5

Fonctions

Dans ce chapitre, nous verrons qu'on peut définir une fonction pour isoler une instruction qui revient plusieurs fois dans un programme. Une fonction est définie par :

- *son nom ;*
- *ses arguments qui porteront les valeurs communiquées par le programme principal à la fonction au moment de son appel ;*
- *éventuellement une valeur de retour communiquée au programme par la fonction en fin d'exécution.*

On souhaite écrire un programme qui affiche de la manière suivante les horaires de trois trains pour Brest, Strasbourg et Toulouse :

```
Le train en direction de Brest partira à 9h
------------------------------------------------------------
Le train en direction de Strasbourg partira à 9h30
------------------------------------------------------------
Le train en direction de Toulouse partira à 9h45
------------------------------------------------------------
```

On peut écrire pour cela le programme suivant :

```
print("Le train en direction de", "Brest", "partira à", "9h")
print("------------------------------------------------------------")
print("Le train en direction de", "Strasbourg", "partira à", "9h30")
print("------------------------------------------------------------")
print("Le train en direction de", "Toulouse", "partira à", "9h45")
print("------------------------------------------------------------")
```

Dans ce programme, on fait trois fois la même chose : on affiche la chaîne `"Le train en direction de"`, puis la destination, puis la chaîne `"partira à"`, puis l'horaire et enfin la ligne de séparation. Répéter plusieurs fois la même chose dans un programme n'est pas souhaitable. En particulier, si on procède par *copier-coller*, alors toute erreur sera dupliquée autant de fois. Si par exemple on fait une faute d'orthographe à `"partira"`, alors il faudra la corriger trois fois. De même, si on souhaite modifier le comportement du programme, par exemple pour afficher des étoiles à la place des tirets, alors il faudra effectuer la modification à trois endroits dans le code. Ici, une solution pour éviter la duplication consiste à utiliser une boucle **`for`** qui parcourt une liste de paires destination-horaire :

```
for (d, h) in [("Brest", "9h"), ("Strasbourg", "9h30"), ("Toulouse", "9h45")]:
    print("Le train en direction de", d, "partira à", h)
    print("------------------------------------------------------------")
```

Ainsi, pour chaque couple d'une destination *d* et d'un horaire *h*, on effectue les deux instructions suivantes :

```
    print("Le train en direction de", d, "partira à", h)
    print("------------------------------------------------------------")
```

Cela ne suffit pas nécessairement à éliminer toute redondance. En effet, on peut imaginer avoir besoin d'afficher un départ de train à un autre endroit du programme. Il faudrait alors copier les deux instructions, avec les valeurs de *d* et *h* pertinentes. Ce qu'il nous faut donc, c'est un moyen d'*isoler* ces deux instructions. C'est exactement ce que permet de faire une *fonction*.

5.1 La notion de fonction

Une fonction est une suite d'instructions qui dépend de paramètres. Ainsi, dans l'exemple précédent, on peut définir une fonction appelée **annoncer_train** et paramétrée par une destination *d* et un horaire *h* de la façon suivante :

```
def annoncer_train(d, h):
    print("Le train en direction de", d, "partira à", h)
    print("-----------------------------------------------------------")
```

Comme on le voit sur cet exemple, une fonction est introduite par le mot-clé **def**, suivi du nom de la fonction, de ses paramètres entre parenthèses, de deux-points, puis d'un bloc d'instructions. Ce dernier s'appelle le *corps* de la fonction. Il doit donc être indenté et la fin de ce bloc marque la fin de la définition de fonction. Les variables *d* et *h*, qui figurent comme paramètres dans la définition de la fonction, s'appellent les *arguments formels*.

En utilisant la fonction **annoncer_train**, le programme se réduit alors aux trois instructions suivantes :

```
annoncer_train("Brest", "9h")
annoncer_train("Strasbourg", "9h30")
annoncer_train("Toulouse", "9h45")
```

Chaque instruction `annoncer_train(...)` est un *appel* à la fonction. Dans l'appel `annoncer_train("Brest", "9h")`, les expressions `"Brest"` et `"9h"`, que l'on donne en arguments, s'appellent les arguments *effectifs* de l'appel.

Organiser un programme à l'aide de fonctions évite les redondances. En outre, cela les rend plus clairs et plus faciles à lire : pour comprendre le programme précédent, il n'est pas nécessaire de savoir comment la fonction **annoncer_train** est programmée, il suffit de savoir ce qu'elle fait. Enfin, cela permet d'organiser l'écriture du programme. On peut décider d'écrire la fonction un jour et le programme principal le lendemain. On peut aussi organiser une équipe de manière à ce qu'un programmeur écrive la fonction et un autre le programme principal.

Ce mécanisme peut se comparer à celui des définitions du langage mathématique qui permet d'utiliser le mot « groupe » au lieu de répéter la locution « ensemble muni d'une loi interne, associative, ayant un élément neutre, dans lequel tout élément a un symétrique ». Pour poursuivre la comparaison, le langage mathématique utilise aussi de telles définitions paramétrées : « Une fonction de classe $\mathcal{C}^n$ est... », « Un $\mathbb{K}$-espace vectoriel est... », etc.

5.1.1 Le retour de valeur

Certaines fonctions sont capables d'effectuer un *calcul*, et non pas seulement un affichage comme dans **annoncer_train**. Par exemple, on peut souhaiter écrire une fonction **hypotenuse**, avec deux paramètres x et y, pour calculer $\sqrt{x^2 + y^2}$. En Python, on la définit de la manière suivante :

```
def hypotenuse(x, y):
    z = math.sqrt(x*x + y*y)
    return z
```

On l'utilise, par exemple pour calculer $\sqrt{3^2 + 4^2}$, de la manière suivante :

```
In [1]: hypotenuse(3.0, 4.0)
Out[1]: 5.0
```

Dans cette instruction, l'appel **`hypotenuse`**`(3.0, 4.0)` transmet les arguments effectifs `3.0` et `4.0` à la fonction, en lieu et place des arguments formels *x* et *y*. La fonction **`hypotenuse`** calcule alors la valeur **`math.sqrt`**`(x*x + y*y)` et la renvoie. En Python, ce retour de valeur est effectué par l'instruction **`return`**. C'est le processus inverse du passage d'arguments, qui transmet des informations du programme principal vers le corps de la fonction. L'appel à la fonction **`hypotenuse`** constitue alors une expression, dont la valeur est la valeur renvoyée, à savoir `5.0`. Celle-ci est affichée par l'instruction **`print`**.

Si on oublie de faire précéder l'expression renvoyée par le mot-clé **`return`**, comme dans la fonction **`hypotenuse`** suivante :

```
def hypotenuse(x, y):
    math.sqrt(x*x + y*y)
```

celle-ci est toujours acceptée, mais elle ne renvoie pas de valeur. Ainsi, si on affiche le résultat du calcul de cette fonction

```
In [2]: hypotenuse(3.0, 4.0)
Out[2]: None
```

on obtient, non pas la valeur attendue `5.0`, mais une valeur particulière **`None`**. De manière générale, l'absence d'instruction **`return`** dans une fonction se traduit par une instruction implicite **`return None`**. C'était en particulier le cas pour la fonction **`annoncer_train`** précédente. Par opposition aux fonctions qui renvoient des valeurs, on pourra en général considérer l'appel à une fonction sans **`return`** (ou qui renvoie **`None`**) comme une instruction. On verra cependant dans la prochaine section que certaines fonctions sont à la fois des expressions et des instructions.

ATTENTION **Ne pas confondre `return` et `print`**

Il arrive que l'on confonde ces deux intructions à cause de l'interpréteur interactif, qui affiche la valeur renvoyée par **`f`** lorsqu'on y saisit **`f`**`(...)`. Leurs rôles sont en réalité totalement différents :

- L'instruction **`print`** n'a pas de valeur, elle a pour seul effet d'afficher un texte à l'écran.
- L'instruction **`return`**, au contraire, n'affiche rien mais décide de ce que renvoie la fonction, et donc de la valeur de l'appel **`f`**`(...)`.

Ainsi, si on commet l'erreur d'utiliser **`print`** à la place de **`return`** dans une fonction, celle-ci affichera la valeur calculée à l'écran mais ne la renverra pas (elle renverra **`None`** comme on l'a vu plus haut).

➥

De manière générale, il est important dans un algorithme de distinguer le résultat calculé des sorties affichées à l'écran. De même, il faut distinguer les données sur lesquelles on travaille des entrées tapées au clavier.

Quand elle ne se trouve pas à la fin d'une fonction, mais au milieu, l'instruction **return** a pour effet d'interrompre le déroulement de la fonction. Ainsi, on peut écrire :

```
def puissance(x, n):
    if (x == 0):
        return 0
    r = 1
    for i in range(n):
        r = r * x
    return r
```

Si la valeur de *x* est nulle, l'instruction **return** 0 interrompra le déroulement de la fonction et aucune des instructions suivantes (à partir de r = 1) ne sera exécutée.

SAVOIR-FAIRE Concevoir une fonction

Quand on conçoit une fonction, il est préférable de lui donner un nom explicite, car elle est susceptible d'apparaître à de nombreux endroits dans le programme (On n'imagine pas que les fonctions de la bibliothèque Python s'appellent **f1**, **f2**, etc.). En revanche, les noms des arguments formels peuvent être courts, car leur portée, et donc leur signification, est limitée au corps de la fonction.

Il convient par ailleurs de documenter convenablement les fonctions, en *spécifiant* les hypothèses faites sur les arguments formels, leur relation avec le résultat renvoyé, mais aussi les effets de la fonction (affichage, etc.) le cas échéant. Python propose un mécanisme pour associer une documentation à toute fonction, sous la forme d'une chaîne de caractères placée au début du corps de la fonction. Ainsi, on peut écrire :

```
def puissance(x, n):
    """calcule x à la puissance n, en supposant x > 0 et n >= 0"""
    r = 1
    for i in range(n):
        r = r * x
    return r
```

La documentation d'une fonction **f** peut être affichée avec **help(f)**.

Exercice 5.1 Écrire une fonction qui prend en arguments deux entiers x et y et qui renvoie -1 si $x < y$, 0 si $x = y$ et 1 si $x > y$.

Exercice 5.2 avec corrigé Écrire une fonction qui prend en arguments deux entiers $n \geqslant 0$ et $d > 0$ et qui renvoie un couple formé du quotient et du reste de la division euclidienne de n par d. Le quotient et le reste seront calculés par soustractions successives.

Le nom à donner à la fonction, à ses arguments et à ses variables sont pour ainsi dire donnés dans l'énoncé. On n'oublie pas de mentionner les conditions d'utilisation dans la documentation.

```
def division_euclidienne(n, d):
    """calcule q et r tels que n = q * d + r et 0 <= r < d,
    en supposant n >= 0 et d > 0
    """
    r = n
    q = 0
    while r > d:
        r = r - d
        q = q + 1
    return (q, r)
```

SAVOIR-FAIRE Utiliser un débogueur avec des fonctions

On a vu jusqu'ici que, en mode débogueur, on pouvait exécuter un programme instruction par instruction à l'aide de la commande *Pas en avant* (ou *next*). Lorsque le programme comporte des fonctions, cette commande considère chaque appel de fonction comme une instruction unique et passe donc directement à l'instruction suivante. Ce n'est pas approprié si on veut voir ce qui se passe dans le corps de la fonction.

Pour entrer dans le corps d'une fonction au moment où elle est appelée, on clique sur *Pas vers l'intérieur* ou on tape *s* comme *step*. Le curseur qui indique la prochaine ligne à exécuter se déplace sur la première ligne du corps de la fonction et on peut ensuite suivre son déroulement instruction par instruction. Les variables utilisées dans le corps de la fonction s'affichent alors également dans l'explorateur de variables.

À l'inverse, si le débogueur est entré dans une fonction et si on souhaite aller directement à la fin de l'exécution de cette fonction, au moment du retour de valeur, on clique sur *Pas vers l'extérieur* ou on tape *r* comme **return**.

Exercice 5.3 avec corrigé On donne le programme suivant :

```
def f(x):
    for i in range(100):
        x = (13 * x + 1) % 256
    return x

def g():
    s = 0.
    for j in range(10):
        a = f(j)
        s = s + 1. / (a-210)
    return s

print(g())
```

1 Que se passe-t-il lors de son exécution ?
2 Jusqu'à quel point la simple lecture du code permet-elle d'expliquer ce comportement ?
3 À quelle itération de la boucle de la fonction **g** se situe le problème ?

1 *Le programme s'arrête sur l'erreur suivante :*

```
ZeroDivisionError: 'float division by zero'
```

2 *Il est clair que la division par zéro provient de la ligne* `s = s + 1. / (a-210)` *dans la fonction* `g`*; en revanche, la lecture seule du code ne nous permet pas de déterminer quand la variable* `a` *prend la valeur* 210.

3 *Si l'on n'utilise que la commande* Pas en avant*, le débogueur n'entre pas dans l'appel à la fonction* `g` *et on ne voit pas ce qui se passe.*
À la ligne `print(g())`*, il faut donc utiliser* Pas vers l'intérieur. *Ensuite, on peut effectuer* Pas en avant *dans le corps de* `g` *jusqu'à déterminer que la division par zéro a lieu à l'itération* `j=6`*. Si par erreur on continue à utiliser* Pas vers l'intérieur*, on peut se retrouver dans le corps de* `f`*. Pour ne pas avoir à exécuter les* 100 *itérations à la main, on peut effectuer* Pas vers l'extérieur.
Remarquons que placer un point d'arrêt à la ligne qui produit la division par zéro évite de dérouler toutes les itérations dans `g` *à la main.*

POUR ALLER PLUS LOIN **Un générateur pseudo-aléatoire**

Le lecteur qui aura eu la curiosité de dérouler une exécution de la fonction `f` dans l'exercice corrigé ci-avant aura pu remarquer le caractère apparemment imprévisible des valeurs successives prises par la variable `x` (apparemment seulement puisqu'il s'agit d'une suite périodique). Ce sont des procédures telles que celle-ci qui permettent aux ordinateurs de simuler le hasard. On parle de générateur *pseudo-aléatoire*.

5.1.2 Variables globales et locales

En Python, on distingue deux sortes de variables : les *globales* et les *locales*. Par exemple, dans le programme suivant, `x` est une variable globale :

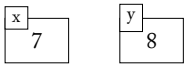
```
x = 7
print(x)
```

À l'inverse, la variable `y` dans la fonction `f` suivante est locale :

```
def f():
    y = 8
    return y
```

Ainsi, avant un appel à `f()`, on se trouve dans l'état :

Pendant l'appel, et après avoir exécuté `y = 8`, on se trouve dans un état augmenté d'une variable locale `y`, c'est-à-dire :



Après l'appel, la variable locale `y` disparaît et on se retrouve dans l'état initial :

En particulier, l'instruction suivante échoue en indiquant que la variable *y* n'est pas définie :

```
print(y)
```

On dit que la *portée* de la variable *y* est limitée au corps de la fonction **f**. Les variables globales, elles, ont une portée qui s'étend généralement sur l'ensemble du programme.

Si on souhaite faire référence à une variable globale dans une fonction, par exemple pour écrire une fonction qui réinitialise la variable globale *x* à 0, alors il ne faut pas écrire :

```
def reinitialise():
    x = 0
```

En effet, cela ne fait qu'affecter une variable locale *x* à la fonction **reinitialise**. On peut s'en convaincre en exécutant le code suivant :

```
reinitialise()
print(x)
```

et en observant qu'il affiche toujours la valeur 7. Pour que la fonction **reinitialise** puisse avoir accès à la variable globale *x*, il faut désigner cette dernière comme telle :

```
def reinitialise():
    global x
    x = 0
```

Ainsi, le programme :

```
reinitialise()
print(x)
```

affiche maintenant 0 et non 7.

De manière générale, si elle n'y est pas explicitement déclarée comme globale, une variable *x* est locale à la fonction dans le corps de laquelle elle est affectée. Elle est globale si elle est utilisée dans la fonction sans être affectée ou si elle est déclarée globale.

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

Dans cette fonction, *a* est globale car elle est déclarée comme telle, *b* est globale car elle est utilisée mais non affectée et *c* est locale car elle est affectée mais n'est pas déclarée globale.

Si une variable *x* est déclarée globale dans une fonction **f** mais pas dans une fonction **g**, et si le nom de variable *x* est utilisé dans **g**, comme dans l'exemple suivant :

```
def f():
    global x
    x = 2

def g():
    x = 3
```

alors il faut considérer qu'on a deux variables *x* différentes : une globale et une locale dans `g`. Dans la fonction `g`, le nom *x* désigne la variable locale et non la globale, qui du fait de son homonymie, ne peut être utilisée. C'est le seul cas où une variable globale a une portée qui ne couvre pas l'ensemble du programme. Ainsi, le programme suivant affiche `2` car l'affectation `x = 3` concerne la variable *x* locale à `g` et non la variable globale *x* :

```
x = 1
f()
g()
print(x)
```

Ainsi, un appel à une fonction peut modifier l'état du programme principal. C'est évidemment possible aussi avec une fonction qui renvoie une valeur : un appel à une telle fonction est donc à la fois une expression (puisqu'il prend une valeur) et une instruction (puisqu'il modifie l'état).

SAVOIR-FAIRE **Du bon usage des variables globales**

De façon générale, une bonne pratique consiste à utiliser les variables globales pour représenter les *constantes* du problème. En pratique, on ne devrait pas recourir souvent à la construction `global` de Python.

Comme pour les fonctions, il est préférable de donner aux variables globales des noms longs et explicites, ce qui les distinguera de fait des variables locales qui portent habituellement des noms courts (comme les paramètres formels).

Exercice 5.4 Quelles sont les variables locales et globales de la fonction `f` ? Qu'affiche le programme suivant ?

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

a = 3
b = 4
c = 5
print(f())
print(a)
print(b)
print(c)
```

5.1.3 Ordre d'évaluation

Comme les expressions peuvent modifier la mémoire, l'ordre dans lequel les arguments d'une fonction sont évalués peut avoir une influence sur le résultat. Ainsi, le programme

suivant affiche le résultat 2 si les arguments de **somme** sont évalués de gauche à droite et 3 sinon :

```
n = 0
def g(x):
    global n
    n = n + 1
    return x + n
def somme(x, y):
    return x + y
print(somme(n, g(1)))
```

Il se trouve qu'en Python, l'ordre d'évaluation est toujours de la gauche vers la droite. Ce n'est pas le cas dans d'autres langages de programmation, où il peut être au contraire de la droite vers la gauche, voire non spécifié. Même en ne considérant que Python, rien ne garantit que l'ordre d'évaluation ne changera pas dans de prochaines versions. Aussi est-il important de ne jamais écrire un programme dont le résultat en dépende. Dans l'exemple précédent, on peut par exemple forcer `g(1)` à être évaluée en premier, en stockant sa valeur dans une variable `a` :

```
a = g(1)
print(somme(n, a))
```

Exercice 5.5 Donner un exemple de programme montrant que l'évaluation des arguments de l'opérateur + se fait également de gauche à droite.

Exercice 5.6 Qu'affiche le programme suivant ? Pourquoi ?

```
def f(x):
    global b
    b = 10
    return x + 1

a = 3
b = 4
print(f(a)+b)
```

Proposer une adaptation de ce programme dans laquelle le résultat affiché ne dépend pas de l'ordre d'évaluation.

5.1.4 Passage par valeur

On considère la fonction **f** suivante, qui incrémente son argument *x* et en renvoie ensuite la valeur :

```
def f(x):
    x = x + 1
    return x
```

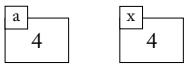
Alors le programme :

```
a = 4
print(f(a))
```

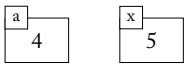
affiche sans surprise la valeur 5.

De manière plus surprenante, la variable `a` contient toujours la valeur `4` après l'appel à `print(f(a))`. En effet, on part de l'état suivant :

Immédiatement après l'appel à `f(a)`, on se retrouve dans un état :



Une nouvelle variable *x* a reçu la *valeur* de `a`. Après l'instruction `x = x + 1`, on se retrouve dans l'état :



Seule *x* a été modifiée. La fonction `f` renvoie alors la valeur de *x*, soit `5`, puis *x* disparaît et on se retrouve dans l'état de départ :

De ce point de vue, il n'y a pas de différence entre un paramètre formel et une variable locale : les deux ont une durée de vie limitée à l'appel de la fonction.

Ce mécanisme de transmission des arguments s'appelle le *passage par valeur*, car c'est seulement la valeur de l'argument effectif qui est transmise à la fonction appelée, et non l'argument effectif lui-même.

ATTENTION Liaison dynamique en Python

On considère le programme suivant qui définit une fonction `f` appelée depuis une fonction `g` :

```
def f():
    print('fonction 1')
def g():
    f()
```

Sans surprise, le message `'fonction 1'` s'affiche si on appelle `g()`. Rien n'empêche de redéfinir ensuite la fonction `f`, par exemple pour qu'elle affiche plutôt `'fonction 2'` :

```
def f():
    print('fonction 2')
```

De manière plus surprenante, c'est maintenant le message `'fonction 2'` qui est affiché quand on appelle de nouveau `g()`, alors qu'il s'agit toujours de la *même* fonction `g`. On appelle ce phénomène la *liaison dynamique*. Dit simplement, au moment de l'appel à une fonction `f`, Python en considère la dernière définition.

D'autres langages de programmation proposent au contraire une liaison *statique* où, dans l'exemple précédent, c'est toujours la fonction `f` initiale qui est appelée par la fonction `g` et donc toujours `'fonction 1'` qui est affiché.

Exercice 5.7 Expliquer pourquoi il n'est pas possible d'écrire une fonction **echange** qui échange le contenu des deux variables entières passées en arguments ?

5.2 Mécanismes avancés

5.2.1 Fonctions locales

Il arrive que l'utilisation d'une fonction soit limitée à la définition d'une autre fonction. On va supposer, par exemple, qu'on veut écrire une fonction **max3** calculant le maximum de trois entiers. Pour cela, il est élégant de commencer par la définition d'une fonction **max2** calculant le maximum de deux entiers, pour l'utiliser ensuite deux fois. Cependant, on ne souhaite pas nécessairement rendre visible la fonction **max2**. On la définit donc localement à la fonction **max3**, de la manière suivante :

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

On parle alors de *fonction locale* à une autre fonction.

Plus subtilement encore, une fonction locale peut faire référence à des arguments, ou à des variables locales, de sa fonction englobante :

```
def f(x, y, z):
    a = x * x
    def g(n):
        return n + a
    return g(y) + g(z)
```

5.2.2 Fonctions comme valeurs de première classe

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

Une application directe est la définition d'un opérateur mathématique par une fonction. Par exemple, la somme $\sum_{i=0}^{n} f(i)$, pour une fonction f quelconque, peut être ainsi définie :

```
def somme_fonction(f, n):
    s = 0
    for i in range(n+1):
        s = s + f(i)
    return s
```

Pour calculer la somme des carrés des entiers de 1 à 10, on commence par définir une fonction **carre**.

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

Puis, on la passe en argument à la fonction **somme_fonction** :

```
In [3]: somme_fonction(carre, 10)
Out[3]: 385
```

On peut même éviter de nommer la fonction **carre**, puisqu'elle est réduite à une simple expression, en utilisant une *fonction anonyme*. Une telle fonction s'écrit **lambda** *x*: *e*, où *e* est une expression pouvant comporter la variable *x*. Elle désigne la fonction $x \mapsto e(x)$. Ainsi l'exemple précédent se réécrit-il plus simplement :

```
In [4]: somme_fonction(lambda x: x*x, 10)
Out[4]: 385
```

De même qu'on peut passer une fonction en argument, on peut renvoyer une fonction comme résultat. En particulier, il est possible d'écrire une fonction qui prend comme arguments deux fonctions f et g et qui renvoie la composée $f \circ g$:

```
def composee(f, g):
    def h(x):
        return f(g(x))
    return h
```

Là encore, on simplifie l'écriture en utilisant une fonction anonyme :

```
def composee(f, g):
    return lambda x: f(g(x))
```

Il est maintenant possible de composer la fonction **carre** avec elle-même et de stocker le résultat dans une variable *a* :

```
In [5]: a = composee(carre, carre)
```

Ensuite, on applique la fonction contenue dans *a* à un argument :

```
In [6]: a(4)
Out[6]: 256
```

Les fonctions de première classe servent à exprimer de façon élégante des algorithmes génériques qu'on serait sinon obligé de réécrire pour chaque fonction. Des exemples d'application en sont donnés dans les chapitres 8 et 9.

Exercice 5.8 * Écrire une fonction **derive** qui prend en argument une fonction f et un flottant $\epsilon > 0$ et renvoie la fonction :

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

Vérifier que la fonction renvoyée par **derive**(**carre**, `0.001`) est effectivement une approximation de $x \mapsto 2x$.

5.2.3 Fonctions partielles

Les fonctions considérées jusqu'à présent sont des *fonctions totales*, c'est-à-dire qu'elles renvoient toujours un résultat, quelle que soit la valeur de leurs arguments. Il existe aussi des fonctions dites *partielles* parce qu'elles ne renvoient pas un résultat pour *toutes* les valeurs possibles des arguments. Par exemple, la fonction **divise** ci-après est partielle :

```
def divise(x, y):
    return x // y
```

En effet, tout appel à **divise** avec un deuxième argument égal à 0 va interrompre l'exécution de la fonction et provoquer l'affichage du message d'erreur suivant :

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in divise
ZeroDivisionError: integer division or modulo by zero
```

Pour définir plus proprement une fonction partielle, il est préférable de vérifier que les valeurs passées en argument sont bien dans son domaine d'utilisation. On peut pour cela vérifier une *précondition* à l'aide d'une instruction **assert** *e*. Cette instruction évalue l'expression booléenne *e* et provoque une erreur *AssertionError* si *e* est fausse. Ainsi, on peut ajouter une instruction **assert** *y* != 0 à la fonction **divise** pour interrompre son exécution si *y* est égal à 0 :

```
def divise(x, y):
    assert y != 0
    return x // y
```

Le message d'erreur suivant est alors affiché dès que l'expression *y* != 0 est évaluée à False :

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in divise
AssertionError
```

Afin d'afficher un message d'erreur plus informatif que *AssertionError*, on peut associer une deuxième expression à **assert**. Cette seconde expression sera évaluée uniquement si la première est fausse et sa valeur sera affichée pour compléter le message d'erreur. Ainsi, on peut écrire :

```
def divise(x, y):
    assert y != 0, 'division par 0 impossible'
    return x // y
```

Le message suivant sera affiché chaque fois que l'expression *y* != 0 sera fausse :

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in divise
AssertionError: division par 0 impossible
```

Il est préférable de placer ces préconditions au *début* du corps de la fonction, afin de s'assurer qu'aucune instruction n'est exécutée si les arguments ne sont pas dans le domaine de

la fonction. Par exemple, on considère la fonction suivante, qui incrémente une variable globale *y* et ne s'assure que très tardivement de la valeur de son argument *x* :

```
def f(x):
    global y
    ...
    y = y + 1
    ...
    assert x != 0
    return 1 // x
```

Bien que chaque appel à `f(0)` provoque une erreur *AssertionError*, toutes les instructions placées avant **assert** *x* `!= 0`, en particulier l'affection *y* `= y + 1`, seront exécutées. Ceci peut être une source d'erreur difficile à détecter dans un programme.

Une autre solution pour écrire une fonction partielle est de ne renvoyer aucun résultat quand les arguments sont en dehors du domaine. Par exemple, on peut écrire la fonction **divise** en utilisant une instruction conditionnelle qui teste la valeur de *y* avant de diviser :

```
def divise(x, y):
    if y != 0:
        return x // y
```

Comme on l'a vu dans la section 5.1.1, la valeur **None** sera renvoyée par **divise** si la variable *y* est égale à `0`. Cette solution présente tout de même l'inconvénient de ne pas informer clairement l'utilisateur quand la fonction est appelée en dehors de son domaine.

5.2.4 Fonctions de bibliothèque

Tous les langages de programmation répandus proposent des fonctions toutes faites pour la plupart des besoins courants, écrites par les concepteurs du langage eux-mêmes, ou par des utilisateurs, puis intégrées au fil du temps dans la distribution du langage. Ces fonctions écrites dans le langage et fournies avec lui sont regroupées dans ce qu'on appelle la *bibliothèque standard*.

Python n'échappe pas à la règle et prétend même être un langage « piles incluses » pour signifier que tout ce dont le programmeur peut avoir besoin est fourni avec le langage. Les différentes fonctions sont organisées en *modules* thématiques variés : représentation exacte des fractions, compression de fichiers, protocoles réseau….

Parmi les modules les plus utiles à l'élève de classes préparatoires — et, pour certains, déjà mentionnés plus haut — on peut citer :

- **math** qui contient toutes les fonctions habituellement utilisées en analyse ;
- **random** qui calcule des nombres pseudo-aléatoires ;
- **fractions** qui sert à manipuler des nombres rationnels en valeur exacte ;
- **numpy** qui fournit des outils variés pour le calcul scientifique. Ce dernier ne fait pas partie de la bibliothèque standard et doit donc être installé à part.

Pour utiliser les fonctions d'un module, on commence par importer ce dernier une fois pour toutes, en début de session interactive ou en début de programme. On prend ici l'exemple du module **`math`** :

```
In [7]: import math
```

Pour toute la suite de la session (ou du programme), on peut alors utiliser les fonctions de ce module en faisant précéder leur nom de celui du module :

```
In [8]: math.cos(14)
Out[8]: 0.1367372182078336
```

```
In [9]: math.sqrt(1 - math.sin(14)**2)
Out[9]: 0.13673721820783322
```

Cette notation permet au programmeur de définir par exemple sa propre fonction **`sqrt`**, sans risquer une homonymie avec celle fournie par le module. De même, plusieurs modules peuvent proposer des fonctions ayant le même nom, mais qui seront différenciées par leurs préfixes.

Une autre forme possible est de n'importer que la fonction dont on a besoin, par exemple **`from math import sqrt`**. On utilise alors directement **`sqrt`** sans le préfixe **`math`**, au prix d'un risque d'homonymie avec une fonction définie par ailleurs.

Enfin, les modules ne fournissent pas que des fonctions : certains proposent des constantes, de nouveaux types de données, etc.

```
In [10]: math.pi
Out[10]: 3.141592653589793
```

SAVOIR-FAIRE Rechercher une information au sein d'une documentation en ligne

Python propose près de deux cents modules, chacun susceptible de contenir jusqu'à plusieurs dizaines de fonctions. Il est donc exclu de les connaître tous, *a fortiori* de mémoriser leurs spécifications.

Un programmeur qui veut utiliser des fonctions de bibliothèque — et il y a intérêt s'il veut être efficace — doit donc savoir chercher l'information qui lui est nécessaire. Beaucoup de langages de programmation proposent une documentation officielle en ligne. Pour Python, il s'agit de `http://docs.python.org/`.

Les bibliothèques des langages sont organisées par thématiques, ce qui restreint assez vite l'espace de recherche. Ensuite, le caractère numérique d'une documentation en ligne fait qu'elle propose souvent des fonctions de recherche par mots-clés.

➥

Enfin, on veillera à consulter la documentation qui correspond à la version du langage que l'on utilise (pour Python, c'est particulièrement important à cause de l'incompatibilité entre les versions 2.x et 3.x).

5.2.5 Méthodes

Le langage Python appartient à la famille des langages dits *orientés objet*. Dans cet ouvrage, cet aspect du langage n'est pas du tout abordé. Néanmoins, l'accès à certaines bibliothèques oblige à introduire la notion de *méthode* liée à la programmation orientée objet.

Pour ce qui concerne cet ouvrage, une méthode n'est rien d'autre qu'une fonction *associée* à une valeur, telle qu'un entier, une chaîne de caractères, une liste, etc. Une valeur à laquelle sont associées des méthodes est appelée un *objet*. Par exemple, on prend `s` une chaîne de caractères définie par :

```
s = 'Ceci est une chaine'
```

On peut obtenir sa longueur de deux manières différentes. La première consiste à appeler la fonction **`len`**, avec la syntaxe :

```
In [11]: len(s)
Out[11]: 19
```

La seconde consiste à appeler sa méthode **`__len__`**, avec la syntaxe :

```
In [12]: s.__len__()
Out[12]: 19
```

Tout comme une fonction, une méthode peut prendre des arguments. Ainsi, on compte le nombre d'occurrences du caractère `'e'` dans la chaîne `s` avec la méthode **`count`** de `s` :

```
In [13]: s.count('e')
Out[13]: 4
```

D'une manière générale, la syntaxe d'appel d'une méthode est *`objet`*`.`**`methode`**`(`*`arguments`*`)`. Dans le contexte de cet ouvrage, ce n'est pas différent de l'appel d'une fonction où l'objet serait passé comme un argument supplémentaire, c'est-à-dire **`methode`**`(`*`objet, arguments`*`)`.

Voici deux autres exemples de méthodes sur les chaînes de caractères. L'opération permettant de passer d'une liste de caractères à la chaîne correspondante s'effectue par la méthode **`join`** d'une chaîne qui va servir de séparateur :

```
In [14]: ''.join(['a','b','c'])
Out[14]: 'abc'
```

```
In [15]: '-'.join(['d','e','f'])
Out[15]: 'd-e-f'
```

Cette méthode fonctionne également avec une liste de chaînes pour argument :

```
In [16]: ' '.join(['Ceci', 'est', 'une', 'phrase'])
Out[16]: 'Ceci est une phrase'
```

On réalise l'opération duale, c'est-à-dire découper une chaîne selon un séparateur donné, à l'aide la méthode **`split`** :

```
In [17]: 'Ceci est une phrase'.split('')
Out[17]: ['Ceci', 'est', 'une', 'phrase']
```

D'autres méthodes seront présentées dans la suite de cet ouvrage, notamment pour manipuler les piles et effectuer des entrées/sorties dans des fichiers.

5.3 La récursivité

Avertissement : les contenus abordés dans cette section sont au programme de seconde année uniquement. En particulier, certains exercices nécessitent d'avoir traité la complexité (section 6.1).

On considère la suite (u_n) suivante, qui calcule une approximation de $\sqrt{3}$:

$$\begin{cases} u_0 &= 2 \\ u_n &= \dfrac{1}{2}\left(u_{n-1} + \dfrac{3}{u_{n-1}}\right) \end{cases}$$

On peut calculer cette suite à l'aide d'une fonction **`u`**, qui prend un argument *`n`*, et telle que **`u`**(*`n`*) renvoie la valeur de u_n. En Python, on peut écrire la fonction **`u`** en suivant *directement* la définition précédente :

```
def u(n):
    if n == 0:
        return 2.
    else:
        return 0.5 * (u(n-1) + 3. / u(n-1))
```

On remarque que dans la dernière ligne, il y a deux appels à la fonction pour calculer u_{n-1}. En effet, rien n'interdit d'appeler la fonction **`u`** à l'intérieur de son propre corps. On nomme alors cela une *fonction récursive*. Pour plus d'efficacité, on peut « factoriser » les deux appels **`u`**(*`n`*-1) en stockant le résultat dans une variable locale, ce qui supprime un appel potentiellement coûteux à **`u`** :

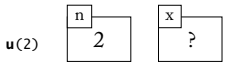
```
    ...
    else:
        x = u(n-1)
        return 0.5 * (x + 3. / x)
```

Ainsi, pour calculer une valeur approchée de $\sqrt{3}$, il suffit d'écrire le programme suivant :

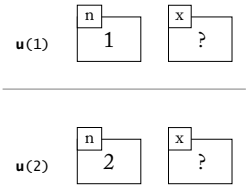
```
print(u(2))
```

Il affiche `1.7321428571428572` à l'écran. Pour comprendre comment ce résultat est calculé, on va suivre pas à pas l'exécution de cet appel de fonction.

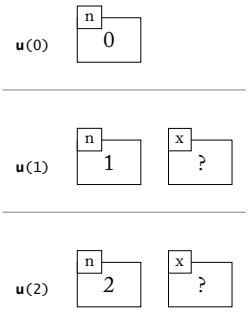
Juste après l'appel `u(2)`, la fonction `u` compare la valeur de son argument formel *n* (qui vaut ici `2`) avec `0` et exécute la branche `else` de l'instruction conditionnelle. Avant d'exécuter l'instruction `x = u(n-1)`, l'état de la mémoire est donc le suivant :



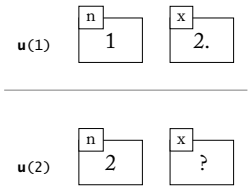
La variable locale *x* n'a toujours pas reçu de valeur. L'exécution se poursuit par l'appel `u(2-1)`. Toute la subtilité des fonctions récursives se dévoile dans ce deuxième appel. Que ce soit les paramètres formels ou les variables locales, toutes les variables dont la portée est limitée à une fonction s'ajoutent à l'état mémoire du programme lorsque celle-ci est appelée. Ainsi, l'état mémoire après ce deuxième appel est constitué de deux variables *n* et de deux variables locales *x* : celles allouées par l'appel `u(2)` et celles allouées pour `u(1)`. Ces variables n'ont de commun que le nom qu'on leur a donné, car elles représentent bien des cases mémoire distinctes.



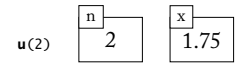
Tout comme pour l'appel précédent, la nouvelle variable locale *x* ne contient toujours pas de valeur avant d'exécuter `x = u(1-1)`. Ce dernier appel à la fonction `u` aboutit à l'état mémoire suivant :



Seule une nouvelle case pour `n` est allouée en mémoire (puisque la branche `else` n'est pas exécutée). L'appel `u(0)` se termine alors par `return 2.`, ce qui a pour effet, non seulement de supprimer de la mémoire les variables locales allouées pour cet appel, mais également d'affecter la valeur `2.` à la variable `x` de l'appel précédent. On se trouve alors dans l'état mémoire suivant :



De la même manière, l'appel `u(1)` se termine par `return 0.5 * (2. + 3. / 2.)` et on revient à l'état mémoire du premier appel :



Enfin, l'appel `u(2)` se termine et la fonction renvoie la valeur `1.7321428571428572` comme approximation de $\sqrt{3}$.

POUR ALLER PLUS LOIN **Dérécursivation**

Il était également possible d'écrire une version non récursive de la fonction `u` en utilisant une boucle, par exemple sous la forme suivante :

```
def u(n):
    r = 2.
    for i in range(n):
        r = 0.5 * (r + 3. / r)
    return r
```

Les mêmes calculs sont effectués, dans le même ordre. Cependant, le rapprochement avec la définition de la suite (u_n) est moins évident. En particulier, dans l'instruction `r = 0.5 * (r + 3. / r)`, il faut comprendre que l'occurrence de `r` dans le membre droit désigne u_i et que celle de `r` dans le membre gauche désigne u_{i+1}. Puisqu'on termine avec $i = n - 1$, on a bien calculé u_n.

5.3.1 Concevoir une fonction récursive

La conception d'une fonction récursive n'est pas éloignée du principe de *démonstration par récurrence*. Par exemple, le principe de récurrence simple permet de démontrer une propriété P_n pour tout $n \in \mathbb{N}$ en démontrant d'une part le cas de base P_0 et d'autre part que P_{n-1} implique P_n pour tout $n > 0$.

De la même façon, on peut définir une fonction `f` prenant en argument un entier naturel *n* en se ramenant au calcul de `f(0)` d'une part et de `f(`*n*`)` en fonction de `f(`*n*`-1)` d'autre part. La fonction `f` prend alors la forme suivante :

```
def f(n):
    if n == 0:
        return ...
    else:
        return ... f(n-1) ...
```

L'exemple le plus classique est sûrement celui de la fonction **`factorielle`**, dont la définition est donnée page 104 :

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

Il est important de noter qu'une telle fonction ne terminera pas sur un argument *n* négatif. En effet, **`factorielle`**`(-1)` appellerait **`factorielle`**`(-2)`, qui appellerait **`factorielle`**`(-3)`, etc. Il s'agit donc d'une fonction partielle, à laquelle on peut appliquer toute solution discutée dans la section 5.2.3. En particulier, on peut s'assurer que *n* est bien un entier naturel en écrivant :

```
def factorielle(n):
    assert n >= 0
    ...
```

Le schéma de récurrence simple peut être appliqué à des fonctions ayant d'autres arguments que *n*. Ainsi, la fonction **`puissance`** qui calcule *x* à la puissance *n* peut facilement être définie par récurrence simple sur *n*, de la manière suivante :

```
def puissance(x, n):
    if n == 0:
        return 1
    else:
        return x * puissance(x, n-1)
```

Comme en mathématiques, le schéma de récurrence simple peut être adapté à des définitions impliquant plusieurs cas de base. Ainsi, on évite une multiplication inutile quand *n* vaut `1` dans la fonction **`puissance`** en la réécrivant de la façon suivante :

```
def puissance(x, n):
    if n == 0:
        return 1
    elif n == 1:
        return x
    else:
        return x * puissance(x, n-1)
```

L'écriture de fonctions récursives n'est pas limitée au schéma de récurrence simple. On peut également utiliser un schéma de récurrence forte, c'est-à-dire effectuer des appels récursifs sur des valeurs strictement inférieures à *n*-1. Reprenant l'exemple du calcul de x^n, on propose un meilleur algorithme qui exploite les deux identités suivantes :

$$\begin{cases} x^{2k} &= (x^k)^2 \\ x^{2k+1} &= x(x^k)^2 \end{cases}$$

Elles permettent de ramener le calcul de x^n à celui de $x^{\lfloor \frac{n}{2} \rfloor}$. Le cas de base reste le même, à savoir $x^0 = 1$. Dans le cas récursif, on commence par calculer $x^{\lfloor \frac{n}{2} \rfloor}$ dans une variable *r*, puis on teste la parité de *n* pour choisir entre les deux identités ci-avant. Finalement, on obtient le code suivant :

```
def puissance_rapide(x, n):
    if n == 0:
        return 1
    else:
        r = puissance_rapide(x, n // 2)
        if n % 2 == 0:
            return r * r
        else:
            return x * r * r
```

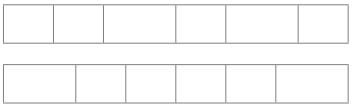
Ainsi, le calcul de `puissance_rapide(3, 5)` se ramène directement au calcul de `puissance_rapide(3, 2)` et évite les appels à `puissance_rapide(3, 3)` et `puissance_rapide(3, 4)`.

Exercice 5.9 * Écrire une variante (toujours récursive) de la fonction `puissance_rapide` qui exploite plutôt les identités suivantes :

$$\begin{cases} x^{2k} &= (x^2)^k \\ x^{2k+1} &= x(x^2)^k \end{cases}$$

Y a-t-il une différence dans le nombre de multiplications effectuées ?

Pour continuer l'analogie avec le principe de récurrence forte en mathématiques, il est parfois nécessaire d'effectuer *plusieurs* appels récursifs pour calculer `f(n)`. On considère le problème consistant à calculer le nombre de façons de construire une rangée de longueur n avec des briques de longueur 2 et 3. Voici par exemple deux rangées de longueur $n = 14$.



On peut en dénombrer 21 au total. Les cas de base correspondent à $n = 1$ (pas de solution) et $2 \leqslant n \leqslant 3$ (une solution unique). Le calcul récursif consiste à se ramener au cas $n - 2$ (ajout d'une brique de longueur 2) et au cas $n - 3$ (ajout d'une brique de longueur 3).

```
def briques(n):
    assert n >= 1
    if n == 1:
        return 0
    elif n == 2 or n == 3:
        return 1
    else:
        return briques(n-2) + briques(n-3)
```

L'exercice 5.24 propose un autre exemple.

Enfin, il est parfois possible que la définition d'une fonction `f` fasse appel à une fonction `g`, et que la définition de `g` fasse appel elle-même à celle de `f`. On parle alors de fonctions *mutuellement récursives*. Par exemple, on peut définir une fonction `pair` pour déterminer si un entier `n` est pair par récurrence mutuelle avec une fonction `impair` qui, elle, détermine si un entier `n` est impair. En Python, il suffit d'écrire ces deux fonctions, l'une à la suite de l'autre :

```
def pair(n):
    return (n == 0) or impair(n-1)

def impair(n):
    return (n != 0) and pair(n-1)
```

5.3.2 Terminaison et correction d'une fonction récursive

Dans cette section, on va montrer comment raisonner à propos d'une fonction récursive, pour démontrer d'une part sa terminaison et d'autre part sa correction, c'est-à-dire le fait qu'elle calcule bien ce qu'elle doit calculer. Sans surprise, on utilisera le principe de démonstration par récurrence pour démontrer la correction d'une fonction récursive.

On va prendre l'exemple de la fonction `factorielle` définie plus haut page 133. On veut montrer par récurrence sur $n \geqslant 0$ la propriété suivante :

$$H_n : \quad \texttt{factorielle}(n) \text{ termine et renvoie la valeur } n!$$

La propriété H_0 est vérifiée car `factorielle(0)` se réduit à `return 1`. Pour $n > 0$, on suppose H_{n-1} et on cherche à montrer H_n. Le calcul de `factorielle(n)` commence par un appel récursif à `factorielle(n-1)`. Par hypothèse de récurrence, cet appel termine et renvoie la valeur $(n-1)!$. Puis l'appel à `factorielle(n)` multiplie ce résultat par `n` et renvoie le produit. Donc, cet appel termine et renvoie bien $n \times (n-1)! = n!$, ce qui démontre H_n.

Il est important de noter que nous n'avons rien démontré quant aux appels à la fonction `factorielle` sur des arguments négatifs. En particulier, ils peuvent ne pas terminer (ce qui est le cas ici), renvoyer des valeurs farfelues, ou encore échouer (ce qui serait le cas avec `assert n >= 0` par exemple).

Si on a utilisé un schéma de récurrence forte pour définir une fonction récursive, alors il faudra bien entendu en démontrer la correction par récurrence forte également. Avec l'exemple de la fonction **puissance_rapide** page 134, on cherche à montrer par récurrence forte sur $n \geqslant 0$ la propriété suivante :

$$H_n : \texttt{puissance\_rapide}(x, n) \text{ termine et renvoie la valeur } x^n$$

La propriété H_0 est vérifiée car `puissance_rapide(x, 0)` se réduit à `return 1` (en admettant que l'on a posé $0^0 = 1$ arbitrairement). Soit maintenant $n > 0$; on suppose H_i pour tout $0 \leqslant i < n$ et on veut montrer H_n. Le calcul de `puissance_rapide(x, n)` commence par un appel récursif `r = puissance_rapide(x, n // 2)`. On pose $k = \lfloor \frac{n}{2} \rfloor$. Comme $n > 0$, on a $k < n$. On peut donc appliquer l'hypothèse de récurrence H_k, qui affirme que l'appel `puissance_rapide(x, n // 2)` termine et renvoie la valeur x^k. On distingue alors deux cas, selon la parité de n. Si n est pair, c'est-à-dire $n = 2k$, alors le programme effectue `return r * r`. Donc il termine et renvoie $x^k \times x^k = x^{2k} = x^n$, ce qui démontre H_n. On procède de même lorsque n est impair.

Exercice 5.10 ** Démontrer la terminaison et la correction de la fonction **puissance** page 133.

Exercice 5.11 Montrer la correction de la fonction **briques** définie plus haut (page 134).

Exercice 5.12 ** Que se passe-t-il avec la fonction **puissance_rapide** si on écrit `n / 2` au lieu de `n // 2` (en Python 3) ou encore `n / 2.` ?

ATTENTION **Limitation de la récursivité en Python**

Le langage Python limite, arbitrairement, le nombre d'appels imbriqués à 1 000. Une fonction qui fait plus de 1 000 appels récursifs provoque l'erreur suivante :

```
RuntimeError: maximum recursion depth exceeded
```

Même si cette limite semble basse, elle n'exclut pas pour autant l'utilisation de fonctions récursives en Python. En effet, il existe de nombreuses situations où l'on sait que le nombre d'appels sera bien inférieur à 1 000. C'est le cas en particulier pour des fonctions qui font un nombre d'appels *logarithmique* en la taille des données. Voir par exemple l'exercice 5.15.

POUR ALLER PLUS LOIN **Démontrer la terminaison d'une fonction récursive**

De même qu'on peut justifier la terminaison d'une boucle en exhibant un entier naturel qui décroît strictement à chaque itération (voir page 98), on peut démontrer la terminaison d'une boucle récursive en exhibant en entier naturel qui décroît strictement à chaque appel récursif. Le plus souvent, il s'agira directement de l'un des arguments de la fonction récursive, comme dans le cas des fonctions définies plus haut dans cette section.

➥

Cependant, rien n'exclut qu'il s'agisse d'une expression plus compliquée. La célèbre « fonction 91 » de McCarthy en est un exemple :

```
def f91(n):
    if n <= 100:
        return f91(f91(n + 11))
    else:
        return n - 10
```

Il existe même des fonctions récursives dont on ne sait pas démontrer la terminaison, comme la célèbre suite de Syracuse :

```
def syracuse(n):
    if n == 1:
        return 1
    elif n % 2 == 0:
        return syracuse(n // 2)
    else:
        return syracuse(3 * n + 1)
```

Exercice 5.13 Démontrer la terminaison de la fonction **f91**.

SAVOIR-FAIRE Déboguer une fonction récursive

Lorsqu'on débogue une fonction récursive, il faut évidemment entrer dans les appels récursifs pour comprendre ce qui se passe : on est naturellement conduit à effectuer *Pas vers l'intérieur*.

Cependant, comme expliqué dans ce chapitre, à chaque appel récursif, il est créé un nouvel ensemble de variables locales à la fonction. L'explorateur de variables ne montre que les variables en cours d'utilisation, autrement dit celles qui sont propres à l'appel dans lequel on se situe.

Il y a cependant une possibilité pour visualiser la suite d'appels récursifs déjà effectués. En tapant *w* (pour *where*) dans l'interpréteur, on voit apparaître la liste des appels de fonctions en cours, le plus récent étant situé tout en bas. Ensuite, il est possible de se déplacer dans cette liste avec les commandes *u* (pour **up**) et *d* (pour **down**). L'explorateur de variables montre alors les contenus des variables correspondant à ces différents appels.

Spyder ne propose pour l'instant pas d'interface graphique pour effectuer ces manipulations.

5.3.3 Complexité d'une fonction récursive

On explique ici comment calculer le *coût* d'une fonction récursive, à savoir le nombre d'opérations élémentaires qu'elle effectue ou son occupation mémoire totale. La notion de complexité sera présentée plus en détail dans le prochain chapitre, section 6.1.

Reprenons l'exemple de la fonction `u` (définie p. 130) :

```
def u(n):
    if n == 0:
        return 2.
    else:
        x = u(n-1)
        return 0.5 * (x + 3. / x)
```

et évaluons le nombre d'opérations arithmétiques (addition, multiplication et division) qu'elle effectue.

Si n désigne la valeur de son argument, notons $C(n)$ ce nombre d'opérations. En suivant la définition de la fonction `u`, on obtient les deux équations suivantes :

$$\begin{aligned} C(0) &= 0 \\ C(n) &= C(n-1) + 3 \end{aligned}$$

En effet, dans le cas $n = 0$, on ne fait aucune opération arithmétique. Et dans le cas $n > 0$, on fait d'une part un appel récursif sur la valeur $n-1$, d'où $C(n-1)$ opérations, puis trois opérations arithmétiques (une multiplication, une addition et une division). Il s'agit d'une suite arithmétique de raison 3, dont le terme général est :

$$C(n) = 3n$$

Le nombre d'opérations arithmétiques effectuées par la fonction `u` est donc proportionnel à n.

Si en revanche on avait écrit la fonction `u` plus naïvement, avec deux appels récursifs `u(n-1)`, c'est-à-dire :

```
def u(n):
    if n == 0:
        return 2.
    else:
        return 0.5 * (u(n-1) + 3. / u(n-1))
```

alors les équations définissant $C(n)$ seraient les suivantes :

$$\begin{aligned} C(0) &= 0 \\ C(n) &= C(n-1) + C(n-1) + 3 \end{aligned}$$

En effet, il convient de prendre en compte le coût $C(n-1)$ des deux appels à `u(n-1)`. Il s'agit maintenant d'une suite arithmético-géométrique, dont le terme général est :

$$C(n) = 3(2^n - 1)$$

Une autre manière d'évaluer le coût d'une fonction récursive est de calculer le nombre d'*appels*, puis d'évaluer le coût de chaque appel. Si on note $A(n)$ le nombre d'appels récursifs dans les deux exemples précédents, on a :

$$\begin{aligned} A(0) &= 0 \\ A(n) &= A(n-1)+1 \end{aligned}$$

dans le premier cas, et :

$$\begin{aligned} A(0) &= 0 \\ A(n) &= A(n-1)+A(n-1)+1 \end{aligned}$$

dans le second cas. Le terme général est donc $A(n) = n$ dans le premier cas et $A(n) = 2^n - 1$ dans le second. Puisqu'on n'a ici aucune opération arithmétique dans le cas de base $n = 0$ et exactement trois opérations arithmétiques dans le cas récursif, on retrouve immédiatement la valeur de $C(n)$ calculée précédemment. D'une manière générale, la valeur de $C(n)$ ne se déduit pas toujours aussi facilement de la valeur de $A(n)$. En effet, il peut y avoir des opérations dans le cas de base et/ou un nombre d'opérations arithmétiques variant selon la valeur de n dans le cas récursif.

Comme nous l'avons expliqué page 131, chaque appel récursif alloue de la mémoire pour les paramètres effectifs et les variables locales de cet appel. L'occupation mémoire d'un calcul récursif admet donc pour majorant le produit du nombre d'appels récursifs par la quantité de mémoire allouée par chaque appel. Dans les deux exemples précédents, on a calculé explicitement le nombre d'appels $A(n)$. L'occupation mémoire est donc $2n$ dans le premier cas (il y a deux cases mémoire, une pour `n` et une autre pour `x`) et $2^n - 1$ dans le second cas (il y a une case mémoire, pour `n`). Cependant, dans le second cas, les $2^n - 1$ cases mémoire ne seront pas utilisées simultanément. En effet, celles allouées pour le premier appel à `u(n-1)` peuvent être réutilisées pour le second (et en pratique elles le sont). Pour une analyse plus fine de l'occupation mémoire, il convient donc de calculer le nombre d'appels *imbriqués*.

Exercice 5.14 On considère la première version de la fonction **`puissance`**, définie p. 133.

1 Combien effectue-t-elle exactement d'appels récursifs pour calculer x^n ?
2 Quel est son coût en mémoire ?

Exercice 5.15 * On considère la fonction **`puissance_rapide`** définie p. 134.

1 Montrer qu'elle calcule x^n en effectuant un nombre total d'appels récursifs proportionnel à $\log n$.
2 A-t-on la même complexité quand on n'utilise pas de variable locale `r`, mais que l'on écrit directement : `puissance_rapide(x, n // 2) * puissance_rapide(x, n // 2)` à la place de `r * r` ?

5.4 Exercices

Exercice 5.16 Écrire en Python une fonction qui prend comme argument un entier n et renvoie l'entier 2^n.

Exercice 5.17 * Écrire en Python une fonction qui prend comme argument un entier n et renvoie un booléen qui indique si cet entier est premier ou non.

Exercice 5.18 Qu'affiche le programme suivant ? Pourquoi ?

```
def g(x):
    global a
    a = 10
    return 2 * x

def f(x):
    v = 1
    return g(x) + v

a = 3
print(f(6)+a)
```

Proposer une adaptation de ce programme dans laquelle le résultat affiché ne dépend pas de l'ordre d'évaluation.

Exercice 5.19 Qu'affiche le programme suivant ? Pourquoi ?

```
def g(x):
    global v
    v = 1000
    return 2 * x

def f(x):
    v = 1
    return g(x) + v

a = 3
print(f(6)+a)
```

Proposer une adaptation de ce programme dans laquelle le résultat affiché ne dépend pas de l'ordre d'évaluation.

Exercice 5.20 Soit `f` la fonction suivante :

```
def f(x):
    while (True):
        x = x + 1
        if (x == 1000):
            return 2 * x
```

Que renvoie l'appel `f(500)` ?

Exercice 5.21 ** Écrire une fonction qui calcule le PGCD des deux entiers naturels passés en arguments en suivant l'algorithme d'Euclide, à l'aide d'une boucle `while`. Démontrer la correction et la terminaison de cette fonction.

Exercice 5.22 On écrit un programme qui gère automatiquement le score au tennis.

1. Écrire une condition booléenne qui permet de décider si un joueur a gagné un jeu.
2. Définir une fonction qui compte les points au cours d'un jeu. En entrée, on demande répétitivement quel joueur, 1 ou 2, gagne le point ; au fur et à mesure, on calcule et on affiche le score. Le programme s'arrête dès qu'un joueur gagne le jeu, après avoir affiché le nom du vainqueur.
3. Pour faciliter les tests, écrire une seconde version de cette fonction avec pour seule entrée une chaîne de caractères qui contient les numéros successifs des joueurs marquant les points ; la fonction lit cette chaîne caractère par caractère pour compter les points. Ainsi, l'entrée `211222` sera comprise comme : le joueur 2 gagne un point, puis le joueur 1 en gagne deux, puis le joueur 2 en gagne trois et le joueur 2 gagne donc le jeu.

4 Écrire une deuxième fonction qui compte les jeux au cours d'un set et s'arrête lorsqu'un joueur gagne le set. Cette fonction fera appel à la précédente pour savoir qui gagne les jeux. On n'oubliera pas de prévoir le cas particulier du jeu décisif.
5 Écrire une troisième fonction qui compte les sets et s'arrête lorsqu'un joueur gagne le match. On pourra, avant de commencer le match, demander en combien de sets gagnants il est joué.

Exercices utilisant la récursivité

Exercice 5.23 Écrire une fonction récursive qui calcule la somme des n premiers entiers. Quelle est la complexité de cette fonction ?

Exercice 5.24 * Suite de Fibonacci. On considère la suite de Fibonacci définie par :

$$\begin{cases} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1} \text{ pour } n \geqslant 2. \end{cases}$$

Écrire une fonction récursive basée sur ces relations qui prend n en argument et renvoie F_n. Quelle est sa complexité ?

Accélérer le calcul de F_n en écrivant plutôt une fonction récursive auxiliaire qui prend en arguments F_{n-1}, F_n et $k \geqslant 0$ et renvoie F_{n+k} (on pourra poser $F_{-1} = 1$). Quelle est la nouvelle complexité ?

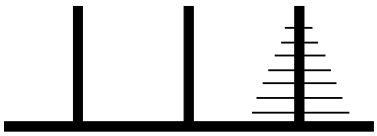
Exercice 5.25 ** Écrire une fonction récursive qui calcule le PGCD des deux entiers naturels passés en arguments en suivant l'algorithme d'Euclide. Démontrer la terminaison et la correction de cette fonction.

Exercice 5.26 Modifier la fonction qui calcule les termes de la suite de Syracuse (voir l'encadré page 137) pour qu'elle affiche les éléments de la suite jusqu'au premier rang n_0 tel que $u_{n_0} = 1$, ainsi que la valeur de n_0.

Exercice 5.27 Définir une fonction récursive qui, étant donné un entier n, décide si l'écriture en base 3 de n ne comporte que des 0 et des 1. Quelle est la complexité de cette fonction ?

Exercice 5.28 ** Les tours de Hanoï. *Les tours de Hanoï* est un jeu inventé par Édouard Lucas en 1883. Il est formé de sept disques de tailles différentes répartis en trois colonnes. Au départ, tous les disques sont empilés sur la colonne de gauche par taille croissante.

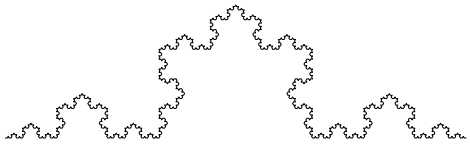
Les seuls mouvements possibles sont les déplacements d'un disque situé au sommet d'une colonne vers le sommet d'une autre colonne, à condition que la colonne d'arrivée soit vide ou que le disque déplacé soit plus petit que son sommet. On note `n -> n'` le déplacement d'un disque de la colonne `n` vers la colonne `n'`. Le but du jeu est de déplacer tous les disques vers la colonne de droite.



Écrire un programme qui affiche une solution du jeu sous la forme d'une suite de mouvements.

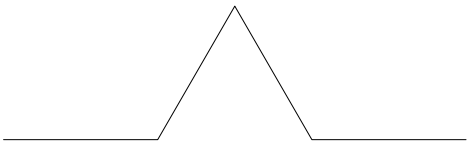
Indication : dans une variante, le jeu n'est formé que de six disques. Si l'on sait résoudre le jeu à six disques, comment résoudre le jeu à sept disques ?

Exercice 5.29 * Le flocon de Von Koch. *Le flocon*, courbe définie par Helge Von Koch en 1906, est un exemple de courbe continue partout et dérivable nulle part. C'est aussi un exemple d'ensemble fractal, c'est-à-dire dont la dimension de Hausdorff n'est pas un nombre entier.

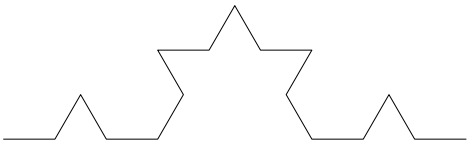


Cette courbe se définit comme la limite de la suite de courbes dont le premier élément est un segment :

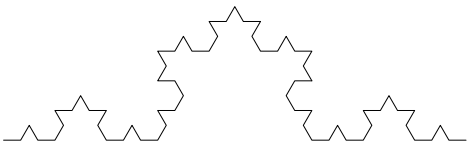
et dans laquelle on passe d'un élément au suivant en divisant chaque segment en trois et en remplaçant le morceau du milieu par deux segments formant un triangle équilatéral avec le segment supprimé. Le deuxième élément est donc :



le troisième :



et le quatrième :



Écrire un programme qui dessine le n-ième élément de cette suite.
Il est recommandé d'utiliser le module **`turtle`** (voir annexe B.3 page 380).