La fonction diag() permet de générer une matrice diagonale :

```
1  a = np.diag((1, 2, 3))
2  a
```

```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

La fonction inv() renvoie l'inverse d'une matrice carrée

```
1  np.linalg.inv(a)
```

```
array([[1.        , 0.        , 0.        ],
       [0.        , 0.5       , 0.        ],
       [0.        , 0.        , 0.33333333]])
```

La fonction det() renvoie le déterminant d'une matrice carrée :

```
1  np.linalg.det(a)
```

```
6.0
```

Enfin, la fonction eig() renvoie les vecteurs et valeurs propres :

```
1  np.linalg.eig(a)
```

```
EigResult(eigenvalues=array([1., 2., 3.]), eigenvectors=array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]]))
```

Fonctions originales :

Définir 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 fort 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[30] pour que ceux-ci restent clairs. D'autre part, il arrivera souvent qu'une même séquence d'instructions doive être utilisée à plusieurs reprises dans un programme, et on souhaitera bien évidemment ne pas avoir à la reproduire systématiquement.

Les *fonctions*[31] 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.

Nous avons déjà rencontré diverses fonctions pré-programmées. Voyons à présent comment en définir nous-mêmes de nouvelles.

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[32], 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* que nous étudierons plus loin).

• 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 (voir plus loin).

Fonction simple sans paramètres :

Pour notre première approche concrète des fonctions, nous allons travailler à nouveau en mode interactif. 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():
...     n = 1
...     while n <11 :
...         print(n * 7, end =' ')
...         n = n +1
...
```

En entrant ces quelques lignes, nous avons défini une fonction très simple qui calcule et affiche les 10 premiers termes de la table de multiplication par 7. Notez bien les parenthèses[33], le double point, et l'indentation du bloc d'instructions qui suit la ligne d'en-tête (c'est ce bloc d'instructions qui constitue le corps de la fonction proprement dite).

Pour utiliser la fonction que nous venons de définir, il suffit de l'appeler par son nom. Ainsi :

```
>>> table7()
```

provoque l'affichage de :

```
7 14 21 28 35 42 49 56 63 70
```

Fonction avec paramètre :

Dans nos derniers exemples, nous avons défini et utilisé une fonction qui affiche les termes de la table de multiplication par 7. Supposons à présent que nous voulions faire de même avec la table par 9. Nous pouvons bien entendu réécrire entièrement une

nouvelle fonction pour cela. Mais si nous nous intéressons plus tard à la table par 13, il nous faudra encore recommencer. Ne serait-il donc pas plus intéressant de définir une fonction qui soit capable d'afficher n'importe quelle table, à la demande ? Lorsque nous appellerons cette fonction, nous devrons bien évidemment pouvoir lui indiquer quelle table nous souhaitons afficher. Cette information que nous voulons transmettre à la fonction au moment même où nous l'appelons s'appelle un ***argument***. Nous avons déjà rencontré à plusieurs reprises des fonctions intégrées qui utilisent des arguments. La fonction **sin(a)**, par exemple, calcule le sinus de l'angle **a**. La fonction **sin()** utilise donc la valeur numérique de **a** comme argument pour effectuer son travail.

Dans la définition d'une telle fonction, il faut prévoir une variable particulière pour recevoir l'argument transmis. Cette variable particulière s'appelle un ***paramètre***. On lui choisit un nom en respectant les mêmes règles de syntaxe que d'habitude (pas de lettres accentuées, etc.), et on place ce nom entre les parenthèses qui accompagnent la définition de la fonction.

Voici ce que cela donne dans le cas qui nous intéresse :

```
>>> def table(base):
...     n = 1
...     while n <11 :
...         print(n * base, end =' ')
...         n = n +1
```

La fonction **table()** telle que définie ci-dessus utilise le paramètre **base** pour calculer les dix premiers termes de la table de multiplication correspondante.

Pour tester cette nouvelle fonction, il nous suffit de l'appeler avec un argument. Exemples :

```
>>> table(13)
13 26 39 52 65 78 91 104 117 130

>>> table(9)
9 18 27 36 45 54 63 72 81 90
```

Dans ces exemples, la valeur que nous indiquons entre parenthèses lors de l'appel de la fonction (et qui est donc un argument) est automatiquement affectée au paramètre **base**. Dans le corps de la fonction, **base** joue le même rôle que n'importe quelle autre variable. Lorsque nous entrons la commande **table(9)**, nous signifions à la machine que nous voulons exécuter la fonction **table()** en affectant la valeur **9** à la variable **base**.

Utilisation d'une variable comme argument :

Dans les 2 exemples qui précèdent, l'argument que nous avons utilisé en appelant la fonction **table()** était à chaque fois une constante (la valeur 13, puis la valeur 9). Cela n'est nullement obligatoire. L'argument que nous utilisons dans l'appel d'une fonction peut être une variable lui aussi, comme dans l'exemple ci-dessous. Analysez bien cet exemple, essayez-le concrètement, et décrivez le mieux possible dans votre cahier d'exercices ce que vous obtenez, en expliquant avec vos propres mots ce qui se passe. Cet exemple devrait vous donner un premier aperçu de l'utilité des fonctions pour accomplir simplement des tâches complexes :

```
>>> a = 1
>>> while a <20:
...     table(a)
...     a = a +1
...
```

Remarque importante :

Dans l'exemple ci-dessus, l'argument que nous passons à la fonction **table()** est le contenu de la variable **a**. À l'intérieur de la fonction, cet argument est affecté au paramètre **base**, qui est une tout autre variable. Notez donc bien dès à présent que :
Ces noms peuvent être identiques si vous le voulez, mais vous devez bien comprendre qu'ils ne désignent pas la même chose (en dépit du fait qu'ils puissent éventuellement contenir une valeur identique).

Fonction avec plusieurs paramètres :

La fonction **table()** est certainement intéressante, mais elle n'affiche toujours que les dix premiers termes de la table de multiplication, alors que nous pourrions souhaiter qu'elle en affiche d'autres. Qu'à cela ne tienne. Nous allons l'améliorer en lui ajoutant des paramètres supplémentaires, dans une nouvelle version que nous appellerons cette fois **tableMulti()** :

```
>>> def tableMulti(base, debut, fin):
...     print('Fragment de la table de multiplication par', base, ':')
...     n = debut
...     while n <= fin :
...         print(n, 'x', base, '=', n * base)
...         n = n +1
```

Cette nouvelle fonction utilise donc trois paramètres : la base de la table comme dans l'exemple précédent, l'indice du premier terme à afficher, l'indice du dernier terme à afficher.
Essayons cette fonction en entrant par exemple :

```
>>> tableMulti(8, 13, 17)
```

ce qui devrait provoquer l'affichage de :

```
Fragment de la table de multiplication par 8 :
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
```

Notes

• Pour définir une fonction avec plusieurs paramètres, il suffit d'inclure ceux-ci entre les parenthèses qui suivent le nom de la fonction, en les séparant à l'aide de virgules.

• Lors de l'appel de la fonction, les arguments utilisés doivent être fournis *dans le même ordre* que celui des paramètres correspondants (en les séparant eux aussi à l'aide de virgules). Le premier argument sera affecté au premier paramètre, le second argument sera affecté au second paramètre, et ainsi de suite.

• À titre d'exercice, essayez la séquence d'instructions suivantes et décrivez dans votre cahier d'exercices le résultat obtenu :

```
>>> t, d, f = 11, 5, 10
>>> while t<21:
...     tableMulti(t,d,f)
...     t, d, f = t +1, d +3, f +5
...
```

Utilisation des fonctions dans un script :

Pour cette première approche des fonctions, nous n'avons utilisé jusqu'ici que le mode interactif de l'interpréteur Python.

Il est bien évident que les fonctions peuvent aussi s'utiliser dans des scripts. Veuillez donc essayer vous-même le petit programme ci-dessous, lequel calcule le volume d'une sphère à l'aide de la formule

que vous connaissez certainement : $V = \frac{4}{3}\pi R^3$

```
def cube(n):
   return n**3

def volumeSphere(r):
   return 4 * 3.1416 * cube(r) / 3

r = input('Entrez la valeur du rayon : ')
print('Le volume de cette sphère vaut', volumeSphere(float(r)))
```

Notes

À bien y regarder, ce programme comporte trois parties : les deux fonctions **cube()** et **volumeSphere()**, et ensuite le corps principal du programme.

Dans le corps principal du programme, on appelle la fonction **volumeSphere()**, en lui transmettant la valeur entrée par l'utilisateur pour le rayon, préalablement convertie en un nombre réel à l'aide de la fonction intégrée **float()**.

À l'intérieur de la fonction **volumeSphere()**, il y a un appel de la fonction **cube()**.

Notez bien que les trois parties du programme ont été disposées dans un certain ordre : *d'abord la définition des fonctions, et ensuite le corps principal du programme.* Cette disposition est nécessaire, parce que l'interpréteur exécute les lignes d'instructions du programme l'une après l'autre, dans l'ordre où elles apparaissent dans le code source.

Pour vous en convaincre, intervertissez cet ordre (en plaçant par exemple le corps principal du programme au début), et prenez note du type de message d'erreur qui est affiché lorsque vous essayez d'exécuter le script ainsi modifié.