Apprendre Python et s'initier à la programmation

Partie 1 : Bases de la programmation

Par Sébastien Combéfis

Date de publication : 14 juin 2018

Cette première partie vous permet de découvrir les *bases de la programmation*. Après une introduction sur l'*ordinateur et son fonctionnement*, et une présentation de la notion de *programmation* dans le premier chapitre, le deuxième présente le langage de programmation *Python*. Le troisième chapitre concerne les *instructions de contrôle* qu'on utilise pour altérer l'exécution séquentielle des instructions d'un programme. Le quatrième chapitre décrit la notion de *fonction*, construction permettant de structurer un programme, pour notamment éviter des répétitions de code et rendre ce dernier générique. Le cinquième chapitre présente les *séquences*, structures permettant d'organiser les données de manière séquentielle. Enfin, le dernier chapitre de cette partie présente la notion d'*algorithme* et résume toutes les notions couvertes dans cette première partie à l'aide de nombreux exemples de plus grande ampleur.

Commentez

I - Ordinateur et programmation

Nous sommes envahis par les *ordinateurs* et autres appareils électroniques modernes tels que les smartphones, consoles de jeu, fours à micro-ondes, smartwatches, distributeurs automatiques de boissons, etc. Tous ces appareils, et encore bien d'autres, fonctionnent grâce aux *programmes* qui les équipent. Un ou plusieurs programmes sont en cours *d'exécution* sur ces appareils, ce qui leur permet de fournir des services à leurs utilisateurs. Ce premier chapitre fournit une description basique de ce qu'est un *ordinateur*, ainsi que de la notion de *programmation*.

I-A - Ordinateur

On peut décrire un ordinateur en suivant l'*architecture de Von Neumann*, un modèle permettant de représenter des systèmes informatiques simples. Dans ce modèle, illustré par la figure 1 à la page suivante, un ordinateur est décomposé en quatre parties :

- L'*unité arithmétique et logique* (*Arithmetic Logic Unit* (ALU) en anglais), aussi appelée unité de traitement, est celle où se déroulent toutes les opérations de base, notamment les opérations arithmétiques comme les additions.
- L'*unité de contrôle* s'assure du déroulement correct de l'exécution des programmes en se chargeant du séquençage des opérations.
- La *mémoire* contient à la fois le programme à exécuter et les données qu'il va utiliser et créer. Elle se divise en deux parties : la mémoire *volatile* est celle utilisée lorsque l'ordinateur est en cours de fonctionnement et la mémoire *permanente* est celle qui permet le stockage permanent d'informations.
- Les dispositifs d'*entrée-sortie* permettent à l'ordinateur de communiquer avec le monde extérieur. Les entrées lui permettent de récupérer des informations (la souris, le clavier, le réseau, etc.) et les sorties lui permettent de produire des résultats (l'écran, l'imprimante, le réseau, etc.).

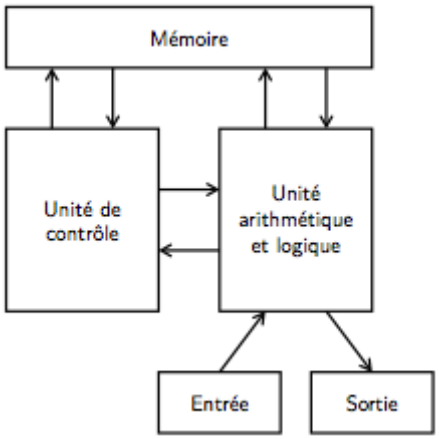

Figure 1. L'architecture de Von Neumann décrit un ordinateur comme étant composé de quatre entités interagissant entre elles pour exécuter des programmes.

Tout ordinateur, ainsi qu'une série d'autres appareils électroniques, peuvent être modélisés grâce à cette simple architecture. Dans ce modèle, de nombreux transferts de données se font entre la mémoire et l'unité de traitement, pouvant mener à un ralentissement global du système. De nos jours, la vitesse des processeurs n'a cessé d'augmenter, tout comme la quantité de mémoire disponible, autant la volatile que la permanente.

Néanmoins, les progrès en termes de *vitesses de transfert de données* n'ont pas été aussi marquants, ce qui conduit à un goulot d'étranglement entre la mémoire et l'unité de traitement. L'architecture de Von Neumann a donc été améliorée pour pallier cette faiblesse, en ajoutant, par exemple, des systèmes de cache, du multithreading, des nouveaux types de mémoire volatile, etc.

Nous allons en rester là pour ce descriptif des ordinateurs, puisque ce n'est pas le but de ce cours, et passer directement à son sujet principal, à savoir la *programmation*.

I-B - Programme

Un *programme* est essentiellement une suite d'actions à entreprendre pour atteindre un but ; il s'agit d'une *séquence d'instructions* à exécuter. On peut faire le parallèle avec une *recette de cuisine* qui est un programme dont le but est la réalisation d'un plat ou avec un *lave-linge* qui propose plusieurs programmes commandant la suite d'actions à exécuter (essentiellement les roulements du tambour) pour laver le linge.

Comme le montre la figure 2, on peut voir un programme comme une boite noire qui reçoit des *données en entrée*, effectue des calculs à partir de ces données et se termine en produisant des *résultats en sortie*. En toute généralité, les entrées et sorties utilisées par un programme proviennent directement de celles de l'ordinateur qui l'exécute. Néanmoins, ces dernières pourraient simplement provenir de la mémoire de l'ordinateur, un programme pouvant ne pas être autorisé à communiquer directement avec le monde extérieur.

Figure 2. Un programme peut être vu comme une boite noire recevant des données en entrée, effectuant un calcul et produisant des résultats en sortie.

Dans le domaine de l'informatique, on écrit des programmes dans le but de résoudre des *problèmes*. Cherchons, par exemple, les racines d'un trinôme du second degré qui est, pour rappel, un polynôme de la forme :

$$ax^2 + bx + c \qquad (a \in \mathbb{R}_0 \text{ et } b, c \in \mathbb{R}).$$

Pour trouver les racines que l'on cherche, on peut suivre cette procédure :

1. Calculer le discriminant
$$\Delta = b^2 - 4ac$$
.
2. Trois cas possibles :
    1. Si
    $$\Delta < 0$$
    , alors il n'y a pas de racine réelle.
    2. Si
    $$\Delta = 0$$
    , alors il y a une racine réelle double :
    $$x = -\frac{b}{2a}.$$
    3. Si
    $$\Delta > 0$$
    , alors il y a deux racines réelles distinctes :
    $$x_1 = \frac{-b + \sqrt{\Delta}}{2a} \quad \text{et} \quad x_2 = \frac{-b - \sqrt{\Delta}}{2a}.$$

Comme on peut l'observer, cette procédure comporte plusieurs *types d'instructions* :

- on peut *effectuer des calculs* à partir de données reçues en entrée (le calcul de $b^2 - 4ac$ lors de la première étape) ;
- on peut *stocker un résultat* intermédiaire (le discriminant stocké dans Δ) ;
- on peut *effectuer des tests* et entreprendre des actions si ces derniers passent (les trois tests de la valeur du discriminant lors de la deuxième étape) ;
- on peut *produire un résultat* final (le verdict sur le nombre de racines et leur valeur).

Comme on le verra dans la suite du cours, il s'agit là de plusieurs constructions clés de la programmation. On peut résumer ce problème par une courte description comme illustrée par la figure 3. On y décrit, en langage naturel, le problème ainsi que les données à fournir en entrée et le résultat à produire en sortie.

Problème	Recherche des racines d'un trinôme du second degré de la forme $ax^2 + bx + c$.
Entrées	Les coefficients $a \in \mathbb{R}_0$ et $b, c \in \mathbb{R}$.
Sorties	Le nombre de racines réelles (0, 1 ou 2), ainsi que leurs valeurs.

Figure 3. Le problème de recherche des racines d'un trinôme du second degré reçoit les coefficients a, b et c et identifie la ou les racines de ce trinôme s'il en existe.

Pour continuer l'analogie avec la cuisine, on peut faire le lien entre entrées, programme et sorties avec, respectivement, les ingrédients, la recette et le plat préparé. Une fois les ingrédients fournis au cuisinier, et après que ce dernier ait exécuté la recette (chacune de ses étapes), le plat désiré est produit comme résultat.

I-C - Programmation

La *programmation* comprend toutes les activités qu'il faut mettre en œuvre afin de produire un programme (informatique). Il s'agit en réalité de l'une des étapes du développement d'un logiciel ou d'une application. La programmation concerne précisément l'écriture du *code source* d'un programme. Généralement, un programme est écrit à l'aide d'un *langage de programmation*, notation souvent textuelle destinée à décrire un programme. Il existe également des langages visuels (un langage de programmation visuel par blocs très répandu et utilisé pour l'enseignement de la programmation aux plus jeunes est le langage Scratch, développé par le MIT) souvent utilisés pour des programmes de petite ampleur. La figure 4 montre une représentation graphique d'un programme qui résout le problème de recherche de racines présenté à la section précédente.

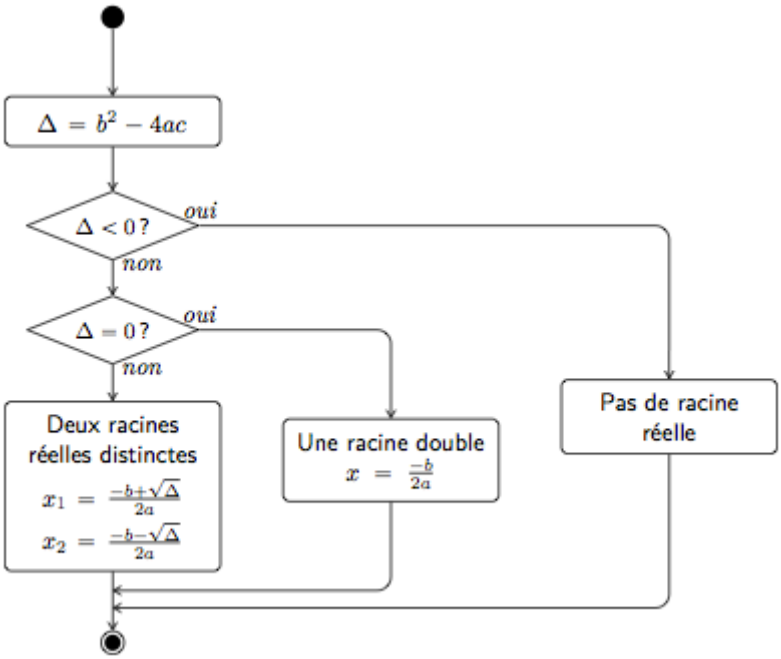

Figure 4. Un programme peut être décrit à l'aide d'une représentation graphique appelée organigramme ou diagramme d'activités.

Le programme commence son exécution au cercle plein ● pour se terminer au cercle semi-plein ◉. Les boites rectangulaires correspondent à des *commandes* décrivant une action à exécuter, ou un résultat à produire. Enfin, les boites en forme de losange correspondent à des *conditions* pouvant être satisfaites ou non. En fonction du résultat, l'exécution du programme continue d'un côté ou de l'autre.

On utilisera cette notation graphique au début de notre apprentissage, afin de mieux comprendre les constructions de base d'un langage de programmation textuel, *Python* en l'occurrence, qui est introduit dans le prochain chapitre.

Cette notation graphique permet en fait de décrire n'importe quel type d'activité. On peut, par exemple, décrire la réalisation d'une omelette comme le montre la figure 5. Dans cet exemple, vous noterez qu'il y a une possible répétition du test *« Est-ce cuit ? »*. Une telle construction est appelée *boucle* et est, comme on le verra plus loin, très utile pour répéter une séquence d'actions.

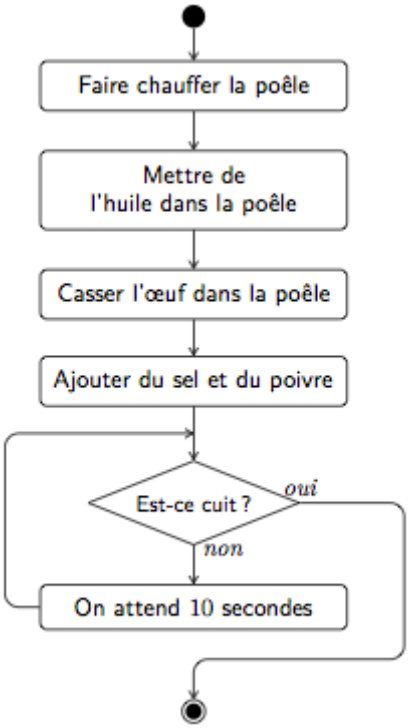

Figure 5. La réalisation d'une omelette peut aussi se représenter à l'aide d'un diagramme d'activités. Cet exemple comporte une boucle permettant de tester si la cuisson est terminée.

I-D - Historique et motivation

Les origines des *sciences informatiques*, et de la programmation en particulier, remontent à *Ada Lovelace* (Augusta Ada King, comtesse de Lovelace, est née le 10 décembre 1815 à Londres et est décédée à 36 ans, le 27 novembre 1852 à Marylebone), dont un portrait se trouve à la figure 6. Elle est connue pour avoir réalisé le premier programme informatique alors qu'elle travaillait sur la machine de Babbage, ancêtre de l'ordinateur. Elle fut ainsi la première à écrire un programme destiné à être exécuté par une machine. Plus tard, son nom a été utilisé pour nommer le langage de programmation *Ada*, conçu entre 1977 et 1983 pour le département de la Défense américain.

Figure 6. Ada Lovelace est connue pour avoir écrit le premier programme informatique, ce qui en fait la première programmeuse de l'histoire.

COBOL, l'un des premiers langages de programmation de haut niveau, c'est-à-dire proche de la langue naturelle, a été conçu en 1959 par l'amiral Grace Hopper, une informaticienne américaine. Depuis, les langages de programmation n'ont cessé d'évoluer, incluant des facilités pour les programmeurs en leur permettant d'exprimer le plus facilement possible les instructions qu'ils désirent exécuter. Dans ce cours, nous allons apprendre à utiliser le *langage Python*, troisième langage le plus utilisé au monde, en 2016 (**The 2016 Top Programming Languages**, par Stephen Cass, dans IEEE Spectrum (le 26 juillet 2016), consulté le 6 août 2016).

Pourquoi est-il important de savoir programmer ? Aujourd'hui, tout le monde dépend des technologies pour communiquer, s'informer, travailler, etc. L'informatique est partout : on s'en sert pour réserver son billet d'avion, faire ses courses, gérer son compte bancaire, etc. Malheureusement, la plupart des gens ne sont pas capables de comprendre les énormes quantités de code qui sont derrière toutes ces applications.

La programmation est le langage utilisé en informatique, tout comme les mathématiques peuvent être vues comme le langage des sciences. Il est dès lors important d'apprendre à comprendre et maitriser un minimum ce *langage de la technologie* si l'on souhaite comprendre le monde numérique dans lequel nous vivons. Tout le monde devrait dès lors être initié à la programmation, et d'autres catégories de personnes devraient en avoir un certain niveau de maitrise. C'est par exemple le cas des scientifiques et ingénieurs, qui seront amenés, tôt ou tard, dans leurs carrières professionnelles, à mettre les mains dans le code. Enfin, aujourd'hui, qu'on le veuille ou non, si on souhaite gagner de grandes quantités d'argent ou changer le monde, l'apprentissage de l'informatique est très certainement un bon départ. Regardez autour de vous et observez tous les projets innovants et originaux ; ils comportent très certainement tous une part non négligeable de code informatique…

Juste pour vous faire rêver, la figure 7 montre un nanosatellite développé par la NASA dans le cadre du projet *PhoneSat* (**le projet PhoneSat**). Ce dernier a été conçu à partir de cartes Arduino et de smartphones grand public,

ces derniers ayant un processeur plus puissant et plus de mémoire qu'un satellite classique moyen. Des compétences informatiques ont évidemment été nécessaires pour mener à bien ce projet.

Figure 7. Le projet PhoneSat, mené par la NASA, consiste à construire des nano-satellites à partir de smartphones et cartes Arduino, non modifiés, et à les placer en orbite basse.

II - Le langage Python

Pour pouvoir écrire des programmes, il faut d'abord choisir un *langage de programmation* qui permette de décrire les différentes opérations que le programme développé doit effectuer. Il existe des centaines de langages de programmation différents, chacun ayant ses spécificités propres. Ce cours a choisi *Python*, un langage récent et moderne utilisable pour coder une grande variété de programmes différents. Ce chapitre présente ce langage et introduit les concepts de base nécessaires à l'écriture d'un premier programme simple.

II-A - Python

Le langage de programmation *Python* a été créé par Guido van Rossum en 1990 et est rendu disponible sous licence libre. Son développement est aujourd'hui assuré par la *Python Software Foundation*, fondée en 2001. Il s'agit d'un langage interprété fonctionnant sur la plupart des plateformes informatiques (notamment Linux, Windows et macOS). Il est également très apprécié des pédagogues qui le considèrent comme étant un bon langage pour s'initier aux concepts de base de la programmation.

La figure 1 montre le logo de Python à gauche de son créateur, *Guido van Rossum*. En 1999, ce dernier a soumis un projet auprès de la *DARPA* (Defense Advanced Research Projects Agency), une agence du département de la Défense des États-Unis chargée du développement des nouvelles technologies destinées à un usage militaire. Dans sa proposition, appelée *« Computer Programming for Everybody »* (**document original**), il définit les buts suivants pour Python :

- le langage doit être simple et intuitif, et aussi puissant que les principaux concurrents ;
- il doit être open source, afin que quiconque puisse contribuer à son développement ;
- le code doit être aussi compréhensible qu'un texte en anglais ;
- il doit être adapté aux tâches quotidiennes, et permettre des courts délais de développement.

Figure 1. Le langage Python a été créé par Guido van Rossum en 1990, programmeur néerlandais qui a travaillé pour Google pendant sept ans. Il travaille maintenant pour Dropbox depuis janvier 2013.

Depuis sa création, le succès du langage n'a cessé de croitre, jusqu'à le porter parmi les dix langages de programmation les plus populaires. Il s'agit d'un *langage de haut niveau, à usage général*. Les dernières versions stables du langage sont la 2.7.12 (25 juin 2016) et la 3.5.2 (27 juin 2016). Ces deux versions, communément appelées Python 2 et Python 3, continuent toutes les deux d'exister, essentiellement pour des raisons de compatibilité. Dans ce cours, nous utiliserons la version 3.

Python est utilisé dans de nombreux projets, de types différents. Il est notamment utilisé comme langage de script pour des applications web. Il est également très présent dans des applications de calculs scientifiques, et est intégré dans de nombreux logiciels de modélisation, comme langage de script. Enfin, il est aussi utilisé dans le développement de jeux vidéo ou dans l'écriture de tâches dans le domaine de l'intelligence artificielle.

II-A-1 - Hello World!

Une fois n'est pas coutume, lorsqu'il s'agit de présenter la *syntaxe* d'un langage de programmation, on utilise comme exemple le programme *Hello World*. Il s'agit simplement d'un programme qui affiche les deux mots « Hello World » à l'écran, avant de se terminer.

Voici ce que cela donne en Python :

```
1. print('Hello World!')
```

Quoi de plus simple ? Python inclut une fonction prédéfinie print qui affiche à l'écran la séquence de caractères qu'on lui fournit. Cette unique ligne de code constitue notre premier programme Python.

II-A-1-a - Mode interactif et mode script

L'exécution d'un programme Python se fait à l'aide d'un *interpréteur*. Il s'agit d'un programme qui va traduire les instructions écrites en Python en *langage machine*, afin qu'elles puissent être exécutées directement par l'ordinateur. Cette traduction se fait à la volée, tout comme les interprètes traduisent en temps réel les interventions des différents parlementaires lors des sessions du parlement Européen, par exemple. On dit donc que Python est un langage *interprété*.

Il y a deux *modes d'utilisation* de Python. Dans le *mode interactif*, aussi appelé mode console, l'interpréteur vous permet d'encoder les instructions une à une. Aussitôt une instruction encodée, il suffit d'appuyer sur la touche ENTER pour que l'interpréteur l'exécute.

La figure 2 à la page suivante montre le programme *Hello World* en mode interactif. Les lignes qui commencent par >>> sont l'*invite de commande* qui vous propose d'encoder une instruction. Si cette dernière produit un résultat, il est affiché une fois l'instruction exécutée. Voici donc l'instruction qui a été encodée et le résultat qu'elle a produit :

```
1. >>> print('Hello World!')
2. Hello World!
```

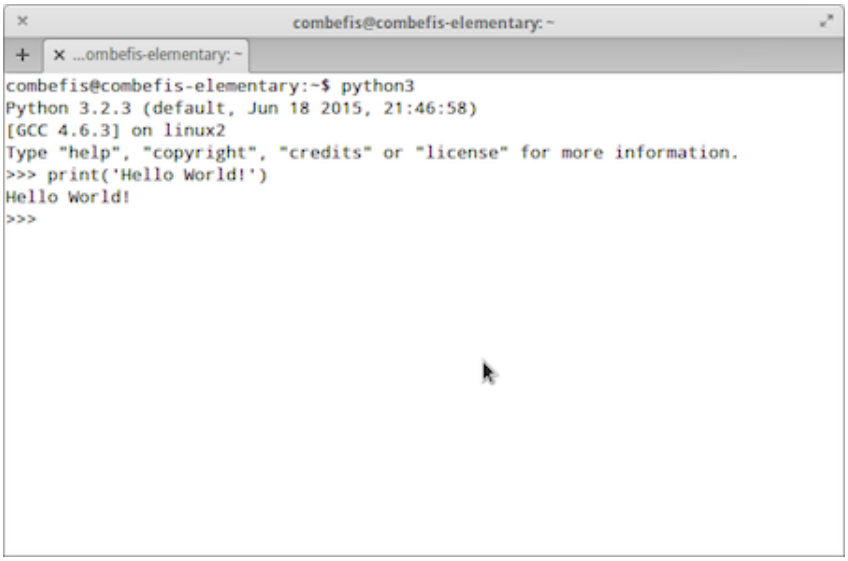

Figure 2. Dans le mode interactif, les instructions sont exécutées une à une par l'interpréteur Python, au fur et à mesure que vous les encodez.

Pour quitter le mode interactif, il suffit d'exécuter l'instruction `exit()`. Il s'agit de nouveau d'une fonction prédéfinie de Python permettant de quitter l'interpréteur.

Le mode interactif est très pratique pour rapidement tester des instructions et directement voir leurs résultats. Son utilisation reste néanmoins limitée à des programmes de quelques instructions. En effet, devoir à chaque fois retaper toutes les instructions s'avèrera vite pénible.

Il existe également le *mode script* où vous devez avoir préalablement écrit toutes les instructions de votre programme dans un fichier texte, et l'avoir enregistré sur votre ordinateur. On utilise généralement l'extension de fichier `.py` pour des fichiers contenant du code Python. Une fois cela fait, l'interpréteur va lire ce fichier et exécuter son contenu, instruction par instruction, comme si vous les aviez tapées l'une après l'autre dans le mode interactif. Les résultats intermédiaires des différentes instructions ne sont par contre pas affichés ; seuls les affichages explicites (avec la fonction `print`, par exemple) se produisent.

II-A-1-b - Calculatrice Python

On peut utiliser l'interpréteur Python en mode interactif comme une *calculatrice*. En effet, si vous y tapez une expression mathématique, cette dernière sera *évaluée* et son résultat affiché comme résultat intermédiaire. Rappelez-vous que cela ne fonctionne pas avec le mode script. Voici, par exemple, une succession d'instructions encodées en mode interactif, ainsi que les résultats produits :

```
>>> 42
42
>>> 2 * 12
24
>>> 2 ** 10
1024
>>> ((19.99 * 1.21) - 5) / 4
4.796975
```

Python propose plusieurs *opérateurs arithmétiques* repris dans la figure 3. Il y a évidemment les classiques : l'*addition* (`+`), la *soustraction* (`-`), la *multiplication* (`*`) et la *division* (`/`). On retrouve également l'*exponentiation* (`**`) qui permet d'élever un nombre à une certaine puissance. Enfin, il reste deux autres opérateurs (`//` et `%`), destinés au calcul avec des nombres entiers.

Description	Notation
Addition	+
Soustraction	-
Multiplication	*
Division	/
Exponentiation	**
Division entière	//
Reste de la division entière	%

II-A-1-b-i - Priorité

Les opérateurs arithmétiques possèdent chacun une *priorité* qui définit dans quel ordre les opérations sont effectuées. Par exemple, lorsqu'on écrit `1 + 2 * 3`, la multiplication va se faire avant l'addition. Le calcul qui sera effectué est donc `1 + (2 * 3)`. Dans l'ordre, l'opérateur d'exponentiation est le premier exécuté, viennent ensuite les opérateurs `*`, `/`, `//` et `%`, et enfin les opérateurs `+` et `-`.

Lorsqu'une expression contient plusieurs opérations de même priorité, ils sont évalués de gauche à droite. Ainsi, lorsqu'on écrit `1 - 2 - 3`, le calcul qui sera effectué est `(1 - 2) - 3`. En cas de doutes, vous pouvez toujours utiliser des parenthèses pour rendre explicite l'ordre d'évaluation de vos expressions arithmétiques.

II-A-1-b-ii - Division entière et son reste

Deux opérations arithmétiques sont exclusivement utilisées pour effectuer des calculs en nombres entiers : la *division entière* (`//`) et le *reste de la division entière* (`%`). Lorsqu'on divise un nombre entier D (appelé dividende) par un autre nombre entier d (appelé diviseur), on obtient deux résultats : un *quotient* q et un *reste* r, tels que :

$$D = q \times d + r \qquad (\text{avec } r < d).$$

La valeur q est le résultat de la division entière et la valeur r celui du reste de cette division. Par exemple, si on divise 17 par 5, on obtient un quotient de 3 et un reste de 2 puisque $17 = 3 \times 5 + 2$.

Ces deux opérateurs sont très utilisés dans plusieurs situations précises. Par exemple, pour déterminer si un nombre entier est pair ou impair, il suffit de regarder le reste de la division entière par deux. Le nombre est pair s'il est nul et est impair s'il vaut 1. Une autre situation où ces opérateurs sont utiles concerne les calculs de temps. Si on a un nombre de secondes et qu'on souhaite le décomposer en minutes et secondes, il suffit de faire la division par 60. Le quotient sera le nombre de minutes et le reste le nombre de secondes restant. Par exemple, 175 secondes correspond à `175 // 60` $= 2$ minutes et `175 % 60` $= 55$ secondes.

II-B - Variable et type de donnée

On souhaite parfois conserver en mémoire le résultat de l'évaluation d'une expression arithmétique en vue de l'utiliser plus tard, dans un autre calcul, par exemple. Si on reprend l'exemple de la recherche des racines du trinôme $ax^2 + bx + c$, on doit mémoriser la valeur du discriminant pour pouvoir calculer les valeurs des deux racines réelles distinctes, lorsqu'il est strictement positif.

II-B-1 - Variable

Une *variable* peut être vue comme une boite (virtuelle) représentant un emplacement en mémoire qui permet de stocker une *valeur*, et à qui on a donné un *nom* afin de facilement l'identifier. La figure 4 illustre ce concept en montrant une variable dont le nom est data et dont la valeur est le nombre entier 42.

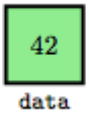

Figure 4. Une variable est une boite virtuelle, représentant un emplacement mémoire, qui possède un nom et une valeur.

Voyons maintenant un exemple de programme qui utilise plusieurs variables pour calculer le discriminant du trinôme $x^2 + 2x - 4$:

```
1. a = 1
2. b = 2
3. c = -4
4.
5. delta = b ** 2 - 4 * a * c
6. print('Le discriminant est :')
7. print(delta)
```

Les trois premières instructions stockent les coefficients du trinôme, à savoir 1, 2 et -4, respectivement dans des variables nommées a, b et c. La quatrième instruction effectue le calcul du discriminant ($b^2 - 4ac$) et stocke le résultat dans la variable delta. Enfin, les deux dernières instructions affichent à l'écran la phrase « Le discriminant est : », suivie de la valeur de la variable delta.

L'utilisation d'une variable est donc assez immédiate. Deux types d'accès peuvent être réalisés :

- Pour *affecter une valeur* à une variable, c'est-à-dire l'initialiser ou modifier sa valeur, on utilise l'*opérateur d'affectation* (=). À gauche de l'opérateur, on retrouve le nom de la variable et à droite la valeur qu'on souhaite lui affecter.
- Pour *accéder au contenu* d'une variable, il suffit d'utiliser son nom. Cet accès peut, par exemple, se faire dans une expression mathématique, ou avec la fonction print.

Le symbole = est à distinguer du signe égal utilisé en mathématiques pour énoncer une égalité. En Python, il permet d'affecter une valeur à une variable. Voyons cela avec l'exemple suivant :

```
1. a = 1
2. b = a
3. print(a)
4. print(b)
5.
6. a = 15
7. print(a)
8. print(b)
```

En mathématiques, écrire $b = a$ signifie que a et b représentent la même valeur. Dès lors, lorsqu'on écrit plus loin a = 15, on pourrait s'attendre à ce que a et b valent 15. En Python, b = a signifie qu'on affecte la valeur de a à la variable b (on fait une copie de la valeur). Après exécution de cette instruction, aucun lien n'est établi entre les deux variables. L'exécution du programme affiche lors de son exécution :

```
1
```

```
1
15
1
```

La figure 5 illustre ce qui se passe en mémoire. Elle montre l'état de cette dernière respectivement juste après l'exécution des instructions a = 1, b = a et a = 15. On voit très bien qu'aucun lien n'a été établi entre les variables a et b et que la dernière modification de a n'a aucun impact sur la valeur de b.

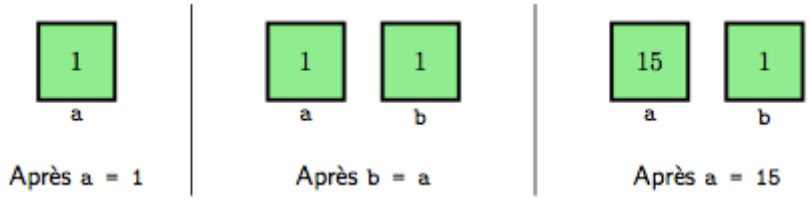

Figure 5. L'opérateur d'affectation permet d'affecter une valeur à une variable. Elle n'établit aucun lien entre deux variables comme on peut le voir avec l'état de la mémoire respectivement après exécution des instructions a = 1, b = a et a = 15.

II-B-1-a - Initialisation de variable

Avant de pouvoir accéder au contenu d'une variable, il faut qu'elle soit *initialisée*, c'est-à-dire qu'elle doit posséder une valeur. Si vous tentez d'utiliser une variable non initialisée, l'exécution du programme va s'arrêter et l'interpréteur Python va produire une *erreur d'exécution*. Voyons cela avec l'exemple de programme suivant :

```
1. height = 178
2. print('Sa taille est :')
3. print(heigth)
```

Le programmeur a fait une malencontreuse faute de frappe à la troisième instruction, et lors de son exécution, l'interpréteur s'arrête et produit une erreur :

```
Sa taille est :
Traceback (most recent call last):
  File "program.py", line 3, in <module>
    print(heigth)
NameError: name 'heigth' is not defined
```

L'avant-dernière ligne reprend l'instruction qui a causé l'erreur d'exécution (à savoir print(heigth) dans notre cas). La dernière ligne fournit une explication sur la cause de l'erreur (celle qui commence par *NameError*). Dans cet exemple, elle indique que le nom heigth n'est pas défini, c'est-à-dire qu'il ne correspond pas à une variable initialisée.

II-B-1-a-i - Modification de variable

Une fois une variable initialisée, on peut donc modifier sa valeur en utilisant de nouveau l'*opérateur d'affectation* (=). La valeur actuelle de la variable est remplacée par la nouvelle valeur qu'on lui affecte. Dans l'exemple suivant, on initialise une variable à la valeur 12 et on remplace ensuite sa valeur par 99 :

```
1. data = 12
2. data = 99
3. print(data)
```

On voit bien sûr le résultat de l'exécution que la valeur de la variable data a été remplacée par 99 et qu'il n'y a plus aucune trace du 12 :

```
99
```

II-B-1-a-ii - Affectation composée

Ce que l'on doit parfois faire, c'est mettre à jour une variable par rapport à la valeur qu'elle possède actuellement. Par exemple, si on veut augmenter la valeur d'une variable data de 1, on doit écrire :

```
1. data = data + 1
```

La nouvelle valeur à affecter à la variable data est sa valeur actuelle à qui on ajoute 1. Dans cette instruction, le nom de la variable est répété deux fois. En programmation, une règle de bonne pratique consiste à éviter au maximum la duplication de code. Pour simplifier cette instruction, Python propose l'opérateur += qui fait l'addition suivie de l'affectation en une fois. L'exemple précédent se réécrit donc comme suit :

```
1. data += 1
```

Un tel opérateur permet de réaliser une *affectation composée*, c'est-à-dire une opération directement suivie d'une affectation. On peut notamment l'utiliser avec les sept opérateurs arithmétiques que l'on vient de voir.

II-B-1-b - Type de donnée

Un programme passe une grande partie de son temps à manipuler des *données*. On a déjà pu voir plusieurs types de données différents dans les sections précédentes, à savoir des nombres et des mots. Il existe de nombreux types prédéfinis en Python, parmi lesquels les nombres et les chaines de caractères.

II-B-1-b-i - Nombre

Il y a trois *types numériques* en Python :

- Le type *entier* (int) permet de représenter n'importe quel nombre entier, peu importe sa taille.
- Le type *flottant* (float) permet de représenter des nombres comportant une partie décimale, compris entre

  $$10^{-308}$$

  et

  $$10^{308}$$

  . La valeur spéciale math.inf représente l'infini.
- Le type *complexe* (complex) permet de représenter des nombres complexes, où le nombre imaginaire se note j.

Il y a deux manières d'introduire de telles données dans un programme. On peut directement écrire le nombre, qu'on appellera *littéral* entier. On obtient également des données numériques à partir d'expressions mathématiques et de fonctions prédéfinies. Deux points d'attention sont à soulever concernant l'écriture littérale de nombres :

- On utilise le point comme séparateur décimal lorsqu'on doit écrire des nombres à virgule.
- On colle directement la partie imaginaire d'un nombre complexe au j sans utiliser un *. En effet, 2j représente le nombre complexe

  $$2i$$

  tandis que 2 * j représente le produit de

  $$2$$

  par la variable j.

Comme le montre l'exemple suivant, les flottants peuvent être écrits en notation scientifique. Il s'agit d'un programme qui calcule les deux racines complexes d'un trinôme du second degré :

```
1. from cmath import sqrt
2.
3. a = 1
4. b = -4
5. c = 2e2
6.
7. delta = b ** 2 - 4 * a * c
8. x1 = (-b + sqrt(delta)) / (2 * a)
9. x2 = (-b - sqrt(delta)) / (2 * a)
10.
11. print('Les deux racines sont :')
12. print(x1)
13. print(x2)
```

La première instruction importe la fonction prédéfinie sqrt depuis le module cmath, afin de calculer la racine carrée d'un nombre complexe. On initialise ensuite trois variables a, b et c avec des données numériques décrites par un littéral (1, -4 et $2 \cdot 10^2$). La cinquième instruction calcule le discriminant qui sera un nombre flottant ($\Delta = b^2 - 4ac$).

Les deux instructions suivantes calculent les racines du trinôme, en faisant notamment appel à la fonction sqrt pour calculer $x_{1,2} = (-b \pm \sqrt{\Delta})/2a$. Les variables x1 et x2 vont donc contenir des données de type complexe. Comme on peut le voir sur ce qu'affiche l'exécution du programme, c'est bien le cas :

```
Les deux racines sont :
(2+14j)
(2-14j)
```

II-B-1-b-ii - Chaine de caractères

Une *chaine de caractères* (str) est une séquence de caractères, délimitée par des guillemets dans sa forme littérale (simple ' ou double "). On peut généralement utiliser indifféremment l'un ou l'autre type de guillemets.

Tous les caractères, à quelques exceptions près, sont acceptés pour définir une chaine de caractères. Tout d'abord, il est évident que si vous souhaitez utiliser le caractère ", vous ne pourrez pas l'utiliser pour délimiter la chaine de caractères, mais vous devrez utiliser ', et inversement.

Dans l'exemple suivant, on est obligé d'utiliser des guillemets doubles pour délimiter la chaine de caractères :

```
1. address = "Promenade de l'Alma 50"
```

Si on veut créer des chaines de caractères contenant à la fois ' et ", il faut utiliser une *séquence d'échappement*. Une telle séquence commence par un backslash (\) suivi d'un ou plusieurs caractères. Elle permet de représenter un caractère spécial. Par exemple, la séquence \" représente le caractère " et la séquence \n représente un saut de ligne. On peut par exemple écrire :

```
1. address = "Promenade de l'Alma 50\n1200 Woluwé-Saint-Lambert"
2. print(address)
```

L'exécution de ce programme affiche deux lignes de texte à l'écran, la séquence d'échappement \n ayant été remplacée par un saut de ligne :

```
Promenade de l'Alma 50
1200 Woluwé-Saint-Lambert
```

La figure 6 ci-dessous reprend les principales séquences d'échappement que propose Python.

Séquence	Description
\\	Backslash
\'	Guillemet simple (apostrophe)
\"	Guillemet double
\n	Saut de ligne
\r	Retour chariot
\t	Tabulation horizontale

Il faut faire bien attention à distinguer les deux littéraux 123 et '123'. Alors que le premier est de type numérique et représente le nombre entier 123, le second est une chaine de caractères constituée des trois caractères 1, 2 et 3, placés l'un après l'autre.

On peut s'en rendre compte grâce à la fonction prédéfinie type qui permet d'obtenir le *type d'une expression*. On voit, sur l'exemple suivant, les mentions <class 'int'> et <class 'str'> qui indiquent respectivement le type numérique entier et le type chaine de caractères :

```
1. >>> type(123)
2. <class 'int'>
3. >>> type('123')
4. <class 'str'>
```

On peut connaitre la *longueur* d'une chaine de caractères (c'est-à-dire le nombre de caractères qui la composent) en utilisant la fonction prédéfinie len. Enfin, on peut joindre deux chaines de caractères l'une derrière l'autre avec l'*opérateur de concaténation* (+). Dans l'exemple suivant, la variable s contiendra Hello World! après exécution :

```
1. a = 'Hello'
2. b = " "
3. c = 'World!'
4. s = a + b + c
```

Attention, cet opérateur ne fonctionne qu'entre deux données de type chaine de caractères. Si vous tentez de l'utiliser avec un autre type, l'interpréteur Python produira une erreur d'exécution.

Le programme suivant initialise une variable de type entier contenant l'année actuelle et définit ensuite une variable s à laquelle on tente d'affecter la *concaténation* de la chaine de caractères 'Nous sommes en ' avec la variable de type entier year :

```
1. year = 2016
2. s = 'Nous sommes en ' + year
```

L'interpréteur Python va générer une erreur d'exécution qui signale qu'il ne parvient pas à convertir implicitement la donnée de type int en une donnée de type str :

```
Traceback (most recent call last):
  File "program.py", line 2, in <module>
    s = 'Nous sommes en ' + year
TypeError: Can't convert 'int' object to str implicitly
```

II-B-1-c - Conversion

Il est possible de *convertir explicitement* une donnée d'un type vers un autre, en utilisant des fonctions prédéfinies. Les fonctions int, float, complex et str permettent de convertir une donnée vers les types correspondants. Voyons un premier exemple :

```
1. a = 12
2. print(type(a))
```

```
3.
4. b = complex(a)
5. print(type(b))
```

La variable a est de type numérique entier. On la convertit ensuite en une donnée de type complexe grâce à la fonction prédéfinie complex, et on stocke le résultat de la conversion dans la variable b. Ceci est confirmé par l'exécution du programme qui affiche :

```
<class 'int'>
<class 'complex'>
```

Grâce à ces fonctions de conversion, on peut maintenant corriger l'exemple de la section précédente. Il suffit en effet d'appliquer la fonction str à la variable year avant d'effectuer la concaténation :

```
1. year = 2016
2. s = 'Nous sommes en ' + str(year)
```

Toutes les conversions ne fonctionnent évidemment pas. Si on tente, par exemple, de convertir une chaine de caractères en un entier, mais qu'elle ne représente pas un nombre entier, on aura une erreur d'exécution :

```
1. a = int("Hello")
2. print(a)
```

Le message d'erreur obtenu signale que la chaine de caractères 'Hello' n'est pas un entier valable en base 10 :

```
Traceback (most recent call last):
  File "program.py", line 1, in <module>
    a = int("Hello")
ValueError: invalid literal for int() with base 10: 'Hello'
```

II-C - Fonction prédéfinie

On a déjà pu rencontrer des *fonctions prédéfinies* à plusieurs reprises. Ces fonctions proviennent de deux sources différentes ; ce sont :

- soit des fonctions faisant partie intégrante du langage Python comme print, len et str, par exemple ;
- soit des fonctions ayant été écrites par d'autres programmeurs et mises à disposition en important le module qui les contient, comme la fonction sqrt du module cmath, par exemple.

II-C-1 - Fonction print

La *fonction* print permet d'afficher une donnée, peu importe son type. En réalité, la fonction affiche une représentation de la donnée sous forme d'une chaine de caractères. En plus, elle ajoute automatiquement un retour à la ligne, ainsi, deux appels successifs à la fonction print résulteront en l'affichage de deux lignes.

La forme générale de la fonction print permet d'afficher un nombre quelconque de valeurs, que l'on doit séparer par des virgules. Par défaut, les différentes valeurs seront séparées par une espace (lorsqu'il désigne le caractère d'espacement blanc, le nom espace est féminin).

L'exemple suivant affiche la phrase « Né en 1961 j'ai 55 ans. » :

```
1. year = 2016
2. birthyear = 1961
3. age = year - birthyear
4.
5. print('Né en', birthyear, "j'ai", age, 'ans.')
```

Remarquez l'alternance entre guillemets simples et doubles pour délimiter les chaines de caractères, contrainte par le fait que la deuxième chaine de caractères contient un guillemet simple. On aurait également pu utiliser des guillemets doubles partout, évidemment.

Par défaut, la fonction print ajoute donc un retour à la ligne après ce qu'elle doit afficher, et elle sépare les différentes valeurs à afficher avec une espace. On peut changer cette configuration par défaut à l'aide des *paramètres nommés*end et sep. Avec ces derniers, on peut spécifier les chaines de caractères à afficher respectivement après le contenu et entre les valeurs.

L'exemple suivant modifie ces deux valeurs par défaut :

```
1. day = 4
2. month = 8
3. year = 1961
4.
5. print('Né le :', end=' ')
6. print(day, month, year, sep='/')
```

Le premier appel à la fonction print va donc afficher la chaine de caractères « Né le : » et terminera par une espace, au lieu d'un retour à la ligne. Le second appel affichera la valeur des trois variables, mais en les séparant par le caractère / au lieu d'une espace :

```
Né le : 4/8/1961
```

On peut évidemment spécifier les deux paramètres nommés end et sep dans le même appel à la fonction print.

II-C-1-a - Sortie formatée

Au lieu de fournir plusieurs valeurs à la fonction print, on aurait également pu construire la chaine de caractères en utilisant la concaténation. Rappelez-vous néanmoins qu'il faudra convertir toutes les valeurs concaténées en chaines de caractères pour que cela fonctionne. L'exemple précédent peut se réécrire comme suit :

```
1. print('Né en ' + str(birthyear) + " j'ai " + str(age) + ' ans.')
```

En fait, on veut construire une chaine de caractères dans laquelle on incruste, à des endroits précis, les valeurs de variables. Python propose des facilités pour *formater* une chaine de caractères. Voyons d'abord comment réécrire l'exemple précédent avant de comprendre dans le détail le fonctionnement du formatage d'une chaine de caractères :

```
1. print("Né en {} j'ai {} ans".format(birthyear, age))
```

On commence donc par construire la chaine de caractères, dans laquelle on place des balises représentées par {}. Il s'agit d'endroits dans la chaine de caractères qui seront remplacés par des valeurs. Après la chaine de caractères, on va devoir indiquer les valeurs à incruster à l'aide de format. Le tout produira une chaine de caractères qui sera affichée par print. Notez qu'on peut également construire une chaine de caractères ainsi, sans nécessairement vouloir l'afficher. De plus, on n'est pas limité à l'incrustation de valeurs de variables, mais on peut insérer n'importe quelle expression. On peut donc, par exemple, écrire :

```
1. s = "Né en {} j'ai {} ans".format(birthyear, 2016 - birthyear)
```

La figure 7 résume cette construction d'une chaine de caractères formatée. Il suffit (1) de placer des balises dans une chaine de caractères et (2) de fournir autant d'expressions que nécessaire à format afin de les remplacer par les valeurs de ces expressions.

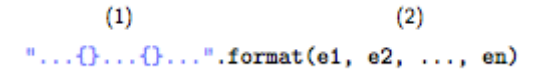

Figure 7. On peut construire une chaine de caractères en y incrustant des valeurs à la place de balises en utilisant format.

II-C-1-b - Fonction input

Alors que la fonction print permet à un programme de produire une sortie, on va pouvoir lire une entrée grâce à la *fonction* input. Cette fonction, une fois appelée, arrête l'exécution du programme et attend que l'utilisateur saisisse un texte. Ce dernier est ensuite rapatrié et peut, par exemple, être stocké dans une variable. Voici un programme qui vous demande votre nom, puis vous souhaite la bienvenue :

```
1. firstname = input('Quel est ton prénom ? ')
2. print('Bonjour', firstname, 'et bienvenue !')
```

L'exécution de ce programme pourrait, par exemple, produire le résultat suivant, sachant que le mot « Sébastien » a été saisi par l'utilisateur :

```
Quel est ton prénom ? Sébastien
Bonjour Sébastien et bienvenue !
```

La fonction input renvoie toujours le texte saisi par l'utilisateur sous forme d'une chaine de caractères. Il faudra donc éventuellement prévoir une conversion, si vous voulez que l'utilisateur saisisse autre chose, comme un nombre entier, par exemple.

La suite du programme d'exemple demande à l'utilisateur quelle est son année de naissance, puis calcule et affiche son âge :

```
1. s = input('Quel est ton année de naissance ? ')
2.
3. year = 2016
4. birthyear = int(s)
5. print('Tu as', year - birthyear, 'ans.')
```

Par contre, si l'utilisateur ne saisit pas un nombre, une erreur se produira lors de l'exécution comme on peut le voir ci-dessous, où l'utilisateur a saisi le texte « deux-mille » :

```
Quel est ton année de naissance ? deux-mille
Traceback (most recent call last):
  File "program.py", line 4, in <module>
    birthyear = int(s)
ValueError: invalid literal for int() with base 10: 'deux-mille'
```

On verra plus loin dans le livre comment gérer ce type d'erreur et pouvoir, par exemple, afficher un message à l'utilisateur.

II-C-1-c - Importation de fonction

Les deux fonctions que l'on vient de voir font partie intégrante du langage. On s'intéresse maintenant aux *fonctions mathématiques* qui sont essentiellement rassemblées dans les *modules* math et cmath, le deuxième étant spécialisé pour les nombres complexes.

Pour utiliser des fonctions faisant partie d'un *module*, il faut avant tout les *importer*. Comme on a déjà pu le voir, on utilise pour cela l'*instruction* from/import à qui on indique le nom du module après le from et la liste des fonctions à importer après le import.

Voici un exemple qui utilise les fonctions sqrt (racine carrée) et sin (sinus) faisant partie du module math :

```
1. from math import sqrt, sin
2.
3. print('Racine carrée de 2 =', sqrt(2))
4. print('Sinus de 2 = ', sin(2))
```

Parfois, il est plus facile d'importer tout le contenu d'un module plutôt que de lister toutes les fonctions dont on a besoin. Pour cela, on utilise un astérisque (*) au lieu de la liste de fonctions. Pour importer toutes les fonctions du module math, il suffit donc d'écrire :

```
1. from math import *
```

De manière générale, on essaie d'éviter autant que possible de brutalement tout importer, afin d'éviter d'éventuels conflits. Prenons pour cela un exemple qui utilise les deux modules math et cmath :

```
1. from cmath import *
2. from math import *
3.
4. print('Racine carrée de -4 =', sqrt(-4))
```

Les deux modules contiennent une fonction sqrt. Dès lors, laquelle sera utilisée à la troisième instruction ? Python va simplement prendre celle du dernier module importé, c'est-à-dire celle du module math. Par conséquent, une erreur d'exécution se produira puisqu'il faut la version complexe pour calculer la racine carrée de -4 :

```
Traceback (most recent call last):
  File "program.py", line 4, in <module>
    print('Racine carrée de -4 =', sqrt(-4))
ValueError: math domain error
```

Pour résoudre ce problème, Python permet de juste importer un module avec l'instruction import suivie du nom du module à importer. Ensuite, lorsqu'on voudra utiliser une fonction du module, il faudra faire précéder son nom de celui du module et de l'*opérateur d'accès* (.) :

```
1. import cmath
2. import math
3.
4. print('Racine carrée de -4 =', cmath.sqrt(-4))
```

Il est maintenant explicite que la fonction sqrt appelée est celle provenant du module cmath, et il n'y aura donc plus d'erreurs d'exécution.

II-D - Code source

Maintenant que l'on a pu découvrir nos premiers programmes écrits en Python, il est temps de prendre un peu de recul en analysant la rédaction d'un code source. En effet, tout comme on n'écrit pas n'importe comment un texte en français, il y a des règles de bonne pratique pour rédiger un code source correct et de qualité.

II-D-1 - Règle de nommage des variables

On a vu qu'il fallait choisir un *nom* pour toute variable, et il y a des règles concernant les noms valides. Tout d'abord, on ne peut pas utiliser certains mots appelés *mots réservés*. Il s'agit de mots qui ont une signification particulière en Python, repris à la figure 8.

```
False   class     finally   is        return
None    continue  for       lambda    try
True    def       from      nonlocal  while
and     del       global    not       with
as      elif      if        or        yield
assert  else      import    pass
break   except    in        raise
```

Figure 8. Il y a en tout 33 mots réservés qui ont une signification particulière en Python et qui ne peuvent donc pas être utilisés comme nom de variable.

Ensuite, le premier caractère doit être une lettre minuscule (a-z), une lettre majuscule (A-Z) ou un tiret de soulignement (_). Pour les autres caractères, on autorise en plus les chiffres (0-9). On peut définir des noms aussi longs que l'on veut (même si c'est recommandé de rester concis), et Python est *sensible à la casse*, c'est-à-dire qu'il fait la différence entre minuscules et majuscules. Comme le montre la figure 9, on peut également utiliser une série de caractères Unicode depuis Python 3.

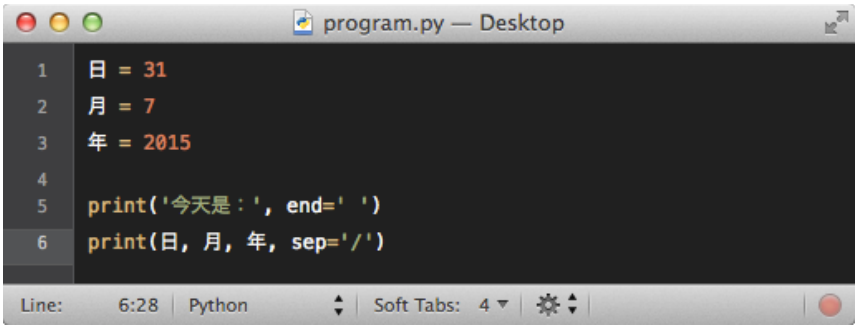

Figure 9. Depuis Python 3, on peut également utiliser des caractères Unicode dans les noms de variable, comme des idéogrammes chinois, par exemple.

II-D-1-a - Commentaire

Lorsqu'on écrit un programme, il est parfois utile d'y ajouter des informations textuelles à destination des êtres humains. Ce genre d'information est appelé *commentaire* et on en ajoute dans un code source à l'aide du caractère #. Tout ce qui suit ce caractère, jusque la fin de la ligne, sera ignoré par l'interpréteur Python.

Le but d'un bon commentaire est d'apporter une information additionnelle au code. Il peut s'agir d'informations liées au problème résolu par le programme ou d'informations sur le programme (auteurs, version, etc.). On pourrait par exemple réécrire le programme qui cherche les racines d'un trinôme du second degré comme suit :

```
1. # Programme de calcul des racines d'un trinôme du second degré
2. # de la forme ax^2 + bx + c
3. # Auteur : Sébastien Combéfis
4. # Version : 7 aout 2015
5.
6. from cmath import sqrt
7.
8. # Coefficients du trinôme
9. a = 1
10. b = -4
11. c = 2e2
12.
13. # Calcul du discriminant
14. delta = b ** 2 - 4 * a * c
15.
16. # Calcul des deux racines
17. x1 = (-b + sqrt(delta)) / (2 * a)
18. x2 = (-b - sqrt(delta)) / (2 * a)
19.
20. print('Les deux racines sont :', x1, 'et', x2)
```

II-D-1-b - Style

Lorsqu'on utilise un langage de programmation, il est très important de suivre les *conventions* et constructions spécifiques du langage. Cela permet de facilement partager du code avec la communauté, et également de facilement comprendre du code écrit par d'autres.

En un mot, vous devez apprendre à écrire avec un *style pythonique*. On peut trouver de nombreuses références à ce propos sur Internet, le point de départ étant le *PEP 0008 — Style Guide for Python code* (**document officiel**) écrit par Guido van Rossum.

Voici les règles du PEP 0008 que l'on peut déjà comprendre :

- la longueur des lignes de code ne devrait pas excéder 79 caractères ;
- on peut utiliser des lignes vides pour séparer des blocs logiques de code, mais avec parcimonie ;
- les fichiers .py devraient être enregistrés en UTF-8 ;
- il faudrait utiliser une instruction import par module ;
- concernant les espaces :
    - on évite de placer des espaces après une parenthèse ouvrante, avant une parenthèse fermante, avant la parenthèse ouvrante d'un appel de fonction ;
    - il faut insérer une espace avant et après l'opérateur d'affectation, et également autour des opérateurs arithmétiques, sauf éventuellement pour faire ressortir les priorités des opérateurs (on écrit par exemple x*x + y*y) ;
    - on ne met pas d'espaces autour du = des paramètres nommés lors d'un appel de fonction ;
- les commentaires devraient être des phrases complètes, commençant donc par une majuscule et finissant par un point, et suivre *Strunk and White* (Strunk, W., Jr.; White, E.B. (1999). *The Elements of Style*. Longman. ISBN 978-0-205-30902-3) lorsqu'ils sont en anglais ;
- il faut éviter les lettres l (L minuscule), O (O majuscule) et I (i majuscule) pour nommer ses variables.

III - Instruction de contrôle

Dans tout programme, à un moment donné, il faut rompre l'exécution séquentielle des instructions, comme on a pu le voir dans les exemples intuitifs du premier chapitre. Ce chapitre présente deux types d'instruction permettant de briser cette exécution linéaire. Les *instructions conditionnelles* permettent de n'exécuter une partie du programme que si une condition donnée est satisfaite. Les *instructions répétitives* permettent de répéter un certain nombre de fois une portion de code.

III-A - Type booléen

On connait déjà deux types de données : les nombres (entiers, flottants et complexes) et les chaines de caractères. Voyons maintenant le *type booléen* dont les données ne peuvent prendre que deux valeurs différentes : vrai et faux. Un intérêt de ce type de donnée est de pouvoir prendre des décisions dans un programme en définissant des *conditions* qui seront satisfaites ou non.

En Python, on peut utiliser les mots réservés True et False qui représentent les deux valeurs booléennes possibles. Ces deux valeurs sont du type bool comme on peut le vérifier avec la fonction type.

On peut, par exemple, écrire :

```
1. v = True
2.
3. print(v)
4. print(type(v))
```

L'exécution de ce programme affiche :

```
True
<class 'bool'>
```

III-A-1 - Opérateur de comparaison

On ne va évidemment pas se contenter des littéraux True et False, mais on va vouloir construire des *expressions booléennes*. Pour cela, une première façon de faire consiste à utiliser un *opérateur de comparaison*. Il en existe au total six en Python, repris à la figure 1.

Description	Notation
Égal	==
Différent	!=
Strictement plus petit	<
Plus petit ou égal	<=
Strictement plus grand	>
Plus grand ou égal	>=

Ces opérateurs permettent de comparer deux valeurs et de tester si elles sont égales ou différentes, ou de savoir si l'une est (strictement) plus petite ou plus grande que l'autre. L'exemple suivant illustre l'utilisation de quelques-uns de ces opérateurs :

```
1. a = 12 == 3 * 4        # a vaut True
2. b = "Eat" > "Drink"    # b vaut True
3. c = a != b             # c vaut False
```

La première expression compare deux nombres entiers. La valeur de 12 étant égale à la valeur de l'expression 3 * 4, la variable a reçoit la valeur True. La deuxième expression compare deux chaines de caractères. Comme Eat est strictement plus grand que Drink (il se trouve après dans l'ordre alphabétique), la variable b reçoit la valeur True. Enfin, la troisième expression compare deux booléens. La valeur de la variable a n'étant pas différente de la valeur de la variable b, la variable c reçoit la valeur False.

De manière générale, on ne peut comparer que des valeurs qui sont du même type. Pour comparer deux valeurs de types différents, il faut avant tout effectuer des conversions explicites. Dans l'exemple suivant, qui compare un nombre entier et une chaine de caractères, la première instruction produit une erreur tandis que la seconde, où une conversion explicite en chaine de caractères a eu lieu, est valide :

```
>>> 19 > 'Hello'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() > str()
>>> str(19) > 'Hello'
False
```

III-A-1-a - Opérateur logique

Une seconde façon de construire des expressions booléennes consiste à en combiner plusieurs pour en construire une nouvelle, en utilisant des *opérateurs logiques*. Python propose trois opérateurs logiques repris à la figure 2.

Description	Notation
« non » logique	not
« et » logique	and
« ou » logique	or

Le premier opérateur logique permet d'inverser la valeur d'une expression booléenne. Appliquée à une expression booléenne qui vaut True, elle produit donc comme valeur False, et inversement :

```
1. a = not 8 > 2          # a vaut False
```

La valeur de l'expression 8 > 2 valant True, puisque 8 est strictement plus grand que 2, la valeur de l'expression not 8 > 2 vaut donc False.

Les deux autres opérateurs logiques combinent deux expressions booléennes. Le résultat du « et » logique ne vaut True que si les deux expressions combinées valent True, et il vaut False sinon. Le résultat du « ou » logique vaut True si au moins une des deux expressions combinées vaut True, et il vaut False sinon.

La figure 3 résume les résultats de ces deux opérateurs, sous la forme d'un tableau appelé *table de vérité*. Une telle table fournit la valeur d'expressions booléennes composées pour toutes les combinaisons possibles des expressions booléennes sur base desquelles elles sont construites.

a	b	a and b	a or b
False	False	False	False
False	True	False	True
True	False	False	True
True	True	True	True

Voici deux exemples d'expressions booléennes complexes construites avec des opérateurs logiques :

```
1. a = 8 > 2 and 12 <= 4                # a vaut False
2. b = 5 != 5 or 'PY' == 'P' + 'Y'      # b vaut True
```

Dans la première expression, la valeur de l'expression à gauche de l'opérateur and est True puisque 8 est strictement plus grand que 2, mais celle de l'expression à droite vaut False puisque 12 n'est pas plus petit ou égal à 4. Dès lors, la variable a reçoit la valeur False puisque les deux expressions combinées par l'opérateur logique and ne valent pas toutes les deux True.

Pour la seconde expression, l'expression à gauche de l'opérateur or vaut False puisque 5 n'est pas différent de 5, et celle de l'expression à droite vaut True puisque la chaine de caractères PY est bien égale à la concaténation des chaines P et Y. Puisqu'au moins une des deux expressions combinées par le or vaut True, la variable b vaut True.

III-A-1-a-i - Propriété de court-circuit

Lorsque l'interpréteur Python évalue une expression booléenne contenant des opérateurs and et or, il le fait de manière *fainéante*, en court-circuitant parfois l'expression à droite de l'opérateur logique. Ce qui se passe, c'est que parfois, en connaissant la valeur de l'expression à gauche de l'opérateur, peu importe celle de l'expression de droite, le résultat final peut déjà être connu.

Si on regarde attentivement les tables de vérité, on se rend compte que si la valeur de a est False, alors a and b vaut False peu importe la valeur de b. De même, si la valeur de a est True, alors a or b vaut True peu importe la valeur de b. Voici deux exemples illustrant cette *propriété de court-circuit* :

```
1. a = False and c        # a vaut False, peu importe la valeur de c
2. b = True or c          # b vaut True, peu importe la valeur de c
```

III-A-1-b - Priorité

Tout comme les opérateurs arithmétiques vus précédemment, les opérateurs de comparaison et les opérateurs logiques possèdent un certain niveau de priorité. Ces opérateurs sont effectués après tout ceux qu'on a déjà rencontrés, dans l'ordre suivant :

- les quatre comparaisons : `<`, `<=`, `>` et `>=` ;
- les égalités : `==` et `!=` ;
- le « non » logique : `not` ;
- le « et » logique : `and` ;
- le « ou » logique : `or`.

De nouveau, on peut utiliser les parenthèses pour rendre explicite l'ordre dans lequel vous souhaitez qu'une expression soit évaluée. La figure 4 montre quelques exemples d'expressions ainsi que l'expression complètement parenthésée équivalente.

Expression	Complètement parenthésée
1 + 2 <= 3 + 4	(1 + 2) <= (3 + 4)
1 - 2 - 3 == 4 + 5 * 6	((1 - 2) - 3) == (4 + (5 * 6))
not 7 + 3 == 10	not ((7 + 3) == 10)
not 9 > 10 or 1 == 1	(not (9 > 10)) or (1 == 1)
True and 4 < 12 and 1 != -1	(True and (4 < 12)) and (1 != -1)
12 < 1 and False or 4 == 1 + 2	((12 < 1) and False) or (4 == (1 + 2))

III-A-1-c - Enchainement de comparaisons

Enfin, on peut *enchainer* les opérateurs de comparaison pour effectuer deux opérations de comparaison qui ont en commun une valeur. Prenons, par exemple, une variable `x` dont on voudrait savoir si sa valeur est comprise entre 0 et 20. Pour cela, on peut écrire :

```
1. check = 0 <= x <= 20
```

Ce que l'interpréteur va faire, c'est effectuer les comparaisons une à une, les combinant avec un « et » logique. L'instruction ci-dessus équivaut donc à l'expression suivante :

```
1. check = 0 <= x and x <= 20
```

Ce type d'enchainement d'opérateurs de comparaison peut se faire avec n'importe quel opérateur. On peut par exemple écrire l'expression suivante :

```
1. 2 == x < 4
```

III-B - Instruction conditionnelle

Maintenant que l'on maitrise les expressions booléennes, on va pouvoir les utiliser pour n'exécuter une partie de programme que si une certaine condition est satisfaite grâce aux *instructions conditionnelles*.

III-B-1 - Instruction if

Commençons avec un exemple qui teste la valeur d'une variable `x` et affiche une phrase seulement si cette valeur est négative :

```
1. x = -5
2. if x <= 0:
3.     print("x est négatif !")
4.     print("Sa valeur absolue vaut ", -x)
```

L'exécution de ce programme produit la sortie suivante :

```
x est négatif !
Sa valeur absolue vaut 5
```

L'*instruction* if se compose du mot réservé if suivi d'une condition, puis du caractère deux-points (:) et enfin d'une séquence d'instructions à exécuter si la valeur de la condition est vraie. Comme vous l'avez remarqué, pour distinguer les instructions à exécuter si la condition est satisfaite, elles sont *indentées*, c'est-à-dire précédées de plusieurs espaces. La figure 5 montre l'organigramme correspondant à ce programme.

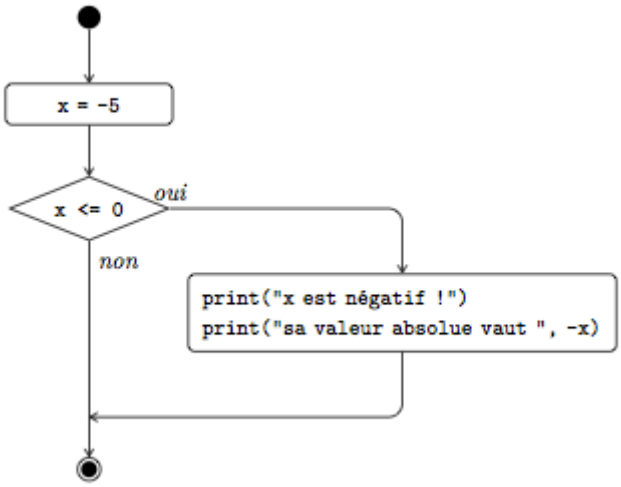

Figure 5. L'instruction if permet de n'exécuter une partie de programme que si une condition est satisfaite, c'est-à-dire que sa valeur est True.

Toutes les instructions à exécuter si la condition est satisfaite forment un ensemble appelé *bloc de code*. Elles doivent toutes être indentées de la même manière, par exemple avec une tabulation horizontale ou avec deux espaces, sans quoi une erreur se produira. Si l'on se réfère au guide de style de Guido van Rossum, il faut utiliser *quatre espaces* par niveau d'indentation.

La condition d'une instruction if est généralement une expression booléenne. Si la valeur de l'expression est True, la condition est satisfaite et le bloc de code du if est exécuté, sinon il est ignoré. Dans tous les cas, le programme continue ensuite son exécution après l'instruction if.

III-B-1-a - Exécution alternative

En plus d'un bloc de code qui n'est exécuté que si une condition est satisfaite, on peut également définir un second bloc qui ne sera exécuté que dans le cas contraire, c'est-à-dire si la condition n'est pas satisfaite. Une telle *exécution alternative* se déclare avec le mot réservé else.

Voici un exemple de programme qui utilise une *instruction* if-else :

```
1. grade = 9.5
2. if grade >= 10:
3.     print("vous avez réussi")
4. else:
5.     print("vous avez raté")
```

Le bloc if sera donc exécuté si la condition placée après le mot réservé if est satisfaite, et sinon, c'est le bloc else qui sera exécuté. Ensuite, le programme continue son exécution après l'instruction if-else. La figure 6 montre l'organigramme correspondant à ce programme.

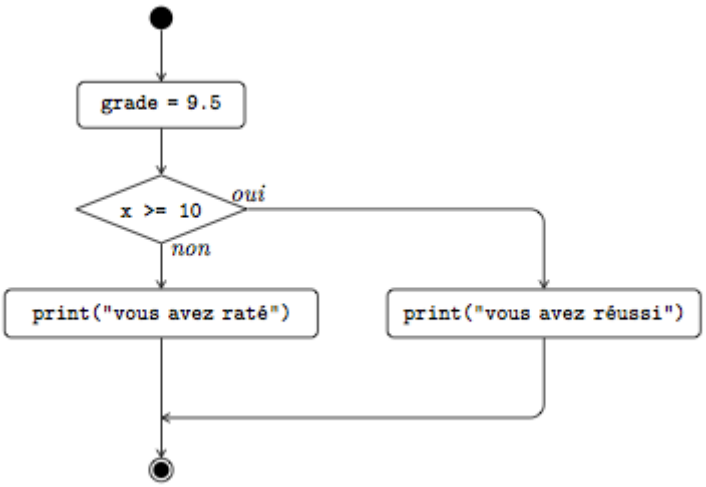

Figure 6. L'instruction if-else permet de n'exécuter une partie de programme que si une condition est vérifiée, et une autre si elle n'est pas vérifiée.

III-B-1-b - Imbrication d'instructions

Avec ce qu'on vient de voir, on peut donc définir une ou deux branches d'exécution différentes, en fonction de la valeur d'une condition. Comment pourrait-on faire si on désire plus de deux branches ?

Supposons que l'on souhaite écrire un programme qui contrôle la température de l'huile d'un moteur d'une voiture. Tant que celle-ci est sous les $100°$ C, tout est normal, si elle est entre $100°$ C et $130°$ C, il faut émettre un avertissement et enfin si elle dépasse les $130°$ C, il faut faire retentir une alarme de danger.

Pour cela, on va utiliser deux instructions if-else que l'on va *imbriquer* l'une dans l'autre. Ce qu'on va faire, c'est qu'on va utiliser comme contenu du bloc else de la première instruction if-else, une seconde instruction if-else :

```
1. temp = 126
2. if temp < 100:
3.     print("tout va bien")
4. else:
5.     if 100 <= temp <= 130:
6.         print("attention")
7.     else:
8.         print("danger")
```

La figure 7 montre l'organigramme de ce programme. Les deux instructions if-else y sont encadrées, ce qui fait bien ressortir qu'elles sont imbriquées l'une dans l'autre. La seconde a été imbriquée dans le bloc else, mais on aurait très bien pu la placer dans le bloc if si on voulait. La raison pour laquelle on ne l'a pas fait est que Python propose une écriture simplifiée lorsque l'imbrication se fait dans le bloc else.

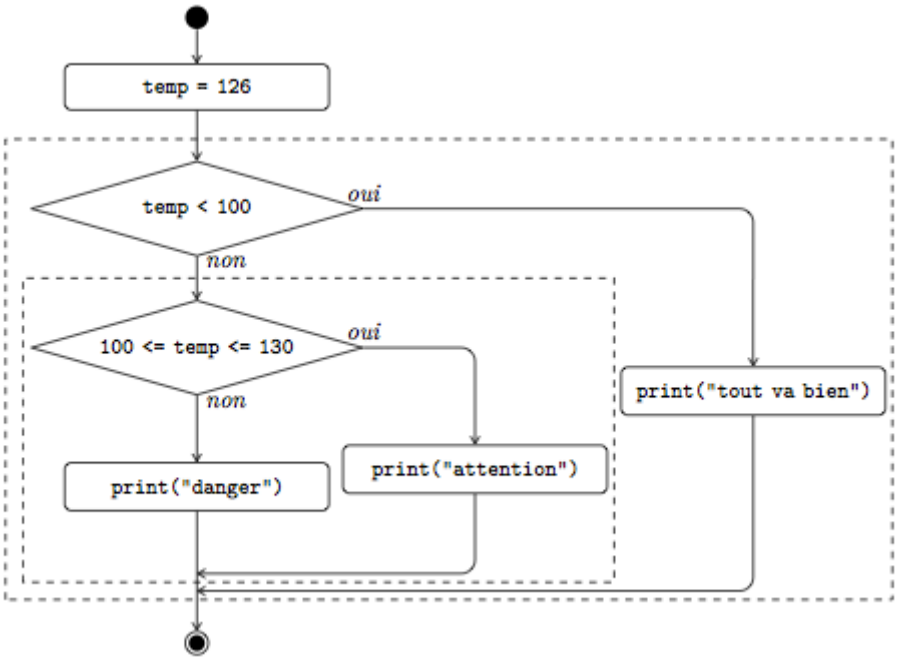

Figure 7. Une instruction if-else peut être imbriquée dans le bloc else d'une autre, produisant ainsi plusieurs branches d'exécution possibles dont une seule sera exécutée en fonction des valeurs de plusieurs conditions.

Vous aurez remarqué que comme le contenu du bloc else est une instruction if-else, cette dernière doit être indentée. Par conséquent, les instructions des blocs if et else de cette dernière ont un niveau d'indentation double. Si on continue à ajouter d'autres branches, on va se retrouver avec des niveaux d'indentation trop grands. Pour éviter cela, Python propose l'instruction elif (contraction de else if) permettant une écriture simplifiée des instructions conditionnelles à plus de deux branches. L'exemple précédent se réécrit comme suit :

```
1. temp = 126
2. if temp < 100:
3.     print("tout va bien")
4. elif 100 <= temp <= 130:
5.     print("attention")
6. else:
7.     print("danger")
```

On est maintenant capable d'écrire un programme complet pour chercher les racines d'un trinôme du second degré, dont le code se trouve au listing suivant. Faites attention aux commentaires qui ont été utilisés, ainsi qu'à la mise en page globale du code.

Le fichier trinomial-root.py contient un programme de recherche des racines d'un trinôme du second degré.

```
1. # Programme de recherche des racines d'un trinôme
2. # du second degré de la forme ax^2 + bx + c
3. # Auteur : Sébastien Combéfis
4. # Version : 7 aout 2015
5.
6. from math import sqrt
7.
8. print("Recherche des racines de ax^2 + bx + c")
9. a = int(input("Coefficient a : "))
10. b = int(input("Coefficient b : "))
11. c = int(input("Coefficient c : "))
12.
13. # Calcul du discriminant
14. delta = b**2 - 4 * a * c
15. print("Discriminant :", delta)
16.
17. # Test des trois cas possibles et affichage des racines du trinôme
18. if delta < 0:
19.     print("Pas de racine réelle")
```

Le fichier trinomial-root.py contient un programme de recherche des racines d'un trinôme du second degré.

```
20. elif delta == 0:
21.     x = -b / (2 * a)
22.     print("Une racine réelle double :", x)
23. else:
24.     x1 = (-b - sqrt(delta)) / (2 * a)
25.     x2 = (-b + sqrt(delta)) / (2 * a)
26.     print("Deux racines réelles distinctes:", x1, "et", x2)
```

III-C - Instruction répétitive

Une grande force de la programmation, et l'une des raisons pour lesquelles elle a été inventée, c'est de pouvoir répéter plusieurs fois la même tâche. Voyons maintenant comment exécuter plusieurs fois une même portion de code à l'aide d'*instructions répétitives*.

III-C-1 - Instruction while

Commençons avec un exemple qui affiche les cinq premiers nombres naturels, en commençant avec 1 :

```
1. n = 1
2. while n <= 5:
3.     print(n)
4.     n += 1
```

L'exécution de ce programme produit la sortie suivante :

```
1
2
3
4
5
```

L'*instruction* while se compose du mot réservé while suivi d'une condition, puis du caractère deux-points (:) et enfin d'un bloc de code appelé *corps de la boucle*. Celui-ci est exécuté intégralement de manière répétée, tant que la condition du while est vraie. La figure 9 montre l'organigramme de ce programme, où la *boucle* ainsi créée apparait.

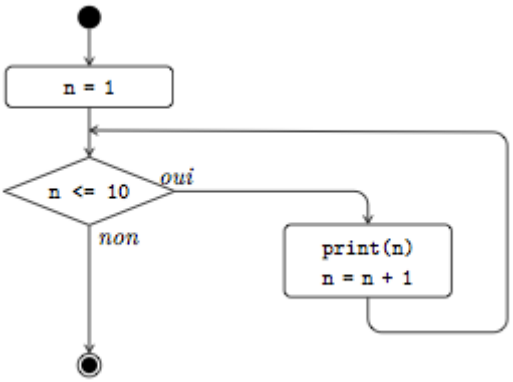

Figure 9. L'instruction while permet de répéter une portion de code tant qu'une condition est satisfaite, c'est-à-dire que sa valeur est True.

Tout comme pour l'instruction if, la condition d'une instruction while est généralement une expression booléenne. Le corps de la boucle est donc répété tant que la condition est satisfaite. Il est dès lors important que les variables qui apparaissent dans la condition voient leur valeur changer dans le corps de la boucle, sans quoi la condition ne changera jamais de valeur. Si celle-ci était vraie au départ, elle le restera donc toujours et on aura une *boucle infinie*. Voici un simple programme qui vous dira perpétuellement bonjour :

```
1. while True:
2.     print("Hello")
```

La première fois que la condition d'une boucle while est testée se fait avant même d'avoir exécuté une seule fois son corps. Il se peut donc qu'il ne soit jamais exécuté si la condition est fausse dès le départ.

Parfois, on souhaite néanmoins toujours exécuter un bloc de code, et on peut pour cela attacher un bloc else à l'instruction while. Le code contenu dans ce bloc else est exécuté une fois lorsque la condition du while devient fausse, avant que l'exécution ne continue après l'instruction while. L'exemple suivant illustre cela :

```
1. n = 10
2. while n <= 5:
3.     print(n)
4.     n += 1
5. else:
6.     print("La boucle est terminée")
```

L'exécution de ce programme produit la sortie suivante :

```
La boucle est terminée
```

Dans ce cas-ci, la condition de la boucle est initialement fausse puisque 10 n'est pas plus petit ou égal à 5. Le corps de la boucle ne va donc pas s'exécuter. Par contre, comme il y a un bloc else attaché à l'instruction while, il va être exécuté une fois. Comme ici la condition est initialement fausse, seul le contenu du bloc else sera donc exécuté, avant que l'exécution ne continue après l'instruction while.

III-C-1-a - Interruption de boucle

Une boucle se termine de manière naturelle lorsque sa condition devient fausse. Parfois, on peut vouloir la quitter prématurément, lorsqu'une autre condition est satisfaite. Python propose pour cela l'instruction break qui quitte directement une boucle, sans exécuter l'éventuel bloc else attaché à l'instruction while.

L'exemple suivant recherche le plus petit naturel non nul divisible par 38 et 46. Pour cela, on utilise une boucle qui parcoure toutes les valeurs entre 1 et 1000000 et, pour chacune d'elles, teste avec une instruction if si elle divise 38 et 46 grâce à l'opérateur modulo (le reste de la division est nul si c'est un diviseur). Dans ce cas, on a trouvé la valeur que l'on cherche et on peut arrêter la boucle, grâce à l'instruction break :

```
1. n = 1
2. while n <= 1000000:
3.     if n % 38 == 0 and n % 46 == 0:
4.         break
5.     n += 1
6. print(n, "est le plus petit nombre divisible par 38 et 46")
```

La figure 10 montre l'organigramme de ce programme, où vous pouvez clairement voir les deux chemins d'exécution possibles quittant la boucle. On voit également très bien que l'instruction break *« casse »* la boucle.

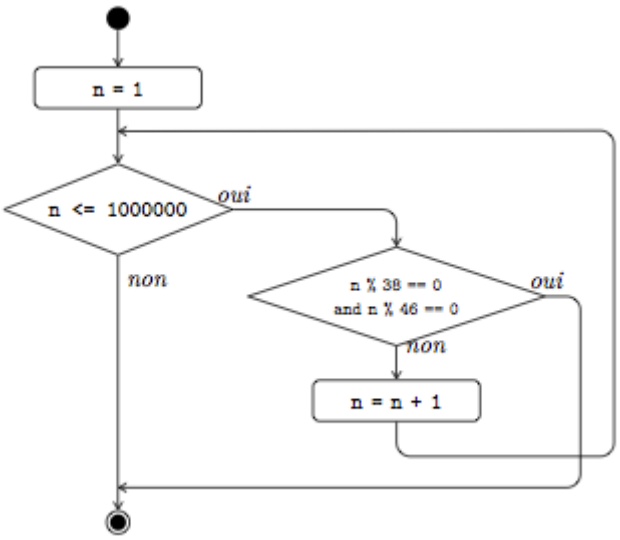

Figure 10. L'instruction break permet de quitter directement une boucle while.

De manière générale, on peut se passer de l'instruction break en adaptant la condition de la boucle while. Au plus vous aurez de chemins d'exécution possibles, au plus votre code deviendra complexe à comprendre. Prenons, par exemple, la situation où votre boucle ne se termine pas quand elle le devrait. Afin de trouver le bug, il vous faudra examiner tous les chemins de sortie possibles, et il vaut donc mieux minimiser leur nombre. L'exemple précédent peut, par exemple, se réécrire comme suit, en intégrant la condition du if dans celle du while :

```
1. n = 1
2. while n <= 1000000 and not (n % 38 == 0 and n % 46 == 0):
3.     n += 1
4. print(n, "est le plus petit nombre divisible par 38 et 46")
```

Néanmoins, vous constaterez qu'on perd en lisibilité, la condition du while devenant complexe. Si l'on s'en réfère aux conventions, un code pythonique doit être concis et clair. Dans ce cas, la version du code avec l'instruction break est préférable.

Pour être complet, il faut également mentionner l'instruction continue. Celle-ci va faire en sorte d'interrompre l'exécution du corps de la boucle, et directement retourner à l'évaluation de la condition du while. Elle permet en quelque sorte de *« passer un tour »* de boucle. De nouveau, il est possible de s'en passer en modifiant le corps de la boucle. L'exemple suivant affiche tous les nombres pairs plus petits que 100 :

```
1. n = 0
2. while n <= 100:
3.     n += 1
4.     if n % 2 != 0:
5.         continue
6.     print(n)
```

La première instruction augmente la valeur de la variable n. Ensuite, si celle-ci est impaire (le reste de la division par deux n'est pas nul), on arrête d'exécuter le corps de la boucle et on passe directement au tour suivant. Sinon, on affiche la valeur de n.

III-D - Dessin avec turtle

Pour terminer ce chapitre, découvrons le *module* turtle qui permet d'effectuer simplement des dessins constitués de lignes. Ce module permet de réaliser des dessins en faisant se déplacer une tortue dans le plan, qui dessine le chemin qu'elle a parcouru. Voyons d'abord un programme qui utilise ce module avant d'en détailler le contenu :

```
1. from turtle import *
```

```
2.
3. i = 0
4. while i < 5:
5.     forward(50)
6.     left(72)
7.     i += 1
8. done()
```

Ce programme comporte une boucle while qui s'exécute cinq fois. À chaque boucle, la tortue avance de 50 unités (forward), puis tourne sur place de 72 degrés (left), dans le sens antihorloger. La figure 11 montre le résultat de l'exécution du programme, sachant que la tortue se trouve initialement au milieu de la fenêtre, orientée vers la droite.

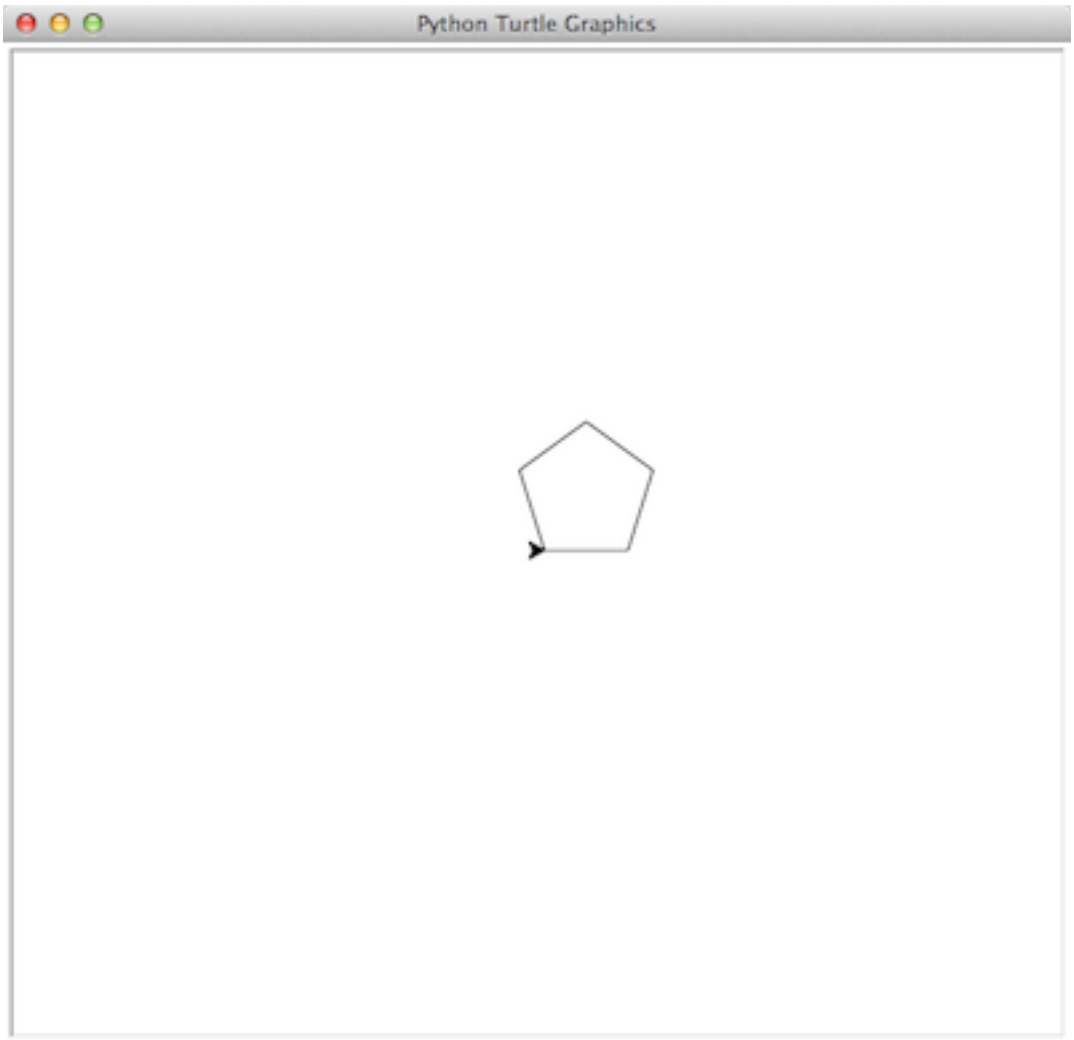

Figure 11. Le module turtle permet de dessiner des formes dans le plan, en faisant se déplacer une tortue à partir de primitives simples.

Une fois le module importé avec from turtle import *, on peut utiliser toutes les fonctions définies dans ce module, dont les principales sont reprises à la figure 12. La fonction forward permet d'avancer d'un nombre d'unités qu'on lui précise tandis que la fonction left permet de tourner sur place d'un angle en degrés qu'on lui précise également. La fonction done, appelée à la fin du programme, permet de garder la fenêtre de dessin ouverte, malgré que la tortue ait fini ses déplacements.

Fonction	Description
forward(dist)	Avancer de la distance spécifiée
backward(dist)	Reculer de la distance spécifiée
left(ang)	Tourne sur place dans le sens antihorloger de l'angle spécifié
right(ang)	Tourne sur place dans le sens horloger de l'angle spécifié
up()	Relever le crayon
down()	Baisser le crayon
color(col)	Changer la couleur du crayon ('red', 'blue'…)
width(wid)	Changer l'épaisseur du crayon par l'épaisseur spécifiée
goto(x, y)	Aller, en ligne droite, au point de coordonnées (x, y)
reset()	Effacer tout
done()	Maintenir la fenêtre ouverte une fois le dessin terminé

Il faut faire attention à la fonction goto(x, y) qui permet de déplacer la tortue vers le point de coordonnées (x, y). En effet, les axes ne sont pas comme vous en avez l'habitude en mathématiques : le coin supérieur gauche se trouve en $(0, 0)$, et l'axe des x va vers la droite tandis que l'axe des y descend vers le bas.

On observe également la présence des fonctions up() et down() qui permettent de lever et baisser le crayon, ce qui permet de dessiner ou non lorsque la tortue se déplace. D'autres fonctions sont disponibles, comme celle pour dessiner des arcs de cercle, par exemple, mais nous n'allons pas les détailler ici.

Le listing suivant montre un autre exemple complet de programme qui utilise une tortue pour réaliser un dessin. Le résultat de l'exécution de ce programme est présenté à la figure 14. Expérimentez ce programme en changeant les valeurs des variables MAX et factor.

Le fichier turtle-drawing.py contient un programme de dessin à l'aide du module turtle.

```
1. # Programme de dessin d'une forme particulière
2. # Auteur : Sébastien Combéfis
3. # Version : 24 aout 2015
4.
5. from turtle import *
6.
7. MAX = 23
8. FACTOR = 1.2
9. len = 10
10. i = 0
11.
12. while i < MAX:
13.     forward(len)
14.     left(90)
15.     i += 1
16.     len *= FACTOR
17.
18. done()
```

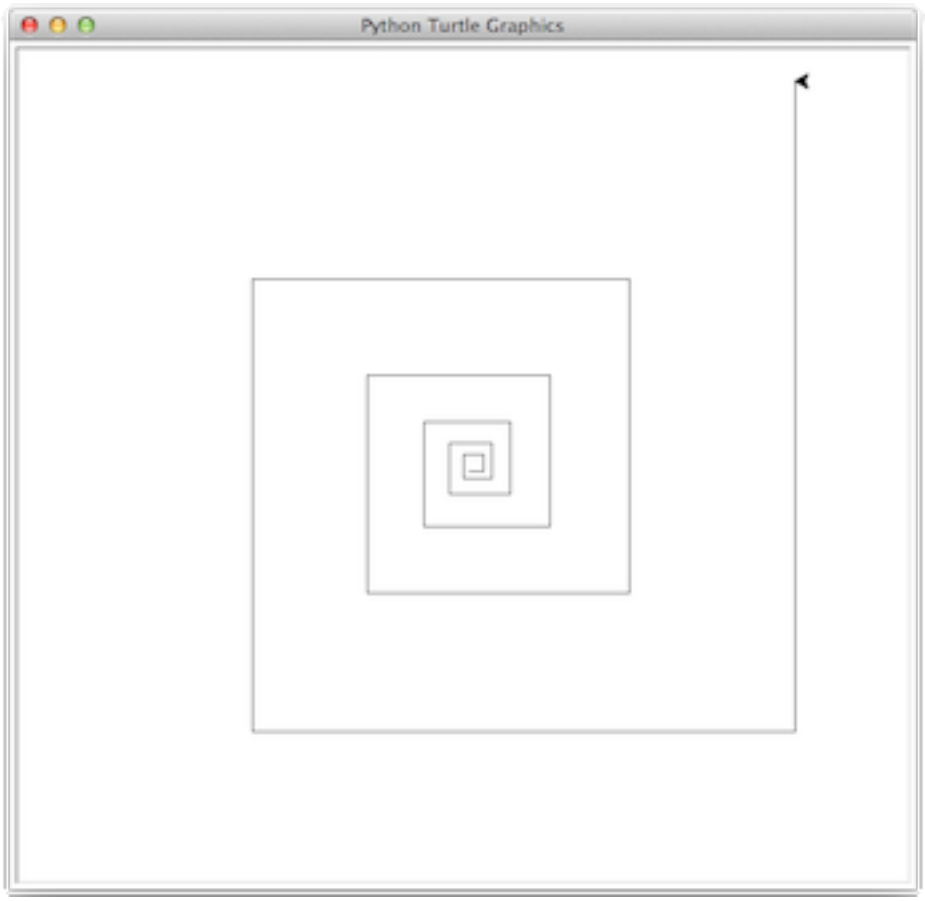

Figure 14. Le programme dessine une séquence de lignes de plus en plus longue, en tournant à chaque fois de 90 degrés vers la gauche entre chaque ligne.

IV - Fonction

On sait maintenant écrire des programmes Python avancés, mais ils vont vite devenir longs en nombre de lignes de code. De plus, on risque très vite de se retrouver avec des répétitions de codes similaires. Ce chapitre décrit le concept de *fonction* grâce auquel on va pouvoir écrire du code plus compact, lisible et réutilisable. On a déjà pu en utiliser dans les chapitres précédents, et on va ici voir comment en définir.

IV-A - Définition et appel de fonction

Une *fonction* se définit avec le mot réservé def, suivi de son nom, d'une liste de *paramètres* (qui peut être vide), du caractère deux-points (:) et enfin d'un bloc de code représentant son *corps*. Une fois définie, elle peut être utilisée autant de fois qu'on le souhaite, en l'*appelant*.

On peut classifier les fonctions selon deux critères. Une fonction peut *renvoyer une valeur* ou non, au terme de son exécution, et une fonction peut admettre ou non des *paramètres*. On va maintenant voir comment définir et utiliser ces différents types de fonctions.

IV-A-1 - Liste de paramètres

Commençons avec un exemple d'une *fonction qui ne renvoie pas de valeur*, et n'admet aucun paramètre. Écrivons, par exemple, une fonction qui affiche la table de multiplication de 7. Pour cela, on va évidemment utiliser une boucle while qui va parcourir les entiers de 1 à 10. La fonction se définit comme suit :

```
1. def table7():
2.     n = 1
3.     while n <= 10:
4.         print(n, "x 7 =", n * 7)
5.         n += 1
```

Ces lignes de code définissent donc une fonction dont le nom est table7. La fonction initialise une variable n à 1, puis une boucle se répète tant que la condition n <= 10 est vraie. Le corps de la boucle affiche une ligne de la table de multiplication, puis incrémente la valeur de n d'une unité. Pour exécuter cette fonction, il suffit simplement d'utiliser son nom, suivi d'une parenthèse ouvrante et d'une fermante. L'*instruction d'appel de fonction* table7() produit donc le résultat suivant :

```
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
5 x 7 = 35
6 x 7 = 42
7 x 7 = 49
8 x 7 = 56
9 x 7 = 63
10 x 7 = 70
```

Un fichier Python est analysé par l'interpréteur ligne par ligne. Dès lors, un appel de fonction ne peut pas se faire avant que celle-ci n'ait été définie, sans quoi l'interpréteur génèrera une erreur vous signalant qu'il ne parvient pas à trouver la fonction demandée :

```
Traceback (most recent call last):
  File "program.py", line 1, in <module>
    table7()
NameError: name 'table7' is not defined
```

Imaginons maintenant que l'on souhaite aussi afficher la table de multiplication de 3, mais aussi celle de 8 et pourquoi pas celle de 42. La manière la plus directe consiste à définir des fonctions table3, table8 et table42, mais ce n'est pas forcément la plus propre, car on va écrire plusieurs fois un code très similaire.

IV-A-1-a - Fonction à un paramètre

Bien évidemment, cela n'est absolument pas pratique de devoir ainsi recopier du code sur lequel on ne fait finalement que de petits changements. Il y a essentiellement deux endroits où on doit effectuer un changement, à savoir dans l'instruction print(n, "x 7 =", n * 7) où il faut remplacer les deux 7.

La solution qu'on va suivre consiste à faire en sorte que la fonction admette un *paramètre*, qui indique le nombre dont on veut la table de multiplication. Un paramètre est identifié par un nom, que l'on place entre les parenthèses de la définition de fonction. Voici une fonction table qui admet un paramètre base et qui affiche sa table de multiplication :

```
1. def table(base):
2.     n = 1
3.     while n <= 10:
4.         print(n, "x", base, "=", n * base)
5.         n += 1
```

Le corps de cette fonction est très similaire à celui de la fonction table7, les occurrences de 7 ayant été remplacées par base. Le grand avantage de cette fonction est qu'elle est beaucoup plus *générique*, c'est-à-dire qu'elle pourra fonctionner dans beaucoup plus de cas. Pour l'appeler, on utilise de nouveau son nom, sans oublier de fournir une valeur à son paramètre, entre parenthèses. Voici comment afficher successivement les tables de multiplication de 3, 8 et 42 :

```
1. table(3)
```

```
2. table(8)
3. table(42)
```

Lors de l'appel table(3), la fonction table est donc appelée et son paramètre base va se voir affecter la valeur spécifiée lors de l'appel, c'est-à-dire l'entier 3. C'est ce qu'on appelle le *passage de paramètres* qui a lieu lors d'un appel de fonction. C'est comme si on avait l'instruction base = 3 au début du corps de la fonction table. On peut d'ailleurs rendre cela plus explicite en écrivant l'appel ainsi :

```
1. table(base=3)
```

On reviendra plus loin dans ce chapitre sur cette notation particulière, qu'on a d'ailleurs déjà rencontrée lors d'appels de la méthode print, avec les paramètres nommés sep et end.

IV-A-1-a-i - Fonction à plusieurs paramètres

Une fonction peut évidemment admettre plus d'un paramètre. Il suffit simplement de les séparer par des virgules, autant dans la définition de la fonction que lors de son appel. Modifions, par exemple, la fonction table afin de pouvoir choisir la première ligne à afficher, et le nombre de lignes que l'on veut en tout :

```
1. def table(base, start, length):
2.     n = start
3.     while n < start + length:
4.         print(n, "x", base, "=", n * base)
5.         n += 1
```

La fonction admet trois paramètres, et il faut en spécifier trois lors de son appel. Voici le résultat de l'appel table(8, 5, 2), où on voit que la première ligne commence bien à 5 et qu'il y en a bien deux affichées :

```
5 x 8 = 40
6 x 8 = 48
```

De nouveau, on peut rendre explicite le passage des paramètres lors de l'appel de la fonction :

```
1. table(base=8, start=5, length=2)
```

De manière générale, cette notation est à éviter, car elle alourdit inutilement le code. Néanmoins, on verra, à la section suivante, qu'elle est nécessaire dans un cas particulier.

IV-A-1-a-ii - Valeur par défaut des paramètres

Dès lors qu'une fonction admet plusieurs paramètres, on doit également en fournir autant qu'il faut lors de son appel. Par exemple, on doit fournir trois paramètres lorsqu'on appelle notre dernière version de la fonction table. Si on tente de l'appeler comme on faisait au début, à savoir avec table(8), par exemple, on aura une erreur :

```
1. Traceback (most recent call last):
2.   File "program.py", line 7, in <module>
3.     table(8)
4. TypeError: table() missing 2 required positional arguments: 'start' and 'length'
```

Pour autoriser un tel appel, il faut que les paramètres start et length possèdent une *valeur par défaut*, c'est-à-dire celle qu'ils auront lors de l'appel si on n'en spécifie pas une autre. En Python, c'est simple, il suffit de déclarer ces valeurs par défaut lors de la définition de la fonction :

```
1. def table(base, start=1, length=10):
2.     n = start
3.     while n < start + length:
4.         print(n, "x", base, "=", n * base)
```

```
5.          n += 1
```

Le paramètre start a donc 1 comme valeur par défaut et le paramètre length a 10 comme valeur par défaut. On peut appeler cette fonction de plusieurs manières différentes :

```
1. table(8, 5, 2)
```

- On peut ne fournir une valeur que pour le premier paramètre, les deux autres recevant leur valeur par défaut :

```
1. table(8)
```

- On peut ne modifier que la valeur par défaut du paramètre start ou length, mais dans le deuxième cas, on doit nommer le paramètre qu'on veut modifier, car on ne suit pas l'ordre de la définition de la fonction :

```
1. table(8, 5)
2. table(8, length=2)
```

Notez que les paramètres pour lesquels on prévoit une valeur par défaut doivent impérativement se trouver après les paramètres sans valeur par défaut, sans quoi l'interpréteur génèrera une erreur.

IV-A-1-b - Valeur de retour

Pour le moment, les fonctions qu'on est capable d'écrire admettent éventuellement des paramètres et permettent d'exécuter plusieurs instructions. Une fois la fonction exécutée, le programme continue son exécution en poursuivant juste après l'instruction d'appel de la fonction.

Une fonction peut également *renvoyer une valeur* qu'il est possible de récupérer lorsqu'on l'appelle. Commençons par voir un exemple d'une fonction qui calcule le produit de deux nombres. Contrairement aux fonctions précédemment vues, cette fonction va effectuer le calcul, mais n'affichera pas le résultat :

```
1. def multiply(a, b):
2.     return a * b
```

La fonction multiply admet donc deux paramètres a et b. Elle calcule ensuite leur produit qui est le résultat que doit produire la fonction. Pour signaler cela, on utilise le mot réservé return qui permet de définir la *valeur de retour* d'une fonction.

On peut appeler cette fonction comme on l'a fait jusqu'à présent, et donc écrire une instruction comme :

```
1. multiply(7, 9)
```

Cette instruction est valable et son exécution ne produira pas d'erreur. La fonction calcule le produit entre 7 et 9 puis renvoie le résultat de ce calcul. Mais on ne récupère pas cette valeur renvoyée lors de l'appel, et dès lors ce dernier est complètement inutile. Le résultat calculé par la fonction est tout simplement perdu à jamais.

Pour avoir accès à la valeur de retour, il faut la stocker dans une variable lors de l'appel, en écrivant par exemple :

```
1. res = multiply(7, 9)
2. print(res)
```

L'exécution de ces deux instructions affiche 63. La première instruction appelle la fonction multiply et stocke la valeur renvoyée par cet appel dans la variable res, c'est-à-dire la valeur de l'expression qui suit l'instruction return à la fin du corps de la fonction (return a * b). Cette valeur est ensuite affichée par la fonction print.

IV-A-1-b-i - Appel de fonction comme expression

L'appel d'une fonction qui renvoie une valeur est une expression, et on peut dès lors l'utiliser partout là où une expression est acceptée. Par exemple, on aurait pu écrire l'exemple précédent comme suit :

```
1. print(multiply(7, 9))
```

On pourrait aussi, par exemple, réécrire la fonction table en utilisant la fonction multiply pour calculer les différents produits :

```
1. def table(base, start=1, length=10):
2.     n = start
3.     while n < start + length:
4.         print(n, "x", base, "=", multiply(n, base))
5.         n += 1
```

On reviendra plus loin dans ce cours sur cette façon de programmer, à savoir en exploitant au maximum les fonctions.

IV-A-1-c - Le type function

Enfin, un dernier point intéressant à savoir est qu'une fonction est une valeur, en ce sens qu'il existe un type de donnée fonction. On peut s'en rendre compte en faisant appel à la fonction type.

Par exemple, si on exécute l'instruction suivante : print(type(multiply)), on obtient le résultat qui suit :

```
<class 'function'>
```

Cette particularité est très puissante comme en témoigne l'exemple présenté au listing de la figure 1, qui permet d'afficher des tables de calcul (d'addition ou de multiplication). Le fichier functions.py contient avant tout les définitions des deux fonctions add et multiply.

Ensuite, on retrouve la définition de la fonction table, quelque peu modifiée par rapport à la précédente version. Celle-ci admet deux nouveaux paramètres qui sont un symbole (un caractère) et une opération à appliquer (une fonction). Par défaut, le symbole est un astérisque (*) et l'opération à appliquer est la fonction multiply.

Viennent enfin deux exemples d'appels à cette nouvelle version de la fonction table :

- le premier appel affiche la table de multiplication de 4, en commençant avec 1 (la valeur par défaut) et en affichant 2 lignes ;
- le second appel affiche la *« table d'addition »* de 4, en commençant également avec 1

(la valeur par défaut), affiche

5

lignes, et utilise le symbole + et la fonction add pour l'opération à appliquer.

Le fichier functions.py contient un programme qui définit et utilise plusieurs fonctions permettant d'afficher des tables de calcul.

```
1. def add(a, b):
2.     return a + b
3.
4. def multiply(a, b):
5.     return a * b
6.
7. def table(base, start=1, length=10, symbol="*", op=multiply):
8.     n = start
9.     while n < start + length:
10.         print(n, symbol, base, "=", op(n, base))
11.         n += 1
12.
13. table(4, length=2)
14. table(4, length=5, symbol="+", op=add)
```

L'exécution du programme du listing précédent affiche ce qui suit à l'écran. On y voit clairement les deux lignes de la table de multiplication et les cinq lignes de la *« table d'addition »* :

```
1 * 4 = 4
2 * 4 = 8
1 + 4 = 5
2 + 4 = 6
3 + 4 = 7
4 + 4 = 8
5 + 4 = 9
```

IV-A-1-d - Variable locale et globale

Lorsqu'on travaille avec des fonctions, il faut distinguer deux sortes de variables : les locales et les globales. Une *variable globale* est définie pour tout le programme ; elle est initialisée en dehors de toute fonction. Une *variable locale* est définie uniquement dans le corps d'une fonction, celle où elle a été initialisée. Pour comprendre ce qu'implique l'existence de ces deux sortes de variables, analysons le programme suivant :

```
1. def tvac(amount):
2.     n = 1 + taxrate / 100
3.     return amount * n
4.
5. taxrate = 21
6. n = 25
7. print(tvac(n))
```

Les trois premières lignes définissent une fonction tvac qui transforme un prix hors taxe en un prix toutes taxes incluses. Le seul paramètre qu'elle admet est le montant hors taxe à transformer. Les trois dernières lignes calculent ce que donnent 25 € avec un taux de taxation de 21 %. Le résultat de l'exécution est tout simplement :

```
30.25
```

Lors de l'exécution de ce programme, il y a en fait deux variables n qui vont exister en même temps :

- Celle initialisée à la deuxième ligne est une variable locale à la fonction tvac. Elle n'existe que dans le corps de la fonction, durant le temps où elle est exécutée et disparait ensuite de la mémoire.
- Celle initialisée à l'avant-dernière instruction est une variable globale. Elle existe dans tout le programme, depuis son initialisation jusque la fin de l'exécution du programme.

Le concept de variable locale permet d'utiliser plusieurs variables différentes avec le même nom, pour autant qu'elles soient dans des fonctions différentes. Dans notre exemple, on a ainsi pu utiliser le nom n dans la fonction tvac, même si une variable globale de même nom existait déjà. Une conséquence immédiate est que la variable globale n n'est plus accessible dans la fonction tvac, au profit de la variable locale portant le même nom. Par contre, comme vous pouvez le voir sur l'exemple, la fonction tvac peut tout à fait accéder à la variable globale taxrate.

IV-A-1-d-i - Portée de variable

La *portée* d'une variable représente les endroits du code où on peut l'utiliser, c'est-à-dire où elle existe. Pour bien comprendre la différence entre les deux sortes de variables, et cette notion de portée de variable, voyons quelques exemples additionnels.

Tentons d'abord d'accéder à une variable locale en dehors de la fonction qui l'a initialisée :

```
1. def fun():
2.     a = 12
3.
4. fun()
5. print(a)
```

L'exécution de ce code provoquera une erreur, car la variable a dont on veut imprimer la valeur n'existe tout simplement pas à cet endroit dans le code. La portée de la variable est limitée au corps de la fonction où elle est initialisée, la variable disparait de la mémoire une fois que l'appel de la fonction est terminé :

```
Traceback (most recent call last):
  File "program.py", line 5, in <module>
    print(a)
NameError: name 'a' is not defined
```

Voyons maintenant un deuxième exemple où on accède à une variable globale depuis une fonction :

```
1. def fun():
2.     print(a)
3.
4. a = 12
5. fun()
```

La variable globale a est initialisée à la valeur 12. Ensuite, la fonction fun est appelée, et affiche la valeur de cette variable, également disponible à cet endroit du code puisque c'est une variable globale.

Voyons maintenant un exemple où une fonction initialise une variable locale portant le même nom qu'une variable globale :

```
1. def fun():
2.     a = 42
3.     print("dans fun :", a)
4.
5. a = 12
6. fun()
7. print("en dehors de fun :", a)
```

Il y a donc deux variables avec le nom a : une globale initialisée à 12 et une locale à la fonction fun initialisée à 42. En fonction de où on se situe dans le code, on accèdera donc à l'une ou l'autre, en se rappelant que la variable locale masque la variable globale. L'exécution du programme affiche donc :

```
dans fun : 42
en dehors de fun : 12
```

Voyons enfin un dernier exemple qui ne se comporte pas forcément comme on pourrait le croire de prime abord. Dans une fonction, on va commencer par tenter un accès à la variable globale a, pour ensuite initialiser une variable locale de même nom et y accéder :

```
1. def fun():
2.     print("globale :", a)
3.     a = 42
4.     print("locale :", a)
5.
6. a = 12
7. fun()
```

L'exécution de ce code produit l'erreur suivante :

```
Traceback (most recent call last):
  File "program.py", line 7, in <module>
    fun()
  File "program.py", line 2, in fun
    print("globale :", a)
UnboundLocalError: local variable 'a' referenced before assignment
```

L'interpréteur Python signale que la variable locale a est utilisée avant d'avoir été initialisée. Elle existe donc, mais ne possède pas de valeur. Lorsqu'une fonction contient des variables locales, elles masquent les variables globales de même nom dans tout le corps de la fonction.

IV-A-1-d-ii - Modifier une variable globale

Comment *modifier* la valeur d'une variable globale depuis une fonction ? Si on utilise une instruction d'affectation, cela va créer une nouvelle variable locale de même nom. Pour signaler qu'un nom réfère à une variable globale, il suffit d'utiliser le mot réservé global :

```
1. def fun():
2.     global a
3.     a = 42
4.
5. a = 12
6. fun()
7. print(a)
```

Comme on verra plus loin, retenez néanmoins que ce n'est pas une bonne pratique d'utiliser des variables globales et qu'il faut limiter au maximum leur utilisation ainsi que celle du mot réservé global.

IV-A-1-e - Instruction return

Revenons un moment sur l'*instruction* return qui permet donc de définir la valeur de retour d'une fonction. En plus de cet effet, elle permet également d'arrêter l'exécution de la fonction et de poursuivre l'exécution du programme après son appel. Tout code se trouvant après un return ne sera donc pas exécuté. Par exemple, la seconde instruction de la fonction suivante ne sera jamais exécutée :

```
1. def fun():
2.     return 42
3.     print('This will not be printed')
```

Le fait que return arrête l'exécution de la fonction peut parfois être utilisé pour simplifier son code. Écrivons par exemple une fonction abs qui calcule la valeur absolue d'un nombre qu'on lui passe en paramètre. Voici une première version de cette fonction :

```
1. def abs(x):
2.     if x < 0:
```

```
3.          return -x
4.      else:
5.          return x
```

Lorsque la valeur de x est négative, la condition du if est satisfaite, et la première instruction return est exécutée. Ensuite, le corps de la fonction est quitté, les instructions se trouvant après le return n'étant pas exécutées. On peut dès lors simplifier le code de la fonction, en éliminant l'instruction else, inutile :

```
1. def abs(x):
2.     if x < 0:
3.         return -x
4.     return x
```

Cela permet de diminuer le niveau d'indentation d'une partie du code et de le raccourcir d'une ligne, ce qui le rend plus lisible. Une autre pratique que l'on retrouve parfois consiste à diminuer le nombre total de return dans une fonction. Cela permet notamment une analyse plus aisée, puisqu'on diminue ainsi le nombre de points de sortie :

```
1. def abs(x):
2.     if x < 0:
3.         x = -x
4.     return x
```

IV-B - Découpe en sous-problèmes

À quoi servent les fonctions en pratique ? Comme on a pu le constater, elles permettent de réutiliser du code et évitent ainsi de la duplication, tout en rendant le programme plus lisible. Les fonctions permettent également de *structurer* un programme, fournissant ainsi une documentation implicite de ce dernier.

Lorsque vous devez écrire du code très similaire à du code déjà écrit, on pourrait sans doute croire que la meilleure solution consiste à copier-coller un bout de code, puis de l'adapter. Il s'agit pourtant d'une très mauvaise pratique, pour deux raisons principales :

- les chances de se tromper à un moment lors des copier-coller ou des adaptations de ces derniers sont grandes ;
- si une erreur est détectée dans le code qui a été copié-collé, il va falloir la corriger partout là où ce code a été copié-collé.

IV-B-1 - Décomposition

Le second intérêt des fonctions est de structurer un programme. Grâce à ces dernières, on va pouvoir *découper* un gros programme en petits blocs, chacun plus simple à comprendre. Voyons cela avec l'exemple suivant qui permet d'afficher les n premiers nombres premiers (pour rappel, un nombre naturel n est premier s'il admet exactement deux diviseurs distincts (qui seront dès lors 1 et n)) :

```
1. n = 1
2. nb = 10
3. while nb > 0:
4.     divisors = 0
5.     d = 1
6.     while d <= n:
7.         if n % d == 0:
8.             divisors += 1
9.         d += 1
10.     if divisors == 2:
11.         print(n)
12.         nb -= 1
13.     n += 1
```

Comprendre ce programme sans être celui qui l'a écrit n'est pas du tout facile. Tout d'abord, il n'y a aucun commentaire permettant d'aider le lecteur et le nom de toutes les variables n'est pas forcément explicite. Ensuite, ce code contient deux boucles imbriquées, ce qui rend sa compréhension moins aisée, de prime abord.

Afin de rendre le programme plus clair, il faut définir et utiliser des fonctions. Pour cela, on va décomposer le *problème* qu'on nous demande de résoudre en *sous-problèmes*. On peut en identifier trois :

- tester si un nombre est un diviseur d'un autre ;
- tester si un nombre est premier ;
- afficher les n premiers nombres premiers.

Une fois ces sous-problèmes résolus, on pourra les combiner pour résoudre le problème principal. Commençons par définir une fonction isDivisor permettant de tester si un nombre est un diviseur d'un autre :

```
1. def isDivisor(d, n):
2.     return n % d == 0
```

Pour tester si le nombre d est un diviseur de n, il suffit de vérifier si le reste de la division entière de n par d est nul ou non. Le corps de la fonction est simplement composé d'une instruction qui renvoie un booléen (True si d est un diviseur de n et False sinon).

Il nous faut ensuite une fonction isPrime qui permet de tester si un nombre n est un nombre premier ou non :

```
1. def isPrime(n):
2.     d = 1
3.     nbDivisors = 0
4.     while d <= n:
5.         if isDivisor(d, n):
6.             nbDivisors += 1
7.         d += 1
8.     return nbDivisors == 2
```

La fonction compte le nombre de diviseurs que possède le nombre n. Pour cela, elle parcourt tous les nombres compris entre 1 et n, et teste, grâce à la fonction isDivisor précédemment définie, s'ils divisent ou non n. On termine en vérifiant que le nombre de diviseurs distincts doit être de deux, en renvoyant donc un booléen (True si n est un nombre premier et False sinon).

Enfin, il nous reste maintenant à définir une fonction printPrimes qui permet d'afficher des nombres premiers. Pour cela, on va parcourir tous les nombres naturels l'un après l'autre, et afficher les nombres premiers, jusqu'à en avoir affiché suffisamment. Le listing suivant montre le programme final, où l'on peut voir la fonction printPrimes qui utilise la fonction isPrime précédemment définie.

Le fichier prime-numbers.py contient un programme permettant d'afficher une séquence de nombres premiers.

```
1. # Programme d'affichage d'une séquence de nombres premiers
2. # Auteur : Sébastien Combéfis
3. # Version : 24 aout 2015
4.
5. def isDivisor(d, n):
6.     return n % d == 0
7.
8. def isPrime(n):
9.     d = 1
10.     nbDivisors = 0
11.     while d <= n:
12.         if isDivisor(d, n):
13.             nbDivisors += 1
```

Le fichier prime-numbers.py contient un programme permettant d'afficher une séquence de nombres premiers.

```
14.         d += 1
15.     return nbDivisors == 2
16.
17. def printPrimes(nb):
18.     n = 1
19.     while nb > 0:
20.         if isPrime(n):
21.             print(n)
22.             nb -= 1
23.         n += 1
24.
25. printPrimes(7)
```

L'exécution du programme présenté au listing précédent affiche donc la séquence des sept premiers nombres premiers :

```
2
3
5
7
11
13
17
```

Découper un problème en sous-problèmes, et par conséquent définir plusieurs fonctions de plus petites tailles, permet de rendre un programme plus *lisible*. Il s'agit de manière générale d'un bon réflexe à suivre. Il ne faut pas avoir peur de définir plusieurs petites fonctions, même si leur corps ne fait qu'une instruction.

De plus, les différentes fonctions définies pourraient être réutilisées dans d'autres programmes ultérieurs. En procédant de la sorte, on se constitue un stock de fonctions à utiliser pour en définir de nouvelles.

IV-B-1-a - Spécification

Lorsqu'on définit des fonctions, c'est évidemment dans le but de les utiliser. Il est donc très important de les *documenter*, c'est-à-dire d'ajouter des commentaires expliquant ce qu'elles font, et comment les utiliser. On pourrait par exemple écrire :

```
1. # Fonction permettant de tester si le nombre entier d
2. # est un diviseur du nombre entier n
3. def isDivisor(d, n):
4.     return n % d == 0
```

Ce commentaire qu'on a ajouté décrit ce que fait la fonction, ainsi que les paramètres qu'il faut lui fournir. Il est donc complet et permet d'utiliser la fonction comme il faut.

On peut décrire une fonction de manière plus systématique en fournissant sa *spécification*. Pour cela, on doit décrire deux choses :

- les *préconditions* d'une fonction sont toutes les conditions qui doivent être satisfaites avant de pouvoir appeler la fonction, que ce soit sur des variables globales ou sur ses paramètres ;
- les *postconditions* d'une fonction sont toutes les conditions qui seront satisfaites après appel de la fonction, si les préconditions étaient satisfaites, que ce soit sur des variables globales ou sur l'éventuelle valeur renvoyée.

Voyons tout de suite comment spécifier la fonction isDivisor en définissant ses préconditions et postconditions :

```
1. # Test de la divisibilité d'un nombre par un autre
2. # Pre  : d et n sont deux entiers positifs
```

```
# Pre  : d != 0
# Post : la valeur renvoyée vaut True si d divise n,
#        et False sinon
def isDivisor(d, n):
    return n % d == 0
```

Pour appeler la fonction, il faut donc lui fournir deux nombres entiers positifs en paramètres, et s'assurer que d soit différent de zéro. Dans ce cas, après avoir appelé la fonction, la valeur qu'elle aura renvoyée contiendra True si d est un diviseur de n et False sinon.

La spécification d'une fonction contient donc toute la documentation nécessaire pour l'utiliser correctement. S'il ne faut pas devoir lire le corps de la fonction pour comprendre ce qu'elle fait et comment l'utiliser, c'est que la spécification est bien écrite.

IV-C - Récursion

Dans tous les exemples qu'on a vu pour le moment, on a défini des fonctions, puis on les a appelées, que ce soit depuis une autre fonction ou en dehors de toute fonction. On peut également appeler une fonction depuis son propre corps. Une telle fonction, qui s'appelle elle-même, est appelée *fonction récursive*.

Commençons par voir un exemple pratique où une telle fonction peut s'avérer utile. Supposons que l'on veuille écrire une fonction permettant de calculer la somme $1 + 2 + \ldots + n$ pour un entier positif n donné. On pourrait utiliser une boucle while pour ce faire et écrire :

```
def sum(n):
    result = 0
    while n > 0:
        result += n
        n -= 1
    return result
```

Cette fonction est tout à fait correcte et calcule la valeur qu'il faut. On peut également l'écrire différemment, à l'aide d'une fonction récursive, en se rendant compte de la propriété suivante :

$$\sum_{i=1}^{n} i = \left(\sum_{i=1}^{n-1} i\right) + n,$$

qui indique donc que la somme des n premiers entiers positifs correspond à la somme des $n-1$ premiers entiers positifs à qui on ajoute n. De plus, lorsque n vaut 1, on obtient directement la valeur de la somme qui est 1. Étant donné ces deux observations, on peut réécrire la fonction sum comme suit :

```
def sum(n):
    if n == 1:
        return 1
    return sum(n - 1) + n
```

Une fonction récursive agit donc comme une boucle, puisqu'elle se rappelle elle-même un certain nombre de fois. Pour éviter une boucle infinie, il faut que dans un cas, appelé *cas de base*, la fonction ne se rappelle pas. Dans notre exemple, le cas de base se produit lorsque $n = 1$, et le *cas récursif* pour $n > 1$. La figure 3 montre l'enchainement des appels récursifs qui se produisent lorsqu'on exécute sum(3).

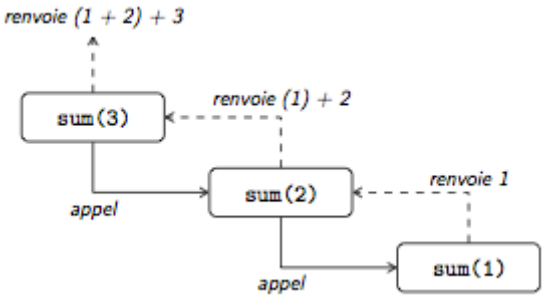

Figure 3. L'appel d'une fonction récursive entraine une succession d'appels en chaine, jusqu'à atteindre le cas de base qui va démarrer une succession de renvois de valeur à la chaine, jusqu'à l'appel initial

Les fonctions récursives peuvent parfois grandement améliorer la lisibilité du code, et sont parfois plus facile à concevoir lorsque le problème à résoudre s'y prête bien. Dès lors que la résolution d'un problème peut être faite en résolvant une version simplifiée du problème initial, il y a lieu de penser à la récursion.

Dans notre cas, la somme des n premiers entiers positifs peut être connue en utilisant le résultat de la somme des $n-1$ premiers nombres entiers, problème plus simple à résoudre.

Prenons un autre exemple, à savoir une fonction qui calcule rapidement la valeur de a^n. Pour cela, on va se baser sur les propriétés mathématiques suivantes :

$$a^n = \begin{cases} a & \text{si } n = 1 \\ (a^2)^{n/2} & \text{si } n \text{ est pair} \\ a \cdot (a^2)^{(n-1)/2} & \text{si } n > 2 \text{ et } n \text{ est impair} \end{cases}$$

Cette méthode de calcul est appelée *exponentiation rapide*. Le premier cas correspond au cas de base, et les deux cas suivants sont des cas récursifs. Voici la fonction récursive `pow` qui traduit directement en code ces trois propriétés :

```
1. def pow(a, n):
2.     if n == 1:
3.         return a
4.     if n % 2 == 0:
5.         return pow(a * a, n / 2)
6.     return a * pow(a * a, (n - 1) / 2)
```

IV-D - Module

Terminons ce chapitre en voyant comment définir ses propres *modules*. On a déjà vu comment importer un module et en utiliser les fonctions dans les chapitres précédents, en particulier à la section 3.4, où on a découvert le module `turtle`.

L'intérêt d'un module est de pouvoir y définir des fonctions que l'on va facilement pouvoir réutiliser dans d'autres programmes, ou dans d'autres modules. L'un des buts d'une fonction étant d'être réutilisée, il est dès lors important d'avoir une structure telle que le module dans un langage de programmation.

IV-D-1 - Utilisation d'un module

Revoyons rapidement comment utiliser un module. La première chose à faire consiste à *importer* le module grâce au mot réservé `import`. On peut importer un module complètement ou uniquement en importer certaines fonctions. Dans le premier cas, il faudra préfixer chaque appel de fonction par le nom du module duquel elle provient. On peut par exemple écrire le programme suivant :

```
1. import turtle
2.
3. turtle.forward(90)
4. turtle.done()
```

L'autre solution importe directement des fonctions depuis des modules et on peut ainsi les appeler directement, juste avec leur nom. Le programme précédent peut se réécrire comme suit :

```
1. from turtle import forward, done
2.
3. forward(90)
4. done()
```

Comme on l'a déjà vu, on peut ne pas lister explicitement toutes les fonctions à importer et simplement écrire `from turtle import *`. Si on fait cela, il faudra faire attention aux *conflits de noms*, c'est-à-dire si plusieurs modules importés contiennent des fonctions portant le même nom. De manière générale, on essaie dès lors d'éviter d'importer `*`, sauf lorsqu'on est certain qu'aucun conflit de noms peut se produire. Si vous n'importez, par exemple, qu'un seul module, cela ne pose aucun souci.

IV-D-1-a - Définition d'un module

La *définition d'un module* est très facile en Python, il suffit essentiellement de créer un fichier `.py` contenant des définitions de fonctions. Un module peut évidemment lui-même importer d'autres modules.

Créons par exemple un module `shapes` contenant des fonctions pour dessiner des formes à l'aide d'une tortue. Le listing suivant montre le code de ce module. On peut y voir une fonction `polygon` qui permet de dessiner des polygones réguliers et une fonction `square` pour dessiner des carrés. Dans les deux cas, on peut spécifier la longueur des côtés et la couleur du trait (noir par défaut). Pour les polygones, on doit en plus préciser le nombre de côtés voulu.

Le fichier shapes.py contient un module reprenant des fonctions pour dessiner des formes complexes à l'aide d'une tortue.

```
1. # Module shapes de dessin de formes avancées
2. # à l'aide d'une tortue
3. # Auteur : Sébastien Combéfis
4. # Version : 25 aout 2015
5.
6. from turtle import *
7.
8. def polygon(nbsides, side, col='black'):
9.     color(col)
10.     angle = 360 / nbsides
11.     i = 0
12.     while i < nbsides:
13.         forward(side)
14.         left(angle)
15.         i += 1
16.
17. def square(side, col='black'):
18.     polygon(4, side, col)
```

Ce fichier, étant enregistré sous le nom `shapes.py`, définit un module `shapes` dont on peut importer les fonctions. Par exemple, pour dessiner trois polygones de couleurs différentes en utilisant ce module, il suffit d'écrire le programme suivant :

```
1. from shape import *
2.
3. square(90)
4. polygon(6, 90, 'red')
5. polygon(10, 90, 'blue')
6.
7. done()
```

Afin que Python puisse retrouver les différents modules existants, on ne peut pas les placer n'importe où sur sa machine. On ne va pas ici rentrer dans les détails techniques de où l'interpréteur Python cherche les modules, le plus facile étant de placer le fichier du module dans le même dossier que le fichier de votre script.

IV-D-1-b - Documentation

Comme on a pu le voir précédemment, il est important d'ajouter un commentaire explicatif pour chaque fonction, afin de savoir ce qu'elle fait et comment l'utiliser. C'est évidemment encore plus important dans le cadre d'un module puisque le but est qu'il soit réutilisé par vous-même ou par d'autres programmeurs.

Pour cela, on va utiliser une forme spéciale de commentaires qui permettra d'automatiquement générer la documentation d'un module Python, appelée *Docstring* (**PEP 0257**). Le principe consiste à déclarer un littéral de type chaine de caractères comme première instruction du corps de la fonction documentée. Par convention, on délimite les littéraux avec des triples guillemets doubles, qui permettent d'écrire la chaine de caractères sur plusieurs lignes.

Voici deux exemples, dont un avec une documentation sur une seule ligne et l'autre avec une documentation sur plusieurs lignes :

```
1. def sum(n):
2.     """Renvoie la somme des entiers de 1 à n (compris)."""
3.     if n == 1:
4.         return 1
5.     return sum(n - 1) + n
6.
7.
8. def isDivisor(d, n):
9.     """Teste la divisibilité d'un nombre par un autre.
10.
11.     Pre  : d et n sont deux entiers positifs
12.            d != 0
13.     Post : la valeur renvoyée vaut True si d divise n,
14.            et False sinon
15.     """
16.     return n % d == 0
```

V - Séquence

Jusqu'à présent, on a vu comment manipuler des nombres, des booléens et des chaines de caractères. Dans ce chapitre, on va découvrir les *séquences*, un type de données permettant de représenter une suite ordonnée de données. En réalité, les chaines de caractères sont un exemple de séquence ; il s'agit en effet d'une suite de caractères rangés dans un ordre bien déterminé. Ce chapitre présente trois types de séquences de base existant en Python : les *listes*, les *tuples* et les *intervalles*. Il présente également deux *structures de données* couramment utilisées. Enfin, il termine avec le concept d'*itérateur* qui permet de parcourir facilement les éléments d'une collection d'éléments.

V-A - Liste

Une *liste* est un type de données qui permet de contenir une liste de valeurs. La manière la plus simple d'en définir une consiste à en préciser toutes les valeurs, dans l'ordre désiré et séparées par des virgules, le tout délimité par des crochets. Une *liste vide* se déclare simplement avec deux crochets ([]). Une liste contenant, dans l'ordre, les entiers de 1 à 5, se déclare comme suit :

```
1. numbers = [1, 2, 3, 4, 5]
```

La variable numbers contient donc une liste de cinq éléments, chacun étant un nombre entier. La *taille* d'une liste est le nombre d'éléments qu'elle contient; on peut l'obtenir grâce à la fonction len. On peut afficher une liste avec la fonction print et obtenir son type avec la fonction type :

```
1. print(numbers)
2. print(type(numbers))
```

L'exécution de ces deux instructions dévoile que la variable numbers contient bien une liste de cinq éléments (les nombres entiers de 1 à 5) et qu'elle est de type list :

```
[1, 2, 3, 4, 5]
```

Chaque élément de la liste possède un *indice* qui permet d'y accéder, c'est-à-dire d'obtenir sa valeur. Le premier élément d'une liste est celui d'indice 0, le deuxième celui d'indice 1…

Pour obtenir un élément de la liste, il suffit de faire suivre le nom de la variable de l'indice désiré entre crochets. L'instruction suivante affiche l'élément à l'indice 0 de la liste stockée dans la variable numbers, c'est-à-dire son premier élément :

```
1. print(numbers[0])
```

La figure 1 représente visuellement la liste numbers. Vous y voyez clairement qu'il s'agit d'une séquence ordonnée d'éléments, chacun ayant un indice. Le premier indice est 0 et le dernier indice est $n-1$, si n représente la taille de la liste.

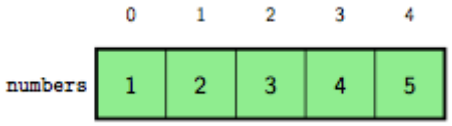

Figure 1. Une liste est une séquence ordonnée d'éléments possédant chacun un indice, utilisé pour y accéder.

À l'aide de la fonction len et avec une boucle while, on peut *parcourir* une liste, c'est-à-dire passer en revue chacun de ses éléments pour, par exemple, les afficher. Pour cela, il suffit d'utiliser une variable comme valeur d'indice, initialisée à 0 et incrémentée de 1 jusqu'à atteindre $n-1$, si n représente la taille de la liste.

L'exemple suivant parcourt tous les éléments de la liste numbers à l'aide d'une boucle while, pour les afficher l'un après l'autre :

```
1. i = 0
2. while i < len(numbers):
3.     print(numbers[i])
4.     i += 1
```

En Python, on peut également utiliser un nombre négatif comme indice, pour accéder aux éléments d'une liste à partir de la fin. Ainsi, l'indice -1 correspond au dernier élément de la liste, l'indice -2 à l'avant-dernier… comme l'illustre la figure 2. Pour parcourir tous les éléments d'une liste à l'envers, on peut écrire la boucle suivante :

```
1. i = -1
2. while i >= -len(numbers):
3.     print(numbers[i])
4.     i -= 1
```

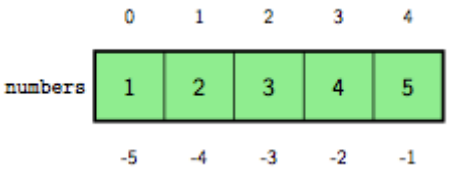

Figure 2. Un indice négatif permet d'accéder aux éléments de la liste en les comptant à partir de la fin.

Enfin, les listes sont *hétérogènes*, c'est-à-dire que leurs éléments peuvent sans soucis être de types différents. Par exemple, on pourrait représenter une adresse par une liste contenant la rue, le numéro, le code postal, la ville et le pays. Voici comment on pourrait stocker l'adresse du cabinet du Premier ministre belge :

```
1. address = ["Rue de la Loi", 16, 1000, "Bruxelles", "Belgique"]
```

V-A-1 - Modification d'une liste

Une fois créée, une liste peut être modifiée en changeant la valeur de ses éléments ou en en supprimant. On peut simplement voir une liste comme plusieurs variables, chacune étant accédée par son indice. Dès lors, on utilise simplement l'opérateur d'affectation pour *modifier un élément d'une liste*. Par exemple, pour modifier son premier élément, il suffit d'écrire :

```
1. numbers[0] = 15
```

Pour *supprimer un élément d'une liste*, il faut utiliser la fonction prédéfinie `del`, en lui indiquant l'élément à supprimer. La fonction s'occupera de décaler tous les autres éléments, de sorte qu'il n'y ait pas de *« trous »* dans la liste. Voyons cela avec l'exemple suivant :

```
1. print(numbers)
2. del(numbers[2])
3. print(numbers)
```

On supprime donc l'élément d'indice 2, à savoir le troisième élément de la liste. Le résultat de l'exécution de ces trois instructions montre bien que l'élément 3 a été retiré et que les autres éléments ont été décalés :

```
[15, 2, 3, 4, 5]
[15, 2, 4, 5]
```

V-A-1-a - Slicing

Une autre opération que l'on peut faire sur les listes est le *slicing*. Elle consiste à extraire une *sous-liste* à partir d'une liste. On utilise de nouveau les crochets, mais en spécifiant deux indices, à savoir celui du début (inclus) et celui de la fin (non inclus) de la sous-liste à extraire, séparés par un deux-points (:).

Par exemple, pour extraire la sous-liste constituée des deuxième et troisième éléments d'une liste, il faut démarrer à l'indice 1 et aller jusque l'indice 3 (c'est-à-dire l'indice 4 non inclus). L'exemple ci-dessous affiche donc [2, 3, 4] lors de son exécution :

```
1. numbers = [1, 2, 3, 4, 5]
2. print(numbers[1:4])
```

Si on ne précise pas de premier indice, la sous-liste commencera au début de la liste (comme si on avait mis 0). De même, ne pas indiquer le second indice fera terminer la sous-liste au bout de la liste (comme si on avait mis la taille de la liste). L'exemple suivant affiche [1, 2, 3] et [4, 5], à savoir les trois premiers éléments et les deux derniers :

```
1. print(numbers[:3])
2. print(numbers[3:])
```

La figure 3 montre les sous-listes correspondant à plusieurs opérations de slicing effectuées sur une liste. On y voit clairement que le premier indice renseigné est inclus alors que le second est exclu.

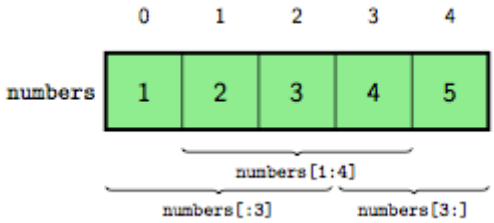

Figure 3. Le slicing permet d'extraire une sous-liste à partir d'une liste, commençant à un indice (inclus) et se terminant à un autre indice (exclu).

V-A-1-a-i - Insertion d'éléments

Le slicing permet également d'*insérer des éléments dans une liste*. Pour cela, il suffit d'affecter une nouvelle valeur à une sous-liste vide. La valeur à affecter doit être une liste. Analysons l'exemple suivant :

```
1. numbers = [1, 2, 3, 4, 5]
2. numbers[2:2] = [0]
3. print(numbers)
```

La sous-liste [2:2] ne contient aucun élément. La deuxième instruction remplace cette sous-liste vide par la liste [0], c'est-à-dire qu'on insère la valeur 0 comme nouvel élément d'indice 2 dans la liste numbers. Les autres éléments de la liste sont automatiquement décalés. L'exécution du programme affiche :

```
[1, 2, 0, 3, 4, 5]
```

La figure 4 illustre comment cette insertion se déroule. On y voit à gauche la sous-liste ainsi que la nouvelle liste qui va la remplacer et à droite le résultat après affectation de la nouvelle valeur à la sous-liste.

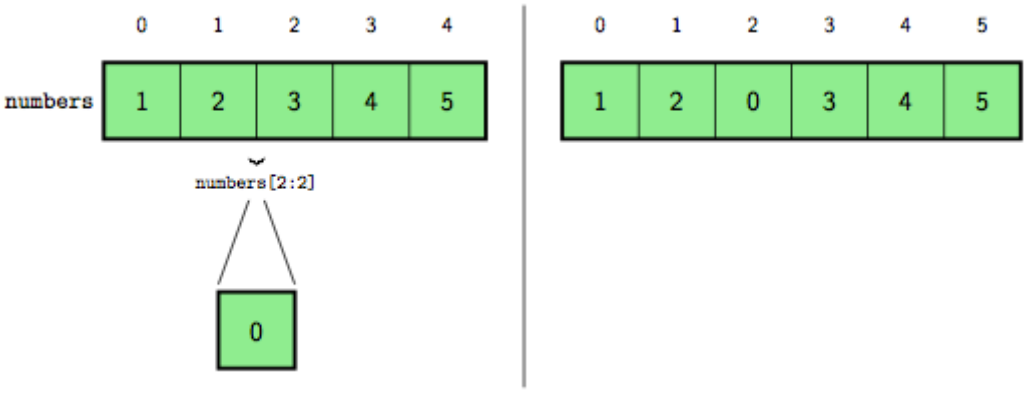

Figure 4. On peut insérer un élément dans une liste en affectant une nouvelle valeur de type liste à une sous-liste de la liste originale.

Voici trois autres exemples qui montrent comment insérer un élément au début et à la fin d'une liste, et comment en modifier un élément quelconque :

```
1. numbers[0:0] = [0]
2. print(numbers)
3.
```

```
4. numbers[6:6] = [6]
5. print(numbers)
6.
7. numbers[3:4] = [42]
8. print(numbers)
```

La première instruction remplace la sous-liste commençant à l'indice 0 et se terminant à l'indice 0 (non inclus), par la sous-liste [0]; c'est-à-dire que l'élément 0 est inséré en début de liste. La troisième instruction, quant à elle, insère l'élément 6 en fin de liste. Enfin, la cinquième instruction remplace l'élément d'indice 3 par 42. On aurait évidemment pu simplement écrire numbers[3] = 42. L'exécution de ces instructions affiche donc :

```
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 4, 5, 6]
[0, 1, 2, 42, 4, 5, 6]
```

Notez que l'on peut simplement utiliser numbers[:0] pour insérer un élément au début d'une liste et numbers[len(numbers):] pour insérer un élément à sa fin. En utilisant le slicing, on peut également remplacer ou insérer plusieurs éléments en une fois, étant donné qu'on remplace en fait une sous-liste par une nouvelle liste. Voici, par exemple, comment remplacer les quatre derniers éléments d'une liste par deux éléments :

```
1. numbers[3:7] = [3, 4]
2. print(numbers)
```

Une fois la première instruction exécutée, la liste contiendra alors cinq éléments au lieu de sept. L'exécution des deux instructions affiche :

```
[0, 1, 2, 3, 4]
```

Les trois premiers éléments de la liste ont été conservés et la sous-liste des quatre derniers éléments a été remplacée par la liste [3, 4].

V-A-1-a-ii - Suppression d'éléments

On peut supprimer *plusieurs éléments d'une liste* en combinant la fonction del avec l'opérateur de slicing. Voici un exemple où l'on supprime tous les éléments de la liste, sauf le premier et le dernier :

```
1. numbers = [1, 2, 3, 4, 5]
2. del(numbers[1:4])
3. print(numbers)
```

Le résultat de l'exécution de ces trois instructions montre bien que les trois éléments centraux de la liste ont été supprimés :

```
[1, 5]
```

La fonction del permet en fait de supprimer une variable de la mémoire. L'exemple suivant provoque une erreur d'exécution, car on tente d'afficher une variable qui n'existe plus, puisqu'elle a été supprimée :

```
1. a = 3
2. del(a)
3. print(a)
```

```
Traceback (most recent call last):
  File "program.py", line 3, in <module>
    print(a)
NameError: name 'a' is not defined
```

V-A-1-b - Concaténation et répétition

Il est possible de créer une nouvelle liste en *concaténant* deux listes existantes. On utilise pour cela l'opérateur de concaténation (+) comme pour les chaines de caractères. L'exemple suivant crée une nouvelle liste en en concaténant deux existantes :

```
1. a = [1, 2, 3]
2. b = [4, 5]
3. numbers = a + b
```

On peut également construire une liste en *répétant* plusieurs fois une liste, c'est-à-dire la concaténer plusieurs fois d'affilée. On utilise pour cela l'*opérateur de copie* (*), comme dans l'exemple suivant :

```
1. a = [1, 2] * 4
2. print(a)
```

L'effet de l'opérateur de copie est que la liste [1, 2] est copiée quatre fois, ces copies étant ensuite concaténées pour créer une nouvelle liste. L'exécution de ces deux instructions affiche donc :

```
[1, 2, 1, 2, 1, 2, 1, 2]
```

V-A-1-c - Appartenance

On doit souvent tester si un élément donné *appartient* à une liste ou non, c'est-à-dire si cet élément fait partie de ceux de la liste. Une solution immédiate à ce problème consiste à utiliser une boucle pour parcourir les éléments de la liste et rechercher celui qu'on veut. On pourrait, par exemple, définir une fonction contains qui fait ce travail :

```
1. def contains(data, element):
2.     i = 0
3.     while i < len(data):
4.         if data[i] == element:
5.             return True
6.         i += 1
7.     return False
```

La boucle compare chaque élément de la liste (data) avec la valeur recherchée (element). Si on l'a trouvée, on quitte immédiatement la fonction en renvoyant la valeur True. Sinon, on finit par atteindre la fin du corps de la fonction et on renvoie False. Mais Python propose en fait un *opérateur d'appartenance* (in) qui permet de directement tester si un élément fait partie d'une liste ou non. Les deux instructions suivantes sont équivalentes et affichent toutes les deux True :

```
1. print(contains(a, 4))
2. print(4 in a)
```

On peut également tester si un élément ne fait pas partie d'une liste en utilisant not in. Les deux instructions suivantes sont équivalentes et affichent toutes les deux False :

```
1. print(not contains(a, 2))
2. print(2 not in a)
```

V-A-1-d - Copie

Si on souhaite faire une *copie* d'une liste, on ne peut pas s'y prendre n'importe comment. Voyons l'exemple suivant :

```
1. a = [1, 2, 3]
2. b = a
3.
```

```
4. a[0] = 42
5. print(a)
6. print(b)
```

La première instruction crée une liste de trois éléments et la stocke dans la variable a. Vient ensuite la deuxième instruction dont on pourrait penser qu'elle fait une copie de la liste stockée dans a pour la stocker dans b. On modifie ensuite le premier élément de la liste a et on affiche les deux listes. Voici le résultat de l'exécution de ces instructions :

```
[42, 2, 3]
[42, 2, 3]
```

On voit que les listes a et b sont toutes les deux modifiées, car il n'y a en fait pas eu de copie de la liste. Les variables a et b permettent d'accéder à la même liste en mémoire, comme l'illustre la figure 5. Une variable contient en fait une *référence* vers la zone mémoire où se trouve stockée la valeur qu'elle représente, et pas la valeur directement. Ce sont ces *adresses mémoires* qui sont pareilles dans les variables a et b.

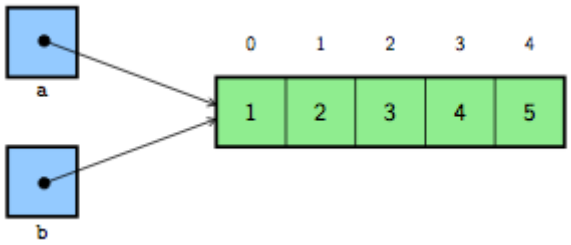

Figure 5. Un alias est une variable qui réfère une liste déjà référée par une autre variable, les deux variables permettant donc d'accéder à la même liste.

Pour réaliser une copie d'une liste, on peut utiliser le slicing. En effet, pour rappel, le slicing extrait une nouvelle liste correspondant à une sous-liste. Dès lors, pour faire une copie d'une liste existante, il suffit de faire un slicing qui reprend simplement toute la liste. On spécifie pour cela les bornes 0 et la longueur de la liste, ou juste rien. Analysons l'exemple suivant :

```
1. a = [1, 2, 3]
2. b = a[0:len(a)]
3. c = b[:]
4.
5. a[0] = -1
6. b[0] = -2
7. print(a)
8. print(b)
9. print(c)
```

On commence par créer une copie de la liste a que l'on stocke dans la variable b. On fait ensuite une copie de la liste b que l'on stocke dans la variable c. On modifie ensuite les listes a et b pour enfin afficher les valeurs des trois listes. Au vu du résultat de l'exécution, on peut confirmer qu'on a bien fait des copies des listes :

```
[-1, 2, 3]
[-2, 2, 3]
[1, 2, 3]
```

V-A-1-e - Comparaison

Enfin, un dernier type d'opération que l'on peut vouloir faire entre deux listes consiste à les *comparer*. Pour tester si deux listes sont identiques, il suffit d'utiliser l'opérateur d'égalité (==). Deux listes sont égales si elles possèdent le même nombre d'éléments et que ceux situés aux mêmes indices sont égaux.

On peut également comparer deux listes en utilisant les opérateurs >, <, >= et <=. Dans ce cas, les listes sont comparées en suivant l'*ordre lexicographique*. Les premiers éléments de chaque liste sont d'abord comparés entre

eux. Si celui de la première liste est plus petit (resp. plus grand) que celui de la seconde liste, alors la première liste est plus petite (resp. plus grande) que la seconde. Si les premiers éléments sont égaux, alors la comparaison continue avec les deuxièmes éléments…

L'exemple suivant montre plusieurs exemples de comparaisons entre deux listes :

```
1. a = [1, 2, 3]
2. b = [1, 2]
3. c = b + [3]
4.
5. print(a == b)          # Affiche False
6. print(a != c)          # Affiche False
7. print(a > b)           # Affiche True, car [1, 2, 3] > [1, 2]
8. print(a > [1, 2, 4])   # Affiche False, car [1, 2, 3] < [1, 2, 4]
```

Comme vous pouvez le constater avec la troisième comparaison faite, si tous les éléments de deux listes sont égaux mais qu'une des deux listes possède moins d'éléments que l'autre, alors elle est d'office plus petite que l'autre liste.

V-B - Tuple

Un *tuple* est une séquence non modifiable d'éléments. Tout comme les listes, un tuple peut être homogène ou hétérogène. Pour en déclarer un, il suffit de séparer ses éléments avec des virgules. Si on veut créer un tuple avec un seul élément, il suffit de placer une virgule derrière l'élément. Enfin, si on souhaite, on peut entourer les éléments du tuple de parenthèses. Voici plusieurs déclarations de tuples :

```
1. # tuple vide
2. a = ()
3. # tuple contenant un seul élément
4. b = 1,
5. c = (1,)
6. # tuple contenant trois éléments
7. d = 1, 2, 3
8. e = (1, 2, 3)
```

On peut choisir de délimiter ou non les valeurs d'un tuple par des parenthèses. Il y a néanmoins des situations où celles-ci sont obligatoires, à savoir lorsqu'on veut un *tuple vide* ou lorsqu'il y a une possible ambiguïté. Définissons par exemple une fonction `sum` qui fait la somme des éléments d'un tuple d'entiers :

```
1. def sum(values):
2.     s = 0
3.     i = 0
4.     while i < len(values):
5.         s += values[i]
6.         i += 1
7.     return s
```

Comme vous pouvez le constater, tout ce qu'on a vu à propos des listes s'applique également aux tuples, à l'exception des opérations de modification, évidemment. Supposons que l'on souhaite appeler la fonction `sum` avec le tuple `(1, 2, 3)`, on pourrait écrire :

```
1. r = sum(1, 2, 3)
```

On se retrouve malheureusement avec l'erreur d'exécution suivante :

```
Traceback (most recent call last):
  File "program.py", line 9, in <module>
    r = sum(1, 2, 3)
TypeError: sum() takes 1 positional argument but 3 were given
```

Ce qui s'est passé est que l'interpréteur Python pense qu'on tente d'appeler la fonction en lui fournissant trois paramètres, alors qu'elle n'en accepte qu'un seul. L'appel correct s'écrit donc :

```
1. r = sum((1, 2, 3))
```

Une autre solution, qui ne forcerait pas l'utilisation des parenthèses consiste à passer par une variable intermédiaire :

```
1. t = 1, 2, 3
2. r = sum(t)
```

Construire un tuple à partir de plusieurs valeurs, comme on l'a fait dans la première instruction, est une opération appelée *emballage*. On combine des valeurs pour former un tuple, référencé par une seule variable grâce à laquelle on pourra accéder aux valeurs combinées à l'aide de leur indice dans le tuple.

V-B-1 - Fonction qui renvoie plusieurs valeurs

L'un des intérêts du tuple est qu'il permet de définir des *fonctions qui renvoient plusieurs valeurs*. Il suffit en fait de renvoyer un tuple. Définissons une fonction qui cherche si un élément donné se trouve ou non dans une liste. Si c'est le cas, elle renvoie True ainsi que le plus petit indice où se trouve l'élément. Sinon, elle renvoie False et la valeur spéciale None comme indice, indiquant une absence de valeur :

```
1. def find(data, element):
2.     i = 0
3.     while i < len(data):
4.         if data[i] == element:
5.             return True, i
6.         i += 1
7.     return False, None
```

Les deux instructions return procèdent à un emballage d'une valeur booléenne et d'un nombre entier (ou None) pour former un tuple qui est renvoyé. Prenons par exemple les deux appels suivants :

```
1. values = [1, 2, 3, 4]
2.
3. print(find(values, 2))
4. print(find(values, 6))
```

L'exécution de ces instructions produit le résultat suivant indiquant que l'élément 2 a été trouvé dans la liste à l'indice 1, et que l'élément 6 n'a pas été trouvé dans la liste :

```
(True, 1)
(False, None)
```

Vous pourriez vous demander, légitimement, pourquoi on ne s'est pas contenté de renvoyer False dans le cas où l'élément recherché n'est pas dans la liste. On aurait pu, mais ce n'est pas une bonne pratique de renvoyer différents types de valeurs ; c'est même parfois impossible.

Dans l'exemple précédent, tout ce qu'on fait avec la valeur de retour, c'est l'afficher avec la fonction print. Cela ne pose donc aucun problème d'avoir deux types de données différents en valeur de retour. De même, on pourrait stocker la valeur de retour de la fonction dans une variable sans que cela ne pose de soucis :

```
1. result = find(values, 2)
```

On a précédemment vu qu'on pouvait construire un tuple à partir d'éléments simples en faisant un emballage. Il est également possible de faire l'opération inverse, appelée *déballage*, c'est-à-dire affecter chaque élément d'un tuple à une variable différente :

```
found, index = find(values, 2)
```

Dans ce cas, il n'est plus possible pour la fonction find de renvoyer tantôt un tuple à deux éléments, tantôt un seul booléen, sans quoi une erreur d'exécution se produira dans certains cas.

V-B-1-a - Opérateur de déballage

On vient de voir qu'on pouvait déballer un tuple lors d'une affectation, en ayant à gauche de l'opérateur d'affectation un tuple de variables avec autant d'éléments qu'il y en a dans le tuple à sa droite :

```
t = 1, 2, 3        # emballage
a, b, c = t        # déballage
```

On peut également déballer un tuple dans les paramètres d'une fonction lors d'un appel, avec l'*opérateur de déballage* (*). Voici une fonction qui renvoie la plus grande valeur entre trois entiers donnés :

```
def max(a, b, c):
    if a > b and a > c:
        return a
    elif b > c:
        return b
    return c
```

Si on a un tuple contenant trois nombres entiers et qu'on souhaite appeler cette fonction avec les trois nombres du tuple, il faut utiliser l'opérateur de déballage et donc écrire :

```
t = 1, 2, 3
result = max(*t)
```

V-B-1-a-i - Affectation multiple

Les propriétés d'emballage et de déballage des tuples permettent de modifier la valeur de plusieurs variables en même temps, avec des valeurs différentes. C'est ce qu'on appelle une *affectation multiple*. On peut par exemple écrire :

```
barcode, name, price = 5449000000996, "Coca-Cola 33cl", 0.70
```

L'affectation des trois variables se déroule en même temps. Cette caractéristique permet d'*échange les valeurs* de deux variables très facilement, grâce à l'instruction suivante :

```
x, y = y, x
```

V-B-1-b - Tuple nommé

Pour accéder à un élément d'un tuple, il faut utiliser son indice. Dans le cas de tuples hétérogènes, ce n'est pas toujours très intuitif. Reprenons l'exemple précédent d'un tuple qui représente un article d'un magasin :

```
item = 5449000000996, "Coca-Cola 33cl", 0.70
```

Pour accéder au code-barres, il faut écrire item[0], pour accéder à la description de l'article, on utilise item[1] et enfin item[2] permet d'avoir le prix de l'article.

Pour rendre plus explicite qu'on a un tuple hétérogène dont chaque élément représente un *attribut* différent, on peut utiliser un *tuple nommé*. Pour cela, il faut avant tout déclarer un nouveau type de tuple nommé, avec namedtuple, en

déclarant un nom et une liste d'attributs. Voici comment déclarer un tuple nommé Item composé d'un code-barres, d'une description et d'un prix :

```
1. from collections import namedtuple
2.
3. Item = namedtuple('Item', ['barcode', 'description', 'price'])
```

Une fois le nouveau type de tuple nommé créé, on peut créer des tuples nommés comme suit :

```
1. coca = Item(5449000000996, "Coca-Cola 33cl", 0.70)
```

La variable coca contient un tuple nommé qui est illustré à la figure 6. Toutes les opérations précédemment vues sur les tuples sont également applicables sur les tuples nommés. On peut, par exemple, écrire les instructions suivantes :

```
1. print(coca)
2. print(len(coca))
3. print(coca[1])
4. print(coca[1:3])
```

L'exécution de ces instructions affiche tout simplement :

```
Item(barcode=5449000000996, description='Coca-Cola 33cl', price=0.7)
3
Coca-Cola 33cl
('Coca-Cola 33cl', 0.7)
```

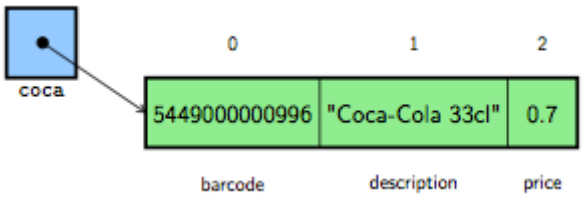

Figure 6. Un tuple nommé est un tuple dont les différents éléments sont associés à un nom permettant d'y accéder de façon intuitive.

L'avantage apporté par les tuples nommés est qu'on peut accéder à leurs attributs directement avec leur nom avec l'*opérateur d'accès* (.). On fait donc suivre le nom de la variable du tuple nommé avec le nom de l'attribut, séparé par un point. On peut, par exemple, écrire :

```
1. print(coca.description)
2. sixpackPrice = 6 * coca.price
```

V-C - Autres types de séquences

Deux autres principaux types de séquences existent en Python. On a déjà rencontré les chaines de caractères, qui sont en fait des séquences non modifiables de caractères, et il y a aussi les intervalles de nombres entiers, également non modifiables.

V-C-1 - Chaine de caractères

Les *chaines de caractères* sont des séquences non modifiables de caractères. On peut appliquer sur ces dernières tout ce qu'on a vu sur les listes, sauf les opérations de modification, évidemment. Voici un exemple qui utilise des chaines de caractères :

```
1. s = "pa" * 2
2. print(s)
```

```
3. print(len(s))
4. print("m" in s)
5. print(s[1:3])
6.
7. p = s + " est là."
8. print(p)
9.
10. p[0] = "P"
11. print(p)
```

L'avant-dernière instruction provoque une erreur d'exécution puisque les chaines de caractères ne sont pas modifiables :

```
papa
4
False
ap
papa est là.
Traceback (most recent call last):
  File "program.py", line 10, in <module>
    p[0] = "P"
TypeError: 'str' object does not support item assignment
```

L'*opérateur d'appartenance* (in) possède un comportement supplémentaire pour les chaines de caractères. Il permet de tester si une chaine donnée est une *sous-chaine* d'une autre ou non. Par exemple, on peut vérifier si une chaine de caractères s contient la sous-chaine "pa" grâce à l'instruction suivante :

```
1. print("pa" in s)
```

V-C-1-a - Intervalle

Un *intervalle* est une séquence d'entiers consécutifs, délimitée par deux bornes (une valeur minimale et une valeur maximale). On crée un nouvel intervalle à l'aide de la fonction prédéfinie range. Voici comment déclarer une séquence d'entiers correspondant à l'intervalle des nombres entiers compris entre 1 (inclus) et 5 (exclu) :

```
i = range(1, 5)
```

Comme on le verra plus loin dans ce chapitre, les intervalles sont, par exemple, utilisés pour parcourir d'autres séquences. Un intervalle est une séquence non modifiable, et on peut utiliser tout ce qu'on a vu sur les listes, à l'exception des opérations de modification, de concaténation et de répétition. Voici quelques instructions que l'on peut écrire :

```
1. print(i)
2. print(len(i))
3. print(i[2])
4. print(i[2:5])
```

Le résultat de leur exécution est conforme à celui attendu :

```
range(1, 5)
4
3
range(3, 5)
```

Par défaut, les éléments d'un intervalle sont distants de une unité. On peut spécifier un *pas* d'intervalle différent avec le troisième paramètre de la fonction range. Par exemple, on peut construire un intervalle avec les cinq premiers nombres pairs comme suit :

```
1. evens = range(2, 12, 2)
```

L'avantage des intervalles est leur occupation mémoire qui est nettement inférieure à celle d'une liste ou d'un tuple. Ces derniers doivent stocker toutes les valeurs de la séquence, alors que l'intervalle ne stocke essentiellement que les premier et dernier éléments, ainsi que le pas.

V-D - File et pile

À partir de séquences, il est possible de construire des *types abstraits de données* (TAD), c'est-à-dire une structure de données et un ensemble d'opérations qu'il est possible de faire avec. La librairie standard Python propose une série de fonctions permettant d'utiliser une liste comme une file ou une pile, deux types abstraits de données très répandus qui sont des cas particuliers de séquences.

V-D-1 - File

Une *file* est une séquence ordonnée d'éléments, c'est-à-dire une liste, tel que l'ajout d'un nouvel élément (enqueue) se fait à sa fin et le retrait d'un élément (dequeue) à sa tête. Une file est une liste de type *FIFO* (First-in First-out), c'est-à-dire que le premier élément qui y a été ajouté sera aussi le premier qui en sortira, comme l'illustre la figure 7.

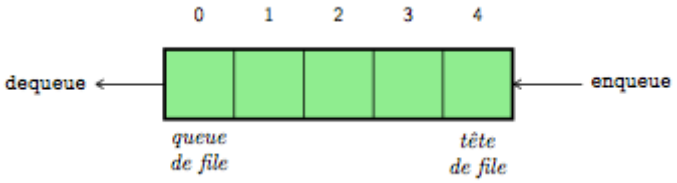

Figure 7. Une file est une liste de type FIFO (First-in First-out), c'est-à-dire que c'est le premier élément qui y a été ajouté qui en sortira en premier.

En Python, on utilise la fonction append appliquée sur la liste pour ajouter un élément à sa fin et on utilise la fonction pop pour retirer un élément à son début. Voici un exemple qui crée une file vide, lui ajoute deux éléments, puis en retire un :

```
1. queue = []                  # la file est vide
2.
3. queue.append(1)             # la file contient [1]
4. queue.append(2)             # la file contient [1, 2]
5. queue.append(3)             # la file contient [1, 2, 3]
6.
7. result = queue.pop(0)       # la file contient [2, 3]
8.
9. print(result)
10. print(queue)
```

Remarquez qu'on a utilisé l'opérateur d'accès (.) sur la variable contenant la file pour appeler les fonctions append et pop. On reviendra plus loin dans ce cours sur la raison pour laquelle on doit procéder ainsi. L'exécution de ces instructions affiche le premier élément retiré de la file, ainsi que les éléments restants dans celle-ci :

```
1
[2, 3]
```

Pour retirer le premier élément, on a dû faire l'appel pop(0). La raison pour laquelle on doit spécifier un indice est que la fonction pop est plus générale et permet en fait de retirer n'importe quel élément dans une liste, en précisant son indice.

Notez qu'on peut obtenir le même résultat avec des opérations de slicing. En effet, l'ajout d'un élément en fin de liste peut se faire en affectant une valeur à queue[len(queue):], et la suppression du premier élément peut se faire en ne gardant que la fin de liste, à savoir queue[1:]. L'exemple précédent peut donc également s'écrire comme suit :

```
queue = []

queue[len(queue):] = [1]
queue[len(queue):] = [2]
queue[len(queue):] = [3]

result, queue = queue[0], queue[1:]

print(result)
print(queue)
```

V-D-1-a - Pile

Une *pile* est également une séquence ordonnée d'éléments, mais tel que l'ajout d'un nouvel élément (push) et le retrait d'un élément (pop) se fait toujours du même côté, en haut de la pile. Une pile est une liste de type *LIFO* (Last-in First-out), c'est-à-dire que l'élément qui en sortira sera toujours celui qui y est entré en dernier comme l'illustre la figure 8.

Figre 8. Une pile est une liste de type LIFO (Last-in First-out), c'est-à-dire que l'élément qui en sort est toujours celui qui y a été ajouté en dernier.

En Python, il va falloir utiliser la fonction append, à appliquer sur la liste, pour lui ajouter un élément à sa fin et la fonction pop pour en retirer un élément à la fin.

Voici un exemple qui crée une pile vide, lui ajoute deux éléments, puis en retire un :

```
stack = []                  # la pile est vide
stack.append(1)             # la pile contient [1]
stack.append(2)             # la pile contient [1, 2]
stack.append(3)             # la pile contient [1, 2, 3]
result = stack.pop()        # la pile contient [1, 2]

print(result)
print(stack)
```

La fonction pop est, cette fois-ci, appelée sans paramètre. En effet, par défaut elle va retirer l'élément en fin de liste. L'exécution de ces instructions affiche le premier élément retiré de la pile, ainsi que les éléments restants dans celle-ci :

```
3
[1, 2]
```

De nouveau, on peut obtenir le même résultat avec des opérations de slicing. L'exemple précédent peut se réécrire comme suit :

```
stack = []

stack[len(stack):] = [1]
stack[len(stack):] = [2]
stack[len(stack):] = [3]

result, stack = stack[len(stack) - 1], stack[:len(stack)-1]

print(result)
print(stack)
```

V-E - Itérateur

Terminons ce chapitre avec le concept d'*itérateur*, mécanisme permettant de parcourir les éléments de n'importe quelle collection d'éléments. Le parcours se fait de manière transparente, sans que l'on doive se soucier de la manière avec laquelle les éléments de la collection sont organisés.

Dans le cas des séquences, étant donné que leurs éléments sont ordonnés, leurs parcours suivront toujours cet ordre. Mais comme on le verra dans la seconde partie du livre, ce n'est pas le cas pour toutes les collections. L'itérateur n'est pas nécessaire, on peut toujours utiliser une boucle while classique. Il rend néanmoins parfois le code bien plus lisible et il est également parfois la seule solution utilisable, lorsqu'il n'est, par exemple, pas possible d'accéder directement aux éléments d'une collection.

V-E-1 - Instruction for

Pour parcourir une collection d'éléments, on peut utiliser l'*instruction* for. Voyons tout de suite un premier exemple :

```
1. numbers = [1, 2, 3, 4, 5]
2. for n in numbers:
3.     print(n)
```

Lors de l'exécution de ces instructions, la variable n prend successivement comme valeur celle des éléments de la liste numbers. Étant donné que c'est une liste, et donc une séquence ordonnée, ses éléments sont parcourus dans l'ordre défini par la liste.

Avec cette méthode de parcours, on ne doit plus utiliser d'indice pour accéder aux éléments de la liste. Si on désire quand même utiliser les indices, il suffit de créer une séquence avec ces derniers, un intervalle par exemple, et de la parcourir avec l'itérateur :

```
1. numbers = [1, 2, 3, 4, 5]
2. for i in range(len(numbers)):
3.     print(numbers[i])
```

On remarque qu'on n'a spécifié qu'un seul paramètre à la fonction range et dans ce cas, la borne inférieure vaut par défaut 0, ce qui correspond bien au premier indice d'une liste.

V-E-1-a - Définition de séquence par compréhension

L'instruction for peut également être utilisée dans un autre contexte. Jusqu'à présent, pour créer une nouvelle séquence, soit on en spécifiait tous les éléments lors de son initialisation, soit on la construisait en faisant du slicing, de la concaténation ou une répétition.

Grâce à l'instruction for, on va pouvoir créer une nouvelle séquence *par compréhension*, c'est-à-dire en sélectionnant un sous-ensemble d'une séquence dont les éléments satisfont une propriété.

Prenons un exemple pour comprendre ce concept. Si on veut construire une séquence qui contient les carrés des nombres entiers compris entre 0 et 100 (inclus), on ne peut pas le faire en listant toutes les valeurs. Une première solution consiste à utiliser une boucle et la fonction append qui permet d'ajouter un élément à une liste. La boucle suivante parcourt les entiers de 0 à 100 et ajoute à la liste squares les carrés de ces entiers :

```
1. squares = []
2. i = 0
3. while i <= 100:
4.     squares.append(i ** 2)
5.     i += 1
```

Cette boucle peut évidemment se réécrire avec l'instruction for et à l'aide de la fonction range à qui on donne en paramètre, ce dernier représentant la borne supérieure de l'intervalle, non incluse : 101

```
1. squares = []
2. for i in range(101):
3.     squares.append(i ** 2)
```

Lorsqu'on a cette structure particulière de boucle, dont le but est d'initialiser une liste, on peut l'écrire de manière plus compacte pour directement définir les éléments de la liste lors de son initialisation :

```
1. squares = [i ** 2 for i in range(101)]
```

Cette notation, plus intuitive, se lit comme : *« la liste est composée d'éléments de la forme* i^2 *pour tout* i *compris entre* 0 *et* 101 *(exclu) »*. C'est ce qu'on appelle une *définition de liste par compréhension*.

De plus, il est possible d'ajouter une condition que doivent satisfaire les éléments générés, à l'aide d'une instruction if. Supposons, par exemple, qu'on ne veuille que les carrés des entiers qui sont divisibles par 3. Pour cela, on écrira simplement :

```
1. squares = [i ** 2 for i in range(101) if i % 3 == 0]
```

Cela revient en fait à écrire le code suivant :

```
1. squares = []
2. for i in range(101):
3.     if i % 3 == 0:
4.         squares.append(i ** 2)
```

VI - Algorithme

Maintenant que l'on maitrise toutes les constructions et mécanismes de base d'un langage de programmation, on va pouvoir s'attaquer à ce qui nous motive à apprendre la programmation, à savoir la *résolution de problèmes*. Ce chapitre présente les notions de *problème* et d'*algorithme* et termine en détaillant quelques exemples de problèmes avec une solution les résolvant.

VI-A - Problème et algorithme

Lorsqu'on écrit un programme, c'est dans le but de résoudre un *problème*. Cette notion a déjà été présentée à la section 4.2, avec celle de décomposition en sous-problèmes. Définir un problème consiste à identifier deux éléments :

- les données dont on dispose ;
- le résultat à produire.

Voici, par exemple, la description d'un problème en langue naturelle :

> « Étant donné un nombre naturel non nul, identifier le nombre de diviseurs stricts qu'il possède. »

On identifie immédiatement les données et le résultat. On reçoit un nombre naturel non nul (une donnée de type int) et on doit calculer le nombre de diviseurs stricts qu'il a (une donnée de type int).

Un *algorithme* est une description d'un ensemble d'opérations à effectuer, et de l'ordre dans lequel elles doivent l'être. Il s'agit de la description d'un processus qu'il est possible d'exécuter. Voici un algorithme, décrit en langue naturelle, qui résout le problème qui nous intéresse :

> « Pour trouver le nombre de diviseurs stricts du nombre n, on va examiner tous les entiers compris entre 1 (inclus) et n (exclu). Pour chacun de ces entiers, on vérifie s'il divise n ou non. En comptant le nombre d'entiers qui passent le test de la division, on obtient la solution au problème. »

Comme le résume la figure 1, un algorithme résout donc un problème. En pratique, on va vouloir résoudre une *instance* donnée d'un problème, c'est-à-dire que l'on connaitra les données dont on dispose. Dans notre exemple, une instance du problème est :

> « Identifier le nombre de diviseurs stricts que possède le nombre naturel 42. »

Alors que l'algorithme n'est qu'une description, lorsqu'il s'agit de devoir l'exécuter sur une instance donnée d'un problème, il faut l'*implémenter*. L'*implémentation* d'un algorithme consiste à en écrire une version exécutable sur une machine à l'aide d'un langage de programmation, c'est-à-dire écrire un *programme*. Le listing suivant montre une implémentation possible de l'algorithme.

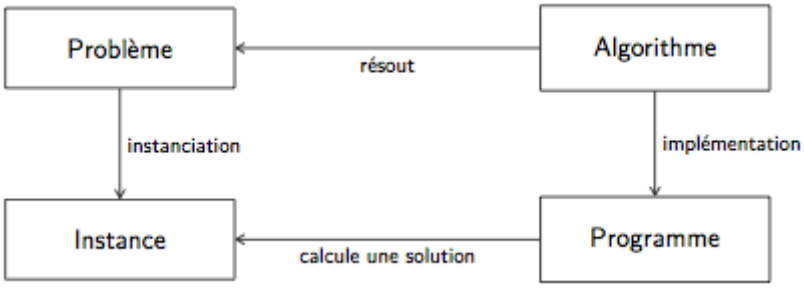

Figure 1. Un algorithme permet de résoudre un problème et pour en résoudre une instance donnée, il faut implémenter l'algorithme à l'aide d'un programme.

Le fichier divisors.py contient une fonction qui compte le nombre de diviseurs stricts d'un nombre naturel non nul.

```
1. # Compte le nombre de diviseurs stricts d'un nombre naturel non nul
2. # Pre  : n est un entier strictement positif
3. # Post : la valeur renvoyée contient le nombre de diviseurs stricts
4. #         de n
5. def nbdivisors(n):
6.     result = 0
7.     for i in range(1, n):
8.         if n % i == 0:
9.             result += 1
```

Le fichier divisors.py contient une fonction qui compte le nombre de diviseurs stricts d'un nombre naturel non nul.

```
10.     return result
11.
12. print('Le nombre 42 possède', nbdivisors(42), 'diviseurs stricts.')
```

Notez que pour un problème donné, il peut exister plusieurs algorithmes différents. Il n'y a en effet pas toujours qu'une seule approche possible à un problème. De même, un algorithme donné peut être implémenté de plusieurs façons différentes. On aurait, par exemple, pu utiliser une boucle while au lieu d'une for dans l'exemple des diviseurs.

VI-B - Exemples

Cette section présente plusieurs problèmes résolus, pour lesquels on ne donne et discute que de l'implémentation en Python, sans passer par la description préalable d'un algorithme, en langue naturelle, par exemple. C'est l'occasion de mettre ensemble tout ce qui a été vu dans les cinq chapitres précédents en action, pour écrire des fonctions dignes de ce nom et qui résolvent des problèmes précis !

VI-B-1 - Manipulation de nombres

Commençons avec quelques exemples simples qui manipulent des nombres à l'aide d'opérations arithmétiques et de fonctions du module math.

VI-B-1-a - Nombre de chiffres

Le premier problème consiste à déterminer le nombre de chiffres que possède un nombre naturel non nul. Pour cela, il suffit de procéder à des divisions entières successives par 10, jusqu'à atteindre 0. En comptant le nombre de divisions que l'on a faites, on obtient le résultat escompté :

```
1. def nbdigits(n):
2.     result = 0
3.     while n != 0:
4.         n //= 10
5.         result += 1
6.     return result
```

Vous remarquerez qu'on initialise d'abord la variable qui contiendra le résultat recherché, qu'elle est mise à jour dans la boucle et enfin renvoyée en fin de fonction. Il existe également une solution plus immédiate qui se base sur les propriétés du logarithme en base 10. En effet, en se rappelant que $\log_{10} 10 = 1$, $\log_{10} 100 = 2$, $\log_{10} 1000 = 3$… on arrive directement à la solution suivante :

```
1. from math import log10
2.
3. def nbdigits(n):
4.     return int(log10(n)) + 1
```

VI-B-1-a-i - Inversion de nombre

Dans le deuxième problème, on reçoit un nombre naturel comme donnée et il faut produire comme résultat le nombre que l'on obtiendrait en le lisant à l'envers (de droite à gauche). Pour résoudre ce problème, on va se baser sur les deux propriétés suivantes, étant donné un nombre :

- le reste de sa division entière par

$$10$$

donne son dernier chiffre ;
- sa division entière par

$$10$$

l'ampute de son dernier chiffre.

Grâce à ces deux propriétés, on va pouvoir parcourir le nombre reçu en paramètre chiffre par chiffre, en partant du dernier. On aura parcouru tous les chiffres lorsqu'on arrivera à zéro, ce qui est la condition d'arrêt de la boucle while. L'implémentation de la fonction est assez directe :

```
1. def reverse(n):
2.     result = 0
3.     while n != 0:
4.         result *= 10
5.         result += n % 10
6.         n //= 10
7.     return result
```

Comme on le constate, les chiffres extraits un à un doivent être combinés pour construire le résultat demandé. Pour cela, on initialise une variable result à zéro, et on la multiplie par 10 à chaque tour de boucle, avant de lui ajouter le chiffre extrait du nombre de départ. On obtient ainsi à la fin le nombre de départ dans le sens inverse.

VI-B-1-a-ii - Nombre de diviseurs stricts communs

Pour le troisième problème, on reçoit deux nombres naturels non nuls et il faut déterminer le nombre de diviseurs stricts qu'ils ont en commun. Pour rappel, un diviseur strict d'un nombre naturel n est un naturel d compris entre 1 et n (exclu) tel que le reste de la division entière de n par d soit nul. Pour résoudre le problème, il suffit de parcourir tous les naturels de 1 au plus petit des deux nombres reçus (exclu), et de compter ceux qui divisent les deux nombres :

```
1. def nbcommondivisors(a, b):
2.     result = 0
3.     for i in range(1, min(a, b)):
4.         if a % i == 0 and b % i == 0:
5.             result += 1
6.     return result
```

Vous pouvez constater l'utilisation de la fonction prédéfinie min qui renvoie le plus petit des deux nombres qu'elle reçoit en paramètres.

VI-B-1-a-iii - Liste des diviseurs

Dans ce dernier problème, il s'agit de calculer et renvoyer la liste de tous les diviseurs d'un nombre naturel donné. On a déjà vu précédemment comment faire pour retrouver les diviseurs d'un nombre, et il s'agira ici d'en plus les stocker dans une liste :

```
1. def divisors(n):
2.     result = []
3.     for i in range(1, n+1):
4.         if n % i == 0:
5.             result.append(i)
6.     return result
```

La variable result est initialisée à une liste vide et elle est complétée au fur et à mesure dans la boucle, par chaque diviseur, à l'aide de la fonction append appliquée sur la liste.

VI-B-1-b - Algorithme récursif

On va maintenant voir des exemples d'algorithmes récursifs, c'est-à-dire qu'ils seront implémentés par une fonction qui se rappelle elle-même. Rappelez-vous qu'il s'agit d'un choix d'implémentation, et que tous les problèmes ne se prêtent pas forcément à une implémentation récursive.

VI-B-1-b-i - Nombre de chiffres

Commençons avec une fonction récursive qui résout le problème du nombre de chiffres. On part des deux propriétés suivantes :

- si le nombre est strictement inférieur à 10, il n'a qu'un seul chiffre ;
- sinon, il possède un chiffre de plus que le nombre de chiffres du résultat de sa division entière par 10.

La première propriété correspond au cas de base qui permettra d'arrêter la récursion et la seconde propriété correspond au cas récursif. En exploitant ces deux propriétés, on obtient l'implémentation suivante :

```
1. def nbdigits(n):
2.     if n < 10:
3.         return 1
4.     return 1 + nbdigits(n // 10)
```

Notez que le choix du cas de base n'est pas unique. Pour cet exemple, on aurait pu se baser sur le fait qu'en faisant les divisions entières par 10 successives, on finirait par obtenir 0. On peut utiliser cette valeur de `n` comme cas de base, en renvoyant la valeur 0 :

```
1. def nbdigits(n):
2.     if n == 0:
3.         return 0
4.     return 1 + nbdigits(n // 10)
```

Ce choix est néanmoins moins intuitif, car cela n'a que peu de sens que de considérer que le nombre de chiffres de 0 est nul. De plus, dans la définition du problème, il est clairement mentionné que `n` doit être un nombre naturel non nul.

VI-B-1-b-ii - Factorielle

La factorielle d'un nombre naturel non nul n, notée $n!$, est le produit $n! = 1 \cdot 2 \cdot \ldots \cdot n$ et avec $0! = 1$, par convention. On peut la calculer en utilisant une simple boucle while ou for, ou alors de manière récursive en exploitant les deux propriétés suivantes :

$$\begin{cases} 0! = 1 \\ n! = n \cdot (n-1)! \quad (\text{si } n > 0) \end{cases}$$

La première propriété correspond au cas de base et la seconde au cas récursif. On obtient dès lors directement l'implémentation suivante :

```
1. def fact(n):
2.     if n == 0:
3.         return 1
4.     return n * fact(n - 1)
```

On remarque que la structure est similaire à celle de l'exemple précédent. Le corps de la fonction commence avec une instruction if pour le cas de base puis vient le return du cas récursif, qui rappelle la fonction.

VI-B-1-b-iii - Plus grand commun diviseur

Le plus grand commun diviseur (PGCD) de deux naturels non nuls est le plus grand nombre entier qui divise à la fois les deux nombres. Pour le trouver, il suffit de tester tous les diviseurs possibles, compris entre le plus petit des deux nombres et 1, et de s'arrêter dès qu'on en trouve un qui divise les deux. On peut également se tourner vers une solution récursive en exploitant les propriétés suivantes :

$$\begin{cases} pgcd(a,0) = a \\ pgcd(a,b) = pgcd(b,a) & (\text{si } a < b \text{ et } b \neq 0) \\ pgcd(a,b) = pgcd(b, a\,\%\,b) & (\text{si } a > b \text{ et } b \neq 0) \end{cases}$$

La première propriété est le cas de base et les deux autres sont des cas récursifs. On obtient directement l'implémentation suivante :

```
1. def pgcd(a, b):
2.     if b == 0:
3.         return a
4.     if a > b and b != 0:
5.         return pgcd(b, a % b)
6.     return pgcd(b, a)
```

On a donc, cette fois-ci, deux cas récursifs. On commence toujours le code avec une instruction if pour le cas de base, puis vient le premier cas récursif, dans une instruction if également. On termine enfin le corps de la boucle avec le second cas récursif.

VI-B-1-b-iv - Nombre de Fibonacci

Dans la suite des nombres de Fibonacci, chaque nombre s'obtient comme la somme des deux termes précédents de la suite, sachant que les deux premiers sont fixés. Si on note par F_n, le n^{e} nombre de la suite, on peut définir la suite comme :

$$\begin{cases} F_1 = 1 \\ F_2 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{cases}$$

On a cette fois-ci un exemple où l'on a deux cas de base (les deux premières propriétés) et un cas récursif (la troisième propriété) qui est une double récursion, c'est-à-dire qu'on se rappelle deux fois. L'implémentation d'une fonction permettant de calculer n'importe quel terme de la suite de Fibonacci est immédiat. On peut même la rendre plus courte en se rendant compte que le résultat à produire pour les deux cas de base est le même :

```
1. def fibo(n):
2.     if n == 1 or n == 2:
3.         return 1
4.     return fibo(n-1) + fibo(n-2)
```

VI-B-1-c - Interroger et manipuler des séquences

Pour terminer ce chapitre, voyons maintenant des algorithmes qui vont prendre comme données des séquences, pour les interroger ou pour effectuer des manipulations et transformations sur ces dernières.

VI-B-1-c-i - Somme des éléments

Le premier problème consiste simplement à faire la somme des éléments d'une liste ou d'un tuple de nombres. La solution est immédiate en ce sens qu'il suffit de parcourir tous les éléments de la séquence et d'en faire la somme à l'aide d'une variable :

```
1. def sumall(data):
2.     result = 0
3.     for elem in data:
4.         result += elem
5.     return result
```

On commence par initialiser une variable result à zéro. On fait ensuite une boucle pour parcourir les éléments de la séquence. Comme on n'a pas besoin des indices, une boucle for est suffisante et plus lisible. On ajoute chacun des éléments parcourus à la variable result qu'on renvoie simplement à la fin.

VI-B-1-c-ii - Valeur minimale

Le deuxième problème consiste à trouver, dans une liste ou un tuple de nombres, la plus petite valeur qu'y s'y trouve. Pour ce faire, il suffit de parcourir tous les éléments de la séquence et de retenir dans une variable la plus petite valeur qu'on a déjà rencontrée.

Si la séquence reçue en paramètre est vide, le problème n'a pas de sens. On ne peut en effet pas trouver la valeur minimale d'une séquence qui ne contient aucun élément. On peut gérer ce cas particulier de deux manières différentes : soit on interdit tout simplement que le paramètre soit une séquence vide, soit on renvoie une valeur particulière dans le cas où la liste est vide, par exemple $+\infty$. En faisant l'hypothèse que la séquence n'est pas vide, l'implémentation est assez directe :

```
1. def findmin(data):
2.     result = data[0]
3.     for elem in data:
4.         if elem < result:
5.             result = elem
6.     return result
```

On peut initialiser la variable result à la valeur de data[0] puisque la séquence n'est pas vide et contient donc au moins un élément. Ensuite, à chaque nouvel élément parcouru, on vérifie s'il est plus petit que celui stocké dans result et, si c'est le cas, on met à jour result avec cette valeur qui est donc la plus petite rencontrée jusqu'à présent.

Pour gérer le cas d'une séquence vide, il suffit simplement d'initialiser la variable result à la valeur math.inf qui représente l'infini positif, en ayant avant tout importé le module math, évidemment.

VI-B-1-c-iii - Recherche d'une sous-séquence

Le troisième problème consiste à vérifier si une séquence est une sous-séquence d'une autre. On a vu précédemment qu'on pouvait utiliser directement l'opérateur in pour les chaines de caractères, mais il n'est pas applicable aux autres séquences.

Une solution assez immédiate en Python se base sur le slicing. On va parcourir la séquence principale et en extraire successivement toutes les sous-séquences possibles et les comparer avec la sous-séquence que l'on cherche :

```
1. def issubsequence(subseq, seq):
2.     n = len(subseq)
3.     for i in range(0,len(seq)-n+1):
4.         if seq[i:i+n] == subseq:
5.             return True
6.     return False
```

On peut visualiser comment ce code fonctionne à l'aide de la figure 2. On va donc positionner la sous-séquence en face de la séquence principale, ce qui pourra se faire de l'indice 0 à l'indice len(seq) - len(subseq) (on doit faire +1 lorsqu'on utilise la fonction range, car la seconde borne n'est pas incluse). Grâce à un slicing, on découpe une partie de la séquence principale, de la même longueur que la sous-séquence, que l'on compare avec cette dernière avec l'opérateur ==. Si on a correspondance, on arrête tout en renvoyant True et si on n'a jamais trouvé correspondance, la boucle se termine et on renvoie False.

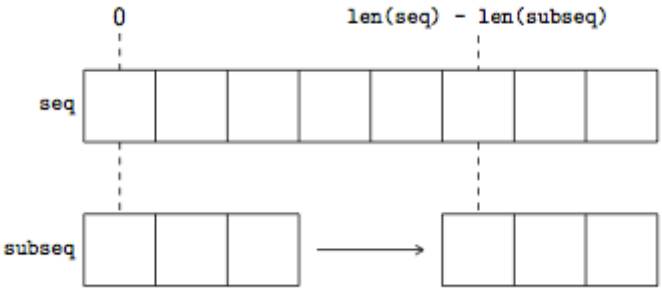

Figure 2. On teste si une chaine est une sous-séquence d'une autre en l'alignant à toutes les positions possibles et en vérifiant si les valeurs aux indices correspondants sont les mêmes.

VI-B-1-c-iv - Nombre de voyelles

Le quatrième problème consiste à compter le nombre de voyelles que contient une chaine de caractères. Pour cela, il suffit de parcourir tous les caractères de la chaine, de tester s'il s'agit ou non d'une voyelle et de les compter pour avoir le résultat attendu :

```
1. def nbvowels(s):
2.     result = 0
3.     for c in s:
4.         if c in 'aeiou':
5.             result += 1
6.     return result
```

Remarquez que la condition c in 'aeiou' est un raccourci pour c == 'a' or c == 'e' or c == 'i' or c == 'o' or c == 'u', que l'on peut se permettre grâce à la présence de l'opérateur in.

VI-B-1-c-v - Caractères uniques

Ce cinquième problème consiste à extraire, depuis une chaine de caractères, la liste des caractères uniques qui y sont présents. Il suffit pour cela de parcourir la chaine de caractères, caractère par caractère. Chaque caractère passé en revue a soit déjà été rencontré ou est un nouveau. Il faut ignorer ceux qu'on a déjà rencontrés et stocker les nouveaux. Pour cela, on va à chaque fois vérifier si on a déjà rencontré le caractère grâce à l'opérateur in.

L'implémentation est directe, et utilise une liste seen qui contient, à tout moment, les caractères uniques déjà rencontrés.

```
1. def uniquechars(s):
2.     seen = []
3.     for c in s:
4.         if c not in seen:
5.             seen.append(c)
```

```
6.      return seen
```

On verra au chapitre suivant, dans la deuxième partie de ce cours, une structure de données plus adaptée pour représenter l'ensemble des caractères déjà rencontrés.

VI-B-1-c-vi - Filtrage

Le sixième problème consiste à filter une séquence de nombres selon un critère donné. Ici, on souhaite ne garder que les nombres qui sont strictement positifs. On parcourt les éléments de la séquence reçue en paramètre, ne gardant que ceux qui sont strictement positifs, en les ajoutant dans une autre liste, initialisée à [] et renvoyée par la fonction :

```
1. def filter_positive(data):
2.     result = []
3.     for elem in data:
4.         if elem > 0:
5.             result.append(elem)
6.     return result
```

On pourrait rendre cette fonction plus générique en passant le critère à appliquer en paramètre, comme on l'a fait à la section 4.1. On définit une nouvelle fonction filter avec un paramètre accept, qui est une fonction qui doit renvoyer un booléen qui signale si on doit garder ou non l'élément qu'on lui fournit en paramètre. On peut réécrire l'exemple précédent comme suit :

```
1. def positive(n):
2.     return n > 0
3.
4. def filter(data, accept):
5.     result = []
6.     for elem in data:
7.         if accept(elem):
8.             result.append(elem)
9.     return result
10.
11. def filter_positive(data):
12.     return filter(data, positive)
```