7

Fonctions originales

La programmation est l'art d'apprendre à un ordinateur comment accomplir des tâches qu'il n'était pas capable de réaliser auparavant. L'une des méthodes les plus intéressantes pour y arriver consiste à ajouter de nouvelles instructions au langage de programmation que vous utilisez, sous la forme de 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 :

---

[30] On appelle **algorithme** la séquence détaillée de toutes les opérations à effectuer pour résoudre un problème.

[31] Il existe aussi dans d'autres langages des **routines** (parfois appelés sous-programmes) et des **procédures**.
Il n'existe pas de **routines** en Python. Quant au terme de **fonction**, il désigne à la fois les fonctions au sens strict (qui fournissent une valeur en retour), et les procédures (qui n'en fournissent pas).

```
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).

[32] La liste complète des mots réservés Python se trouve page 14.

[33] Un nom de fonction doit toujours être accompagné de parenthèses, même si la fonction n'utilise aucun paramètre. Il en résulte une convention d'écriture qui stipule que dans un texte quelconque traitant de programmation d'ordinateur, un nom de fonction soit toujours accompagné d'une paire de parenthèses vides. Nous respecterons cette convention dans la suite de ce texte.

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
```

Nous pouvons maintenant réutiliser cette fonction à plusieurs reprises, autant de fois que nous le souhaitons. Nous pouvons également l'incorporer dans la définition d'une autre fonction, comme dans l'exemple ci-dessous :

```
>>> def table7triple():
...     print('La table par 7 en triple exemplaire :')
...     table7()
...     table7()
...     table7()
...
```

Utilisons cette nouvelle fonction, en entrant la commande :

```
>>> table7triple()
```

l'affichage résultant devrait être :

```
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
```

Une première fonction peut donc appeler une deuxième fonction, qui elle-même en appelle une troisième, etc. Au stade où nous sommes, vous ne voyez peut-être pas encore très bien l'utilité de tout cela, mais vous pouvez déjà noter deux propriétés intéressantes :

- 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. Par exemple, si vous devez afficher la table par 7 plusieurs fois dans un même programme, vous n'avez pas à réécrire chaque fois l'algorithme qui accomplit ce travail.

Une fonction est donc en quelque sorte une nouvelle instruction personnalisée, que vous ajoutez vous-même librement à votre langage de programmation.

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 concrète-

ment, 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 :

> *Le nom d'une variable que nous passons comme argument n'a rien à voir avec le nom du paramètre correspondant dans la fonction.*

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).

Exercice

7.1 Importez le module ***turtle*** pour pouvoir effectuer des dessins simples.

Vous allez dessiner une série de triangles équilatéraux de différentes couleurs.
Pour ce faire, définissez d'abord une fonction **triangle()** capable de dessiner un triangle d'une couleur bien déterminée (ce qui signifie donc que la définition de votre fonction doit comporter un paramètre pour recevoir le nom de cette couleur).

Utilisez ensuite cette fonction pour reproduire ce même triangle en différents endroits, en changeant de couleur à chaque fois.

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
...
```

Variables locales, variables 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. C'est par exemple le cas des variables **base**, **debut**, **fin** et **n** dans l'exercice précédent.

Chaque fois que la fonction **tableMulti()** est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) un nouvel *espace de noms*[34]. Les contenus des variables **base**, **debut**, **fin** et **n** sont stockés dans cet espace de noms qui est *inaccessible depuis l'extérieur de la fonction*. Ainsi par exemple, si nous essayons d'afficher le contenu de la variable **base** juste après avoir effectué l'exercice ci-dessus, nous obtenons un message d'erreur :

```
>>> print(base)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'base' is not defined
```

[34] Ce concept d'*espace de noms* sera approfondi progressivement. Vous apprendrez également plus loin que les fonctions sont en fait des *objets* dont on crée à chaque fois une nouvelle *instance* lorsqu'on les appelle.

La machine nous signale clairement que le symbole **base** lui est inconnu, alors qu'il était correctement imprimé par la fonction **tableMulti()** elle-même. L'espace de noms qui contient le symbole **base** est strictement réservé au fonctionnement interne de **tableMulti()**, et il est automatiquement détruit dès que la fonction a terminé son travail.

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*. Exemple :

```
>>> def mask():
...     p = 20
...     print(p, q)
...
>>> p, q = 15, 38
>>> mask()
20 38
>>> print(p, q)
15 38
```

Analysons attentivement cet exemple :

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 (où elles pourraient entrer en compétition), ce sont les variables définies localement qui ont la priorité.

On constate en effet que lorsque la fonction **mask()** est lancée, la variable globale **q** y est accessible, puisqu'elle est imprimée correctement. Pour **p**, par contre, c'est la valeur attribuée localement qui est affichée.

On pourrait croire d'abord que la fonction **mask()** a simplement modifié le contenu de la variable globale **p** (puisqu'elle est accessible). Les lignes suivantes démontrent qu'il n'en est rien : en dehors de la fonction **mask()**, la variable globale **p** conserve sa valeur initiale.

Tout ceci peut vous paraître compliqué au premier abord. Vous comprendrez cependant très vite combien il est utile que des variables soient ainsi définies comme étant locales, c'est-à-dire en quelque sorte confinées à l'intérieur d'une fonction. 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.

Cet état de choses peut toutefois être modifié si vous le souhaitez. Il peut se faire par exemple que vous ayez à 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'indi-

quer – à l'intérieur de la définition d'une fonction – quelles sont les variables à traiter globalement.

Dans l'exemple ci-dessous, la variable **a** utilisée à l'intérieur de la fonction **monter()** est non seulement accessible, mais également modifiable, parce qu'elle est signalée explicitement comme étant une variable qu'il faut traiter globalement. Par comparaison, essayez le même exercice en supprimant l'instruction **global** : la variable **a** n'est plus incrémentée à chaque appel de la fonction.

```
>>> def monter():
...     global a
...     a = a+1
...     print(a)
...
>>> a = 15
>>> monter()
16
>>> monter()
17
>>>
```

Vraies fonctions et procédures

Pour les puristes, les fonctions que nous avons décrites jusqu'à présent ne sont pas tout à fait des fonctions au sens strict, mais plus exactement des *procédures*[35]. Une « vraie » fonction (au sens strict) doit en effet *renvoyer une valeur* lorsqu'elle se termine. Une « vraie » fonction peut s'utiliser à la droite du signe égale dans des expressions telles que **y = sin(a)**. On comprend aisément que dans cette expression, la fonction **sin()** renvoie une valeur (le sinus de l'argument **a**) qui est directement affectée à la variable **y**.

Commençons par un exemple extrêmement simple :

```
>>> def cube(w):
...     return w*w*w
...
```

L'instruction **return** définit ce que doit être la valeur renvoyée par la fonction. En l'occurrence, il s'agit du cube de l'argument qui a été transmis lors de l'appel de la fonction. Exemple :

```
>>> b = cube(9)
>>> print(b)
729
```

À titre d'exemple un peu plus élaboré, nous allons maintenant modifier quelque peu la fonction **table()** sur laquelle nous avons déjà pas mal travaillé, afin qu'elle renvoie elle aussi une valeur. Cette valeur sera en l'occurrence une liste (la liste des dix premiers termes de la table de multiplication choisie). Voilà donc une occasion de reparler des listes. Dans la foulée, nous en profiterons pour apprendre encore un nouveau concept :

[35] Dans certains langages de programmation, les fonctions et les procédures sont définies à l'aide d'instructions différentes. Python utilise la même instruction **def** pour définir les unes et les autres.

```
>>> def table(base):
...     resultat = []              # resultat est d'abord une liste vide
...     n = 1
...     while n < 11:
...         b = n * base
...         resultat.append(b)     # ajout d'un terme à la liste
...         n = n +1               # (voir explications ci-dessous)
...     return resultat
...
```

Pour tester cette fonction, nous pouvons entrer par exemple :

```
>>> ta9 = table(9)
```

Ainsi nous affectons à la variable **ta9** les dix premiers termes de la table de multiplication par 9, sous la forme d'une liste :

```
>>> print(ta9)
[9, 18, 27, 36, 45, 54, 63, 72, 81, 90]
>>> print(ta9[0])
9
>>> print(ta9[3])
36
>>> print(ta9[2:5])
[27, 36, 45]
>>>
```

(Rappel : le premier élément d'une liste correspond à l'indice 0).

Notes

- Comme nous l'avons vu dans l'exemple précédent, l'instruction **return** définit ce que doit être la valeur « renvoyée » par la fonction. En l'occurrence, il s'agit ici du contenu de la variable **resultat**, c'est-à-dire la liste des nombres générés par la fonction[36].
- L'instruction **resultat.append(b)** est notre second exemple de l'utilisation d'un concept important sur lequel nous reviendrons encore abondamment par la suite : dans cette instruction, nous appliquons la *méthode* **append()** à l'*objet* **resultat**.

  Nous préciserons petit à petit ce qu'il faut entendre par *objet* en programmation. Pour l'instant, admettons simplement que ce terme très général s'applique notamment aux *listes* de Python. Une *méthode* n'est en fait rien d'autre qu'une fonction (que vous pouvez d'ailleurs reconnaître comme telle à la présence des parenthèses), mais *une fonction qui est associée à un objet*. Elle fait partie de la définition de cet objet, ou plus précisément de la *classe* particulière à laquelle cet objet appartient (nous étudierons ce concept de classe plus tard).

  Mettre en œuvre une méthode associée à un objet consiste en quelque sorte à « faire fonctionner » cet objet d'une manière particulière. Par exemple, on met en œuvre la méthode **methode4()** d'un objet **objet3**, à l'aide d'une instruction du type : **objet3.methode4()** ,

[36] **return** peut également être utilisé sans aucun argument, à l'intérieur d'une fonction, pour provoquer sa fermeture immédiate. La valeur retournée dans ce cas est l'objet **None** (objet particulier, correspondant à « rien »).

c'est-à-dire le nom de l'objet, puis le nom de la méthode, reliés l'un à l'autre par un point. Ce point joue un rôle essentiel : on peut le considérer comme un véritable *opérateur*.

Dans notre exemple, nous appliquons donc la méthode **append()** à l'objet **resultat**, qui est une liste. Sous Python, les *listes* constituent donc une *classe* particulière d'objets, auxquels on peut effectivement appliquer toute une série de *méthodes*. En l'occurrence, la méthode **append()** des objets « listes » sert à leur ajouter un élément par la fin. L'élément à ajouter est transmis entre les parenthèses, comme tout argument qui se respecte.

- ***Remarque :*** nous aurions obtenu un résultat similaire si nous avions utilisé à la place de cette instruction une expression telle que « `resultat = resultat + [b]` » (l'opérateur de concaténation fonctionne en effet aussi avec les listes). Cette façon de procéder est cependant moins rationnelle et beaucoup moins efficace, car elle consiste à redéfinir à chaque itération de la boucle une nouvelle liste **resultat**, dans laquelle la totalité de la liste précédente est à chaque fois recopiée avec ajout d'un élément supplémentaire.
  Lorsque l'on utilise la méthode **append()**, par contre, l'ordinateur procède bel et bien à une modification de la liste existante (sans la recopier dans une nouvelle variable). Cette technique est donc préférable, car elle mobilise moins lourdement les ressources de l'ordinateur, et elle est plus rapide (surtout lorsqu'il s'agit de traiter des listes volumineuses).
- Il n'est pas du tout indispensable que la valeur renvoyée par une fonction soit affectée à une variable (comme nous l'avons fait jusqu'ici dans nos exemples par souci de clarté). Ainsi, nous aurions pu tester les fonction **cube()** et **table()** en entrant les commandes :
  ```
  >>> print(cube(9))
  >>> print(table(9))
  >>> print(table(9)[3])
  ```
  ou encore plus simplement encore :
  ```
  >>> cube(9)...
  ```

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.

> *Dans un script, la définition des fonctions doit précéder leur utilisation.*

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.

En fait, le corps principal d'un programme Python constitue lui-même une entité un peu particulière, qui est toujours reconnue dans le fonctionnement interne de l'interpréteur sous le nom réservé **__main__** (le mot « main » signifie « principal », en anglais. Il est encadré par des caractères « souligné » en double, pour éviter toute confusion avec d'autres symboles). L'exécution d'un script commence toujours avec la première instruction de cette entité **__main__**, où qu'elle puisse se trouver dans le listing. Les instructions qui suivent sont alors exécutées l'une après l'autre, dans l'ordre, jusqu'au premier appel de fonction. Un appel de fonction est comme un détour dans le flux de l'exécution : au lieu de passer à l'instruction suivante, l'interpréteur exécute la fonction appelée, puis revient au programme appelant pour continuer le travail interrompu. Pour que ce mécanisme puisse fonctionner, il faut que l'interpréteur ait pu lire la définition de la fonction avant l'entité **__main__**, et celle-ci sera donc placée en général à la fin du script.

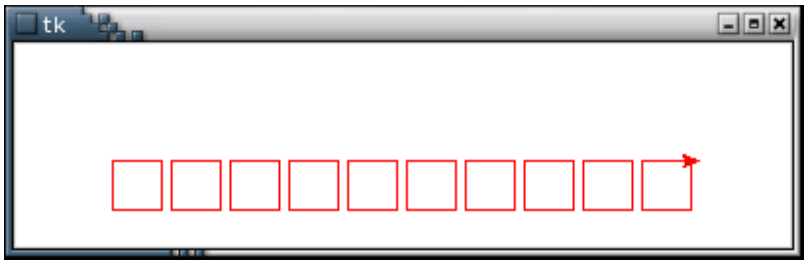
Dans notre exemple, l'entité **__main__** appelle une première fonction qui elle-même en appelle une deuxième. Cette situation est très fréquente en programmation. Si vous voulez comprendre correctement ce qui se passe dans un programme, vous devez donc apprendre à lire un script, non pas de la première à la dernière ligne, mais plutôt en suivant un cheminement analogue à ce qui se passe lors de l'exécution de ce script. Cela signifie donc concrètement que vous devrez souvent analyser un script en commençant par ses dernières lignes !

Modules de fonctions

Afin que vous puissiez mieux comprendre encore la distinction entre la définition d'une fonction et son utilisation au sein d'un programme, nous vous suggérons de placer fréquemment vos définitions de fonctions dans un module Python, et le programme qui les utilise dans un autre.

Exemple :

On souhaite réaliser la série de dessins ci-dessous, à l'aide du module ***turtle*** :



Écrivez les lignes de code suivantes, et sauvegardez-les dans un fichier auquel vous donnerez le nom ***dessins_tortue.py*** :

```
from turtle import *

def carre(taille, couleur):
    "fonction qui dessine un carré de taille et de couleur déterminées"
    color(couleur)
    c =0
    while c <4:
        forward(taille)
        right(90)
        c = c +1
```

Vous pouvez remarquer que la définition de la fonction **carre()** commence par une chaîne de caractères. Cette chaîne ne joue aucun rôle fonctionnel dans le script : elle est traitée par Python *comme un simple commentaire*, mais qui est mémorisé à part dans un système de documentation interne automatique, lequel pourra ensuite être exploité par certains utilitaires et éditeurs « intelligents ».

Si vous programmez dans l'environnement IDLE, par exemple, vous verrez apparaître cette chaîne documentaire dans une « bulle d'aide », chaque fois que vous ferez appel aux fonctions ainsi documentées.

En fait, Python place cette chaîne dans une variable spéciale dont le nom est **__doc__** (le mot « doc » entouré de deux paires de caractères « souligné »), et qui est associée à l'objet fonction comme étant l'un de ses attributs (vous en apprendrez davantage au sujet de ces attributs lorsque nous aborderons les classes d'objets, page 166). Ainsi, vous pouvez vous-même retrouver la chaîne de documentation d'une fonction quelconque en affichant le contenu de cette variable. Exemple :

```
>>> def essai():
...     "Cette fonction est bien documentée mais ne fait presque rien."
...     print("rien à signaler")
...
>>> essai()
rien à signaler
>>> print(essai.__doc__)
Cette fonction est bien documentée mais ne fait presque rien.
```

Prenez donc la peine d'incorporer une telle chaîne explicative dans toutes vos définitions de fonctions futures : il s'agit là d'une pratique hautement recommandable.

Le fichier que vous aurez créé ainsi est dorénavant un véritable *module de fonctions* Python, au même titre que les modules *turtle* ou *math* que vous connaissez déjà. Vous pouvez donc l'utiliser dans n'importe quel autre script, comme celui-ci, par exemple, qui effectuera le travail demandé :

```
from dessins_tortue import *

up()                        # relever le crayon
goto(-150, 50)              # reculer en haut à gauche

# dessiner dix carrés rouges, alignés :
i = 0
while i < 10:
    down()                  # abaisser le crayon
    carre(25, 'red')        # tracer un carré
    up()                    # relever le crayon
    forward(30)             # avancer + loin
    i = i +1
a = input()                 # attendre
```

> *- Attention -*
> *Vous pouvez à priori nommer vos modules de fonctions comme bon vous semble. Sachez cependant qu'il vous sera impossible d'importer un module si son nom est l'un des 33 mots réservés Python signalés à la page 14, car le nom du module importé deviendrait une variable dans votre script, et les mots réservés ne peuvent pas être utilisés comme noms de variables. Rappelons aussi qu'il vous faut éviter de donner à vos modules – et à tous vos scripts en général – le même nom que celui d'un module Python préexistant, sinon vous devez vous attendre à des conflits. Par exemple, si vous donnez le nom* ***turtle.py*** *à un exercice dans lequel vous avez placé une instruction d'importation du module* ***turtle****, c'est l'exercice lui-même que vous allez importer !*

Résumé : structure d'un programme Python type

```
# -*- coding:Utf8 -*-

###################################
# Programme Python type           #
# auteur : G.Swinnen, Liège, 2009 #
# licence : GPL                   #
###################################


#####################################
# Importation de fonctions externes :

from math import sqrt


#################################
# Définition locale de fonctions :

def occurrences(car, ch):
    "Cette fonction renvoie le \
     nombre de caractères <car> \
     contenus dans la chaîne <ch>"

    nc = 0

    i = 0

    while i < len(ch):

        if ch[i] == car:
            nc = nc + 1

        i = i + 1

    return nc


##############################
# Corps principal du programme :

print("Veuillez entrer un nombre :")
nbr = eval(input())

print("Veuillez entrer une phrase :")
phr = input()
print("Entrez le caractère à compter :")
cch = input()

no = occurrences(cch, phr)
rc = sqrt(nbr**3)

print("La racine carrée du cube", end=' ')
print("du nombre fourni vaut", end=' ')
print(rc)

print("La phrase contient", end=' ')
print(no, "caractères", cch)
```

Un programme Python contient en général les blocs suivants, dans l'ordre :
- *Quelques instructions d'initialisation (importation de fonctions et/ou de classes, définition éventuelle de variables globales).*
- *Les définitions locales de fonctions et/ou de classes.*
- *Le corps principal du programme.*

Le programme peut utiliser un nombre quelconque de fonctions, lesquelles sont définies localement ou importées depuis des modules externes.
Vous pouvez vous-même définir de tels modules.

La définition d'une fonction comporte souvent une liste de PARAMÈTRES.
Ce sont toujours des VARIABLES, qui recevront leur valeur lorsque la fonction sera appelée.

Une boucle de répétition de type 'while' doit toujours inclure au moins quatre éléments :
- *l'initialisation d'une variable 'compteur' ;*
- *l'instruction while proprement dite, dans laquelle on exprime la condition de répétition des instructions qui suivent ;*
- *le bloc d'instructions à répéter ;*
- *une instruction d'incrémentation du compteur.*

La fonction "renvoie" toujours une valeur bien déterminée au programme appelant.
Si l'instruction 'return' n'est pas utilisée, ou si elle est utilisée sans argument, la fonction renvoie un objet vide : 'None'.

Le programme qui fait appel à une fonction lui transmet d'habitude une série d'ARGUMENTS, lesquels peuvent être des valeurs, des variables, ou même des expressions.

Exercices

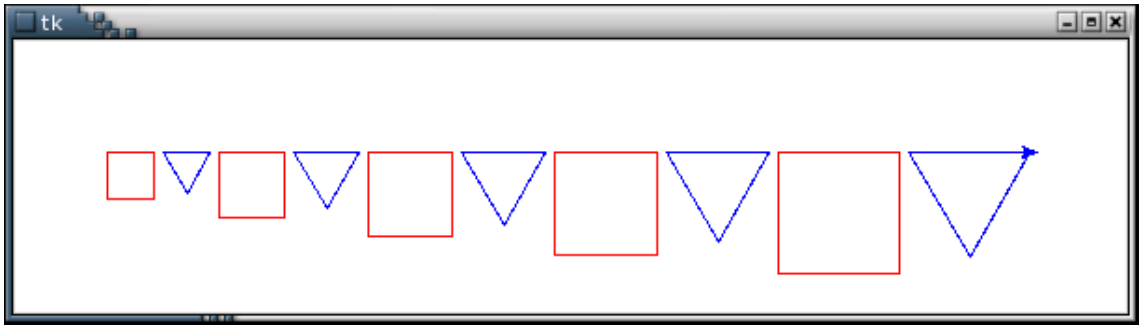
7.2 Définissez une fonction **ligneCar(n, ca)** qui renvoie une chaîne de **n** caractères **ca**.

7.3 Définissez une fonction **surfCercle(R)**. Cette fonction doit renvoyer la surface (l'aire) d'un cercle dont on lui a fourni le rayon **R** en argument. Par exemple, l'exécution de l'instruction :
print(surfCercle(2.5)) doit donner le résultat : **19.63495...**

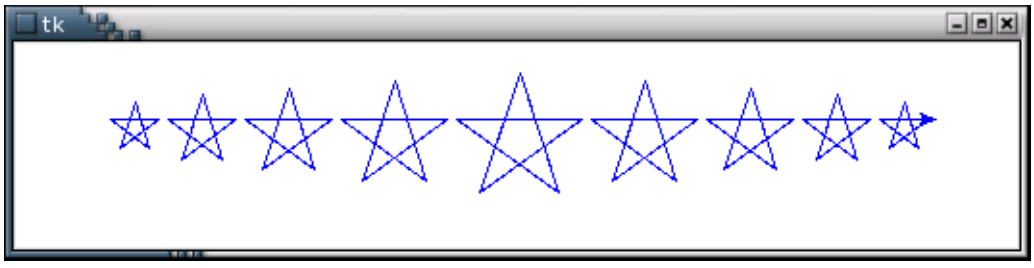
7.4 Définissez une fonction **volBoite(x1,x2,x3)** qui renvoie le volume d'une boîte parallélépipédique dont on fournit les trois dimensions **x1, x2, x3** en arguments.
Par exemple, l'exécution de l'instruction :
print(volBoite(5.2, 7.7, 3.3)) doit donner le résultat : **132.132.**

7.5 Définissez une fonction **maximum(n1,n2,n3)** qui renvoie le plus grand de 3 nombres **n1, n2, n3** fournis en arguments. Par exemple, l'exécution de l'instruction :
print(maximum(2,5,4)) doit donner le résultat : **5.**

7.6 Complétez le module de fonctions graphiques **dessins_tortue.py** décrit à la page 72.
Commencez par ajouter un paramètre **angle** à la fonction **carre()**, de manière à ce que les carrés puissent être tracés dans différentes orientations.
Définissez ensuite une fonction **triangle(taille, couleur, angle)** capable de dessiner un triangle équilatéral d'une taille, d'une couleur et d'une orientation bien déterminées.
Testez votre module à l'aide d'un programme qui fera appel à ces fonctions à plusieurs reprises, avec des arguments variés pour dessiner une série de carrés et de triangles :



7.7 Ajoutez au module de l'exercice précédent une fonction **etoile5()** spécialisée dans le dessin d'étoiles à 5 branches. Dans votre programme principal, insérez une boucle qui dessine une rangée horizontale de 9 petites étoiles de tailles variées :



7.8 Ajoutez au module de l'exercice précédent une fonction **etoile6()** capable de dessiner une étoile à 6 branches, elle-même constituée de deux triangles équilatéraux imbriqués. Cette nouvelle fonction devra faire appel à la fonction **triangle()** définie précédemment.

7.9 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**. Par exemple, l'exécution de l'instruction :
`print(compteCar('e', 'Cette phrase est un exemple'))` doit donner le résultat : `7`

7.10 Définissez une fonction **indexMax(liste)** qui renvoie l'index de l'élément ayant la valeur la plus élevée dans la liste transmise en argument. Exemple d'utilisation :
```
serie = [5, 8, 2, 1, 9, 3, 6, 7]
print(indexMax(serie))
4
```

7.11 Définissez une fonction **nomMois(n)** qui renvoie le nom du n-ième mois de l'année.
Par exemple, l'exécution de l'instruction :
`print(nomMois(4))` doit donner le résultat : `Avril.`

7.12 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.

7.13 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.

Typage des paramètres

Vous avez appris que le *typage* des variables sous Python est un *typage dynamique*, ce qui signifie que le type d'une variable est défini au moment où on lui affecte une valeur. Ce mécanisme fonctionne aussi pour les paramètres d'une fonction. Le type d'un paramètre devient automatiquement le même que celui de l'argument qui a été transmis à la fonction. Exemple :

```
>>> def afficher3fois(arg):
...     print(arg, arg, arg)
...

>>> afficher3fois(5)
5 5 5

>>> afficher3fois('zut')
zut zut zut

>>> afficher3fois([5, 7])
[5, 7] [5, 7] [5, 7]

>>> afficher3fois(6**2)
36 36 36
```

Dans cet exemple, vous pouvez constater que la même fonction **afficher3fois()** accepte dans tous les cas l'argument qu'on lui transmet, que cet argument soit un nombre, une chaîne de ca-

ractères, une liste, *ou même une expression*. Dans ce dernier cas, Python commence par évaluer l'expression, et c'est le résultat de cette évaluation qui est transmis comme argument à la fonction.

Valeurs par défaut pour les paramètres

Dans la définition d'une fonction, il est possible (et souvent souhaitable) de définir un argument par défaut pour chacun des paramètres. On obtient ainsi une fonction *qui peut être appelée avec une partie seulement des arguments attendus*. Exemples :

```
>>> def politesse(nom, vedette ='Monsieur'):
...     print("Veuillez agréer ,", vedette, nom, ", mes salutations cordiales.")
...

>>> politesse('Dupont')
Veuillez agréer , Monsieur Dupont , mes salutations cordiales.

>>> politesse('Durand', 'Mademoiselle')
Veuillez agréer , Mademoiselle Durand , mes salutations cordiales.
```

Lorsque l'on appelle cette fonction en ne lui fournissant que le premier argument, le second reçoit tout de même une valeur par défaut. Si l'on fournit les deux arguments, la valeur par défaut pour le deuxième est tout simplement ignorée.

Vous pouvez définir une valeur par défaut pour tous les paramètres, ou une partie d'entre eux seulement. Dans ce cas, cependant, *les paramètres sans valeur par défaut doivent précéder les autres* dans la liste. Par exemple, la définition ci-dessous est incorrecte :

```
>>> def politesse(vedette ='Monsieur', nom):
```

Autre exemple :

```
>>> def question(annonce, essais =4, please ='Oui ou non, s.v.p.!'):
...     while essais >0:
...         reponse = input(annonce)
...         if reponse in ('o', 'oui','O','Oui','OUI'):
...             return 1
...         if reponse in ('n','non','N','Non','NON'):
...             return 0
...         print(please)
...         essais = essais-1
...
>>>
```

Cette fonction peut être appelée de différentes façons, telles par exemple :

```
rep = question('Voulez-vous vraiment terminer ? ')
```

ou bien :

```
rep = question('Faut-il effacer ce fichier ? ', 3)
```

ou même encore :

```
rep = question('Avez-vous compris ? ', 2, 'Répondez par oui ou par non !')
```

Prenez la peine d'essayer et de décortiquer cet exemple.

Arguments avec étiquettes

Dans la plupart des langages de programmation, les arguments que l'on fournit lors de l'appel d'une fonction doivent être fournis *exactement dans le même ordre* que celui des paramètres qui leur correspondent dans la définition de la fonction.

Python autorise cependant une souplesse beaucoup plus grande. Si les paramètres annoncés dans la définition de la fonction ont reçu chacun une valeur par défaut, sous la forme déjà décrite ci-dessus, on peut faire appel à la fonction en fournissant les arguments correspondants *dans n'importe quel ordre, à la condition de désigner nommément les paramètres correspondants*. Exemple :

```
>>> def oiseau(voltage=100, etat='allumé', action='danser la java'):
...     print('Ce perroquet ne pourra pas', action)
...     print('si vous le branchez sur', voltage, 'volts !')
...     print("L'auteur de ceci est complètement", etat)
...

>>> oiseau(etat='givré', voltage=250, action='vous approuver')
Ce perroquet ne pourra pas vous approuver
si vous le branchez sur 250 volts !
L'auteur de ceci est complètement givré

>>> oiseau()
Ce perroquet ne pourra pas danser la java
si vous le branchez sur 100 volts !
L'auteur de ceci est complètement allumé
```

Exercices

7.14 Modifiez la fonction **volBoite(x1,x2,x3)** que vous avez définie dans un exercice précédent, de manière à ce qu'elle puisse être appelée avec trois, deux, un seul, ou même aucun argument. Utilisez pour ceux ci des valeurs par défaut égales à 10.
Par exemple :

`print(volBoite())` doit donner le résultat : `1000`
`print(volBoite(5.2))` doit donner le résultat : `520.0`
`print(volBoite(5.2, 3))` doit donner le résultat : `156.0`

7.15 Modifiez la fonction **volBoite(x1,x2,x3)** ci-dessus de manière à ce qu'elle puisse être appelée avec un, deux, ou trois arguments. Si un seul est utilisé, la boîte est considérée comme cubique (l'argument étant l'arête de ce cube). Si deux sont utilisés, la boîte est considérée comme un prisme à base carrée (auquel cas le premier argument est le côté du carré, et le second la hauteur du prisme). Si trois arguments sont utilisés, la boîte est considérée comme un parallélépipède. Par exemple :

`print(volBoite())` doit donner le résultat : `-1` (indication d'une erreur)
`print(volBoite(5.2))` doit donner le résultat : `140.608`
`print(volBoite(5.2, 3))` doit donner le résultat : `81.12`
`print(volBoite(5.2, 3, 7.4))` doit donner le résultat : `115.44`

7.16 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). Exemples :

```
>>> phrase = 'Ceci est une toute petite phrase.'
>>> print(changeCar(phrase, ' ', '*'))
Ceci*est*une*toute*petite*phrase.
>>> print(changeCar(phrase, ' ', '*', 8, 12))
Ceci est*une*toute petite phrase.
>>> print(changeCar(phrase, ' ', '*', 12))
Ceci est une*toute*petite*phrase.
>>> print(changeCar(phrase, ' ', '*', fin = 12))
Ceci*est*une*toute petite phrase.
```

7.17 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). Exemples de la fonctionnalité attendue :

```
>>> serie = [9, 3, 6, 1, 7, 5, 4, 8, 2]
>>> print(eleMax(serie))
9
>>> print(eleMax(serie, 2, 5))
7
>>> print(eleMax(serie, 2))
8
>>> print(eleMax(serie, fin =3, debut =1))
6
```