4 Fonctions

1 Composants d'un programme Python

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 pour que ceux-ci restent clairs.

Les **fonctions**[1] 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 rendre les programmes plus modulaires. Cette modularité dans l'architecture d'un programme offre des nombreux avantages :

- Rendre un programme plus facile à lire et à debugger en regroupant sous un même nom des instructions opérant une tâche spécifique du programme.
- Éviter les redondances en définissant à un endroit du programme les fonctions et les classes d'objets nécessaires et en les appelant plus tard aussi souvent qu'il le faut dans le programme. En outre, un seul changement dans la définition d'une fonction ou d'une classe d'objets permet d'avoir des répercutions à plusieurs endroits du programme.
- Diviser une tâche complexe en sous-programmes plus lisibles et plus faciles à gérer.
- Créer des blocs d'instructions réutilisables dans d'autres programmes.

Les fonctions étant les éléments structurants de base de tout langage de programmation, nous allons décrire ici leur définition sous Python. Les **objets** et les **classes** seront quant à eux examinés plus tard.

En programmation, une **fonction** est un ensemble d'instructions regroupées sous un nom et s'exécutant à la demande lorsque la fonction est appelée. En définissant une fonction, le programmeur doit spécifier le nom attribué à la fonction, la liste des paramètres sur lesquels opère la fonction ainsi que la séquence d'instructions définissant son action. L'appel d'une fonction spécifie quant à lui le nom de cette dernière et fournit, entre parenthèses, les arguments qui lui sont nécessaires pour exécuter sa tâche. Vous avez déjà rencontré des fonctions pré-programmées, comme la fonction `len`, par exemple, qui permet de connaître la longueur d'une chaîne de caractères :

```
>>> len('Ma chaîne')
>>> 9
```

Dans l'exemple ci-dessus, le **nom** de la fonction est `len`, l'expression entre parenthèses est appelée l'**argument** de la fonction et le résultat retourné est communément nommé **valeur de retour**. Remarquons que l'utilisation de la fonction `len` ne présuppose aucunement la connaissance de sa définition. En effet, seuls le nombre et le type d'arguments qu'elle attend, l'action qu'elle entreprend et la valeur qu'elle retourne sont nécessaires à son utilisation. Cette encapsulation des instructions définissant une fonction est analogue aux relations hiérarchiques d'une entreprise : un patron (la *fonction appelante*) a besoin d'un employé (la *fonction appelée*) pour effectuer une tâche et lui retourner le résultat de l'opération entreprise. La fonction-*patron* n'est pas consciente de la manière dont la fonction-*employé* effectue sa tâche ; seul lui importe le résultat retourné. Ainsi, l'employé peut appeler d'autres fonctions-*employés* pour mener à bien son action sans que le patron le sache.

1. 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)

La figure ci-dessous montre la fonction-*patron* communiquant avec les fonctions *employé1, employé2* et *employé3* de manière hiérarchique. Notez que l'*employé1* agit à la manière d'un patron pour l'*employé4* et l'*employé5*. En appelant l'*employé1*, la fonction-*patron* n'a pas connaissance de la relation qu'entretient l'*employé1* avec l'*employé4* et l'*employé5*.

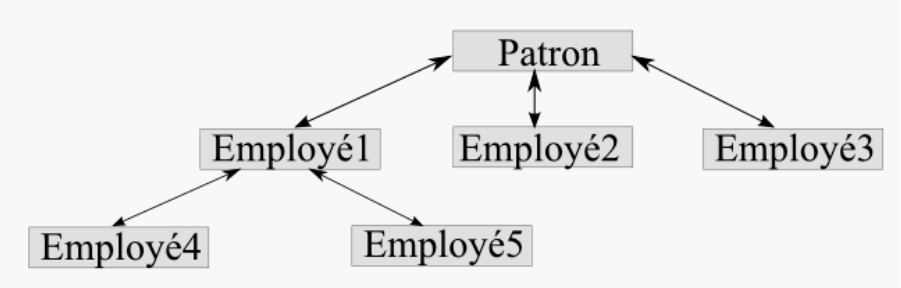


FIGURE 4.1 – Structure hiérarchique d'une entreprise

2 Modules de fonctions

La fonction `len` utilisée dans la section précédente est une **fonction intégrée** au langage lui-même. 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. 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 fonctions [2] réalisant des tâches communes.

Vous verrez au fil de votre apprentissage 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**.

2.1 Module math

Le module `math` 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 :

```
>>> import math
```

Cette ligne indique à Python qu'il lui faut inclure dans le programme courant le module `math`, lequel contient une bibliothèque de fonctions mathématiques pré-programmées. Pour accéder à l'une de ces fonctions, il suffit de spécifier le nom du module ainsi que celui de la fonction séparés par un point :

```
>>> math.sqrt(9)          # appel de la fonction sqrt du module math sur l'argument 9
3.0
```

La fonction `sqrt` située dans le module `math` calcule la racine carrée d'un nombre placé entre parenthèses. Dans l'exemple ci-dessus, la valeur 9 est l'**argument** de la fonction `sqrt` et 3.0 est sa **valeur de retour**. Remarquons que la valeur retournée par la fonction `sqrt` est un nombre réel.

Si un programme a besoin d'importer plusieurs modules, il peut inclure une instruction **import** par module ou alors importer tous les modules en une seule étape en les séparant d'une virgule.

2. En toute rigueur, un module peut contenir aussi des définitions de variables ainsi que des classes. Nous pouvons toutefois laisser ces précisions de côté, provisoirement.

L'accès aux fonctions d'un module importé se fait à l'aide du point situé entre le nom du module et celui de la fonction appelée, ce qui peut rendre l'écriture un peu pénible. Pour pallier à ce problème, il est possible d'importer une, plusieurs voire toutes les fonctions d'un module en une seule instruction :

```
>>> from math import *
>>> degre = 45
>>> radian = (degre/180) * pi          # conversion de l'angle en radian
>>> sin(radian)
0.7071067811865475
```

La première ligne de l'exemple ci-dessus indique à Python qu'il lui faut inclure dans le programme courant *toutes* les fonctions (c'est là la signification du *) du module `math`. Remarquons que la fonction `sin` du module `math` calcule le sinus de l'angle (en radians !) placé entre parenthèses. La transformation en radians d'un angle initialement donné en degrés nécessite alors l'utilisation de la variable `pi` fourni par le module `math`. Cette variable donne une approximation à 15 chiffres de π.

Veuillez noter au passage que les éléments mathématiques utilisés 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 sur IDLE via le système d'aide :

Help → Python Docs → Python v3.7.1 documentation → Global Module Index → math

ou à l'adresse Internet http://docs.python.org/3/py-modindex.html, rubrique math.

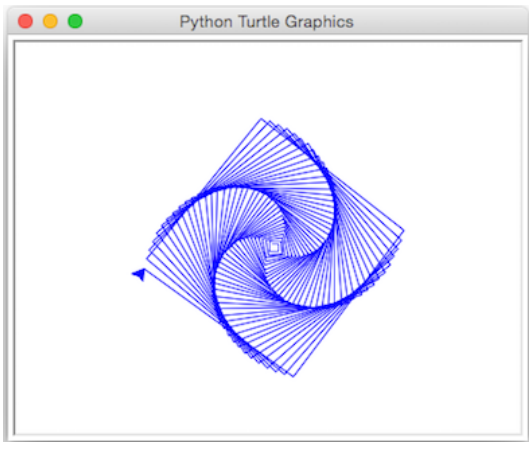
2.2 Module turtle

Comme nous venons de le voir, l'une des grandes qualités de Python est qu'il est extrêmement facile de lui ajouter de nombreuses fonctionnalités par importation de divers **modules**.

Pour illustrer cela, nous allons explorer un module Python qui permet de réaliser des *graphiques tortue*, c'est-à-dire des dessins géométriques correspondant à la piste laissée derrière elle par une petite *tortue* virtuelle, dont nous contrôlons les déplacements sur l'écran de l'ordinateur à l'aide d'instructions simples.

Activer cette tortue est un vrai jeu d'enfant. Plutôt que de vous donner de longues explications, nous vous invitons à essayer tout de suite :

```
>>> from turtle import *
>>> forward(120)
>>> left(90)
>>> color('red')
>>> forward(80)
```



L'exercice est évidemment plus riche si l'on utilise des boucles :

```
>>> reset()
>>> i = 5
>>> while i < 150:
        forward(i)
        right(89)
        i += 1
```

Avant de lancer un tel script, assurez-vous toujours qu'il ne comporte pas de boucle infinie, car si c'est le cas vous risquez de ne plus pouvoir reprendre le contrôle des opérations (en particulier sous **Windows**).

Les principales fonctions mises à disposition par le module sont les suivantes :

`reset()`	On efface tout et on recommence
`goto(x, y)`	Aller à l'endroit de coordonnées x, y
`forward(distance)`	Avancer d'une distance donnée
`backward(distance)`	Reculer d'une distance donnée
`up()`	Relever le crayon (pour pouvoir avancer sans dessiner)
`down()`	Abaisser le crayon (pour recommencer à dessiner)
`color(couleur)`	`couleur` peut être une chaîne prédéfinie ('red', 'blue', etc.)
`left(angle)`	Tourner à gauche d'un angle donné (exprimé en degrés)
`right(angle)`	Tourner à droite
`width(épaisseur)`	Choisir l'épaisseur du tracé
`begin_fill()`	Commencer le remplissage d'une figure fermée
`end_fill()`	Terminer le remplissage d'une figure fermée
`speed(i)`	Gérer la vitesse de déplacement de la tortue (i prend des valeurs entières entre 1 et 10)
`circle(r)`	Tracer un cercle de rayon r
`hideturtle()`	Rendre la tortue invisible
`showturtle()`	Rendre la tortue à nouveau visible

Aussi longtemps que le crayon est posé sur la feuille, la tortue dessine une trace. Il est possible d'interrompre ce comportement avec la fonction `up()` qui va lever le crayon de la feuille et empêcher la tortue de dessiner. Il est ensuite possible de reposer le crayon sur la feuille avec `down()` de sorte qu'elle se remette à dessiner la trace.

La fonction `color()` permet de changer la couleur du crayon. Il est capital de placer le nom de la couleur en anglais et entre doubles guillemets : *yellow, orange, red, maroon, violet, purple, navy, blue, skyblue, cyan, turquoise, lightgreen, green, darkgreen, chocolate, brown, black, gray, white,...*

Le programme suivant ordonne à la tortue de dessiner une bougie composé d'un rectangle rouge surmonté d'un cercle jaune à l'intérieur duquel est placé un trait noir épais pour simuler la mèche.
Le remplissage d'une figure fermée est géré par les fonctions `begin_fill()` (au début du tracé) et `end_fill()` (à la fin du tracé).
Pour dessiner la flamme jaune, il suffit d'utiliser `circle()`. Il y a une partie du trajet durant laquelle la tortue ne dessine plus de trace puisque que le crayon a été levé avec `up()`. Dès qu'elle voit la fonction `down()`, elle se remet à dessiner.
La fonction `hideturtle()` permet de rendre la tortue invisible.

```
# Tracé du pied de la bougie
color('red')
begin_fill()
i = 0
while i <= 1:
    forward(30)
    left(90)
    forward(100)
    left(90)
    i += 1
end_fill()

# Déplacement vers la flamme
up()
forward(15)
left(90)
forward(110)
right(90)

# Tracé de la flamme
down()
color('yellow')
begin_fill()
circle(20)
end_fill()

# Déplacement vers le début de la mèche
```

```
left(90)
up()
backward(5)

# Tracé de la mèche
down()
color('black')
width(5)
forward(20)

hideturtle()
```

3 Définition de fonctions

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 nom_fonction(paramètres):
    "Documentation de la fonction"
    <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** que nous étudierons plus tard).
- Comme les instructions `if`, `while` et `for` 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** et qui décrit l'ensemble des opérations entreprises par la fonction.
- La **liste de paramètres** spécifie quelles informations il faudra fournir en guise d'**arguments** lorsque l'on voudra appeler cette fonction (les parenthèses peuvent parfaitement rester vides si la fonction ne nécessite pas d'arguments). 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, comme nous le verrons par la suite.
- La ligne d'en-tête de la définition de fonction peut être suivie d'une chaîne de caractères qui donne des informations nécessaires à l'utilisation de la fonction (types d'arguments à transmettre, descriptif de l'action entreprise, type de la valeur retournée). Cette *chaîne de documentation* est traitée par Python comme un simple commentaire mais est placée dans une variable spéciale dont le nom est **__doc__** et qui est associée à l'objet fonction comme étant l'un de ses attributs. Afin de pouvoir écrire cette chaîne de documentation sur plusieurs lignes, il convient de la délimiter à l'aide de triple guillemets.

3.1 Fonction sans paramètre ni valeur de retour

Pour notre première approche concrète des fonctions, nous allons travailler à nouveau en mode interactif :

```
>>> def table7():
        """
        table7() -> None.
        Affiche la liste des 10 premiers multiples de 7
        """
        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 éléments suivants :

- les parenthèses qui doivent toujours être utilisées, même si la fonction n'utilise aucun paramètre ;
- le double point ;
- la chaîne de documentation de la fonction (écrite entre triple guillemets afin de permettre une écriture sur plusieurs lignes) ;
- 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()
7 14 21 28 35 42 49 56 63 70
```

Pour obtenir la documentation de la fonction, il suffit d'afficher le contenu de la variable **__doc__ :**

```
>>> table7.__doc__
'table7() -> None. Affiche la liste des 10 premiers multiples de 7'
```

Le premier élément de la documentation informe que la fonction ne prend aucun argument et ne retourne rien. En fait, la valeur retournée par **table7** est l'objet **None** (objet particulier correspondant à "rien") :

```
>>> a = table7()              # affecter la valeur de retour de table7 à la variable a
7 14 21 28 35 42 49 56 63 70
>>> print(a)
None
```

Prenez la peine d'incorporer une chaîne explicative dans toutes vos définitions de fonctions futures.

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. Ainsi, dans l'exemple suivant, la fonction **table7** devient l'*employé* de la fonction-*patron* **table7triple**, évitant à cette dernière des redondances d'écriture au niveau de son code :

```
>>> def table7triple():
        """
        table7triple() -> None.
        Affiche 3 fois la liste des 10 premiers multiples de 7
        """
        print('La table par 7 en triple exemplaire :')
        table7()
        table7()
        table7()
```

Utilisons cette nouvelle fonction, en entrant la commande :

```
>>> 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
```

3.2 Fonction avec paramètres mais sans valeur de retour

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 monter quelque peu en abstraction et 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. L'expression **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):
        """
        table(int base) -> None.
        Affiche la liste des 10 premiers multiples de <base>
        """
        n = 1
        while n <11 :
            print(n * base, end =' ')
            n = n +1
```

La fonction `table` telle que définie ci-dessus utilise le paramètre entier `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 :

```
>>> 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`.

Notons que l'**argument que nous utilisons dans l'appel d'une fonction peut être une variable** lui aussi, comme dans l'exemple ci-dessous. 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
```

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.

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).

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. 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):
        "tableMulti(int base, int debut, int fin) -> None. Affiche la liste des \
multiples de <base> allant du terme d'indice <debut> à celui d'indice <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 (paramètre `base`), l'indice du premier terme à afficher (paramètre `début`), l'indice du dernier terme à afficher (paramètre `fin`) :

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

Finalement, notons que :

- 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 (séparés 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.

3.3 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 :**

```
>>> def tableMulti(base, debut=1, fin=6):
        """
        tableMulti(int base, int debut, int fin) -> None.
        Affiche la liste des multiples de <base> allant du terme d'indice <debut>
        à celui d'indice <fin>.
        """
        print('Fragment de la table de multiplication par', base, ':')
        n = debut
        while n <= fin :
            print(n, 'x', base, '=', n * base)
            n = n +1
```

Lorsqu'on appelle cette fonction en ne lui fournissant que le premier argument, le second et le troisième reçoivent tout de même les valeurs 1 et 6 par défaut. Si l'on fournit deux arguments sur trois, la valeur par défaut pour le deuxième argument est ignorée mais celle pour le troisième est conservée :

```
>>> tableMulti(8)
Fragment de la table de multiplication par 8 :
1 x 8 = 8
2 x 8 = 16
3 x 8 = 24
4 x 8 = 32
5 x 8 = 40
6 x 8 = 48

>>> tableMulti(5, 3)
Fragment de la table de multiplication par 5 :
3 x 5 = 15
4 x 5 = 20
5 x 5 = 25
6 x 5 = 30
```

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 tableMulti(base=7, debut=1, fin):
```

3.4 Arguments avec étiquettes

Dans la plupart des langages, les arguments fournis lors de l'appel d'une fonction doivent apparaître **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 :**

```
>>> def tableMulti(base=7, debut=1, fin=3):
        print('Fragment de la table de multiplication par', base, ':')
        n = debut
        while n <= fin :
            print(n, 'x', base, '=', n * base)
            n = n +1

>>> tableMulti(debut=4, base=5, fin=6)
Fragment de la table de multiplication par 5 :
4 x 5 = 20
5 x 5 = 25
6 x 5 = 30

>>> tableMulti()
Fragment de la table de multiplication par 7 :
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
```

3.5 Fonction avec valeur de retour

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**. 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`.

Voici 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 :

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

Remarquons que l'instruction `return` n'affiche pas le résultat calculé par la fonction mais le retourne à l'appelant. Cette instruction ne doit donc pas être confondue avec la fonction `print()` qui se contente d'afficher le résultat sans le retourner. L'exemple suivant permet de constater cette importante différence entre `return` et `print` :

```
>>> def cube(w):
        print(w*w*w)
>>> b = cube(9)
729
>>> print(b)
None
```

Puisque la fonction `cube` ci-dessus affiche le cube de l'argument qui lui est transmis sans rien retourner, l'instruction `b = cube(9)` ne fait qu'affecter la valeur `None` (symbolisant rien) à la variable `b`. En outre, comme cette affectation fait appel à la fonction `cube` qui utilise la fonction `print`, l'affectation engendre l'affichage du résultat de `9*9*9`, c'est-à-dire 729.

3.6 Fonction anonyme

Le langage Python permet de définir des **fonctions anonymes** (aussi appelées **fonctions-lambda**), c'est-à-dire des fonctions non forcément associées à un nom. Ces fonctions, limitées à une seule instruction, sont définies à l'aide du mot réservé `lambda` selon la syntaxe suivante :

```
lambda para1, para2,..., paraN : instruction de retour
```

La définition d'une fonction anonyme se distingue de la définition d'une fonction normale par quatre caractéristiques :

- le mot-clé `def` est remplacé par le mot réservé `lambda` ;
- la liste des paramètres de la fonction n'est pas entourée de parenthèses ;
- la valeur de retour n'est pas précédée du mot réservé `return` ;
- la définition ne peut se faire que sur une seule ligne ;

En appliquant la syntaxe ci-dessus, nous pouvons donc définir une fonction anonyme qui permet d'élever un nombre au carré de la manière suivante :

```
>>> lambda x : x**2
<function <lambda> at 0x10298c050>
```

Comme toute fonction anonyme, la fonction ci-dessus est définie sur une seule ligne. Sa définition commence par le mot-clé `lambda`, suivi de l'unique paramètre attendu puis d'un double-point qui vient séparé le paramètre de la fonction de l'opération qu'elle réalise. Remarquons que l'expression `lambda` renvoie un objet fonction, similaire à ceux que vous créez avec l'instruction `def`

L'appel de la fonction-lambda sur un argument peut dès lors se faire de deux manières différentes :

- L'appel suit directement la définition de la fonction :

  ```
  >>> (lambda x : x**2)(3)
  9
  ```

  Deux couples de parenthèses sont alors nécessaires pour distinguer la définition de la fonction de la liste de paramètres sur lesquels elle est appelée.

- La fonction est tout d'abord affectée à une variable, puis est appelée au travers de cette variable comme lors d'un appel traditionnel :

  ```
  >>> f=lambda x : x**2
  >>> f(3)
  9
  ```

Bien qu'il n'existe aucune opération faite par une fonction anonyme qui ne puisse pas être réalisée par une fonction standard, les fonctions anonymes sont néanmoins fréquemment utilisées pour créer des fonctions jetables, c'est-à-dire utilisables qu'une seule fois. En effet, leur principal avantage réside dans le fait qu'il est possible de les définir et de les appeler en une seule ligne de code ce qui, dans certains contextes que nous étudierons par la suite, s'avérera très utile.

Notons finalement que les fonctions-lambda peuvent s'avérer utiles pour créer des générateurs de fonctions :

```
>>> def increment(n): return lambda x: x+n
>>> f = increment(2)
>>> g = increment(6)
>>> print(f(42), g(42))
44 48
>>> print(increment(22)(23))
55
```

Le code ci-dessus commence par définir une fonction `increment` qui créer une fonction anonyme et la retourne. L'opération réalisée par cette fonction anonyme consiste à incrémenter son argument de la valeur qui lui a été associée lors de sa création par la fonction `increment`. Il est dès lors possible de créer différentes fonctions d'incrémentation et de les affecter à des variables puis de les utiliser indépendamment les unes des autres. D'autre part, il est également possible d'utiliser une fonction d'incrémentation sans lui assigner de nom comme le propose la dernière instruction de l'exemple ci-dessus.

4 Corps principal d'un programme

4.1 Structure d'un programme fonctionnel

Dans une première approche des fonctions, nous n'avons utilisé que le mode interactif de l'interpréteur Python. Il est bien évident que les fonctions peuvent aussi être utilisées dans des scripts. Dans ce cas, l'emplacement de leur définitions ne doit pas être aléatoire. 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 :

$$V = \frac{4}{3}\pi R^3$$

```
# Importation de fonctions ou variables nécessaires
from math import pi

# Définitions locales de fonctions

def cube(n):
    return n**3

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

# Corps principal du programme:

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

À bien y regarder, ce programme comporte deux parties : les définition des deux fonctions `cube` et `volumeSphere` suivies du 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`.

Notez bien que les deux 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. En d'autres termes, **la définition des fonctions doit précéder leur utilisation.**

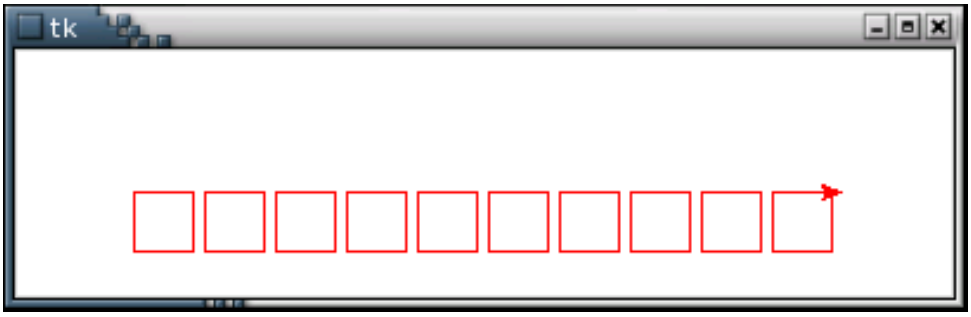
Pour vous en convaincre, intervertissez cet ordre (en plaçant par exemple le corps principal du programme au début), et prenez note du type de message d'erreur qui est affiché lorsque vous essayez d'exécuter le script ainsi modifié.

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 !

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.

Par exemple, imaginons qu'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):
    """
    carre(int taille, str couleur)-> None.
    Dessine un carré de taille et de couleur déterminées
    """

    color(couleur)
    c =0
    while c <4:
        forward(taille)
        right(90)
        c = c + 1
```

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
```

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.

4.2 Tests de fonction

Comme nous l'avons vu précédemment, les chaînes de documentation qui suivent la ligne d'en-tête de la définition d'une fonction et qui donnent des informations nécessaires à son utilisation sont traitées par Python comme de simples commentaires placés dans une variable spéciale, `__doc__`, associée à l'objet fonction. Ainsi, l'instruction suivante permet de récupérer la documentation de la fonction intégrée `len` :

```
>>> len.__doc__
'len(object) -> integer\n\nReturn the number of items of a sequence or mapping.'
```

La chaine de documentation d'une fonction peut être complétée de différents tests à exécuter avant l'utilisation de la fonction. Ces test permettront d'identifier rapidement les éventuelles erreurs de sémantique que contiendrait le code de la fonction créée. Afin de pouvoir recourir à cet outil puissant, une syntaxe particulière doit être utilisée dans la chaîne de documentation de la fonction :

- La ligne de test doit être précédé d'un triple chevron `>>>` ;
- La ligne suivant le test doit contenir le résultat escompté du test.

Lors de l'exécution du module contenant la définition de la fonction à tester, la fonction `testmod()` du module `doctest` exécutera les tests précédés de chevrons et vérifiera si le résultat produit concorde avec le résultat escompté, faute de quoi un message d'erreur sera retournée. Ainsi, l'exécution du module suivant, contenant une erreur à la ligne 12,

```
1  def cube(n):
2      """
3      cube(float n) --> float
4      Retourne le cube d'un nombre <n>
5
6      Test d'utilisation:
7
8      >>> cube(2)
9      8
10     """
11
12     return n*3
13
14 if __name__ == "__main__":
15     import doctest
16     doctest.testmod()
```

provoquera le message d'erreur suivant :

```
**********************************************************************
File "__main__", line 9, in __main__.cube
Failed example:
    cube(2)
Expected:
    8
Got:
    6
**********************************************************************
1 items had failures:
   1 of   1 in __main__.cube
***Test Failed*** 1 failures.
```

En cas de succès, aucun message d'erreur n'est retourné. Finalement, remarquons que l'instruction

```
if __name__ == "__main__":
```

placée à la fin du module permet d'exécuter les instructions qui suivent, à savoir l'appel de la fonction `testmod()` du module `doctest`, uniquement si le module est lancé en tant que programme autonome. Si au contraire, le module est utilisé comme une bibliothèque de fonctions importée par un autre programme, cette partie du code est sans effet.

5 Portée des variables

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 de la variable `x` de l'exemple suivant :

```
>>> def modif():
    x = 20
    x = x + 1
    print(x)
```

Chaque fois que la fonction `modif` est appelée, Python réserve pour elle (dans la mémoire de l'ordinateur) un nouvel **espace de noms**[3]. Le contenu de la variable `x`, réinitialisée à chaque appel de la fonction `modif`, est stocké 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 `x` juste après avoir appelé la fonction `modif`, nous obtenons un message d'erreur :

```
>>> modif()
21
>>> modif()
21
>>> print(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

La machine nous signale clairement que le symbole `x` lui est inconnu, alors qu'il était correctement imprimé par la fonction `modif` elle-même. L'espace de noms qui contient le symbole `x` est strictement réservé au fonctionnement interne de `modif`, 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 :**

```
>>> def modif2():
        x = 20
        print(x, y)

>>> x, y = 15, 38
>>> modif2()
20 38
>>> print(x, y)
15 38
```

Dans l'exemple précédent, nous commençons par définir une fonction très simple (qui n'utilise d'ailleurs aucun paramètre). À l'intérieur de cette fonction, une variable `x` est définie, avec `20` comme valeur initiale. Cette variable `x` 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 `x` et `y` 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 `x` 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 `modif2()` est lancée, la variable globale `y` y est accessible, puisqu'elle est imprimée correctement. Pour `x`, par contre, c'est la valeur attribuée localement qui est affichée.

On pourrait croire d'abord que la fonction `modif2()` a simplement modifié le contenu de la variable globale `x` (puisqu'elle est accessible). Les lignes suivantes démontrent qu'il n'en est rien : en dehors de la fonction `modif2()`, la variable globale `x` conserve sa valeur initiale.

3. 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.

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

Dans l'exemple ci-dessous, la variable `y` utilisée à l'intérieur de la fonction `modif3()` 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 `y` n'est plus incrémentée à chaque appel de la fonction.

```
>>> def modif3():
        global y
        x = 20
        y = y + 1
        print(x, y)

>>> y = 38
>>> modif3()
20 39
>>> modif3()
20 40
>>> print(y)
40
```

Les variables globales pouvant être utilisées partout dans un programme sans avoir été transmises préalablement en paramètres de fonction, leur utilisation simplifie la conception et l'écriture de programme.

De manière plus générale, en informatique, une fonction est dite **à effet de bord** si elle modifie autre chose que sa valeur de retour. Par exemple, une fonction peut modifier une variable globale ou l'un ou plusieurs de ses arguments. La fonction `modif3` ci-dessus est à effet de bord puisqu'elle modifie la variable globale `y`. Les effets de bord rendant souvent le comportement des programmes difficile à comprendre, ils sont **à éviter** dans la mesure du possible.

Pour éviter les effets de bords, une possibilité serait de passer les variables globales comme arguments de la fonction et les modifier. Malheureusement, en Python, les arguments ne peuvent être modifiés que s'ils sont d'un type[4] **mutable** :

Types	Mutables
Chaîne de caractères	Non
Valeur numérique	Non
Liste	Oui
Tuple	Non
Dictionnaire	Non

FIGURE 4.2 – Types mutables et non mutables

Comme `y` correspond à un entier, elle n'est pas mutable et ne pourra pas être modifié.

4. Les liste, tuples et dictionnaires seront étudiés dans le chapitre suivant.

L'effet de bords produit par la fonction `modif3` sur la variable globale `y` peut être éliminé en utilisant un paramètre et une valeur de retour dans la fonction `modif4` :

```
>>> def modif4(nombre):
    x = 20
    nombre += 1
    print(x, nombre)
    return nombre

>>> y = 38
>>> y = modif4(y)
20 39
>>> print(nombre)
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    print(nombre)
NameError: name 'nombre' is not defined
>>> print(y)
39
>>> y = modif4(y)
20 40
>>> print(y)
40
```

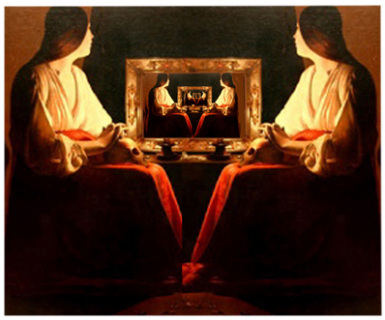
En plus d'afficher la valeur des variables locales `x` et `nombre`, cette fonction retourne la valeur de `nombre` qu'il faudra stocker dans la variable globale `y` permettant ainsi de lui faire changer de valeur. Comme la variable `nombre` est un argument de la fonction `modif4`, elle lui est locale et ne peut pas être appelée de l'extérieur, contrairement aux variables globales. Si d'aventure, il y avait dans le code des variables de même nom, que ce soit dans le programme principal ou dans d'autres fonctions, il s'agirait de variables différentes !

La durée de vie des variables locales est limitée. Elles n'existent que pendant l'exécution de la fonction dans laquelle elles se trouvent. Leur emplacement en mémoire n'est pas fixé à l'avance comme les variables globales mais un nouvel espace mémoire leur est alloué à chaque exécution de la fonction. Cela a pour conséquence qu'elles ne conservent pas leur contenu d'un exécution à l'autre du sous-programme.

6 Récursivité

Bon nombre d'algorithmes reposent sur la répétition - autant de fois que nécessaire - de tâches identiques. L'implémentation de tels algorithmes peut généralement se faire de deux manières différentes :

- de manière **itérative**, en recourant à une boucle permettant de répéter l'ensemble d'instructions similaires aussi longtemps qu'une condition est satisfaite. Ce type d'implémentation nécessite alors la définition d'un **compteur** comptant le nombre d'itérations de la boucle et d'un **accumulateur** gardant temporairement en mémoire le résultat des calculs couramment effectués ;
- de manière **récursive**, en définissant une fonction qui s'appelle elle-même sur des arguments plus simples que ceux invoqués initialement et ceci jusqu'à ce que le programme atteigne un **ancrage** qui autorise la fonction à ne plus faire appel à elle-même. En d'autres termes, la récursivité consiste à scinder successivement un problème en parties plus simples à résoudre jusqu'à atteindre un état soluble de manière triviale. Dans le domaine des arts, la récursivité est connue sous le terme de **mise en abyme** :



(a) Georges de la Tour Marie Madeleine

(b) Publicité restée célèbre

FIGURE 4.3 – Exemples de mise en abyme.

Illustrons ces deux manières d'implémenter la répétition en nous penchant sur le calcul de la *factorielle* d'un entier positif n, définie comme le produit des entiers strictement positifs, inférieurs ou égaux à n et notée $n!$:

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 \quad \text{pour } n \in \mathbb{N}^*.$$

La méthode itérative pour calculer $n!$ consiste à multiplier d'abord n par $(n-1)$, puis à multiplier ce résultat par $(n-2)$, puis par $(n-3)$ et ainsi de suite jusqu'à 1. Plus formellement, il suffit de mettre à jour le produit courant, mémorisé dans un accumulateur et d'utiliser un compteur balayant les entiers de n à 1. L'implémentation itérative en Python de la factorielle s'obtient dès lors à l'aide d'une boucle et d'une variable `resultat` permettant de mémoriser à chaque itération le résultat courant du calcul à effectuer :

```
def factorielle(n):
    "Calcule la factorielle de l'entier <n> de manière itérative."
    resultat = 1     # accumulateur
    while n > 0:     # computer
        resultat = resultat * n
        n = n - 1
    return resultat
```

Nous pouvons dès lors suivre le déroulement du processus de calcul de 5! à l'aide d'un tableau d'états :

Itération	`n`	`resultat`
0	5	1
1	4	5
2	3	20
3	2	60
4	1	120
5	0	120

FIGURE 4.4 – Tableau d'états de l'appel `factorielle(5)`

En analysant la liste des premières factorielles de n, nous constatons que cette fonction peut facilement être définie de manière récursive :

$1! = 1$
$2! = 2 \cdot 1 = 2 \cdot 1! = 2 \cdot 1 = 2$
$3! = 3 \cdot 2 \cdot 1 = 3 \cdot 2! = 3 \cdot 2 = 6$
$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 4 \cdot 3! = 4 \cdot 6 = 24$
$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 5 \cdot 4! = 5 \cdot 24 = 120$
$6! = 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 6 \cdot 5! = 6 \cdot 120 = 720$

Par conséquent, la factorielle de n répond à la formule de récurrence suivante :

$$\begin{cases} 1! &= 1, \\ n! &= n \cdot (n-1)! \quad \text{pour } n > 1. \end{cases}$$

En Python, cette relation de récurrence peut facilement être traduite en une fonction récursive illustrant les caractéristiques élémentaires de tout programme récursif : la fonction s'appelle elle-même avec, à chaque itération, une valeur inférieure de l'argument et arrête son appel récursif une fois l'ancrage atteint :

```
def factorielle(n):
    "Calcule la factorielle de l'entier <n> de manière récursive."
    if n == 1:      # ancrage
        return 1
    else:           # appel récursif
        return n * factorielle(n-1)
```

Nous pouvons suivre le déroulement du processus de calcul de 5! en explicitant chaque appel récursif, comme le suggère la figure ci-dessous :

```
                factorielle(5)
              5 * factorielle(4)
            5 * 4 * factorielle(3)
          5 * 4 * 3 * factorielle(2)
        5 * 4 * 3 * 2 * factorielle(1)
            5 * 4 * 3 * 2 * 1
              5 * 4 * 3 * 2
                5 * 4 * 6
                  5 * 24
                    120
```

FIGURE 4.5 – Déroulement récursif de l'appel `factorielle(5)`

En comparant les deux processus de calcul de la factorielle, nous constatons que dans sa version itérative, l'état de la fonction est entièrement déterminé par le compteur `n` et l'accumulateur `resultat` qui sont mis à jour à chaque itération de la boucle `while` selon une règle fixée d'avance. Au contraire, dans sa version récursive, le déroulement de l'appel à la fonction révèle une forme de **développement suivie d'une contraction** : le développement se produit lorsque le processus construit la chaîne de multiplications à réaliser et la contraction intervient une fois l'ancrage atteint et que chaque multiplication est finalement effectuée. Le déroulement nécessite donc que l'interpréteur Python garde en mémoire une trace des multiplications à réaliser plus tard.

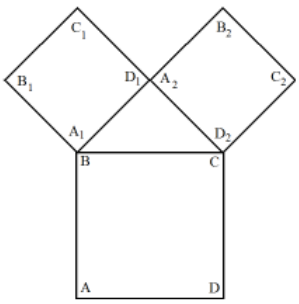
6.1 Dessins récursifs

Les algorithmes récursifs permettent de réaliser de magnifiques dessins comme celui proposé ci-dessous :

(1) En partant d'un point A, dessiner un carré $ABCD$ en s'assurant que sa base soit le côté AD.

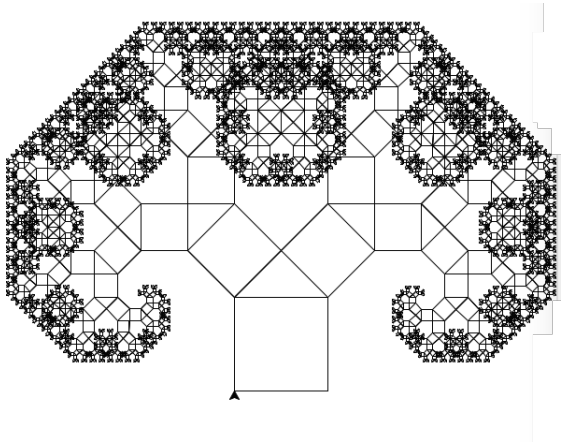
(2) Ajouter un triangle rectangle isocèle BD_1C au-dessus du côté BC du carré.

(3) Recommencer la procédure en prenant les cathètes du triangle rectangle isocèle comme bases des prochains carrés à dessiner.



Un tel algorithme permet d'obtenir un arbre de Pythagore :

```
1  from turtle import *
2  import math
3
4  def tree(s):
5      if s >= 2:
6          square(s)
7          forward(s)
8          s1 = s / math.sqrt(2)
9          left(45)
10         tree(s1)
11         right(90)
12         forward(s1)
13         tree(s1)
14         backward(s1)
15         left(45)
16         backward(s)
17
18 def square(s):
19     i = 0
20     while i < 4:
21         forward(s)
22         right(90)
23         i += 1
24
25 # Corps principal du programme:
26
27 left(90)
28 speed(-1)
29 tree(100)
```



Il est bien établi que l'implémentation d'un programme récursif est inhabituelle. De ce fait, voici une procédure servant à bien comprendre l'élaboration de ce programme :

- Définir une fonction `square(int s)` (lignes 18 à 23) qui ordonne à la tortue de dessiner un carré de côté s et qui s'assure qu'elle achève le dessin dans la même position et la même orientation qu'au début du dessin.
- Définir une fonction `tree(int s)` (lignes 4 à 16) permettant de dessiner un arbre en commençant par un carré dont la longueur du côté est s. Cette fonction peut s'appeler elle-même de manière récursive. Il est capital qu'après avoir dessiné l'arbre en question, la tortue se retrouve dans la même position et avec la même orientation que lorsqu'elle a débuté le dessin de cet arbre. Il est toujours judicieux dans ces cas de se mettre à la place de la tortue pour bien comprendre le programme. Les étapes de cette fonction sont mises en évidence ci-dessous :

  Ligne 6 : Commencer par dessiner le carré dont le côté mesure s en partant du point A.

  Lignes 7 à 10 : Se rendre au point B, tourner de 45° vers la gauche et considérer cette position et cette orientation comme le point de départ pour le dessin d'un prochain sous-arbre de longueur réduite s_1. D'après le théorème de Pythagore, on a $s_1 = \frac{s}{\sqrt{2}}$.

  Lignes 11 à 13 : Étant donné qu'après avoir dessiné un arbre, on se retrouve dans la position et l'orientation de départ, on se retrouve au point B avec une orientation de 45° vers la gauche (Nord-Ouest) dans la direction du sommet B_1.

  En se tournant de 90° et en avançant d'une distance s_1, la tortue se retrouve au point D_1, orientée vers B_2. Il s'agit de la condition initiale permettant de redessiner un arbre avec `tree(s1)`.

  Lignes 14 à 16 : Finalement, il est nécessaire de retourner au point initial A en adoptant l'orientation de départ. Pour ce faire, il suffit de reculer sur une distance s_1, de tourner de 45° vers la gauche et de reculer sur une distance s.

  Notre programme ne fonctionnerait cependant pas en l'état, car il ne traite pas le cas de base d'un arbre formé sur un carré de côté inférieur à 2 pixels. On peut facilement régler ce problème en forçant la fonction à ne rien faire lorsque le côté du carré est inférieur à 2 pixels (ligne 5).

6.2 Danger de la récursivité

Bien qu'elle soit souvent plus simple à implémenter que la méthode itérative, la méthode récursive s'avère parfois beaucoup plus gourmande en mémoire. Pour s'en convaincre, prenons la suite de Fibonacci définie par la relation de récurrence suivante :

$$\begin{cases} F_1 &= 1, \\ F_2 &= 1, \\ F_k &= F_{k-2} + F_{k-1}, \quad \text{pour } k \geq 3. \end{cases}$$

Chaque terme de cette suite correspond à la somme des deux précédents : $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

En Python, la relation de récurrence de cette suite peut facilement être traduite en une fonction récursive :

```
def fibRec(n):
    "Retourne le <n>-ème terme de la suite de Fibonacci de manière récursive."
    if n == 1 or n == 2:        # ancrage
        return 1
    else:                       # appel récursif
        return fibRec(n-2) + fibRec(n-1)
```

La version itérative de cette fonction nécessite l'utilisation d'accumulateurs permettant de garder en mémoire l'état du processus répétitif à chaque itération de la boucle de calcul :

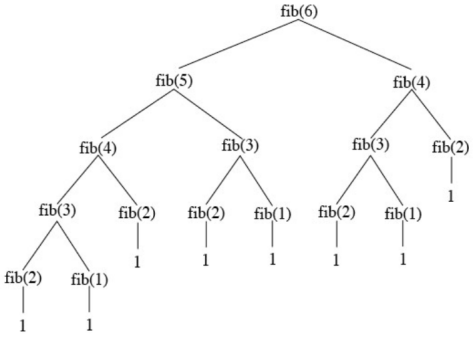
```
def fibIter(n):
    "Retourne le <n>-ème terme de la suite de Fibonacci de manière itérative."
    if n == 1 or n == 2:
        return 1
    else:
        i = 2           # position du cadre scannant la suite
        a = 1           # valeur du premier terme présent dans le cadre
        b = 1           # valeur du deuxième terme présent dans le cadre
        while i < n:
            a, b = b, a+b
            i = i + 1
        return b
```

Bien que les deux fonctions permettent de calculer le n-ème terme de la suite de Fibonacci, leurs temps de calcul diffèrent énormément comme l'illustre le tableau ci-dessous décrivant la durée du processus de calcul pour chacune des deux méthodes lancées avec un même processeur :

	Version récursive (`fibRec`)	Version itérative (`fibIter`)
n	Temps (en secondes)	Temps (en secondes)
10	$3.91 \cdot 10^{-5}$	$8.11 \cdot 10^{-6}$
20	$4.9 \cdot 10^{-3}$	$1.10 \cdot 10^{-5}$
30	0.49	$1.5 \cdot 10^{-5}$
40	60.61	$1.5 \cdot 10^{-5}$
50	7264	$1.5 \cdot 10^{-5}$

FIGURE 4.6 – Temps nécessaire au calcul du n-ème terme de la suite de Fibonacci

Cette différence s'explique par le fait que dans sa version récursive, la fonction `fibRec` fait un très grand nombre d'appels inutiles comme l'illustre le déroulement du calcul de `fibRec(6)` proposé dans l'arbre ci-contre.



Nous pouvons constater par exemple que `fibRec(3)` est appelé trois fois, engendrant une perte de temps. De plus, nous pouvons bien imaginer que l'arbre devient vite gigantesque, ce qui, déjà à moyen terme, augmente considérablement le temps de calcul.

6.3 Glossaire du chapitre

La liste suivante répertorie dans l'ordre alphabétique les principaux termes introduits dans ce chapitre. La connaissance de leur définition est indispensable à la suite de votre apprentissage :

appel de fonction : instruction constituée du nom de la fonction suivi, entre parenthèses, d'une liste d'arguments et qui permet d'exécuter l'action de la fonction.

argument avec étiquette : argument incluant le nom du paramètre correspondant.

argument : valeur fournie à une fonction lorsque cette dernière est appelée. Cette valeur est assignée au paramètre correspondant dans la définition de fonction.

chaîne de documentation (DocString) : chaîne de caractères situées dans la définition d'une fonction afin de la documenter. Une chaîne de documentation comprend généralement le type des paramètres de la fonction, l'action entreprise par la fonction et le type de la valeur de retour.

encapsulation : procédé consistant à placer une séquence d'instructions dans la définition d'une fonction.

fonction : groupe d'instructions regroupées sous un nom et s'exécutant à la demande.

fonction anonyme : expression d'une seule ligne utilisant le mot-clé `lambda` pour définir une fonction non affectée à une variable.

module : fichier contenant la définition d'un ensemble de fonctions, variables et classes d'objets réalisant des tâches communes.

paramètre : variable utilisée dans la définition d'une fonction pour référencer la valeur passée en argument lors de l'appel de la fonction.

procédure : fonction n'engendrant aucune valeur de retour.

récursivité : procédé informatique consistant à définir une fonction qui s'appelle elle-même sur des arguments plus simples que ceux invoqués initialement et ceci jusqu'à ce que le programme atteigne un ancrage qui autorise la fonction à ne plus faire appel à elle-même.

valeur de retour : valeur retournée lors de l'appel d'une fonction.

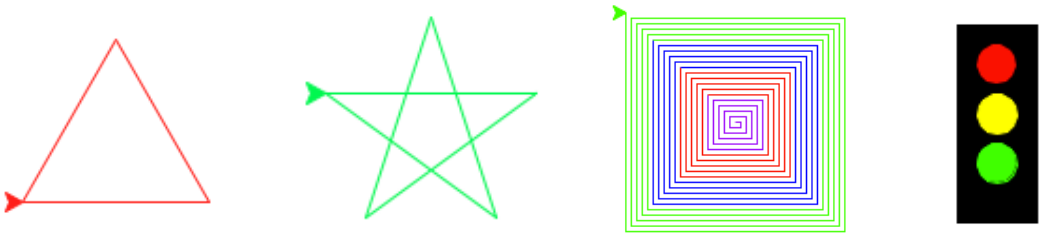
variable globale : variable définie à l'extérieur d'une fonction et accessibles dans tout le script y compris à l'intérieur des fonctions. Une variable globale n'est pas modifiable à l'intérieur d'une fonction sauf en la déclarant `global`.

variable locale : variable définie et accessible qu'à l'intérieur du corps d'une fonction. Une variable locale est redéfinie à chaque nouvel appel de la fonction.

7 Exercices

Exercice 4.1
Dans chacun des cas suivants, écrivez un programme reproduisant le dessin proposé à l'aide du module `turtle` :

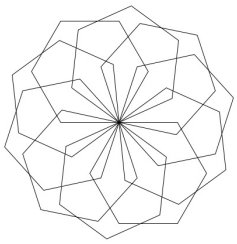
(1) (2) (3) (4)



Exercice 4.2
Définissez une fonction `hexagone() → None` capable de dessiner un hexagone de 50 pixels de côté.

Utilisez ensuite cette fonction pour définir une fonction `fleur() → None` capable de tracer une fleur à 10 pétales hexagonaux comme le propose la figure ci-contre.

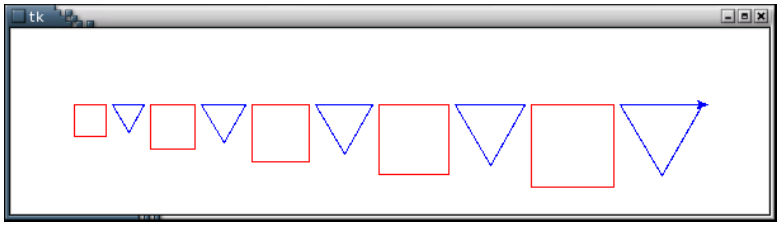
La partie principale du programme permettra de tracer 4 fleurs l'une à côté de l'autre.



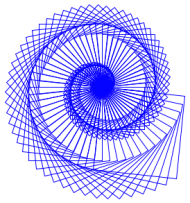
Exercice 4.3
Définissez une fonction `petale() → None` capable de dessiner un pétale de fleur rouge à l'aide de deux arcs de cercle de rayon 100 pixels et d'angle au centre de 90°. Utilisez ensuite cette fonction pour définir une fonction `fleur() → None` capable de dessiner une fleur à 5 pétales rouges.

La partie principale du programme permettra de tracer une fleur à partir de la fonction ci-dessus.

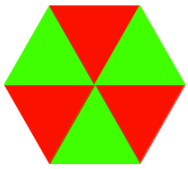
Exercice 4.4
Définissez un module `dessins_turtle` contenant une fonction `carre(int taille, str couleur) → None` et une fonction `triangle(int taille, str couleur) → None` capables de dessiner un carré, respectivement un triangle, de côté et de couleur déterminée. La partie principale du programme fera appel à ces fonctions à plusieurs reprises avec des arguments variés pour dessiner une série de carrés bleus et de triangles rouges :



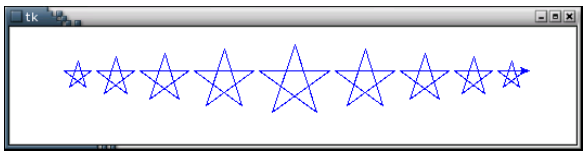
Exercice 4.5
A partir du module de l'exercice 4.4, définissez une fonction `spirale() --> None` permettant de dessiner des carrés avec des côtés croissants et tournés de 6° jusqu'à ce que la longueur du côté soit 120.



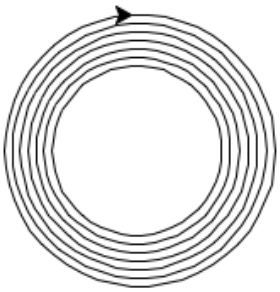
Exercice 4.6
A partir du module de l'exercice 4.4, définissez une fonction `hexagone(float cote) --> None` permettant de dessiner un hexagone dont la longueur du côté est choisi par l'utilisateur et qui est composé de triangles équilatéraux de 2 couleurs différents.



Exercice 4.7
Ajoutez au module de l'exercice précédent une fonction `etoile5(float taille, str couleur) → None` spécialisée dans le dessin d'étoiles à 5 branches. La partie principale du programme fera appel à cette fonction pour dessiner une rangée horizontale de 9 étoiles de tailles variées :



Exercice 4.8
Ajoutez au module des exercices précédents une fonction `polygone(int n, int R) → None` capable de dessiner un polygone régulier à `n` côtés inscrit dans un cercle de rayon `R` donné. A partir de cette fonction, définissez une fonction `cercle(int R) → None` permettant de dessiner un cercle de rayon `R` donné. L'approximation du cercle se fera à l'aide d'un polygone à 50 côtés de périmètre égal à celui du cercle à tracer. La partie principale du programme fera appel à cette fonction pour dessiner une série de 7 cercles de tailles variées comme proposé ci-contre.



Exercice 4.9
Définissez une fonction `surfDisque(float R) → float` qui renvoie l'aire d'un disque dont le rayon `R` est fourni en arguments. La partie principale du programme demandera à l'utilisateur d'entrer la valeur du rayon d'un disque et lui retournera l'aire de ce dernier.

Exercice 4.10
Définissez une fonction `compteCar(str car, str ch) → int` qui renvoie le nombre de fois que l'on rencontre le caractère `car` dans la chaîne de caractères `ch`. La partie principale du programme demandera à l'utilisateur d'entrer une phrase ainsi qu'un caractère et retournera le nombre d'occurrences du caractère entré dans la phrase en question.

Exercice 4.11
Définissez une fonction `comptePattern(str pattern, str ch) → int` qui renvoie le nombre de fois que l'on rencontre la sous-chaîne de caractères `pattern` dans la chaîne de caractères `ch`. La partie principale du programme demandera à l'utilisateur d'entrer une phrase ainsi qu'une sous-chaîne de caractères et retournera le nombre d'occurrences de la sous-chaîne entrée dans la phrase en question.

Exercice 4.12
Définissez une fonction `estVoyelle(str car) → bool` qui renvoie `True` ou `False` selon que le caractère `car` entré est une voyelle ou non. Définissez ensuite une fonction `compteVoyelle(str ch) → int` qui retourne le nombre de voyelles contenues dans la chaîne de caractères `ch`. La partie principale du programme demandera à l'utilisateur d'entrer une phrase et lui retournera le nombre de voyelles contenues dans celle-ci.

Exercice 4.13
Définissez une fonction `majuscule(str ch) → str` qui renvoie la chaîne de caractères `ch` en majuscules après avoir enlevé les éventuels accents. La partie principale du programme demandera à l'utilisateur d'entrer une phrase et retournera cette dernière écrite en majuscules.

Exercice 4.14
Définissez une fonction `maximum(obj1, obj2, obj3) → obj` qui renvoie le maximum des objets `obj1`, `obj2` et `obj3` qu'il s'agisse d'entiers, de nombres réels ou de chaîne de caractères. La partie principale du programme demandera à l'utilisateur d'entrer trois objets de même type, de préciser leur type et retournera le maximum des trois objets.

Exercice 4.15
Définissez une fonction `estPremier(int n) → bool` qui renvoie `True` ou `False` selon que l'entier naturel `n` entré est un nombre premier ou non. La partie principale du programme affichera tous les nombres premiers situés entre 2 et 1000.

Exercice 4.16
Définissez une fonction `volBoite(float x1, float x2, float x3) → float` qui renvoie le volume d'un parallélépipède rectangle dont sont fournis les trois dimensions `x1, x2` et `x3` en arguments. La définition de la fonction doit être telle que la fonction puisse être appelée avec trois, deux, un seul, ou même aucun argument. Utilisez pour ceci des valeurs par défaut égales à 10 comme le proposent les exemples suivants :

```
>>> print(volBoite())
1000
>>> print(volBoite(5.2))
520.0
>>> print(volBoite(5.2,3))
156.0
```

Exercice 4.17
Modifiez la fonction `volBoite(x1, x2, x3)` ci-dessus de manière à ce qu'elle puisse être appelée avec un seul, deux ou trois arguments. Si un seul est utilisé, la boîte est considérée comme cubique, l'argument étant l'arête du cube. Si deux sont utilisés, la boîte est considérée comme un prisme à base carrée, le premier argument faisant office de côté du carré alors que le second joue le rôle de hauteur du prisme. Si trois argument sont utilisés, la boîte est considérée comme un parallélépipède rectangle. Ainsi, par exemple :

```
>>> print(volBoite())
-1  # indication d'une erreur
>>> print(volBoite(5.2))
140.608
>>> print(volBoite(5.2, 3))
81.12
>>> print(volBoite(5.2, 3, 7.4))
115.44
```

Exercice 4.18
Définissez la fonction `volBoite` de l'exercice 4.16 sous la forme d'une fonction-lambda.

Exercice 4.19
Définissez une fonction `changeCar(str ch, str car1, str car2, int debut, int fin) → str` qui remplace tous les caractères `car1` par des caractères `car2` 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 traîtée d'une extrémité à l'autre). Ainsi, par exemple :

```
>>> 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.
```

Exercice 4.20
Définissez une fonction `inverse(str chaine) → str` retournant l'inverse de la chaîne de caractère entrée en argument en recourant à un procédé ...

(1) ... itératif. (2) ... récursif.

La partie principale du programme déterminera si un mot entré par l'utilisateur est un palindrome.

Exercice 4.21
L'algorithme d'Euclide permet de calculer le plus grand diviseur commun (PGDC) de deux entiers positifs a et b (avec $a > b$) en se basant sur l'observation suivante : si r est le reste de la division entière de a par b, alors les diviseurs communs de a et b sont exactement les mêmes que ceux de b et r. En d'autres termes,

$$PGDC(a, 0) = a \quad \text{et} \quad PGDC(a, b) = PGDC(b, r), \text{ où } r \text{ est le reste de la division entière de } a \text{ par } b.$$

Définissez une fonction `pgdc(int a, int b) → int` permettant de calculer le PGDC des entiers a et b selon l'algorithme d'Euclide ...

(1) ... de manière itérative. (2) ... de manière récursive.

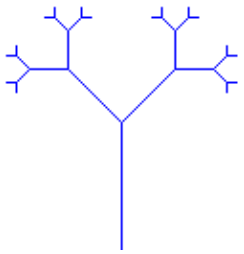
Exercice 4.22
La suite de Fibonacci $(F_n)_{n\in\mathbb{N}^*}$ est constituée d'entiers positifs tels que chacun est la somme des deux précédents. En d'autres termes, la suite $(F_n)_{n\in\mathbb{N}^*}$ est définie par la relation de récurrence suivante :

$$F_1 = 1, F_2 = 1 \text{ et } F_k = F_{k-2} + F_{k-1}, \quad \text{pour } k \geq 3.$$

Définissez une fonction `Fibonacci(int n) → int` qui calcule le `n`-ème terme de la suite de Fibonacci ...

(1) ... de manière itérative. (2) ... de manière récursive.

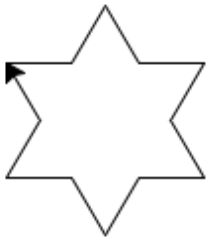
Exercice 4.23
L'arbre binaire complet est un graphe bien connu. La figure ci-contre représente un arbre binaire de profondeur 4. Le tronc unique est rajouté en plus et n'en fait pas partie à strictement parler. Définissez une fonction récursive `tree(int s) --> None` qui dessine un arbre binaire complet (sans le tronc qu'il faut rajouter) de profondeur s.

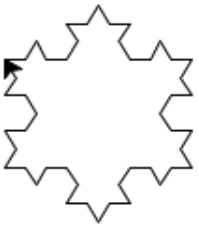


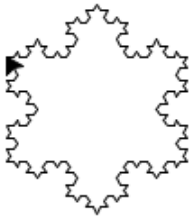
Exercice 4.24
Le *flocon de von Koch* est une courbe fractale construite de la manière suivante :

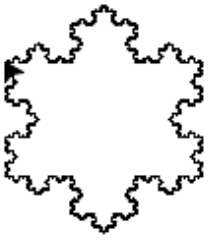
(1) Tracer un triangle équilatéral ;
(2) Remplacer le tiers central de chaque côté par une pointe dont la longueur de chaque côté est aussi égale au tiers du côté ;
(3) Recommencer cette construction sur chaque côté des triangles ainsi formés.

En itérant plusieurs fois l'étape (3) ci-dessus, nous obtenons différentes figures possibles du *flocon de von Koch* :


1 itération


2 itérations


3 itérations


5 itérations

A l'aide du module `turtle`, définissez une fonction `flocon(int l, int n) → None` permettant de dessiner un flocon de von Koch basé sur un triangle équilatéral de côté `l` et construit en itérant `n`-fois l'étape (3) ci-dessus.