Eléments de Programmation (en Python)

Avant-propos

Ce document est la version électronique du livre :

> **Éléments de Programmation**
> **(de l'Algorithme au Programme Python)**
> de *Frédéric Peschanski* et *Romain Demangeon*
> publié aux *Éditions Ellipses* (première édition, Septembre 2020)

https://www.editions-ellipses.fr/accueil/10671-21103-elements-de-programmation-de-lalgorithme-au-programme-python-9782340041066.html

---

En comparaison de la version publiée, ce document ne contient pas les énoncés et corrigés d'exercices qui sont disponibles dans un recueil séparé (recueil d'exercices).

Table des matières

Chapitre 1

Premiers pas

Dans ce chapitre, nous abordons nos premiers *éléments de programmation*, à partir d'un exemple de programme Python court (une seule fonction) mais complet. Parmi ces *éléments* nous étudierons dans un premier temps les notions fondamentales d'*expression* et de *fonction*. Les autres *éléments* seront étudiés dans les prochains chapitres.

1.1 Un exemple de programme

One donne ci-dessous le code Python d'une fonction dont le nom est `liste_premiers` et qui permet de calculer, pour un entier naturel `n` fixé, la liste triée des nombres premiers [1] inférieurs (strictement) à `n`.

Voici le code de cette fonction :

```
from typing import List

def liste_premiers(n : int) -> List[int]:
    """Précondition : n >= 0
       Retourne la liste des nombres premiers inférieurs à n."""

    i_est_premier : bool = False  # indicateur de primalité

    l : List[int] = []   # liste des nombres premiers en résultat

    i : int # Entier courant
    for i in range(2, n):
        i_est_premier = True

        j : int # Candidat diviseur
```

1. On rappelle qu'un nombre est premier s'il n'est divisible que par 1 ou par lui-même.

```
        for j in range(2, i - 1):
            if i % j == 0:
                # i divisible par j, donc i n'est pas premier
                i_est_premier = False

        if i_est_premier:
            l.append(i)

    return l
```

On ne cherche pas, pour l'instant, à comprendre ce que fait cette fonction (il faudra pour cela atteindre le chapitre 5, au minimum). Notre objectif est d'extraire un petit catalogue (non-exhaustif mais significatif) des *éléments de programmation* que cette fonction met en œuvre.

Avant cela, nous pouvons peut-être *tester* notre fonction dans l'interprète de commandes de Python :

```
>>> liste_premiers(30)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

Voici quelques *éléments de programmation* à l'œuvre dans le programme ci-dessus :

— **Définitions de fonction**

La construction :

```
def liste_premiers(n : int) -> List[int]:
    ... etc ...
```

est une définition de fonction dont le principe est de nommer un calcul. Ici le calcul concerne la construction d'une liste des nombres premiers, et le nom choisi pour la fonction, `liste_premiers`, semble ainsi parfaitement adapté. Ce calcul est également *paramétré* par un entier `n` appelé un **paramètre formel**. Le rôle de ce paramètre est clairement expliqué dans la documentation de la fonction dont il est expliqué qu'elle *retourne la liste des nombres premiers inférieurs à* `n`.

— **Variables**

L'identifiant `i_est_premier` est un nom de variable. Les variables s'apparentent à des cases mémoire que l'on identifie par un nom (plutôt que directement une adresse mémoire dans l'ordinateur). Elles sont abordées au chapitre 2.

— **Expressions**

On trouve de nombreuses expressions dans notre programme. C'est le cas par exemple de l'expression atomique `1`, l'expression arithmétique `i-1` ou encore l'expression arithmético-logique `i % j == 0`. Une expression dénote une *valeur* obtenue d'après un *principe d'évaluation*. Nous discutons des expressions et de leur évaluation un peu plus loin dans ce chapitre.

— **Instructions**

Une instruction est, de façon simplifiée, un *ordre* donné à l'ordinateur (dans notre cas, plus précisément, à l'interprète Python) dans le but de réaliser une *action*.

Par exemple, la construction `i_est_premier = True` est une instruction dite d'*affectation* à la variable de nom `i_est_ premier`. L'action réalisée est une écriture dans la mémoire.

La construction :

```
if i_est_premier:
   ... etc ...
```

est une instruction dite *alternative*. Les alternatives permettent les calculus conditionnés, et seront vues au chapitre 2.

La construction :

```
for i in range(1, n):
   ... etc ...
```

est une instruction dite de *boucle* (ici une boucle d'itération), que l'on abord au chapitre 3.

1.2 Notion d'expression

Nous allons commencer par expliquer de façon un peu plus approfondie le premier type fondamental d'élément de programmation que l'on nomme **expression**. Une expression en informatique est une écriture formelle textuelle que l'on peut saisir au clavier. Le langage des expressions que l'on peut saisir est un langage informatique, beaucoup plus contraint que les langages naturels comme le Français par exemple. En effet, l'ordinateur doit «comprendre» ce que l'on écrit.

Dans ce livre, les expressions sont écrites dans le langage Python, mais il existe bien sûr de nombreux autres langages informatiques (C, C++, java, ocaml, etc.), et dans tous ces langages la même notion d'expression est présente.

Les expressions sont principalement de deux types :

— **expressions atomiques** (ou **atomes**) : la forme la plus simple d'expression. On parle également d'*expressions simples*.
— **expressions composées** : expressions (plus) complexes composées de **sous-expressions**, elles-mêmes à nouveau simples ou composées.

La propriété fondamentale d'une expression est d'être *évaluable*, c'est-à-dire que chaque expression possède une **valeur** que l'on peut obtenir par calcul. Ce calcul, effectué par la machine, est appelé un **principe d'évaluation**.

1.2.1 Expressions atomiques

Une expression atomique `v` possède un type `T` et - c'est ce qui caractérise la propriété d'atomicité - sa valeur est également notée `v`.

Considérons l'interaction suivante avec l'intérprète Python :

```
>>> 42
42
```

Ici, on a saisi l'entier `42` au clavier et Python nous a répondu également `42`. Le type de cette expression atomique est `int`, le type des nombres entiers. Nous pouvons vérifier ce fait par la fonction prédéfinie `type` qui retourne la description du type d'une expression.

```
>>> type(42)
int
```

Le processus d'évaluation mis en jeu pour évaluer l'expression `42` est en fait plus complexe qu'il n'y paraît. Lorsque l'on tape `42` au clavier et qu'on soumet cette expression à l'*évaluateur* de Python, ce dernier doit transformer ce texte en une valeur entière représentable sur ordinateur. Cette traduction est assez complexe, il faut notamment représenter `42` par une suite de 0 et de 1 - les fameux *bits* d'information - au sein de l'ordinateur (on parle alors de *codage binaire*). Cette représentation dans la machine se nomme en python un **objet**. On dit que le *type de l'expression* `42` est `int` mais pour la représentation interne, on dit que la représentation de 42 en mémoire est un objet instance de la **classe** `int`.

Cette terminologie fait de Python un **langage objet** et nous reviendrons sur ce point mais pas avant le dernier chapitre du livre (et nous expliquerons pourquoi ce n'est pas un thème central de ce livre).

Le processus dit d'**auto-évaluation** des expressions atomiques n'est pas terminé. Dans un second temps, Python nous «explique» qu'il a bien interprété l'expression saisie en produisant un affichage de la valeur. Dans cette deuxième étape l'objet en mémoire qui représente 42 est converti en un texte finalement affiché à l'écran.

Pour résumer, nous avons fait la distinction entre :

- une *expression d'un type donné*, saisie par le programmeur, par exemple l'expression `42` de type `int`,
- la valeur de l'expression : un *objet représenté en mémoire* d'une *classe* donnée, par exemple la représentation interne (codée en binaire) de l'entier 42, objet de la classe `int`,
- *l'affichage de cet objet en sortie*, par exemple la suite de symboles `42` affiché par l'interprète Python.

Avec un peu de pratique, le programmeur ne voit qu'un seul 42 à toutes les étapes mais il faut être conscient de ces distinctions pour comprendre pourquoi et comment l'ordinateur est capable d'effectuer *nos* calculs.

Retenons donc **le principe simplifié d'évaluation des expressions atomiques**:

> Une expression atomique s'évalue en elle-même, directement, sans calcul ni travail particulier.

Effectuons maintenant un tour d'horizon des principales expressions atomiques fournies par Python.

1.2.1.1 Les constantes logiques (ou booléens)

La vérité logique est représentée par l'expression `True` qui signifie *vrai* et l'expression `False` qui signifie *faux*. Ces deux atomes forment le type booléen noté `bool` du nom du logicien *George Boole* qui, au XIXème siècle, a établi un lien fondamental entre la logique et le calcul.

```
>>> type(True)
bool
```

```
>>> type(False)
bool
```

Nous étudions les booléens de façon plus approfondie au chapitre 2.

1.2.1.2 Les entiers

Les entiers sont écrits en notation mathématique usuelle.

```
>>> type(4324)
int
```

Une remarque importante est que les entiers Python peuvent être de taille arbitraire.

```
>>> 232392873298372983827392847398473948374394873984792837293823922832323928732983729838273928473984739483743948739847928372938239228323239287329837298382739284739847394837439487398479283729382392283
```

Dans beaucoup de langages de programmation (exemple : le langage C) les entiers sont ceux de l'ordinateur et le résultat aurait donc été tronqué.

Sur une machine 32 bits, l'entier (signé) maximal que l'on peut stocker dans une seule case mémoire est 2^{31}.

```
>>> 2 ** 31   #  l'opérateur puissance se note **  en python.
2147483648
```

Ceci illustre un aspect important de Python : l'accent est mis sur la précision et la généralité des calculs plutôt que sur leur efficacité. Le langage C, par exemple, fait plutôt le choix opposé de se concentrer sur l'efficacité au détriment de la précision et de la généralité. On dit que Python est (plutôt) un *langage de haut-niveau*, et que le langage C est (plutôt) un *langage de bas-niveau*.

1.2.1.3 Les constantes à virgule flottante.

Les expressions atomiques `1.12` ou `-4.3e-3` sont de type flottant noté `float`.

Les flottants sont des approximations informatiques des *nombres réels* de $\mathbb{R}$, qui eux n'existent qu'en mathématiques. Dans ce livre d'introduction, nous ne nous occuperons pas trop des problèmes liés aux approximations, mais il faut savoir que ces problèmes sont très complexes et sources de nombreux *bugs* informatiques, certains célèbres comme le *bug* de la division en flottant du *Pentium*[2].

```
>>> type(-4.3e-3)
float
```

Il est important de remarquer que les entiers et les flottants sont des types *disjoints* mais comme beaucoup d'autres langages de programmation, Python convertit implicitement les entiers en flottants si nécessaire.

Par exemple :

```
>>> type(3 + 4.2)
float
```

Ici `3` est un entier de type `int` et `4.2` est de type `float`. Le résultat de l'addition privilégie les flottants puisqu'en effet on s'attend au résultat suivant :

```
>>> 3 + 4.2
7.2
```

Remarque : lors de certains calculs, on ne veut pas distinguer entre entiers et flottants (par exemple le calcul de la valeur absolue). Dans ce cas on utilisera le type `float`. En effet, dans la spécification des types pour Python[3], les entiers du type `int` doivent être acceptés en remplacement de `float`. C'est un peu comme l'ensemble des entiers que l'on peut considérer en mathématiques comme sous-ensemble des nombres réels.

1.2.1.4 Les chaînes de caractères

Les chaînes de caractères de type `str` ne sont pas à proprement parler atomiques, mais elles s'évaluent de façon similaire.

Une chaîne de caractères est un texte encadré :

— soit par des apostrophes ou guillemets simples `'` :

```
>>> 'une chaîne entre apostrophes'
'une chaîne entre apostrophes'
```

— soit par des guillemets à l'anglaise `"` ou guillemets doubles :

```
>>> "une chaîne entre guillemets"
'une chaîne entre guillemets'
```

2. à propos du Bug du Pentium : http://fr.wikipedia.org/wiki/Bug_de_la_division_du_Pentium
3. cf. PEP 484 https://www.python.org/dev/peps/pep-0484/

On remarque que Python privilégie les apostrophes pour les affichages des chaînes de caractères. Ceci nous rappelle d'ailleurs bien ici le processus en trois étapes : lecture de l'expression (avec guillemets doubles), conversion en un objet en mémoire, puis écriture de la valeur correspondante (avec guillemets simples).

Il existe d'autres types d'expressions atomiques que nous abordons pour certains dans d'autres chapitre de ce livre, mais dans l'immédiat nous nous contenterons des types `bool`, `int`, `float` et `str`.

1.2.2 Expressions composées

Les expressions composées sont formées de combinaisons de sous-expressions, atomiques ou elles-mêmes composées.

Pour ne pas trop charger ce premier chapitre nous nous limiterons dans nos exemples aux expressions arithmétiques, c'est-à-dire aux expressions usuelles des mathématiques. Nous aborderons d'autres types d'expressions lors des prochains chapitres, mais il faut retenir que la plupart des concepts étudiés ici dans le cadre arithmétique restent valables dans le cadre plus général.

Pour composer des expressions arithmétiques, le langage fournit diverses constructions, notamment :

— les expressions atomiques entiers et flottants vues précédemment
— des opérateurs arithmétiques
— des applications de fonctions prédéfinies en langage Python ou définies par le programmeur.

1.2.2.1 Opérateurs arithmétiques

Le langage Python fournit la plupart des opérateurs courants de l'arithmétique :

— les opérateurs *binaires* d'addition, de soustraction, de multiplication et de division
— l'opérateur *moins unaire*
— le parenthésage

La notation utilisée suit l'usage courant des mathématiques.

Par exemple, on peut calculer de tête assez rapidement les expressions suivantes :

```
>>> 2 + 1
3
```

```
>>> 2 + 3 * 9
29
```

```
>>> (2 + 3) * 9
45
```

```
>>> (2 + 3) * -9
-45
```

Remarque importante sur la division : en informatique on distingue généralement deux types de division entre nombres :

1. la division entière ou *euclidienne* qui est notée `//` en Python
2. la division flottante qui est notée `/`.

Voici quelques exemples illustratifs.

```
>>> 7.0 / 2.0
3.5
```

Ici on a divisé deux flottants par la division flottante. Le résultat est aussi un flottant. Divisons maintenant dans les entiers :

```
>>> 7 // 2
3
```

Ici le résultat est bien un entier : 3 est le quotient de la division entière de 7 par 2. Nous pouvons d'ailleurs obtenir le reste de la division entière avec l'opérateur *modulo* (qui est noté `%` en Python).

```
>>> 7 % 2
1
```

Le reste de la division entière de 7 par 2 est bien 1, on a : 7 qui vaut $2 * 3 + 1$

Mais maintenant, que se passe-t-il si on utilise la division flottante pour diviser des entiers ?

```
>>> 7 / 2
3.5
```

Puisque la division flottante nécessite des opérandes de type `float`, l'entier 7 a été converti implicitement en le flottant 7.0 et l'entier 2 a été converti en le flottant 2.0. C'est donc *presque* sans surprise que le résultat produit est bien le flottant 3.5.

On retiendra de cette petite digression que les divisions informatiques ne sont pas simples.

Priorité des opérateurs

Les règles que nous appliquons implicitement dans nos calculs mentaux doivent être explicitées pour l'ordinateur. Pour cela, les opérateurs sont ordonnés par priorité.

Retenons **les règles de base de priorité des opérateurs** :

— la multiplication et la division sont prioritaires sur l'addition et la soustraction
— le moins unaire est prioritaire sur les autres opérateurs
— les sous-expressions parenthésées sont prioritaires

Dans l'expression `2 + 3 * 9` ci-dessus, on évalue la multiplication *avant* l'addition. Le processus de calcul est donc le suivant :

```
2 + 3 * 9
==> 2 + 27
==> 29
```

Pour *forcer* une priorité et changer l'ordre d'évaluation, on utilise, comme en mathématiques, des parenthèses. On a ainsi :

```
(2 + 3) * 9
==> 5 * 9
==> 45
```

Exercice - Donner la valeur des expressions suivantes :

- `4 + 7 * 3 - 9 * 2`
- `(4 + 7) * 3 - 9 * 2`
- `4 + 7 * (3 - 9) * 2`
- `4 + (7 * 3 - 9) * 2`
- `4 + (7 * - 3 - 9) * 2`
- `4 + 3 / 2`
- `4 + 3 // 2`

1.2.2.2 Applications de fonctions prédéfinies

Le langage Python fournit un certain nombre de fonctions prédéfinies (en fait plusieurs centaines !). Les fonctions prédéfinies dites *primitives* sont accessibles directement.

Pour pouvoir utiliser une fonction, prédéfinie ou non, il nous faut sa **spécification**.

Prenons l'exemple de la fonction `max` qui retourne le maximum de deux nombres et qui pourrait être spécifiée de la façon suivante [4] :

```
def max(a : int, b : int) -> int:
  """Retourne le maximum des nombres a et b."""
```

Par exemple :

```
>>> max(2,5)
5
```

```
>>> max(-5.12, -7.4)
-5.12
```

Dans la spécification de fonction, par exemple `max`, on trouve trois informations essentielles :

- la **signature de la fonction**, `def max(a : int, b : int) -> int`, qui donne le *nom de la fonction* (`max`) et de ses *paramètres formels* (`a` et `b`). Elle indique aussi les *types des paramètres* ainsi que le *type de retour* de la fonction. Ici, il est indiqué que le premier paramètre (`a`) est de type `int`, le second paramètre (`b`) également de type `int` et le type de retour indiqué après la flèche `->` est également

4. En pratique, la spécification de la fonction `max` est plus complexe et ne s'applique pas qu'aux entiers, mais elle reste compatible avec la spécification proposée ici.

`int`. Ainsi la fonction prend deux valeurs entières en entrée, et retourne un entier en sortie. Un notation alternative pour cette signature est la suivante : `int * int -> int` (l'étoile `*` est le produit cartésien qui sépare les types des paramètres).
— la **description de la fonction** qui explique, de façon textuelle, le problème résolu par la fonction. Ici, la fonction *«retourne le maximum des nombres a et b»*.

Nous reviendrons longuement dans ce livre sur ce concept fondamental de spécification.

Le **principe d'évaluation des appels de fonctions prédéfinies** est expliqué ci-dessous :

Pour une expression de la forme `fun(arg1, ..., argn)`

— on évalue d'abord les expressions en arguments d'appels: `arg1, ..., argn`
— puis on remplace l'appel par la valeur calculée par la fonction.

Considérons l'exemple suivant :

```
max(2+3, 4*12)
==> max(5, 48)    [évaluation des arguments d'appel]
==> 48            [remplacement par la valeur calculée]
```

Un second exemple :

```
3 + max(2, 5) * 9
```

L'opérateur de multiplication est prioritaire mais on doit d'abord évaluer son premier argument, c'est-à-dire l'appel à la fonction `max`. Cela donne :

```
==> 3 + 5 * 9
```

Et on procède comme précédemment :

```
==> 3 + 45
==> 48
```

Ce que l'on vérifie aisément avec l'interprète python :

```
>>> 3 + max(2, 5) * 9
48
```

Les fonctions prédéfinies primitives ne sont pas très nombreuses. En revanche, Python fournit de nombreuses *bibliothèques de fonction prédéfinies* que l'on nomme des **modules**.

Pour ce qui concerne l'arithmétique, la plupart des fonctions intéressantes se trouvent dans le module `math`.

Pour pouvoir utiliser une fonction qui se trouve dans une bibliothèque, il faut tout d'abord *importer* le module correspondant à cette bibliothèque.

Par exemple, pour importer le module `math` on écrit :

```
import math # bibliothèque mathématique
```

Prenons l'exemple de la fonction de calcul du *sinus* d'un angle, dont la spécification est la suivante :

```
# fonction présente dans le module math
def sin(x : float) -> float:
    """Retourne le sinus de l'angle x exprimé en radians."""
```

Attention : la fonction `sin` du module `math` s'appelle en fait `math.sin`.

Voici quelques exemples d'utilisation.

```
>>> math.sin(0)
0.0
```

```
>>> math.sin(math.pi / 2)
1.0
```

On remarque ici que la constante π est aussi définie dans le module `math` et qu'elle s'écrit `math.pi` en Python.

```
>>> math.pi
3.141592653589793
```

1.2.2.3 Principe d'évaluation des expressions arithmétiques

Nous pouvons maintenant résumer **le principe d'évaluation des expressions arithmétiques**.

Pour évaluer une expression e:

1. si e est une expression atomique alors on termine l'évaluation avec la valeur e.
2. si e est une expression composée alors :
   a. on détermine la sous-expression e' de e à évaluer en premier (cf. règles de priorité des opérateurs).
      — si e' est unique (il n'y a qu'une seule sous-expression prioritaire) alors on lui applique le principe d'évaluation (on passe à l'étape 1 donc).
      — s'il existe plusieurs expressions à évaluer en premier, alors on procède de gauche à droite et de haut en bas.
   b. on réitère le processus d'évaluation jusqu'à sa terminaison.

Considérons l'exemple suivant :

```
(3 + 5) * ( max(2, 9) - 5 ) - 9
```

Il s'agit bien sûr d'une expression composée. Les sous-expressions prioritaires sont entre parenthèses (le parenthésage est toujours prioritaire). Il y en a deux ici : `(3 + 5)` et `( max(2, 9) - 5 )`. On procède de gauche à droite donc on commence par la première :

```
==> 8 * ( max(2, 9) - 5 ) - 9    [parenthésage - gauche]
```

La deuxième sous-expression prioritaire est la soustraction dont on évalue les arguments de gauche à droite, en commençant donc par l'appel de la fonction `max` :

```
==> 8 * ( 9 - 5 ) - 9            [appel de la fonction max]
```

Maintenant on peut évaluer la soutraction prioritaire car entre parenthèses :

```
==> 8 * 4 - 9                    [soustraction entre parenthèses]
```

On s'occupe ensuite de la multiplication prioritaire :

```
==> 32 - 9                       [multiplication prioritaire]
```

Et finalement la dernière soustraction est possible :

```
==> 23                           [soustraction]
```

```
>>> (3 + 5) * ( max(2, 9) - 5 ) - 9
23
```

Si ces principes d'évaluation peuvent apparaître un peu complexes (et en pratique, ils le sont!), mais pour comprendre les programmes, on peut la plupart du temps (en particuliers pour les entiers) retenir qu'ils sont assez proches de notre propre manière de calculer.

1.3 Définition de fonctions simples

Considérons le problème suivant :

> Calculer le périmètre d'un rectangle défini par sa `largeur` et sa `longueur`.

La solution mathématique du problème consiste à calculer la valeur de l'expression suivante :

```
2 * (largeur + longueur)
```

Par exemple, si on souhaite calculer en Python le *périmètre* d'un rectangle de `largeur` valant 2 unités (des mètres par exemple) et de `longueur` valant 3 unités, on saisit l'expression suivante :

```
>>> 2 * (2 + 3)
10
```

Le périmétre obtenu est donc de 10 unités.

Maintenant si l'on souhaite calculer le périmètre d'un rectangle de `largeur` valant 4 unités et de `longueur` valant 9 unités, on saisit alors l'expression :

```
>>> 2 * (4 + 9)
26
```

Le premier usage que nous ferons des fonctions est de permettre de paramétrer, et donc de généraliser, les calculs effectués par une expression arithmétique.

Pour le calcul du périmètre, une traduction presque directe est possible en Python :

```
def perimetre(largeur : int, longueur : int) -> int:
    """Précondition : (longueur >= 0) and (largeur >= 0)
    Précondition : longueur >= largeur
    retourne le périmètre du rectangle défini par
    sa largeur et sa longueur.
    """
    return 2 * (largeur + longueur)
```

Une fois écrite (et avant de l'étudier un peu plus loin), on peut interagir avec cette fonction avec quelques *applications* :

```
>>> perimetre(2, 3)
10
```

```
>>> perimetre(4, 9)
26
```

De ces exemples d'application de la fonction `perimetre` nous pouvons déduire un **jeu de tests** nous servant à *valider* la fonction.

```
# Jeu de tests
assert perimetre(2, 3) == 10
assert perimetre(4, 9) == 26
assert perimetre(0, 0) == 0
assert perimetre(0, 8) == 16
```

Remarque : le mot-clé `assert` permet de définir un test. Le premier test ci-dessus signifie *«Le programmeur de la fonction certifie que le périmètre d'un rectangle de largeur 2 et de longueur 3 vaut 10»*.

Si une telle assertion ne correspond pas à la définition de fonction, une exception est levée et une erreur se produit. Par exemple :

```
>>> assert perimetre(2, 3) == 12
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
  ...
----> 1 assert perimetre(2, 3) == 12

AssertionError:
```

Illustrons un autre intérêt de bien spécifier les fonctions :

```
>>> help(perimetre)
Help on function perimetre in module __main__:

perimetre(largeur:int, longueur:int) -> int
    Précondition : (longueur >= 0) and (largeur >= 0)
    Précondition : longueur >= largeur
    retourne le périmètre du rectangle défini par
```

```
    sa largeur et sa longueur.
```

La commande `help` de l'interprète Python permet d'obtenir la documentation des fonctions. Et ici notre documentation est précise (et elle sera bientôt claire).

Détaillons le processus qui conduit du problème de calcul de périmètre à sa solution informatique sous la forme de la fonction `perimetre`.

Les trois étapes fondamentales sont les suivantes :

1. la **spécification** du problème posé et devant être résolu par la fonction, comprenant :

   a. la **signature** de la fonction
   b. les **préconditions** éventuelles pour une bonne application de la fonction
   c. la **description** du problème résolu par la fonction

2. l'**implémentation** dans le **corps de la fonction** de l'**algorithme** de calcul fournissant une solution au problème posé
3. la **validation** de la fonction par le biais d'un **jeu de tests**

Détaillons ces trois étapes pour la fonction `perimetre`.

1.3.1 la spécification de la fonction

La spécification d'une fonction permet de décrire le problème que la fonction est censée résoudre. Cette spécification s'énonce dans un cadre standardisé pour ce livre.

Le rôle fondamental de la spécification est de permettre à un programmeur de comprendre comment utiliser la fonction sans avoir besoin d'une lecture détaillée de son code d'implémentation en Python. On retiendra la maxime suivante :

> On n'écrit pas une fonction uniquement pour soi, mais également et surtout pour toute personne susceptible de l'utiliser dans un futur plus ou moins proche.

La partie spécification de la fonction `perimetre` est répétée ci-dessous.

```
def perimetre(largeur : int, longueur : int) -> int:
    """Précondition : (longueur >= 0) and (largeur >= 0)
    Précondition : longueur >= largeur

    retourne le périmètre du rectangle défini par
    sa largeur et sa longueur.
    """
```

La ligne

```
def perimetre(largeur : int, longueur : int) -> int:
```

se nomme la **signature** de la fonction. Son rôle est de donner *le nom* de la fonction (qui est souvent à la fois le nom porté par le problème et sa solution), ici `perimetre`, ainsi que les noms et les types de ses *paramètres formels* (ou plus simplement *paramètres* tout court). Pour notre problème de périmètre, les paramètres se nomment naturellement `largeur` et `longueur` et sont tous les deux entiers (type `int`). Le type de la *valeur de retour* de la fonction se situe à droite de la flèche `->` est également de type `int`.

En Python les annotations de type sont optionnelles, mais on verra que la signature joue un rôle essentiel notamment lorsque l'on manipule des types de données complexes comme les listes, les ensembles ou les dictionnaires. C'et pour cela que dans ce livre nous utilisons la version typée du langage Python. La signature de la fonction peut également être résumée de la façon suivante :

```
int * int -> int
```

Dans cette écriture compacte, on retient uniquement que la fonction possèdent deux paramètres entiers, et retourne un entier en sortie (on rappelle que l'étoile `*` représente ici le produit Cartésien). On peut appeler cette écriture *le type de la fonction* `perimetre`.

La partie de la spécification en dessous de l'en-tête se trouve entre des triples guillemets `""" ... """` que l'on nomme en Python la *docstring* (chaîne de caracatères de documentation).

> **Important** : Python impose les indentations dans les programmes. Ainsi, après l'en-tête de fonction, il faut indenter par 4 espaces les lignes qui composent la spécification et le corps de la fonction.

La spécification des fonctions est souvent complétée par une ou plusieurs **préconditions** permettant de préciser les domaines de valeurs possibles pour les paramètres. La fonction `perimetre` indique deux préconditions :

1. La largeur et la longueur doivent être positives, ce qui correspond à l'expression booléenne : `(longueur >= 0) and (largeur >= 0)`
2. La largeur doit être inférieure ou égale à la longueur : `largeur <= longueur`.

On retiendra que les préconditions sont des expressions logiques Python, c'est-à-dire de type `bool`, qui s'évaluent donc soit en `True`, soit en `False`. Nous revenons sur les expressions logiques avec l'alternative dans le chapitre suivant.

Finalement, on donne la **description** en français du problème résolu par la fonction. L'objectif de cette description est d'indiquer le QUOI de la fonction, mais *sans* expliquer le COMMENT, c'est-à-dire sans préciser les détails des calculs à effectuer.

1.3.2 l'implémentation du corps de la fonction

L'implémentation de la fonction consiste à programmer en Python un algorithme de calcul permettant de résoudre le problème posé. Cette implémentation représente **le**

corps de la fonction composé d'une *instruction* ou d'une suite d'instructions.

Dans le cadre des problèmes consistant à paramétrer une expression arithmétique, comme pour notre calcul de périmètre, le corps de la fonction se résume à une instruction de la forme suivante :

```
return <expression>
```

Cette instruction s'exécute ainsi :

1. évaluation de l'`<expression>`
2. retour de la fonction à son appelant (voir plus loin) avec la valeur calculée, dite *valeur de retour*.

Dans notre cas c'est bien sûr notre expression paramétrée par la largeur et la longueur du rectangle :

```
return 2 * (largeur + longueur)
```

Important : en règle générale, un même problème peut être résolu de différentes façons. Ainsi on parle de *la* spécification et *d'une* implémentation (possible) de la fonction. Par exemple, pour notre fonction de calcul du périmètre, nous pouvons proposer une implémentation différente :

```
return (largeur + longueur) + (largeur + longueur)
```

Pour l'instant nos définitions de fonctions sont très simples et concises en comparaison de la spécification du problème. Mais plus nous avancerons dans le livre, plus cette tendance s'inversera, avec des fonctions de plus en plus complexes.

1.3.3 la validation de la fonction par un jeu de tests

Il faut distinguer deux parties distinctes dans l'approche d'un problème informatique :

— le *client* qui pose le problème à résoudre (par exemple : un enseignant qui rédige un exercice pour un cours).
— le *fournisseur* qui propose une solution informatique au problème posé par le client (comme l'étudiant qui répond à l'exercice).

Pour chaque problème posé par le client, une définition de fonction est rédigée par le fournisseur.

On retiendra donc la maxime suivante :

> un problème = une fonction pour le résoudre.

Cependant, proposer une spécification et une définition de fonction pour résoudre le problème ne suffit pas pour ce que l'on appelle l'étape de *validation*, c'est-à-dire l'acceptation (ou non !) de la solution par le client. Pour assurer cette validation, l'objectif est de proposer un **jeu de tests** sous la forme d'une suite d'applications de la fonction définie.

Une expression d'appel est de la forme suivante :

```
nom_fonction(arg1, arg2, ...)
```

Il s'agit d'appeler la fonction en donnant des valeurs précises à ses paramètres. Les expressions qui décrivent la valeur prise par les paramètres se nomment les **arguments d'appel**.

Par exemple, dans notre premier test pour la fonction `perimetre` :

```
perimetre(2, 3)
```

— l'argument d'appel pour le paramètre `largeur` est l'expression `2`
— l'argument d'appel pour la `longueur` est l'expression `3`.

Si la définition de fonction répond bien au problème posé, la valeur retournée par l'expression d'appel sera valide.

```
>>> perimetre(2, 3)
10
```

Les expressions `2` et `3` sont ici des expressions simples, mais on peut aussi utiliser des expressions de complexité arbitraire.

```
>>> perimetre(1 * 14 - 2 * 6, (3 * 121 // 11) - 30)
10
```

Question : que pensez-vous de l'expression d'appel suivante ?

```
perimetre(2, 3.1)
```

Ici, on *casse* une **règle fondamentale** dans le cadre de ce livre (et plus généralement dans le cadre des *bonnes pratiques de programmation*) :

> Les expressions d'application de fonction doivent respecter la signature et les préconditions spécifiées par la fonction. Dans le cas contraire, aucune garantie n'est fournie concernant les effets éventuellement désastreux de cette expression d'appel erronée.

Il est possible (voire probable) que Python «accepte» l'appel et produise une valeur, mais cette valeur obtenue ne correspond à aucun problème posé, en particulier celui censé être résolu par la fonction.

Dans l'application ci-dessus, l'expression `3.1` de type `float` ne respecte pas la signature de la fonction `perimetre` qui indique que le second paramètre, la `longueur`, doit être de type `int`. On dit que cette expression n'est pas valide, et dans ce cas la meilleure correction est de tout simplement l'effacer ... ou la modifier pour la rendre valide.

Dans l'expression d'application ci-dessous :

```
perimetre(-2, 3)
```

l'erreur commise, qui rend l'expression invalide, est que le premier argument d'appel est négatif alors qu'une précondition de la fonction `perimetre` indique que les valeurs prises par le premier paramètre `largeur` doivent être positives.

Dans le même ordre d'idée, l'expression :

```
perimetre(3, 2)
```

est invalide puisqu'elle contredit la précondition indiquant que la largeur doit être inférieure à la longueur.

On retiendra finalement que le jeu de tests permettant de valider la fonction doit :

- être composé uniquement d'expressions d'appel valides, qui respectent les signatures et préconditions de la fonction appelée
- couvrir suffisamment de cas permettant d'avoir confiance dans la validité de la solution proposée.

Bien sûr, c'est finalement le client qui décidera si la solution proposée est bien valide ou non.

L'écriture du jeu de test proprement dit se compose d'une suite d'assertions. Nous répétons ci-dessous le jeu de tests proposé pour la fonction périmètre.

```
# Jeu de tests
assert perimetre(2, 3) == 10
assert perimetre(4, 9) == 26
assert perimetre(0, 0) == 0
assert perimetre(0, 8) == 16
```

Chapitre 2

Instructions, variables et alternatives

Il existe une différence fondamentale entre *expressions* et *instructions*.

Les principes d'évaluation des expressions (arithmétiques) vus lors du chapitre précédent expliquent comment passer d'une expression à sa valeur. En comparaison, une instruction ne possède pas de valeur et ne peut donc pas être évaluée. On parle donc plutôt de *principe d'interprétation des instructions*.

La seule instruction que nous avons utilisée jusqu'à présent est le *retour de fonction*[1] :

```
return <expression>
```

Le **principe d'interprétation du retour de fonction** est le suivant :

- évaluation de l'`<expression>` en une **valeur de retour**
- sortie directe de la fonction (depuis l'endroit où se trouve le `return`) avec la valeur de retour calculée précédemment.

Dans ce chapitre, nous allons enrichir le répertoire d'instructions que nous pouvons placer dans le corps des fonctions.

Nous allons notamment introduire :

- les **suites d'instructions** qui permettent de combiner les instructions de façon séquentielle,
- les **variables** qui permettent de stocker en mémoire des résultats intermédiaires,
- les **alternatives** qui permettent d'effectuer des branchements conditionnels dans les calculs.

1. L'instruction `return` n'a de sens, bien sûr, que dans le corps d'une fonction.

2.1 Suites d'instructions

Il est assez naturel de décomposer des activités de façon séquentielle dans le temps. Par exemple, si on doit réaliser deux tâches successives X et Y pour obtenir le résultat Z :

- on commence par X
- puis on effectue Y
- enfin on termine par Z

Il n'est pas difficile de trouver des exemples pratiques dans la vie courante ; ainsi, en cuisine :

Recette de l'omelette (pour 3 personnes) :

- casser 6 œufs dans un saladier
- battre les œufs avec une fourchette
- ajouter 1 pincée de sel fin et 1 pincée de poivre
- verser 1 cuillerée à soupe d'huile et une noisette de beurre dans une poêle
- chauffer à feu vif et verser les œufs battus
- faire cuire jusqu'à l'obtention d'une belle omelette.

Dans le langage Python, on indique une telle suite d'instructions en les juxtaposant de façon verticale :

```
<instruction_1>
<instruction_2>
...
<instruction_n>
```

Le **principe d'interprétation des suites d'instructions** est très simple :

- on commence par l'interprétation de `<instruction_1>`. Une fois cette étape terminée,
- on poursuit par l'interprétation de `<instruction_2>`. Une fois cette étape terminée,
- ... etc ...
- on termine par l'interprétation de `<instruction_n>`.

Pour compléter notre répertoire d'instructions, nous allons également utiliser ici l'**instruction d'affichage** `print`.

L'instruction :

```
print(<expression_1>, <expression_2>, ..., <expression_n>)
```

a pour effet d'afficher les résultats des calculs des `<expression_1>`, `<expression_2>`, ..., `<expression_n>` séparées par des espaces. Un passage à la ligne est ajouté en fin d'affichage.

L'utilisation la plus simple et sans doute la plus courante du `print` est d'afficher une chaîne de caractères :

```
>>> print('Bonjour')
Bonjour
```

Voici un autre exemple avec deux expressions affichées :

```
>>> print('La valeur de pi est à peu près :', 3.14159254)
La valeur de pi est à peu près : 3.14159254
```

En Python 3, `print` est en fait une fonction qui retourne systématiquement la valeur `None` qui signifie «absence de valeur» (intéressante). Le type de `None` s'écrit `None` aussi, ce qui peut prêter à confusion [2].

Constatons ce fait :

```
>>> print('Bonjour') == None
Bonjour
True
```

Ici, l'affichage de la chaîne `'Bonjour'` s'est produit mais la valeur de l'expression est `True`.

Dans la terminologie usuelle, on dit que le `print` réalise un **effet de bord**, on peut également parler d'**action**, consistant ici l'affichage d'un texte sur la console.

Le fait de retourner `None` implique que l'instruction `print` est tout à fait inutile du point de vue du calcul. En revanche elle est très utile pour afficher des informations relatives au déroulement des calculs.

Considérons ainsi la fonction suivante :

```
def recette_omelette() -> None:
    """Affiche la recette de l'omelette.
    """
    print('- casser 6 oeufs dans un saladier')
    print('- battre les oeufs avec une fourchette')
    print('- ajouter 1 pincée de sel fin et 1 pincée de poivre')
    print("- verser 1 cuillerée à soupe d'huile")
    print('  et une noisette de beurre dans une poêle')
    print('- chauffer à feu vif et verser les oeufs battus')
    print("- faire cuire jusqu'à l'obtention d'une belle omelette.")
```

```
>>> recette_omelette()
- casser 6 oeufs dans un saladier
- battre les oeufs avec une fourchette
- ajouter 1 pincée de sel fin et 1 pincée de poivre
- verser 1 cuillerée à soupe d'huile
  et une noisette de beurre dans une poêle
- chauffer à feu vif et verser les oeufs battus
- faire cuire jusqu'à l'obtention d'une belle omelette.
```

2. La classe d'implémentation de `None` s'appelle en fait `NoneType` mais d'après PEP484 (spécifications du typage de Python) le nom du type à utiliser est bien `None`.

Les affichages ci-dessus confirment le principe d'interprétation des suites d'instructions : les instructions d'affichage sont bien interprétées les unes après les autres dans l'ordre séquentiel «du haut vers le bas». Dans le cas contraire, notre recette de cuisine ne serait pas très effective.

Notre définition de la fonction `recette_omelette` n'est pas complète : il manque la validation par un jeu de tests. Mais puisqu'elle ne prend aucun paramètre et retourne systématiquement `None`, il n'existe qu'un unique test possible.

```
# Jeu de tests
assert recette_omelette() == None
```

Ceci rappelle encore une fois que les affichages sont destinés principalement à produire des informations concernant le déroulement des calculs, mais ne participent pas directement à ces derniers.

Exercice : modifier la fonction `recette_omelette` en ajoutant un paramètre entier `nb` représentant le nombre de personnes. Les quantités affichées par la recette doivent prendre en compte ce paramètre.

Par exemple, l'appel `recette_omelette(6)` devra produire l'affichage :

```
- casser 12 oeufs dans un saladier
- battre les oeufs avec une fourchette
- ajouter 2 pincée(s) de sel fin et 2 pincée(s) de poivre
- verser 2 cuillerée(s) à soupe d'huile
  et une noisette de beurre dans une poêle
- chauffer à feu vif et verser les oeufs battus
- faire cuire jusqu'à l'obtention d'une belle omelette.
```

2.2 Variables et affectations

Les deux composants primordiaux d'un ordinateur sont :

- le **processeur** qui effectue des calculs,
- la **mémoire** qui permet de stocker de l'information de façon temporaire (mémoire centrale) ou permanente (disques).

Dans les calculs d'expressions paramétrées que nous avons vus, comme le calcul du périmètre, nous avons exploité le processeur pour effectuer des opérations arithmétiques, et nous avons exploité la mémoire avec les paramètres des fonctions.

Rappelons la définition de la fonction `perimetre` :

```
def perimetre(largeur : int, longueur : int) -> int:
    """Précondition : (longueur >= 0) and (largeur >= 0)
    précondition : longueur >= largeur

    retourne le périmètre du rectangle défini par
    sa largeur et sa longueur.
```

```
    """
    return 2 * (largeur + longueur)
```

```
# Jeu de tests
assert perimetre(2, 3) == 10
assert perimetre(4, 9) == 26
assert perimetre(0, 0) == 0
assert perimetre(0, 8) == 16
```

Dans cette fonction, les paramètres `largeur` et `longueur` sont à considérer comme des *cases mémoires*. Chaque case mémoire contient une unique valeur. Pour les paramètres de fonctions, cette valeur est décidée au moment de l'appel de la fonction et reste la même pendant toute l'exécution de la fonction. Toutes les cases mémoires sont ensuite effacées au moment du retour de la fonction.

Considérons par exemple l'appel suivant :

```
>>> perimetre(2, 3)
10
```

Pendant l'interprétation de cet appel :

— la case mémoire `largeur` contient la valeur 2
— la case mémoire `longueur` contient la valeur 3

Pour de nombreux calculs, la mémoire que l'on peut utiliser avec seulement les paramètres n'est pas suffisante, pour deux raisons principales :

1. des calculs intermédiaires doivent être conservés pour composer d'autres calculs, c'est le principe des calculatrices à mémoire.
2. la valeur contenue dans une case mémoire doit être modifiée *pendant* la durée de l'appel de fonction.

Dans ce chapitre, nous allons principalement nous intéresser au premier cas, et nous étudierons en détail le second cas lors du prochain chapitre sur *les calculs répétitifs et les boucles*.

2.2.1 Exemple de calcul avec variable : l'aire d'un triangle

Considérons le problème simple du calcul de l'aire d'un triangle à partir des longueurs de ses trois côtés. Nous allons profiter de ce problème pour rappeler les étapes principales de la définition d'une fonction.

2.2.1.1 Etape 1 : spécification de la fonction

Cette étape est primordiale. Il faut formuler le problème posé et en dégager les éléments essentiels :

— que doit calculer la fonction ?

— combien de paramètres sont nécessaires ?
— quel est le type de chacun de ces paramètres ?
— existe-t-il des préconditions sur ces paramètres ?
— quel est le type de la valeur de retour ?

Dans le cas de notre énoncé, en réponse à ces questions, on obtient :

— la fonction doit calculer *l'aire du triangle dont les côtés sont de longueurs a, b et c*
— trois paramètres sont nécessaires : les côtés a, b et c du triangle
— ces trois paramètres sont des nombres (type `float`) qui représentent des longueurs entières ou flottantes
— une précondition importante est que les trois paramètres sont des longueurs donc des nombres strictement positifs.
— une précondition plus difficile à caractériser est que toutes les valeurs positives de `a`, `b` et `c` ne définissent pas un triangle. Par exemple, il n'existe pas de triangle avec des côtés respectivement de longueur 3, 4 et 10.
— le type de retour est `float` : c'est l'aire d'un triangle.

Nous pouvons traduire ces réponses sous la forme de la spécification suivante :

```
def aire_triangle(a : float, b : float, c : float) -> float:
    """Précondition : (a>0) and (b>0) and (c>0)
       Précondition : les côtés a, b et c définissent bien un triangle.

       Retourne l'aire du triangle dont les côtés sont de longueur
       a, b, et c.
    """
```

2.2.1.2 Etape 2 : implémentation de l'algorithme de calcul

Dans cette étape, l'objectif est de *trouver* l'algorithme de calcul permettant de résoudre le problème posé. En général, c'est l'étape la plus difficile car elle demande souvent des connaissances (connaître des algorithmes proches ou similaires qui doivent être adaptés), de l'imagination (pour trouver de nouvelles approches de résolution) et de nombreux essais (au brouillon) avant d'arriver à la résoudre.

De nombreux algorithmes sont également décrits dans diverses sources : ouvrages d'algorithmique, sites internet spécialisés, Wikipedia, etc. Il faut parfois aussi *inventer* de nouveaux algorithmes. Nous verrons dans le cadre de ce livre certains algorithmes relativement complexes, mais pour l'instant, nous nous limitons à des algorithmes de calcul simples.

Le calcul de l'aire d'un triangle correspond à une simple expression paramétrée, une formule inventée selon la légende par *Héron d'Alexandrie* au premier siècle de notre ère.

Formule de Héron

Soit ABC un triangle quelconque ayant pour longueurs des côtés a, b et c.

A partir de la valeur du demi-périmètre $p = \frac{a+b+c}{2}$, l'aire du triangle est donnée par :

$$Aire = \sqrt{p(p-a)(p-b)(p-c)}$$

Une première traduction possible de cette formule en Python est la suivante :

```
import math  # nécessaire pour pouvoir utiliser la racine carrée
```

```
def aire_triangle(a : float, b : float, c : float) -> float:
    """ ... (cf. ci-dessus)
    """
    return math.sqrt( ((a + b + c) / 2)
                     * (((a + b + c) / 2) - a)
                     * (((a + b + c) / 2) - b)
                     * (((a + b + c) / 2) - c) )
```

L'implémentation de la définition ci-dessus n'est pas satisfaisante, pour deux raisons principales :

1. elle est beaucoup moins lisible que la formule mathématique de départ,
2. le calcul du demi-périmètre est effectué quatre fois alors qu'une seule fois devrait suffire.

Comme suggéré par l'énoncé mathématique, on aimerait d'abord calculer le demi-périmètre et stocker sa valeur pour pouvoir la réutiliser ensuite quatre fois. Pour cela, on a besoin d'une case mémoire que l'on introduit sous la forme d'une **variable**. Pour stocker en mémoire le résultat du demi-périmètre, la variable que nous introduisons se nomme `p` (soit le même nom que celui de la formule mathématique).

La définition de la fonction devient alors :

```
def aire_triangle(a : float, b : float, c : float) -> float:
    """ ... (cf. ci-dessus)
    """
    # demi-périmètre
    p : float = (a + b + c) / 2

    return math.sqrt(p * (p - a) * (p - b) * (p - c))
```

Cette dernière définition est nettement plus satisfaisante :

— elle est beaucoup plus lisible. En particulier, l'implémentation est très proche de la formulation mathématique du problème
— elle ne contient aucun calcul inutile : le demi-périmètre n'est calculé qu'une seule fois (cf. ci-dessous).

La variable `p` que nous utilisons est une *variable locale*, plus précisément locale à la fonction `aire_triangle`, c'est-à-dire qu'elle ne peut être utilisée que dans le corps de cette fonction, et nulle part ailleurs. Dans ce livre, toutes nos variables seront locales et nous expliquerons la raison au fur et à mesure, mais l'idée générale est que modifier la mémoire est une des principales source de bugs dans les programmes. En utilisant uniquement des variables locales, on s'assure que les modifications de mémoire sont également locales, et on évite ainsi de nombreux bugs potentiels, nos programmes sont plus sûrs.

2.2.1.3 Etape 3 : validation par un jeu de tests

La fonction `aire_triangle` est un bon exemple de fonction qui n'est pas évidente à valider. En effet, la seconde précondition - que les côtés `a`, `b` et `c` définissent bien un triangle - n'est pas triviale à garantir.

Pour notre jeu de tests, on peut faire une étude empirique, c'est-à-dire essayer de construire des triangles et d'en calculer les longueurs. Une autre approche possible, plus satisfaisante mais plus complexe, est de trouver les contraintes mathématiques sur les longueurs des côtés d'un triangle. Heureusement, pour ce cas précis les contraintes sont simples : ce sont les fameuses *inégalités triangulaires* qui encadrent la notion de distance.

Ces contraintes sont les suivantes :

- $a \leq b + c$
- $b \leq a + c$
- $c \leq a + b$

Par exemple, $a = 3$, $b = 4$ et $c = 10$ ne définit par un triangle puisque $c > a + b$.

De ces contraintes nous déduisons le jeu de tests suivant :

```
# Jeu de tests (Etape 3)
assert aire_triangle(3, 4, 5) == 6.0
assert aire_triangle(13, 14, 15) == 84.0
assert aire_triangle(1, 1, 1) == math.sqrt(3 / 16)
assert aire_triangle(2, 3, 5) == 0.0  # c'est un triangle plat...
```

Remarque : suite à notre petite réflexion concernant les tests, nous pouvons d'affiner un peu notre spécification.

```
def aire_triangle(a : float, b : float, c : float) -> float:
    """Précondition : (a>0) and (b>0) and (c>0)
    Précondition : (a <= b + c) and (b <= a + c) and (c <= a + b)

    Retourne l'aire du triangle dont les côtés sont de longueur
    a, b, et c.
    """
```

2.2.2 Utilisation des variables

Une **variable** représente donc une case mémoire dans lequel on peut stocker une valeur d'un type donné.

Une variable possède :

— un **nom** choisi par le programmeur,
— un **type** de contenu : `int`, `float`, etc.
— une **valeur** qui correspond au contenu de la case mémoire associée à la variable.

Le nom de la variable permet de **référencer** la valeur contenue dans la variable.

Les trois principales manipulations liées aux variables sont les suivantes :

1. la **déclaration de variable** avec son **initialisation** (ou première **affectation**)
2. l'**occurrence (du nom) d'une variable** au sein d'une expression
3. sa mise à jour ou **réaffectation**.

2.2.2.1 Déclaration de variable et initialisation

Pour pouvoir être utilisée, une variable doit préalablement être déclarée et (généralement) initialisée [3].

La syntaxe utilisée pour une telle déclaration est la suivante :

```
<var> : <type> = <init>
```

où `<var>` est un nom de variable et `<type>` le type du contenu de la variable (`int`, `float`, etc.) et `<init>` est une expression d'initialisation (du type `<type>`)

Il est également possible de séparer la déclaration et l'initialisation, avec la syntaxe alternative :

```
<var> : <type>
<var> = <init>
```

Par exemple, dans la fonction `aire_triangle`, nous avons effectué la déclaration suivante :

```
p : float = (a + b + c) / 2
```

On a ici déclaré une variable de nom `p` et de type `float` et on précise également son initialisation à la valeur du demi-périmètre. On aurait également pu écrire, de façon équivalente :

3. Dans certaines situations, notamment lorsque nous aborderons les itérations, on pourra omettre la valeur initiale de la variable.

```
p : float
p = (a + b + c) / 2
```

En général, nous utiliserons le premier style de déclaration, mais dans certaines situations (par exemple la décomposition des n-uplets), il sera nécessaire de déclarer le type séparément.

Important : à quelques exceptions près (notamment les variables d'itération que nous aborderons au chapitre 4), il est fortement conseillé de déclarer toutes les variables nécessaires *au début* du corps de la fonction : juste en dessous de la spécification et *avant* l'implémentation proprement dite de l'algorithme de calcul. Cela simplifie beaucoup la lecture du code de la fonction : les besoins en mémoire sont exprimés à un endroit précis, et avant la réalisation des calculs.

2.2.2.2 Occurrence d'une variable dans une expression

Après sa déclaration et son initialisation, une variable peut être utilisée dans les expressions de calcul. Il suffit pour cela d'indiquer le nom de la variable dans l'expression. On dit que l'expression contient **une occurrence de la variable**.

Par exemple, dans la fonction `aire_triangle`, l'expression :

```
(p - a)
```

contient une occurrence de la variable `p` (ainsi qu'une occurrence du paramètre `a`).

De même, l'expression :

```
math.sqrt(p * (p - a) * (p - b) * (p - c))
```

contient quatre occurrences de la variable `p`.

Le **principe d'évaluation** d'une expression **occurrence de variable** `<var>` dans une expression est le suivant :

> L'occurrence de la variable est remplacée par la valeur qu'elle contient.

Donc si la valeur de la variable `p` est 10, la valeur de l'expression atomique `p` est également 10. En fait cela fonctionne exactement comme une calculatrice à mémoire, il s'agit d'une simple opération de *lecture mémoire*.

Remarque : le contenu d'une variable est une valeur et non une expression. Ainsi, si l'on demande plusieurs fois la valeur d'une même variable, il n'y a aucun calcul supplémentaire effectué. Par exemple, dans la fonction `aire_triangle`, le calcul du demi-périmètre est effectué une fois pour toute au moment de l'initialisation de la variable `p`. Les quatre occurrences de la variable `p` dans l'expression du calcul de l'aire ne conduisent à *aucun* calcul supplémentaire du demi-périmètre.

2.2.2.3 Affectation de variable

La syntaxe :

`<var> = <expression>`

est appelée une **affectation de variable** (on pourrait dire *réaffectation* puisque dans la plupart des cas il y a déjà une valeur initiale).

Le **principe d'interprétation des affectations** est le suivant :

- on évalue tout d'abord l'`<expression>`
- on place la valeur obtenue à l'étape précédente comme contenu de la variable `<var>`.

Par exemple, si on suppose `a=2`, `b=3` et `c=5` alors l'initialisation :

```
p : float = (a + b + c) / 2
```

place comme contenu de la variable `p` la valeur $\frac{2+3+5}{2}$ c'est-à-dire 10. A l'issue de cette initialisation, on pourra représenter explicitement la case mémoire correspondante de la façon suivante :

variable	p
valeur	10

Lors d'une affectation, la valeur contenue dans la case mémoire associée à la variable est *écrasé* par la nouevelle valeur calculée.

Considérons la suite d'instructions suivante :

```
n : int = 0

m  : int = 58

n = m -16

m = m + 1
```

Question : Quels sont les contenus des variables `n` et `m` après interprétation de cette suite d'instructions ?

Pour répondre à cette question précisément, nous pouvons placer quelques `print` judicieux et décortiquer l'interprétation de cette suite d'instructions.

```
n : int = 0
print("Initialisation de n :")
print("   n =", n)
# remarque : m n'existe pas encore

m : int = 58
```

```
print("Initialisation de n :")
print("   n =", n)
print("   m =", m)

n = m - 16
print("Mise à jour de n :")
print("   n =", n)
print("   m =", m)

m = m + 1
print("Mise à jour de m :")
print("   n =", n)
print("   m =", m)
```

Les affichages produits sont les suivants :

```
Initialisation de n :
   n = 0
Initialisation de n :
   n = 0
   m = 58
Mise à jour de n :
   n = 42
   m = 58
Mise à jour de m :
   n = 42
   m = 59
```

Lors de la première étape, la variable `n` est initialisée à la valeur `0`, on peut donc la représenter ainsi :

variable	`n`
valeur	0

Bien sûr, la variable `m` n'existe pas encore à ce stade donc tenter de l'afficher conduirait à une erreur :

```
n : int = 0
print("Initialisation de n :")
print("   n =", n)
print("   m =", m)

Initialisation de n :
   n = 0
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
   ...
```

```
      3 print("Initialisation de n :")
      4 print("   n =", n)
----> 5 print("   m =", m)

NameError: name 'm' is not defined
```

Lors de la seconde étape, la variable `m` est à son tour initialisée. Puisque les deux variables existent, la table variable/valeur possède désormais deux colonnes :

variable	`n`	`m`
valeur	0	58

L'étape suivante est une affectation de la variable `n`. Puisque cette variable possède déjà une valeur, cette dernière sera écrasée par la nouvelle valeur calculée.

L'interprétation de :

```
n = m - 16
```

procède ainsi :

1. l'expression `m - 16` est évaluée en premier.
   — la valeur de la sous-expression atomique `m` est 58 : la valeur de la variable `m`
   — la valeur de l'expression complète est donc $58 - 16$ soit 42
2. la valeur obtenue - donc 42 - est placée comme *nouveau* contenu de la variable `n`.

La table des variables devient donc :

variable	`n`	`m`
valeur	42	58

Selon les mêmes principes, la table des variables après la dernière instruction de affectation `m = m + 1` est la suivante :

variable	`n`	`m`
valeur	42	59

Tout se passe donc comme si nous avions «ajouté» 1 au contenu de la variable `m`. Mais si l'on veut être précis, il faut plutôt dire que :

1. nous avons d'abord récupéré la valeur de `m`,
2. puis nous avons calculé `58 + 1`, donc 59,
3. et nous avons stocké cette valeur 59 comme nouvelle valeur de la variable `m`.

Remarque : ajouter ou retrancher 1 d'une variable entière est si courant que l'on donne un nom spécifique à ces opérations :

- une instruction de la forme `v = v + 1` où `v` est une variable (de type `int`) se nomme une **incrémentation** de la variable `v`. On peut aussi écrire cette instruction de façon compacte : `v += 1`.
- une instruction de la forme `v = v - 1` où `v` est une variable (de type `int`) se nomme une **décrémentation** de la variable `v`. L'écriture compacte correspondante est `v -= 1`.

2.2.2.4 Complément : règles de base concernant le nommage des variables et des fonctions

Le nom de variable est très important car il véhicule l'intention du programmeur concernant le rôle de la variable. Il ne faut pas hésiter à prendre des noms de variables explicites et suffisamment longs (sans exagérer bien sûr...).

La convention que nous suivrons [4] est que les noms de variables sont formés de mots en minuscules en utilisant uniquement les lettres de `a` à `z` sans accent. Si plusieurs mots sont combinés pour créer le nom, ils doivent être séparés par le caractère souligné '`_`'. On peut éventuellement accoler des chiffres en fin du nom de variable (sans les séparer des lettres précédentes).

Voici quelques exemples de noms de variable valides :

- `compteur`
- `plus_grand_nombre`
- `calcul1`, `calcul2`
- `min_liste1` , `min_liste2`

Les noms suivants ne respectent pas les conventions de nommage :

- `Compteur` : **incorrect** car il y a un `C` majuscule (seules les minuscules sont autorisées).
- `plusgrandnombre` : **incorrect** car il n'y a pas de séparation avec des `_` entre les éléments du nom
- `1calcul` : **incorrect** car il y a un chiffre au début du nom (alors que les chiffres ne doivent être qu'en fin)
- `min_liste_1` : **incorrect** car le chiffre est séparé du reste par `_`
- `élève` : **incorrect** car les accents sont interdits

2.3 Alternatives

Lors d'un calcul, il est souvent nécessaire d'effectuer des choix de sous-calculs dépendants d'une condition donnée. L'expression d'un tel choix se fait par l'intermédiaire d'une

4. Le document PEP-8 propose un ensemble de convention à suivre concernant l'écriture des programmes Python, cf. https://www.python.org/dev/peps/pep-0008/

instruction nommée *alternative*.

Pour illustrer l'intérêt et l'usage des alternatives, considérons la définition mathématique de la *valeur absolue* d'un nombre.

Définition : La valeur absolue d'un nombre x est notée $|x|$ et définie ainsi :

$$|x| = \begin{cases} x & \text{si } x \geq 0 \\ -x & \text{sinon} \end{cases}$$

Pour calculer la valeur absolue d'un nombre, nous avons ici à faire un **choix** entre deux calculs selon le signe de x.

Notre objectif est de définir une fonction `valeur_absolue` réalisant ce calcul avec choix, avec la spécification suivante :

```
def valeur_absolue(x : float) -> float:
    """Retourne la valeur absolue de x.
    """
```

Pour implémenter l'algorithme décrit ci-dessus, nous devons introduire les alternatives simples.

2.3.1 Syntaxe et interprétation des alternatives simples

La syntaxe des alternatives simples est la suivante :

```
if <condition>:
    <conséquent>
else:
    <alternant>
```

où

- la `<condition>` est une **expression booléenne** (à valeur dans `bool` donc soit `True` soit `False`)
- le `<conséquent>` est une instruction ou suite d'instructions
- l'`<alternant>` est également une instruction ou suite d'instructions.

Remarques :

- les « `:` » sur les lignes `if` et `else` sont importants ! Ils font partie de la syntaxe de l'alternative
- bien noter l'indentation de **4 espaces** avant les instructions du `<conséquent>` ou celles de l'`<alternant>`

Le **principe d'interprétation des alternatives simples** est le suivant :

- on évalue tout d'abord la `<condition>`

— si la valeur de la condition est `True` alors on interprète *uniquement* le `<conséquent>`
— sinon, la valeur de la condition est `False` et on interprète *uniquement* l'`<alternant>`.

On réalise donc ici un *choix exclusif* entre «interpréter le conséquent» ou «interpréter l'alternant» selon le résultat de l'évaluation de la condition de l'alternative.

Grâce à l'alternative, nous pouvons traduire presque directement le calcul mathématique de valeur absolue :

```
def valeur_absolue(x : float) -> float:
    """Retourne la valeur absolue de x.
    """
    #  stockage de la valeur absolue
    abs_x : float = 0 # Choix arbitraire de la valeur initiale
                      # qui pourrait donc être omise
    if x >= 0:
        abs_x = x  # conséquent
    else:
        abs_x = -x # alternant

    return abs_x
```

```
# Jeu de tests
assert valeur_absolue(3) == 3
assert valeur_absolue(-3) == 3
assert valeur_absolue(1.5 - 2.5) == valeur_absolue(2.5 - 1.5)
assert valeur_absolue(0) == 0
assert valeur_absolue(-0) == 0
```

Décrivons étape-par-étape l'interprétation de l'expression `valeur_absolue(3)` :

La première instruction est l'initialisation de la variable `abs_x` à la valeur 0. Ce choix est arbitrarire mais au moins on a une valeur disponible dans la case mémoire dès le début. La table des variables est donc :

variable	`abs_x`
valeur	0

La seconde instruction est l'alternative :

— la condition `x >= 0` est tout d'abord évaluée : puisque `x` vaut 3 la valeur de la condition est `True`. On évalue donc uniquement le conséquent `abs_x = x`. La table des variables devient :

variable	`abs_x`
valeur	3

La troisième et dernière instruction de la suite est `return abs_x`. On sort donc de la fonction avec la valeur de `abs_x` en retour, donc la valeur 3 qui est bien la valeur absolue attendue.

Recommençons avec cette fois-ci l'expression `valeur_absolue(-3)` :

La première instruction est l'initialisation de la variable `abs_x` à la valeur 0. La table des variables est donc :

variable	`abs_x`
valeur	0

La seconde instruction est l'alternative :

— la condition `x >= 0` est tout d'abord évaluée : puisque `x` vaut -3 la valeur de la condition est `False`. On évalue donc uniquement l'alternant `abs_x = -x`. La table des variables devient :

variable	`abs_x`
valeur	3

La troisième et dernière instruction de la suite est `return abs_x` donc on sort de la fonction avec la valeur de `abs_x` en retour, donc la valeur 3 qui est bien la valeur absolue attendue.

2.3.1.1 Complément : sortie anticipée d'une fonction

L'implémentation de la fonction `valeur_absolue` utilise la variable `abs_x` pour stocker la valeur absolue à calculer et à retourner. Une question que l'on peut se poser est la suivante :

> la variable `abs_x` est-elle nécessaire dans le calcul effectué ?

Dans le cas de `aire_triangle`, la variable `p` utilisée pour stocker la valeur du demi-périmètre était, elle, nécessaire pour éviter de répéter inutilement plusieurs fois le même calcul.

En revanche, dans `valeur_absolue` la variable `abs_x` n'est pas nécessaire au calcul. Par exemple, si le nombre `x` est positif, on sait que `x` lui-même est la valeur absolue et donc on pourrait sortir de la fonction et retourner la valeur de `x` directement. De même si `x` est négatif, on peut sortir directement de la fonction avec la valeur de `-x`. En exploitant cette propriété du `return` de sortir directement d'une fonction, on peut proposer la définition suivante :

```
def valeur_absolue(x : float) -> float:
    """  ... cf. valeur_absolue ...
```

```
    """
    if x >= 0:
        return x  # conséquent
    else:
        return -x # alternant
```

Une question se pose finalement ici : Quelle définition est à privilégier ?

En fait, il serait dommage de trancher trop rapidement cette question. Du point de vue de la concision, la seconde définition est sans doute préférable : on a économisé une variable et quelques lignes de programme. En revanche, on peut trouver la première définition avec variable plus simple à comprendre. Notamment, on sait qu'après l'alternative la variable `abs_x` contient dans tous les cas la valeur absolue du paramètre `x`. Nous avons effectué le calcul complet avant de sortir de la fonction. L'utilisation un peu trop approximative du `return` peut conduire à des *bugs* parfois difficiles à dénicher.

En pratique, un programmeur aguerri choisira sans doute la seconde solution, du fait de sa concision.

2.3.2 Expressions booléennes

La **condition d'une alternative** est une **expression booléenne** de type `bool`.

Cela signifie qu'il n'y a que deux valeurs possibles :

— soit la valeur est `True`, ce qui signifie que la condition est vraie (et qu'il faut interpréter le conséquent uniquement)
— soit la valeur est `False`, ce qui signifie que la condition est fausse (et qu'il faut interpréter l'alternant uniquement)

Dans le cadre de calculs numériques, les expressions booléennes sont le plus souvent constituées :

— de comparaisons de nombres (`x` et `y` sont du type `int` ou `float`) :

opérateur	notation Python
égalité	`x == y`
inégalité	`x != y`
inférieur strict	`x < y`
inférieur ou égal	`x <= y`
supérieur strict	`x > y`
supérieur ou égal	`x >= y`

— d'expressions composées d'**opérateurs logiques** (`a` et `b` sont du type `bool`) :

opérateur	notation Python
conjonction (*et logique*)	`a and b`
disjonction (*ou logique*)	`a or b`

opérateur	notation Python
négation (*non logique*)	`not a`

Attention : il est important de distinguer le symbole `=` qui exprime une affectation de variable et le symbole `==` qui représente l'opérateur d'égalité. Des *bugs* célèbres proviennent de la confusion entre ces deux symboles dans le cadre du langage C. Heureusement, cette confusion est détectée par l'interprète Python.

```
>>> 42 == 21 * 2  ## ok, c'est une égalité
True
```

```
>>> 42 = 21 * 2 # oops, ce n'est pas une égalité
                # mais une tentative ratée d'affectation
```

```
  42 = 21 * 2
             ^
SyntaxError: can't assign to literal
```

2.3.2.1 Principe d'évaluation de la négation

Le **principe d'interprétation de la négation** est le suivant.

Dans `not <expression>` (avec `<expression>` de type `bool`) :

- on évalue tout d'abord `<expression>`
  - si la valeur obtenue est `True` alors on renvoie `False`
  - si la valeur obtenue est `False` alors on renvoie `True`

Par exemple :

```
>>> 3 < 9
True
```

```
>>> not (3 < 9)
False
```

```
>>> 9 < 3
False
```

```
>>> not (9 < 3)
True
```

Remarque : le `not` est prioritaire sur les comparateurs (`<`, `>`, `<=`, etc.), il faut donc bien placer les parenthèses.

Par exemple :

```
not 9 < 3
```

signifie :

```
(not 9) < 3
```

qui ne veut en fait rien dire puisque l'on essaye de comparer un booléen et un entier !

2.3.2.2 Principe d'évaluation de la conjonction

Le **principe d'évaluation de la conjonction (*et logique*)** est le suivant :

Dans `<expression1> and <expression2>`

où `<expression1>` et `<expression2>` sont des expressions booléennes

- on évalue tout d'abord `<expression1>` ce qui produit une certaine valeur booléenne b_1
  - si b_1 est la valeur `False` alors on *stoppe immédiatement le calcul* et l'expression finale vaut `False`.
  - si b_1 est la valeur `True` alors on évalue `<expression2>` ce qui produit une certaine valeur b_2
    - la valeur finale de la conjonction est b_2.

Ce mode d'évaluation est dit *paresseux* car si la première expression est fausse, alors il est impossible que la conjonction soit vraie. Autrement dit : `a and b` ne peut être vraie si `a` est faux, ce qui est tout à fait logique !

Par exemple :

```
>>> (3 <= 9) and (9 <= 27)
True
```

Pour réaliser l'évaluation de cette expression :

- on évalue tout d'abord `(3 <= 9)` qui produit la valeur `True`
  - on évalue donc `(9 <= 27)` ce qui produit la valeur `True`
  - la valeur finale de la conjonction est donc `True`

```
>>> (3 <= 9) and (9 >= 27)
False
```

Pour cette expression :

- on évalue tout d'abord `(3 <= 9)` qui produit la valeur `True`
  - on évalue donc `(9 >= 27)` ce qui produit la valeur `False`
  - la valeur finale de la conjonction est donc `False`

```
>>> (3 >= 9) and (9 <= 27)
False
```

Pour cette expression :

- on évalue tout d'abord `(3 >= 9)` ce qui produit la valeur `False`
  - on stoppe donc immédiatement le calcul et l'expression entière vaut `False`.

```
>>> (3 >= 9) and (9 >= 27)
False
```

Cette expression s'évalue ainsi :

— on évalue tout d'abord `(3 >= 9)` ce qui produit la valeur `False`
— on stoppe donc immédiatement le calcul et l'expression entière vaut `False`.

Remarque : on voit bien dans les deux derniers exemples que puisque la première est expression est fausse, la conjonction entière est fausse et ce, indépendamment de la valeur de la seconde expression (qui n'est donc, en fait, pas calculée).

L'intérêt de ce mode d'évaluation est multiple :

— on peut exploiter le fait que la seconde expression n'est évaluée que si la première est vraie.
— si la seconde expression nécessite un calcul coûteux, alors ce calcul n'est pas effectué si la première expression est fausse.

Pour illustrer l'intérêt du calcul paresseux, considérons la fonction suivante :

```
def est_divisible(n : int, m : int) -> bool:
    """Précondition : (n >= 0) and (m >= 0)
    Retourne True si n est divisible par m, False sinon.
    """
    return (m != 0) and (n % m == 0)
```

```
# Jeu de tests
assert est_divisible(5, 2) == False
assert est_divisible(8, 2) == True
assert est_divisible(8, 0) == False
```

La fonction `est_divisible` permet de tester si un entier naturel `m` divise un autre entier naturel `n`. Cette fonction retourne un booléen et est de ce fait ce que l'on appelle un **prédicat**.

Le test de divisibilité consiste à vérifier si le reste de la division entière de `n` par `m` vaut `0`, soit la condition suivante en Python :

```
n % m == 0
```

Cependant, cette condition n'est vérifiable que si `m` est différent de 0 sinon la division et le reste ne sont pas définis. Or, la spécification de la fonction accepte que `m` soit égal à 0.

On a donc contraint la condition de l'alternative pour finalement obtenir :

```
(m != 0) and (n % m == 0)
```

D'après le principe d'évaluation de la conjonction :

— si `m` est strictement positif alors `m != 0` retourne `True` et on évalue alors `n % m == 0`
— en revanche, si `m` vaut zéro alors `m != 0` retourne `False` et la conjonction entière vaut `False`

— en particulier, quand `m` vaut zéro on n'évalue donc pas `n % m == 0` et heureusement car le reste n'est pas défini dans ce cas !

2.3.2.3 Principe d'évaluation de la disjonction

Le **principe d'évaluation de la disjonction (ou logique)** est le suivant : soit

`<expression1> or <expression2>`

où `<expression1>` et `<expression2>` sont des expressions booléennes

— on évalue tout d'abord `<expression1>` ce qui produit une certaine valeur booléenne b_1
  — si b_1 est la valeur `True` alors on *stoppe immédiatement le calcul* et l'expression entière vaut `True`.
  — si b_1 est la valeur `False` alors on évalue `<expression2>` ce qui produit une certaine valeur b_2
    — la valeur finale de la disjonction est b_2.

Ce mode d'évaluation est encore une fois *paresseux* car si la première expression est vraie, alors on sait déjà que la disjonction entière est vraie. Autrement dit, `A ou B` ne peut être faux si `A` est vrai, c'est logique !

Par exemple :

```
>>> (3 <= 9) or (9 <= 27)
True
```

Pour évaluer cette expression :

— on évalue tout d'abord `3 <= 9` ce qui produit la valeur `True`
  — on *stoppe donc immédiatement le calcul* et l'expression entière vaut `True`.

```
>>> (3 <= 9) or (9 >= 27)
True
```

Pour cette expression :

— on évalue tout d'abord `3 <= 9` ce qui produit la valeur `True`
  — on *stoppe donc immédiatement le calcul* et l'expression entière vaut `True`.

Remarque : on constate dans les exemples ci-dessus que la disjonction est vraie quelle que soit la valeur de la seconde expression.

```
>>> (3 >= 9) or (9 <= 27)
True
```

— on évalue tout d'abord `3 >= 9` ce qui produit la valeur `False`
  — on évalue alors `9 <= 27` ce qui produit la certaine valeur `True`
  — la valeur finale de la disjonction est donc `True`.

```
>>> (3 >= 9) or (9 >= 27)
False
```

— on évalue tout d'abord `3 >= 9` ce qui produit la valeur `False`
  — on évalue alors `9 >= 27` ce qui produit la certaine valeur `False`
  — la valeur finale de la disjonction est donc `False`.

Pour illustrer l'utilisation de la disjonction, considérons la définition d'un prédicat (donc une fonction retournant un booléen) indiquant si au moins deux nombres parmi trois de ses paramètres sont égaux.

```
def deux_egaux(x : float, y : float, z : float) -> bool:
    """Retourne la valeur True si au moins 2 de ces 3 nombres sont égaux,
       et False sinon
    """
    return (x == y) or (x == z) or (y == z)
```

```
# Jeux de test
assert deux_egaux(1, 2, 3) == False
assert deux_egaux(1, 2, 1) == True
assert deux_egaux(1, 1, 3) == True
assert deux_egaux(2, 2, 2) == True
```

2.3.3 Alternatives multiples

Les alternatives simples proposent un choix à deux possibilités. En pratique, il est parfois nécessaire d'effectuer des choix à n possibilités où n est strictement supérieur à 2.

Pour illustrer ce besoin considérons l'énoncé suivant :

Définir la fonction `nb_solutions` qui, étant donné trois nombres `a`, `b` et `c`, renvoie le nombre de solutions du trinôme du second degré $ax^2 + bx + c = 0$.

Rappel : le nombre de solutions dépend de la valeur du discriminant de ce trinôme

— si le discriminant est strictement positif il y a 2 solutions
— si le discriminant est nul il y a 1 solution
— si le discriminant est négatif il n'y a aucune solution

Pour répondre au problème énoncé, il est nécessaire d'effectuer un choix à trois possibilités selon que le discriminant est strictement positif, nul ou strictement négatif.

Pour cela, on utilise une *alternative multiple* dont la syntaxe est la suivante :

```
if <condition1>:
    <conséquent1>
elif <condition2>:
    <conséquent2>
elif ...
    ...
```

```
else:
    <alternant>
```

Le **principe d'interprétation des alternatives multiples** est le suivant :

- on évalue tout d'abord la `<condition1>`
  - si la valeur de la condition est `True` alors on interprète *uniquement* le `<conséquent1>`
  - sinon, on évalue la `<condition2>`
    - si la valeur de la condition est `True` alors on interprète *uniquement* le `<conséquent2>`
    - sinon, ...
  - ...
  - sinon, la valeur de toutes les conditions sont `False` et on interprète *uniquement* l'`<alternant>`.

Voici donc une solution pour répondre à l'énoncé :

```
def nb_solutions(a : float, b : float,c : float) -> float:
    """Retourne le nombre de solutions de l'équation aX² + bX + c = 0
    """
    discriminant : float = b*b - 4*a*c

    if discriminant == 0:
        return 1
    elif discriminant > 0:
        return 2
    else:
        return 0
```

```
# Jeu de tests
assert nb_solutions(1, 2, 3) == 0
assert nb_solutions(1, 3, 2)== 2
assert nb_solutions(1, 2, 1) == 1
```

Remarque : ici la variable `discriminant` est nécessaire pour ne pas répéter inutilement de calcul.

Chapitre 3

Répétitions et boucles

Nous avons vu dans les chapitres précédents qu'un langage qui serait constitué simplement d'expressions simples et composées, même muni d'instructions d'affectation des variables ne permet pas de faire beaucoup plus que ce que l'on fait avec une calculatrice. Pour accroître la puissance d'un langage de programmation, il faut également des *structures de contrôle* dont nous avons déjà vu un exemple avec l'*alternative*. Mais il nous manque le plus important : la possibilité de faire des *calculs répétitifs* avec le concept de **boucle**.

3.1 Motivation : répéter des calculs

Supposons que l'on veuille calculer le produit des 5 premiers entiers naturels, c'est-à-dire la factorielle de 5, ou 5!. Une solution possible, à partir des éléments de langage que nous connaissons serait de définir la fonction suivante :

```
def fact_5() -> int:
    """Retourne la factorielle de 5.
    """
    return 1 * 2 * 3 * 4 * 5
```

```
# Test
assert fact_5() == 120
```

Cette solution fonctionne, bien sûr, mais on se rend assez vite compte ici qu'une telle approche n'est pas du tout raisonnable si l'on désire calculer, non pas la factorielle de 5 mais disons de 100 par exemple ... Notre programme comporterait 99 signes de multiplications et ne fonctionnerait que pour exactement 100 entiers. Si l'on veut rendre le calcul générique, c'est-à-dire calculer la factorielle de n'importe quel entier naturel n, alors le nombre de multiplications à effectuer dépend maintenant de ce paramètre n. On ne peut plus écrire le calcul à effectuer de façon directe en indiquant explicitement toutes les multiplications à effectuer.

Il nous faut un moyen de définir des calculs dont le nombre d'étapes n'est pas fixé à l'avance. Nous dirons que ce sont des *calculs répétitifs*.

Pour le calcul de la factorielle, nous aimerions un moyen de décrire le calcul générique :

$$n! = 1 * 2 * \ldots * n$$

sans fixer à l'avancer la borne n.

Pour résoudre ce problème, regardons comment nous pourrions calculer ce produit pour $n = 5$ mais *au fur et à mesure* que l'on énumère les entiers de 1 à 5.

Nommons `r` le produit ainsi construit étape par étape. Initialement `r` possède la valeur 1 qui est l'élement neutre pour la multiplication. On peut ensuite :

1. multiplier `r` par la valeur du prochain entier naturel 1 : `r` vaut désormais 1
2. multiplier `r` par la valeur du prochain entier naturel 2 : `r` vaut désormais 2
3. multiplier `r` par la valeur du prochain entier naturel 3 : `r` vaut désormais 6
4. multiplier `r` par la valeur du prochain entier naturel 4 : `r` vaut désormais 24
5. multiplier `r` par la valeur du prochain entier naturel 5 : `r` vaut désormais 120

On a alors atteint notre borne $n = 5$ et on peut constater que `r` vaut bien 120, la factorielle de 5.

Dans la suite des 5 instructions ci-dessus, on peut remarquer que l'on effectue toujours le même traitement : la seule chose qui change est la valeur de l'entier que l'on a ajouté. Or, on sait en programmation mémoriser une valeur dans une variable et changer la valeur d'une variable. On voudrait donc pouvoir faire, à partir d'une variable `i`, initialisée à la valeur 1 :

— multiplier la valeur `r` (le produit partiel) par la valeur de la variable `i`, en conservant le résultat dans `r`
— passer à la valeur suivante de `i` (par une incrémentation `i = i + 1` ou `i += 1`)
— recommencer les 2 instructions précédentes (`n` fois)

Il nous reste à expliquer que ce `i` ne doit pas seulement énumérer les entiers de 1 à 5 mais plus généralement les entiers de 1 à n pour un n quelconque.

Les langages de programmation fournissent pour cela les **boucles** qui sont les principales structures de contrôle pour décrire des calculs répétitifs.

Nous allons étudier dans ce chapitre l'une des plus simples et des plus générales : la boucle `while` qui permet de répéter un bloc d'instructions *tant que* la valeur d'une certaine expression booléenne (appelée la *condition de boucle*) est vraie (`True` en Python).

Pour reproduire le calcul du produit des entiers de 1 à n, nous supposons avoir nos deux variables `r` (pour le résultat) et `i` pour l'entier courant. Au départ, le produit cumulé vaut `1`, et le premier entier courant est `1`.

Le principe de répétition est alors le suivant :

— répéter tant que `i <= n`:

— multiplier `r` par la valeur de l'entier naturel stocké dans `i`, et écrire ce résultat dans `r`,
— incrémenter `i` (affecter à `i` la valeur `i + 1`)

Cette forme de boucle est fournie en Python (et dans d'autres langages) par le mot clé `while`. Utilisons-la pour effectuer notre calcul de la factorielle.

```
def factorielle(n : int) -> int:
    """Précondition: n >= 0
    Retourne la factorielle de n.
    """
    # Entier courant
    i : int = 1 # élément neutre de la multiplication

    # Produit cummulé
    r : int = 1 # r est la factorielle de i

    while i <= n:
        r = r * i   # ou  r *= i
        i = i + 1   # ou  i += 1

    return r
```

```
# Jeu de tests
assert factorielle(0) == 1
assert factorielle(1) == 1
assert factorielle(5) == 120
assert factorielle(6) == 720
```

Nos tests semblent indiquer que notre fonction effectue le bon calcul. Mais pour nous en convaincre, nous devons étudier la construction `while ...` de façon plus détaillée.

3.2 La boucle while

Dans cette section, nous détaillons les mécanismes d'interprétation des boucles `while ...`.

3.2.1 Principe d'interprétation

La **syntaxe** de l'instruction de contrôle `while` pour les boucles dite «*tant que*» est la suivante :

```
while <condition logique>:
    <instruction 1>
    ...
    <instruction n>
```

```
<instruction_suite>
```

La `<condition logique>` est une expression booléenne (de type `bool`) appelée la **condition de boucle**.

La suite d'instructions :

```
        <instruction 1>
        ...
        <instruction n>
```

se nomme le **corps de la boucle**.

Nous avons ajouté l'instruction `<instruction_suite>` qui sera la première instruction à interpréter *après* l'exécution complète de la boucle. Cette instruction se trouve au même niveau d'indentation que la construction `while ...`.

Le principe d'interprétation correspondant de la boucle `white` est le suivant :

Dans :

```
while <condition logique>:
    <instruction 1>
    ...
    <instruction n>

<instruction suite>
...
```

1. On évalue tout d'abord la `<condition logique>`. De deux choses l'une :

— 2.a) Soit cette dernière est *différente* de `False` et alors :
  — on interprète la séquence des instructions dans le corps de la boucle :

```
<instruction 1>
...
<instruction n>
```

  — on retourne à l'étape 1.
— 2.b) Soit la `<condition logique>` s'évalue en `False` et on sort de la boucle pour interpréter l'instruction `<instruction suite>` (et celles qui la suivent).

3.2.2 Simulation de boucle

Pour étudier l'interprétation des boucles nous introduisons la notion de **simulation de boucle** que nous allons illustrer sur la fonction `factorielle`.

La simulation de boucle conduit à la construction d'une *table de simulation* selon les principes qui suivent.

La première étape est de fixer une valeur pour chaque paramètre de la fonction qui joue un rôle dans la boucle que l'on veut simuler.

Pour `factorielle` il y a un unique paramètre qui est `n` qui joue bien un rôle dans la boucle (il apparaît dans la condition). Nous allons fixer `n` à la valeur 5 pour notre simulation.

Pour la seconde étape, on construit une table avec une première colonne *tour de boucle* (ou *tour*), puis on ajoute une colonne par *variable modifiée dans le corps de la boucle*. L'ordre des colonnes correspond à l'ordre des affectations dans le corps.

Par exemple dans la fonction `factorielle` la variable `r` est modifiée avant la variable `i`, on obtient donc l'en-tête suivant pour notre table de simulation :

tour de boucle	variable `r`	variable `i`

On réalise une simulation en remplissant les lignes de notre table : une ligne pour chaque tour de boucle.

La première ligne correspond à l'étape *avant* l'entrée de la boucle. On écrit donc `entrée` dans la colonne `tour de boucle` et on donne pour chaque variable sa valeur avant d'effectuer le premier tour de boucle.

Pour `factorielle` la valeur initiale de `r` est `1` et la valeur de `i` est `1` avant le premier tour de boucle, on obtient donc :

tour de boucle	variable `r`	variable `i`
entrée	1	1

Pour la seconde ligne et les suivantes, on indique dans la première colonne le nombre de tours de boucle effectués ainsi que la valeur *après* ce tour de boucle pour les variables modifiées. Il est important de respecter l'ordre dans lequel ces modifications sont effectuées, qui devrait correspondre à l'ordre de gauche à droite dans la table.

Pour `factorielle` la valeur de `r` après le premier tour est `1` et la valeur de `i` est `2`.

On obtient donc :

tour de boucle	variable `r`	variable `i`
entrée	1	1
1er tour	1	2

La condition est boucle `i <= n` est toujours vraie après ce premier tour (on se rappelle que l'on a fixé la valeur de `n` à `5` pour cette simulation).

A la fin du deuxième tour cela donne :

tour de boucle	variable `r`	variable `i`
entrée	1	1
1er tour	1	2
2e tour	2	3

La condition est boucle `i <= n` est toujours vraie après le tour 2.

On continue ainsi pour le tour 3, le tour 4 et le tour 5 :

tour de boucle	variable `r`	variable `i`
entrée	1	1
1er tour	1	2
2e tour	2	3
3e tour	6	4
4e tour	24	5
5e tour (sortie)	120	6

A l'issue de ce cinquième tour, la condition `i <= n` n'est plus vérifiée car `i` vaut 6, la condition de boucle vaut donc `False` et cela provoque donc la *sortie de boucle*. Pour préciser que le cinquième tour est le dernier, on rajoute la mention `(sortie)` dans la colonne `tour de boucle`. Cette ligne précise alors les valeurs finales de chaque variable, et la simulation est finie.

À la fin de la simulation, la valeur de `r` est donc 120 qui correspond bien à la factorielle de 5. Dans la fonction `factorielle`, l'instruction qui suit la boucle est `return r` qui retourne donc le résultat attendu pour ce calcul.

3.2.3 Tracer l'interprétation des boucles avec print

Construire une simulation de boucle permet de comprendre et de vérifier sur quelques exemples que les calculs effectués sont corrects (ou en tout cas *semblent* corrects, nous reviendrons sur cette nuance importante lors du prochain chapitre). Cependant, sur machine on aimerait également pouvoir observer le comportement des programmes sans systématiquement faire des simulations à la main. Pour cela, nous pouvons utiliser l'instruction `print` pour *tracer* l'interprétation des programmes et en particulier les boucles.

Voici comment reproduire notre simulation sur `factorielle` :

```
def factorielle(n : int) -> int:
    """ ... cf. ci-dessus ...
    """
    i : int = 1
    r : int = 1
```

```
    print("====================")
    print("r en entrée vaut ", r)
    print("i en entrée vaut ", i)

    while i <= n:
        r = r * i
        i = i + 1
        print("------------------")
        print("r après le tour vaut ", r)
        print("i après le tour vaut ", i)

    print("------------------")
    print("sortie")
    print("====================")

    return r
```

```
>>> factorielle(5)
====================
r en entrée vaut  1
i en entrée vaut  1
------------------
r après le tour vaut  1
i après le tour vaut  2
------------------
r après le tour vaut  2
i après le tour vaut  3
------------------
r après le tour vaut  6
i après le tour vaut  4
------------------
r après le tour vaut  24
i après le tour vaut  5
------------------
r après le tour vaut  120
i après le tour vaut  6
------------------
sortie
====================
120
```

Bien sûr, ces instructions `print` doivent être supprimées (ou commentées) lorsque l'on pense que le calcul effectué est correct. Les programmeurs Python aguéris utilisent en complément des outils de mise au point de programmes, en particulier le débogueur (en anglais *debugger*) qui permet par exemple d'exécuter un programme pas-à-pas. Mais dans le contexte de ce livre, les traces avec `print` seront sans doute suffisantes pour la mise au point.

3.3 Problèmes numériques

La boucle `while` est notamment utilisée pour effectuer des calculs répétitifs sur les nombres entiers ou flottants.

Nous allons nous intéresser dans cette section aux problèmes suivants :

- calcul des éléments d'une suite,
- calcul d'une somme (série numérique),
- un exemple de problème plus complexe : le calcul du PGCD de deux entiers naturels
- un problème nécessitant des boucles imbriquées : le nombre de couples d'entiers distincts dans un intervalle donné.

3.3.1 Calcul des éléments d'une suite

Une suite arithmétique $(u_n)_{n\geq 0}$ est généralement définie en mathématiques de la façon suivante :

$$\begin{cases} u_0 &= k \\ u_n &= f(u_{n-1}) \text{ pour } n > 0 \end{cases}$$

où k est une constante dite *condition initiale* de la suite, et f est une fonction permettant de calculer l'élément de la suite au rang n à partir de la valeur de l'élément de rang $n-1$. On dit d'une telle définition qu'elle est *récursive*.

Considérons en guise d'exemple la suite récursive $(u_n)_{n\geq 0}$ ci-dessous :

$$\begin{cases} u_0 &= 7 \\ u_n &= 2u_{n-1} + 3 \text{ pour } n > 0 \end{cases}$$

Les premiers termes de cette suite sont: 7, 17, 37, 77, 157, 317, 637, ...

On peut définir, avec une boucle `while`, une fonction dont l'argument est n et qui calcule la valeur de u_n :

```
def suite_u(n: int) -> int:
    """Précondition : n >= 0
    Retourne la valeur au rang n de la suite U.
    """
    # valeur au rang 0
    u : int = 7

    # initialement rang 0
    i : int = 0

    while i < n:
        u = 2 * u + 3
```

```
        i = i + 1

    return u
```

Par exemple :

```
>>> suite_u(0)
7
```

```
>>> suite_u(1)
17
```

```
>>> suite_u(2)
37
```

```
>>> suite_u(6)
637
```

Voici la simulation correspondante de cette dernière application (donc pour `n` fixé à `6`) :

tour de boucle	variable `u`	variable `i`
entrée	**7**	**0**
1er tour	17	1
2e	37	2
3e	77	3
4e	157	4
5e	317	5
6e (sortie)	**637**	**6**

Exercice : proposer un jeu de tests pour la fonction `suite_u`.

3.3.2 Calcul d'une somme

De façon similaire au calcul des éléments d'une suite, on peut s'intéresser au calcul de sommes ou de produits d'éléments d'une suite.

Par exemple, considérons le n-ième terme S_n de la série S défini, pour $n \geq 0$, de la façon suivante :

$$S_n = \sum_{k \geq 0}^{n} (\frac{1}{2})^k$$

Nous pouvons traduire ce calcul en Python de la façon suivante :

```
def serie_s(n : int) -> float:
    """Précondition : n >= 0
    Retourne le n-ième terme de la série  :
```

```
        1 + 1/2 + 1/4 + ... + (1/2)^n
    """
    # la somme vaut 0 initialement
    s : float = 0.0

    # on commence au rang 0
    k : int = 0

    while k <= n:
        s = s + ((1/2) ** k)
        k = k + 1

    return s
```

```
>>> serie_s(1)
1.5
```

```
>>> serie_s(5)
1.96875
```

```
>>> serie_s(10)
1.9990234375
```

```
>>> serie_s(50)
1.9999999999999991
```

```
>>> serie_s(51)
1.9999999999999996
```

```
>>> serie_s(52)
1.9999999999999998
```

```
>>> serie_s(53)
2.0
```

Complément : validation des calculs flottants

Cette fonction est intéressante pour étudier la problématique de test concernant les nombres flottants (de type `float`). Avec les entiers et la plupart des types autres ceux impliquant des flottants, l'opérateur d'égalité `==` est l'outil le plus communément utilisé pour tester les fonctions. Malheureusement, l'égalité sur les nombres flottants n'est pas un outil très utile. La raison principale est que les flottants sont des approximations de nombres réels, et que les nombres réels sont eux même de nature non-triviale (on pourrait dire «complexe» mais il existe aussi les «nombres complexes» en mathématiques). En fait la plupart des nombres réels ne sont pas représentables sur ordinateur, et les flottants sont ce que les processeurs utilisent pour en représenter certains, permettant ce que l'on appelle des *calculs numériques* sur ordinateur.

Prenons l'exemple suivant :

```
>>> 15 * 0.1
1.5
```

Un test de validation classique en utilisant l'égalité serait le suivant :

```
assert 15 * 0.1 == 1.5
```

Avec un raisonnement mathématique classique, on pourrait penser que le test suivant soit aussi correct :

```
assert 1.5 * 0.1 == 0.15
```

Essayons avec l'interprète Python :

```
>>> 1.5 * 0.1
0.15000000000000002
```

Le résultat n'est pas vraiment celui attendu ! En fait on a ici une approximation raisonnable (puisque fausse à la 17ème décimale) mais notre raisonnement est faux, le résultat n'est pas *exactement* `0.15`. La raison, que nous ne pouvons détaillé ici, est liée à la représentation binaire des flottants. En particulier le nombre `0.1` ne possède pas de représentation flottante exacte. On retiendra de cette petite expérience que les propriétés mathématiques des nombres réelles ne sont souvent pas respectées par les flottants, et qu'ainsi l'égalité n'est pas un bon test de comparaison entre une valeur flottante estimée (par exemple `0.15` qui est le résultat réel attendu) et une valeur calculée par l'ordinateur (qui est une bonne approximation, mais une approximation quand même).

La question est donc de comprendre comment valider une fonction retournant des nombres flottants. La méthode la plus robuste est d'utiliser un *principe d'encadrement* de la forme :

```
assert <borne_inférieure_estimée>  <=  <expression_a_calculer> <= <borne_supérieure_estimée>
```

Si on retourne à notre exemple, un test de validation «acceptable» est le suivant :

```
assert 0.15 <= 1.5 * 0.1 <= 0.16
```

Bien sûr, on peut choisir des bornes inférieures et supérieures plus fines, en fonction des résultats produits par l'interprète Python. Cela dépasse un petit peu le cadre de notre livre, mais nous donnerons quelques illustrations de cette problématique dans des exercices.

Pour notre fonction `serie_s` nous pouvons pointer un autre problème intéressant. On peut montrer, mathématique, que la série S_n qu'elle implémente tend vers 2 lorsque n tend vers l'infini. Cela veut dire que la valeur 2 n'est jamais atteinte exactement. Pourtant avec `serie_s` on a par exemple :

```
>>> serie_s(53)
2.0
```

Dès le 53-ème terme nous avons déjà atteint la valeur de convergence, et pourtant nous sommes assez loin de «l'infini» ! Mathématiquement, l'égalité $S_{53} = 2.0$ est parfaitement fausse, alors il semble douteux de l'utiliser comme assertion dans un jeu de

test. Une autre propriété mathématique intéressante peut nous être un peu plus utile : $\forall n > 0,\ S_{n-1} < S_n \leq 2.0$. Avec les flottants ce n'est toujours pas utilisable puisque par exemple :

```
>>> serie_s(53) == serie_s(54)
True
```

Mais en utilisant une inégalité large on trouve un critère de validation correcte. Et on en déduit le jeu de test suivant :

```
# Jeu de test (pour la fonction serie_s)
assert serie_s(0) == 1.0   # ici l'égalité est utilisable
assert serie_s(0) <= serie_s(1) <= 2.0
assert serie_s(9) <= serie_s(10) <= 2.0
assert serie_s(99) <= serie_s(100) <= 2.0
```

On remarque que pour S_0 la valeur est bien exactement 1 donc, une fois n'est pas coutûme, on peut utiliser l'égalité.

3.3.3 Exemple de problème plus complexe : le calcul du PGCD

Certains problèmes nécessitent des calculs répétitifs un peu plus complexes que les suites ou les séries simples. En guise d'illustration, considérons le problème du calcul du PGCD de deux entiers naturels.

La spécification proposée est la suivante :

```
def pgcd(a : int, b : int) -> int:
    """Précondition: (a >= b) and (b > 0)
    Retourne le plus grand commun diviseur de a et b.
    """
```

Par exemple :

```
>>> pgcd(9, 3)
3
```

```
>>> pgcd(15, 15)
15
```

```
>>> pgcd(56, 42)
14
```

```
>>> pgcd(4199, 1530)
17
```

Pour écrire la fonction `pgcd`, nous allons exploiter une variante de l'algorithme d'Euclide. Pour calculer le PGCD des deux entiers a et b tels que $a \geq b$:

— si $b \neq 0$ alors le PGCD de a et b est le PGCD de b et du reste dans la division euclidienne de a par b ($a\%b$)
— sinon le PGCD de a et b est a.

Remarque: on vérifie aisément que si $a \geq b$ et $b \neq 0$, alors on a $b \geq a\%b$

Par exemple, le PGCD de 56 et 42 est égal :

— au PGCD de 42 et $56\%42 = 14$, qui est égal :
— au PGCD de 14 et $42\%14 = 0$, donc le PGCD cherché est 14

Comme autre exemple, on peut calculer le PGCD de 4199 et 1530 qui est égal :

— au PGCD de 1530 et $4199\%1530 = 1139$, qui est égal :
— au PGCD de 1139 et $1530\%1139 = 391$, qui est égal :
— au PGCD de 391 et $1139\%391 = 357$, qui est égal :
— au PGCD de 357 et $391\%357 = 34$, qui est égal :
— au PGCD de 34 et $357\%34 = 17$, qui est égal :
— au PGCD de 17 et $34\%17 = 0$, donc le PGCD cherché est 17 (ouf !).

Cet algorithme peut être traduit de la façon suivante en Python :

```python
def pgcd(a : int, b : int) -> int:
    """Précondition: (a >= b) and (b > 0)
    Retourne le plus grand commun diviseur de a et b.
    """
    # le quotient
    q : int = a

    # le diviseur
    r : int = b

    # mémoire auxiliaire
    temp : int = 0

    while r != 0:
        temp = q % r
        q = r
        r = temp

    return q
```

```python
# Jeu de tests
assert pgcd(9, 3) == 3
assert pgcd(15, 15) == 15
assert pgcd(56, 42) == 14
assert pgcd(4199, 1530) == 17
```

On remarque l'utilisation d'une variable `temp` dite *temporaire*, qui permet de stocker un calcul intermédiaire. Ici, on doit modifier les contenus des deux variables `q` et `r` donc le calcul `q % r`, qui porte sur les valeurs des `q` et `r` avant qu'elles soient modifiées, doit être effectués en premier, et stocké dans une mémoire temporaire.

Exercice : Donner la simulation correspondant à `pgcd(56, 42)`. Même question pour `pgcd(4199, 1530)`. Les trois variables à observer dans la simulation sur `temp`, `q` et `r`

(dans cet ordre).

3.3.4 Complément : les boucles imbriquées

Considérons la question suivante :

> Combien existe-t-il de couples (i, j) distincts d'entiers naturels inférieurs ou égaux à un entier n fixé ?

Pour répondre à cette question, on ne peut pas se contenter d'une unique boucle `while` car :

— nous devons considérer tous les entiers i de 0 à n
— pour chaque i nous devons considérer également tous les entiers j de 0 à n

Il est donc nécessaire de combiner deux boucles, l'une à l'intérieur de l'autre. On parle alors de *boucles imbriquées*.

Ceci se traduit par la fonction suivante :

```
def nb_couples_distincts(n : int) -> int:
    """Précondition : n >= 0
    Retourne le nombre de couples distincts (i,j) d'entiers
    naturels inférieurs ou égaux à n.
    """

    # premier élément du couple
    i : int = 0

    # second élément du couple
    j : int = 0

    # pour compter les couples
    compte : int = 0

    while i <= n:
        j = 0  # il faut réinitialiser j pour chaque "nouveau" i
        while j <= n:
            if i != j:
                compte = compte + 1  # on a trouvé un nouveau couple

            j = j + 1
        # sortie de la boucle sur j

        i = i + 1  # après avoir "regardé" tous les j,
                   # on passe au i suivant
    # sortie de la boucle sur i
    return compte
```

```
>>> nb_couples_distincts(3)
12
```

```
>>> nb_couples_distincts(5)
30
```

Il n'y a pas de difficulté particulière dans cette fonction. Cependant, les boucles imbriquées sont un peu complexes à simuler. Le principe est de faire une simulation pour la boucle intérieur, pour chaque tour de la boucle extérieur. Cela dépasse un peu le cadre d'une première approche des boucles, donc, dans l'immédiat, nous n'aborderons pas de façon plus approfondie cette problématique.

Exercice : proposer un jeu de tests pour la fonction `nb_couples_distincts`.

3.4 Complément : les fonctionnelles

On s'intéresse dans cette section à la généralisation des calculs numériques du type de ceux décrits précédemment. Nous reposons sur un principe fondamental : la possibilité d'utiliser des fonctions en paramètres d'autres fonctions.

Vocabulaire : Une fonction qui prend une fonction en argument est appelée une **fonctionnelle**.

3.4.1 Exemple 1 : calcul des éléments d'une suite arithmétique

Plutôt que de résoudre un problème particulier, comme le calcul des éléments d'une suite arithmétique particulière, on peut résoudre le problème plus général du calcul des éléments d'une suite.

Reprenons la description du problème présenté précédemment :

En mathématiques, une suite arithmétique $(u_n)_{n\geq 0}$ est définie de la façon suivante :

$$\left\{ \begin{array}{lcl} u_0 & = & k \\ u_n & = & f(u_{n-1}) \text{ pour } n > 0 \end{array} \right.$$

où k est une constante dite *condition initiale* de la suite, et f est une fonction permettant de calculer l'élément de la suite au rang n à partir de la valeur de l'élément de rang $n-1$.

Nous avons considéré, en guise d'exemple, la suite $(u_n)_{n\geq 0}$ ci-dessous :

$$\left\{ \begin{array}{lcl} u_0 & = & 7 \\ u_n & = & 2u_{n-1} + 3 \text{ pour } n > 0 \end{array} \right.$$

Et nous avons déduit la fonction `suite_u` de cette dernière définition.

Mais le problème de départ, posé en toute généralité, peut être résolu de façon générale.

Pour cela, il faut considérer une fonction `suite` paramétrée par :

- le rang `n` de l'élément de la suite que nous désirons calculer
  - la condition initiale `k` pour donner une valeur à u_0.
  - la fonction `f` qui permet de passer de u_{n-1} à u_n donc de l'élément précédent à l'élément courant.

Ceci conduit à la spécification suivante :

```
def suite(n : int, k : float, f : Callable[[float], float]) -> float:
    """Précondition: n >= 0
    Retourne l'élément de rang n de la suite
    U définie par U_0=k et pour tout n>0, U_n = f(U_(n-1)).
    """
```

La principale difficulté ici est la signature. Les deux premiers paramètres sont «classiques» mais ce n'est pas le cas du dernier : `Callable[[float], float]`. Cette signature indique que le troisième paramètre `f` doit être une référence à une fonction unaire paramétrée par un nombre de type `float` et retournant un nombre du même type. L'écriture mathématique de cette signature est donc `float -> float` mais malheureusement elle n'est pas disponible en Python. On n'utilisera pas les paramètre de type `Callable` en dehors de cette section du livre.

La définition proprement dite de la fonction `suite` est en fait moins compliquée que sa signature puisqu'il s'agit d'une simple «paramétrisation» de la fonction `suite_u` définie précédemment. Cela conduit à la définition suivante :

```
from typing import Callable

def suite(n : int, k : float, f : Callable[[float], float]) -> float:
    """ ... cf. ci-dessus ...
    """
    # élément courant de rang i (on démarre au rang 0)
    i : int = 0

    # valeur de l'élément courant (au rang i=0 la valeur est k)
    u : float = k

    while i < n:
        u = f(u) # calcul de la valeur de l'élément suivant, grâce à f
        i = i + 1  # on passe au rang suivant

    return u
```

Pour retrouver la suite particulière :

$$\begin{cases} u_0 &= 7 \\ u_n &= 2u_{n-1} + 3 \text{ pour } n > 0 \end{cases}$$

il faut considérer que `k` vaut 7 et nous avons également besoin d'une fonction permettant de calculer la valeur de u_n en fonction de la valeur de u_{n-1}.

Voici une définition de cette fonction :

```
def suivant_u(x : float) -> float:
    """Retourne l'élément suivant pour la suite U.
    """
    return 2 * x + 3
```

```
# Jeu de tests
assert suivant_u(2) == 7
assert suivant_u(7) == 17
```

Nous avons maintenant tous les ingrédients pour calculer des termes de la suite :

```
>>> suite(0, 7, suivant_u)
7
```

```
>>> suite(1, 7, suivant_u)
17
```

```
>>> suite(5, 7, suivant_u)
317
```

```
>>> suite(5, 7, suivant_u) == suite_u(5)  # cf. définition suite_u
True
```

```
>>> suite(2000, 7, suivant_u) == suite_u(2000)
True
```

Etant donné que notre fonction `suite` est plus générale que `suite_u` nous pouvons par exemple modifier les conditions initiales.

```
>>> suite(5, 3, suivant_u)
189
```

Et nous pouvons également bien sûr modifier la fonction de calcul de l'élément suivant pour calculer les élément d'une suite complètement différente.

Exercice : selon les mêmes principes, proposer une généralisation du calcul de la somme des éléments d'une suite arithmétique.

3.4.2 Exemple 2 : annulation d'une fonction sur un intervalle

Certains problèmes ont une définition intrinsèquement générale, comme par exemple la résolution des équations du type $f(x) = 0$ très courantes en analyse.

Nous souhaitons définir une fonction `annulation` permettant de tester si une fonction `f` s'annule sur un intervalle donné.

Pour simplifier un peu le problème, considérons que l'on cherche à savoir si une fonction f donnée s'annule sur un intervalle $[a; b]$ entier (donc dans $f(x)$ on considère x entier)

et que $f(x)$ est un flottant. Ceci nous conduit à la spécification suivante :

```
def annulation(a : int, b : int, f : Callable[[int],float]) -> bool:
    """Précondition: a <= b
    Retourne True si la fonction f  s'annule sur  l'intervalle [a;b].
    """
```

Encore une fois, c'est le troisième paramètre de type `Callable[[int],float]` qui fait de `annulation` une fonctionnelle. Ici, il s'agit naturellement de prendre la fonction f que l'on veut tester, donc une fonction qui prend un `int` en entrée, et retourne un `float` en sortie (signature que l'on souhaiterait écrire `int -> float`).

Comme exemple, nous allons définir une variante de la fonction racine carrée : la fonction `racine` qui calcule la racine carrée d'un entier naturel :

```
import math

def racine(n : int) -> float:
    """Précondition : n >=0
    Retourne la racine carrée de l'entier n.
    """
    return math.sqrt(n)

# Jeu de tests
assert racine(1) == 1.0
assert racine(25) == 5.0
```

Remarquons que pour appliquer la fonction `annulation`, l'intervalle passé en argument ne doit contenir que des entiers positifs ou nuls pour la fonction racine.

Par exemple :

```
>>> annulation(1, 5, racine)
False
```

```
>>> annulation(0, 5, racine)
True
```

Sans surprise, la fonction `racine` s'annule sur l'intervalle [0;5] (elle s'annule pour la valeur 0).

Considérons un deuxième exemple, le calcul de l'inverse d'un nombre entier :

```
def inverse(n : int) -> float:
    """Précondition : n != 0
    Retourne le rationnel inverse de l'entier n.
    """
    return 1 / n

# Jeu de tests
assert inverse(2) == 0.5
assert inverse(4) == 0.25
```

Ici, la fonction inverse n'est pas définie en 0, il faut donc faire attention à cette contrainte lors de l'utilisation de `annulation`.

Voici quelques exemples :

```
>>> annulation(1, 5, inverse)
False
```

```
>>> annulation(1, 1000, inverse)
False
```

```
>>> annulation(-500, -1, inverse)
False
```

La fonction inverse semble ne s'annuler nulle part ... cela semble cohérent.

Exercice : proposer d'autres exemples de tests d'annulation pour des fonctions diverses. Existe-t-il une fonction pouvant être annulée à plusieurs valeurs de son domaine ?

Une définition possible pour la fonction `annulation` est la suivante :

```
def annulation(a : int, b : int, f : Callable[[int],float]) -> bool:
    """ ... cf. ci-dessus ...
    """
    # élément courant, au début de l'intervalle
    x : int = a

    while (x <= b):
        if f(x) == 0.0:
            return True  # la fonction s'annule !
        else:  # sinon on continue avec l'élément suivant
            x = x + 1

    return False  # on sait ici que la fonction ne s'annule pas
```

```
# Jeu de tests
assert annulation(1, 5, racine) == False
assert annulation(0, 5, racine) == True
assert annulation(1, 5, inverse) == False
assert annulation(-1, -5, inverse) == False
```

Exercice : proposer une généralisation de la fonction `annulation` permettant de prendre un intervalle réel en paramètre (donc `a` et `b` sont des réels). On pensera à ajouter un paramètre flottant `delta` dont la valeur doit être inférieure à 1. Par exemple, pour un intervalle entier on choisira `delta` égal à 1. On considère donc le problème de l'annulation d'une fonction `f` entre `a` et `b` par pas d'incrémentation de `delta`.

Chapitre 4

Plus sur les boucles

Dans ce chapitre, nous approfondissons nos connaissances sur les répétitions et les boucles en étudiant trois questions fondamentales concernant les fonctions implémentées avec des boucles :

- la fonction répond-elle de façon **correcte** au problème posé ?
- est-ce-que la fonction retourne bien une valeur, donc est-ce que le programme **termine** ?
- l'algorithme de calcul est-t-il **efficace**

Ces questions de *correction*, de *terminaison* et d'*efficacité* sont fondamentales en programmation en général. Pourquoi étudie-t-on ces problématiques dans le cadre d'un (second) chapitre sur les boucles ? La raison est simple : les boucles représentent la principale source de problème potentiel concernant ces trois questions.

En complément, nous abordons une approche alternative pour traiter les calculs répétitifs : la **récursion**. Il s'agit d'une approche qui facilite grandement les questions de correction et de terminaison.

4.1 Notion de correction

Lorsque l'on écrit une définition de fonction, *la* question fondamentale que l'on se pose est la suivante :

> La fonction répond-elle correctement au problème posé ?

Pour illustrer cette problématique centrale, nous allons considérer le problème de l'élévation d'un nombre à une puissance entière positive.

Voici une première proposition :

```python
def puissance(x : float, n : int) -> float:
    """Précondition : n >= 0
    Retourne la valeur de x élevé à la puissance n.
    """
    # valeur de x^0
    res : float = 1

    # compteur
    i : int = 1

    while i != n + 1:
        res = res * x
        i = i + 1

    return res
```

```python
# Jeu de tests
assert puissance(2, 5) == 32
assert puissance(2, 10) == 1024
assert puissance(2.5, 1) == 2.5
```

Effectuons une simulation pour `x=2` et `n=5`.

tour de boucle	variable `res`	variable `i`
entrée	**1**	**1**
1	2	2
2	4	3
3	8	4
4	16	5
5 (sortie)	**32**	**6**

On obtient bien 32, soit 2^5, par simulation. La fonction semble donc à première vue correcte mais on aimerait un argument plus fort et plus général qu'un seul exemple pour des valeurs fixées. Une des approches possibles (et probablement la plus connue et la plus utilisée) est de faire intervenir la notion suivante.

Un **invariant de boucle** est une expression logique (donc vraie ou fausse) :

- exprimant une relation «intéressante» impliquant les variables modifiées dans le corps de la boucle,
- qui doit vraie en *entrée* de boucle (avant le premier tour de boucle),
- et qui doit rester vraie *après chaque tour* de boucle.

Trouver un invariant de boucle «intéressant» n'est pas toujours chose aisée. En général,

cela nécessite une connaissance précise de l'algorithme mis en œuvre. Il n'existe pas de méthode «exacte» pour trouver un invariant, , mais l'approche proposée ci-dessous est souvent fructueuse.

Pour trouver un invariant de boucle :

1. Il est nécessaire de bien comprendre le problème posé

Pour la fonction `puissance` on sait que l'on calcule une puissance, un problème dont nous avons à priori une bonne compréhension mathématique.

2. il faut effectuer quelques simulations avec des valeurs de paramètres bien choisies

Pour la puissance, nous avons déjà effectué une simulation et nous allons l'exploiter (en pratique, il en faudrait probablement quelques autres). L'objectif est de trouver une relation entre les différentes variables qui sont modifiées à chaque tour de boucle. Cette relation doit être vérifiée pour chaque ligne de la table de simulation (cf. exemple ci-dessous).

3. et finalement l'expérience compte

Pour trouver des invariants, il faut bien sûr de la pratique et finalement un peu d'imagination, nous sommes en présence d'un processus créatif.

Pour illustrer ce processus, revenons à notre fonction `puissance` et en particulier à la simulation pour `x=2` et `n=5`.

L'invariant doit relier de façon logique les variables indiquées dans la simulation, et rester vrai à chaque étape et donc sur chaque ligne.

Pour notre calcul de puissance, nous proposons comme invariant de boucle la propriété suivante :

Candidat invariant de boucle: $res = x^{i-1}$

Remarque : l'invariant de boucle est une expression mathématique, le symbole égal = ci-dessus est bien l'égalité mathématique. De plus ce n'est pas encore un invariant mais simplement un candidat.

Regardons ce que cela donne sur notre simulation :

tour de boucle	variable `res`	variable `i`	**invariant** : $res = x^{i-1}$
entrée	**1**	**1**	$1 = 2^{1-1}$ (vrai)
1	2	2	$2 = 2^{2-1}$ (vrai)
2	4	3	$3 = 2^{3-1}$ (vrai)
3	8	4	$8 = 2^{4-1}$ (vrai)
4	16	5	$16 = 2^{5-1}$ (vrai)
5 (sortie)	**32**	**6**	$32 = 2^{6-1}$ (vrai)

Notre candidat invariant est bien vrai à l'entrée de la boucle et après chaque tour, ainsi qu'à la sortie de boucle il est donc vérifié par notre simulation.

On peut démontrer formellement que cet invariant est correct, c'est-à-dire qu'il correspond bien à la boucle de la fonction `puissance`. Mais cette preuve nous entraînerait un peu trop loin pour une introduction des concepts. Donc nous nous limiterons comme ici à vérifier les candidats invariants de boucle sur des simulations (on dit alors que l'on *teste* l'invariant de boucle).

Si l'on suppose que cet invariant est correct, alors il est facile de montrer que le calcul effectué par notre fonction `puissance` est bien x^n car en sortie de boucle, on sait que la condition `i != n + 1` est fausse, et l'on peut aussi facilement se convaincre que `i` est toujours plus petit que `n + 1`. En effet, par hypothèse `n >= 0` donc `n >= 1` qui est la valeur initiale de `i`. Donc en sortie de boucle on a $i = n + 1$ et notre invariant devient : $res = x^{i-1} = x^{n+1-1} = x^n$ (CQFD).

Supposons maintenant que nous modifions légèrement la définition de la fonction, de la façon suivante :

```
def puissance_modif(x : float, n : int) -> float:
    """Précondition : n >= 0
    Retourne la valeur de x élevé à la puissance n.
    """
    # valeur de x^0
    res : float = 1

    # compteur
    i : int = 1

    while i != n:
        res = res * x
        i = i + 1

    return res
```

Reprenons notre simulation avec invariant pour `x=2` et `n=5` :

tour de boucle	variable `res`	variable `i`	**invariant** : $res = x^{i-1}$
entrée	**1**	**1**	$1 = 2^{1-1}$ (vrai)
1	2	2	$2 = 2^{2-1}$ (vrai)
2	4	3	$3 = 2^{3-1}$ (vrai)
3	8	4	$8 = 2^{4-1}$ (vrai)
4 (sortie)	**16**	**5**	$16 = 2^{5-1}$ (vrai)

Ici, en sortie de boucle on à `i = 5` donc l'invariant en sortie devient $res = 2^{i-1} = 2^4 = 16$.

Dans le cas général on ne calcule donc pas x^n mais x^{n-1} et l'invariant nous montre bien que le calcul n'est pas le bon : la fonction n'est *pas correcte* par rapport à sa spécification.

Retenons qu'une légère modification d'un programme peut fausser le résultat des calculs effectués. Cependant dans le cas des boucles, les invariants de boucle nous aident à comprendre précisément *pourquoi* ces calculs ne sont pas corrects.

Il est important de remarquer que la correction d'une fonction est *toujours* relative à sa spécification. Essayons par exemple d'effectuer un appel qui sort du domaine de la spécification. Prenons une valeur flottante pour le paramètre `n` et non un entier naturel, comme par exemple $2^{\frac{1}{2}}$.

```
>>> puissance(2, 1/2)
# ...  le programme ne s'arrête pas ...
```

En fait, ici l'appel de fonction ne retourne jamais, on met en lumière un *problème de terminaison de boucle* dont nous allons discuter dans la section suivante.

Pourtant on sait qu'en mathématique $2^{1/2} = \sqrt{2}$, ce que l'on peut vérifier directement en Python :

```
>>> 2 ** (1/2)
1.4142135623730951
```

Par conséquent, il est clair que notre fonction `puissance` ne calcule correctement que les puissances entières naturelles, ce que sa spécification précise bien sûr.

On retiendra plus généralement :

> La correction d'une fonction est toujours relative à sa spécification.

4.2 Notion de terminaison

Il y a deux façons principales pour une fonction de ne pas répondre à un problème posé :

1. le calcul effectué n'est pas correct
2. le calcul ne termine pas

La notion de correction discutée précédemment n'a de sens que si le calcul se termine, nous l'avons bien vu sur notre tentative de calcul pour une puissance non-entière.

Pour comprendre ce problème de terminaison, effectuons quelques tours de simulation pour `puissance(2, 1/2)`.

tour de boucle	variable `res`	variable `i`
entrée	**1**	**1**
1	2	2
2	4	3
3	8	4
4	16	5
...	...	...

Dans cet exemple, `n=1/2` donc la condition de boucle `i != n + 1` devient `i != 1/2 +1`. Or avec `i=1` en entrée de boucle et en l'incrémentant à chaque tour, on voit bien que la condition ne peut jamais est fausse : `i` est toujours différent de 1.5 donc on ne sort jamais de la boucle.

Question : d'après-vous que se passe-t-il pour `puissance(2, -5)` ?

Pour offrir des garanties concernant la terminaison d'une boucle, on utilise une sorte de *mesure* que nous définissons ci-dessous.

Un **variant de boucle** est une expression définie sur un ensemble muni d'un *ordre bien fondé* (donc sans suite infinie décroissante) et qui *décroît strictement* à chaque tour de boucle.

Le variant de boucle est le plus souvent une expression à valeur dans $\mathbb{N}$ muni de l'*ordre naturel* $<$, ordre dont l'une des caractéristiques fondamentales est d'être bien fondé. En effet, comme le zéro est la borne inférieure des entiers naturels, donc une suite décroissante strictement dans $\mathbb{N}$ ne *peut pas* être infinie.

Dans ce livre, pour simplifier un peu la méthode générale, on imposera que le variant de boucle :

- est un entier naturel positif en entrée de boucle,
- qui décroît strictement à chaque tour,
- et qui vaut 0 en sortie de boucle, c'est-à-dire quand la condition de boucle devient fausse.

Pour la fonction puissance, le variant de boucle proposé est :

Candidat variant de boucle : $n + 1 - i$

Vérifions la valeur du variant sur la simulation pour `puissance(2, 5)`:

tour de boucle	variable `res`	variable `i`	**variant** : $n + 1 - i$
entrée	**1**	**1**	**5**
1	2	2	4
2	4	3	3
3	8	4	2
4	16	5	1
5 (sortie)	**32**	**6**	**0**

Ce variant de boucle est bien vérifié par la simulation : il décroît strictement après chaque tour de boucle, et en sortie il vaut 0.

On peut montrer de façon formelle que $n + 1 - i$ est bien un variant de boucle pour toute simulation avec `x` et `n` satisfaisant la spécification de la fonction `puissance`. Cependant, dans le cadre de ce livre nous allons en rester à de simples vérifications pour

des simulations données.

En faisant l'hypothèse que $n+1-i$ est bien un variant de boucle dans tous les cas, il est immédiat de conclure que la boucle termine : puisque le variant décroît strictement à chaque tour de boucle et que ce variant est à valeur dans $\mathbb{N}$ alors il finira forcément par valoir 0. Et s'il vaut 0 il est bien sûr impossible de continuer : la boucle termine forcément.

Bien sûr, la terminaison - et donc notre variant - est liée à la condition de boucle et l'on retiendra la règle suivante :

> Le variant de boucle vaut 0 lorsque la condition du `while` devient fausse.

On peut lire cette proporiété dans les deux sens.

- lorsque le variant vaut 0, on a $n+1-i=0$ donc $i=n+1$. De ce fait, la condition de boucle est fausse et donc on sort bien de la boucle.
- de façon complémentaire, lorsque la condition de boucle `i != n + 1` est fausse (et en ajoutant l'argument que $i \leq n+1$) on peut en déduire que i vaut $n+1$ en sortie de boucle. Donc le variant devient $n+1-i=n+1-n-1=0$ (CQFD).

4.3 Notion d'efficacité

Les informaticiens apprécient, certes, de résoudre des problèmes mais ils apprécient aussi (et surtout pour nombre d'entre eux) de les résoudre le plus efficacement possible.

Concernant l'efficacité, voici quelques questions typiques que les informaticiens se posent :

1. ne fait-on pas de calculs redondants ?
2. peut-on trouver un raccourci ?
3. existe-t-il un calcul algorithmiquement plus efficace ?

Etudions ces questions tour à tour sur quelques exemples.

4.3.1 Factoriser les calculs

Nous avons vu dans le chapitre 2 la fonction de calcul de l'aire d'un triangle. Notre première définition était la suivante :

```
def aire_triangle(a : float, b : float, c : float) -> float:
    """ ... cf. chapitre 2 ...  """
    return math.sqrt(((a + b + c) / 2)
                     * (((a + b + c) / 2) - a)
                     * (((a + b + c) / 2) - b)
                     * (((a + b + c) / 2) - c))
```

Comme nous en avons discuté au chapitre 2, cette définition est clairement loin d'être satisfaisante. Elle est illisible mais surtout le calcul du demi-périmètre est répété quatre fois ... donc trois fois de trop ! C'est un cas typique de redondance inutile dans les calculs.

Nous avons résolu simultanément ces deux problèmes en introduisant une variable locale dédiée au calcul du demi-périmètre. Intuitivement, nous avons introduit une case mémoire supplémentaire pour pouvoir effectuer un calcul intermédiaire et en stocker le résultat.

```
def aire_triangle(a : float, b : float, c : float) -> float:
    """ ... cf. chapitre 2 ... """
    # p : float
    p = (a + b + c) / 2   # demi-périmètre

    return math.sqrt(p * (p - a) * (p - b) * (p - c))
```

On retiendra que l'introduction de variables locales (et donc de cases mémoires supplémentaires) permet souvent d'éliminer des calculs redondants, tout en améliorant la lisibilité des programmes.

4.3.2 Sortie anticipée

Un second outil que nous avons déjà exploité est la sortie anticipée d'une boucle. Considérons la fonction suivante :

```
def plus_petit_diviseur(n : int) -> int:
    """Précondition : n >= 2
    Retourne le plus petit diviseur de n (autre que 1).
    """
    # Diviseur trouvé, 0 pour démarrer (donc pas de diviseur)
    d : int = 0

    # Compte le nombre de tours de boucle
    nb_tours : int = 0

    # Candidat diviseur
    m : int = 2   # on commence par 2  (car 1 divise tout, et 0 rien)

    while m <= n:
        nb_tours = nb_tours + 1
        if (d == 0) and (n % m == 0):
            d = m

        m = m + 1

    print("Tours de boucles =",nb_tours)
```

```
    return d
```

```
# Jeu de tests
assert plus_petit_diviseur(9) == 3
assert plus_petit_diviseur(121) == 11
assert plus_petit_diviseur(17) == 17
assert plus_petit_diviseur(1024) == 2
```

Les affichages générés par `print` sur ce jeu de tests sont les suivants :

```
Tours de boucles = 8
Tours de boucles = 120
Tours de boucles = 16
Tours de boucles = 1023
```

Pour trouver le plus petit diviseur de n, on effectue $n-1$ tours de boucle. Pourtant, si ce diviseur est beaucoup plus petit que n alors on effectue de nombreux tours de boucles inutiles.

Par exemple, dans notre dernier test, 2 est le plus petit diviseur de 1024 et lors du premier tour de boucle, on teste justement si 2 est un diviseur de 1024, ce qui est le cas. Donc un unique tour de boucle devrait suffire ici et pourtant nous en réalisons 1023 !

Une première façon de résoudre ce problème est de travailler sur la condition de boucle. Dans le corps de la fonction, la variable `d` sert à stocker la valeur du plus petit diviseur trouvé. Cette variable est initialisée à 0 avant la boucle. Dans la boucle, dès que l'on trouve un entier qui divise `n`, celui-ci est stocké dans la variable `d`. On aimerait alors ne pas effectuer de tour de boucle supplémentaire dès que la valeur stockée dans `d` est différente de 0, ce qui indique que l'on a justement trouvé un diviseur.

Voici la version modifiée de la définition :

```
def plus_petit_diviseur(n : int) -> int:
    """ ... cf. ci-dessus ..."""
    # Diviseur trouvé, 0 pour démarrer (donc pas de diviseur)
    d : int = 0

    # Compte le nombre de tours de boucle
    nb_tours : int = 0

    # Candidat diviseur
    m : int = 2   # on commence par 2  (car 1 divise tout,  et 0 rien)

    while (d == 0) and (m <= n):
        nb_tours = nb_tours + 1
        if (d == 0) and (n % m == 0):
            d = m

        m = m + 1
```

```
    print("Tours de boucles =",nb_tours)

    return d
```

```
# Jeu de tests
assert plus_petit_diviseur(9) == 3
assert plus_petit_diviseur(121) == 11
assert plus_petit_diviseur(17) == 17
assert plus_petit_diviseur(1024) == 2
```

Les affichages correspondants sont :

```
Tours de boucles = 2
Tours de boucles = 10
Tours de boucles = 16
Tours de boucles = 1
```

Ici, le nombre de tours de boucle a baissé pour la plupart des tests. En particulier, pour 1024 il ne faut qu'un tour de boucle comme on le souhaitait. En revanche, pour un nombre premier comme 17 on ne gagne rien puisque ce nombre n'est divisible que par lui-même (et 1 mais on ne le prend pas en compte).

Il existe une troisième façon un peu plus drastique de résoudre le problème : sortir directement de la fonction en plaçant un `return` au bon endroit. On ne sort plus de la boucle, mais directement de la fonction.

Ceci conduit à la définition suivante :

```
def plus_petit_diviseur(n : int) -> int:
    """ ... cf. ci-dessus ..."""
    # Compte le nombre de tours de boucle
    nb_tours : int = 0

    # Candidat diviseur
    m : int = 2   # on commence par 2  (car 1 divise tout,  et 0 rien)

    while m <= n:
        nb_tours = nb_tours + 1
        if (n % m == 0):
            print("Tours de boucles =",nb_tours)
            return m  # sortie directe de la fonction

        m = m + 1

    print("Tours de boucles =",nb_tours)

    return n
```

```
# Jeu de tests
assert plus_petit_diviseur(9) == 3
assert plus_petit_diviseur(121) == 11
assert plus_petit_diviseur(17) == 17
assert plus_petit_diviseur(1024) == 2
```

Les affichages sont les suivants :

```
Tours de boucles = 2
Tours de boucles = 10
Tours de boucles = 16
Tours de boucles = 1
```

Nous obtenons ici les mêmes résultats avec les mêmes performances.

La question qui s'impose ici est la suivante : quelle version choisir ?

Il est bien sûr hors de question de choisir la première solution car elle n'est pas efficace. Effectuer 1023 tours de boucle pour décider que 2 est diviseur de 1024 n'est pas raisonnable. La seconde solution à la mérite de ne pas nécessiter de structure de contrôle supplémentaire en dehors du `while` et du `if`. Il est également plus facile de l'analyser du point de vue de la correction et de la terminaison. Dans la troisième solution, on peut supprimer la variable locale `d` ce qui conduit à une définition assez simple à lire. Mais analyser des boucles avec sortie anticipée directement au niveau de la fonction n'est pas chose évidente. Nous avons constaté dans nos cours que de nombreux étudiants ont du mal à placer le `return` correctement.

En pratique, il faudra être capable de comprendre les deux types de solution : condition explicite de sortie de boucle ou sortie anticipée de fonction avec `return`, ainsi que la relation entre les deux.

4.3.3 Efficacité algorithmique

Posons-nous la question :

> Notre fonction puissance calcule-t-elle efficacement ? Peut-on être plus rapide ?

Avant d'améliorer un algorithme, il faut en mesurer les performances. Pour la fonction `puissance`, la mesure qui semble la plus pertinente de ce point de vue est le nombre de multiplications effectuées pour parvenir au résultat x^n.

Décorons un peu notre fonction pour effectuer cette mesure.

```
def puissance(x : float, n : int) -> float:
    """ ... cf. ci-dessus ..."""

    # Résultat (initialement valeur de x^0)
    res : float = 1
```

```
    # Compteur
    i : int = 1

    # Nombre de multiplication(s), initialement 0
    nb_mults : int = 0

    while i <= n:
        res = res * x
        nb_mults = nb_mults + 1
        i = i + 1

    print("Nombre de multiplications =", nb_mults)

    return res
```

```
>>> puissance(2, 5)
Nombre de multiplications = 5
32
```

```
>>> puissance(2, 10)
Nombre de multiplications = 10
1024
```

```
>>> puissance(2, 100)
Nombre de multiplications = 100
1267650600228229401496703205376
```

```
>>> puissance(2, 1000)
Nombre de multiplications = 1000
107150860718626732094842504906000181056140481170553360744375 0 ...
```

```
>>> puissance(2, 10000)
Nombre de multiplications = 10000
19950631168807583848837421626835850838234968318861924548520 08 ...
```

Comme le suggèrent nos affichages, et on peut le confirmer formellement, il faut n multiplications pour calculer x^n.

La question qui se pose est donc :

> Peut-on calculer x^n en effectuant moins de n multiplications ?

La réponse est *oui*, en exploitant notamment l'additivité des puissances : pour tout x, pour tout j et pour tout k,

$$x^j \times x^k = x^{j+k}$$

En particulier, on peut proposer la décomposition suivante : pour tout réel x et pour tout entier $n \geq 0$,

$$x^n = \begin{cases} x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor} & \text{si } n \text{ est pair} \\ x^{\lfloor n/2 \rfloor} \times x^{\lfloor n/2 \rfloor} \times x & \text{si } n \text{ est impair} \end{cases}$$

Par exemple :

$$\begin{cases} x^4 = x^{\lfloor 4/2 \rfloor} \times x^{\lfloor 4/2 \rfloor} = x^2 \times x^2 \\ x^5 = x^{\lfloor 5/2 \rfloor} \times x^{\lfloor 5/2 \rfloor} \times x = x^2 \times x^2 \times x \end{cases}$$

Remarque : ici la division est donc entière, on se rappelle cette dernière est notée `//` en Python.

Voici une version «rapide» de notre calcul de puissance qui exploite cette décomposition :

```python
def puissance_rapide(x : float, n : int) -> float:
    """ Précondition : n >= 0
    Retourne x élevé à la puissance n.
    """
    # Résultat
    res : float = 1

    # Accumulateur pour les puissances impaires
    acc : float = x

    # Variant de boucle
    i : int = n

    # Compteur de multiplications
    nb_mults : int = 0

    while i > 0:
        if i % 2 == 1:
            res = res * acc
            nb_mults = nb_mults + 1

        acc = acc * acc
        nb_mults = nb_mults + 1
        i = i // 2

    print("Nombre de multiplications =", nb_mults)

    return res
```

```
>>> puissance_rapide(2, 5)
Nombre de multiplications = 5
32
```

```
>>> puissance_rapide(2, 10)
Nombre de multiplications = 6
1024
```

```
>>> puissance_rapide(2, 100)
Nombre de multiplications = 10
1267650600228229401496703205376
```

```
>>> puissance_rapide(2, 1000)
Nombre de multiplications = 16
107150860718626732094842504906000181056140481170553360744375 0 ...
```

```
>>> puissance_rapide(2, 10000)
Nombre de multiplications = 19
199506311688075838488374216268358508382349683188619245485200 8 ...
```

Les résultats sont les mêmes que pour `puissance` donc la fonction `puissance_rapide` semble bien calculer x^n (on verra plus loin l'invariant de boucle correspondant). Le nombre de multiplications effectuées pour calculer x^n est cependant bien inférieur à n. En fait, on peut montrer que ce nombre est proche de $\frac{3}{2} \times \log_2(n)$ pour n «suffisamment grand».

```
>>> import math
>>> 3 / 2 * math.log(5, 2)
3.4828921423310435
```

```
>>> 3 / 2 * math.log(10, 2)
4.982892142331044
```

```
>>> 3 / 2 * math.log(100, 2)
9.965784284662089
```

```
>>> 3 / 2 * math.log(1000, 2)
14.948676426993131
```

```
>>> 3 / 2 * math.log(10000, 2)
19.931568569324178
```

Pour `n=100` nous sommes déjà très proche du nombre effectif de multiplications (10). L'information la plus importante est que ce nombre de multiplications est fonction du logarithme $\log_2(n)$ qui croît extrêmement lentement, et notamment beaucoup plus lentement que n. On comprend pourquoi les informaticiens sont de grands adeptes du logarithme !

Tout ceci illustre un point important:

> Les gains les plus importants en efficacité résultent d'une étude algorithmique.

En revanche, il faut comprendre une règle importante et très souvent vérifiée :

> Plus un algorithme est efficace, plus les questions concernant sa correction et sa terminaison sont complexes.

La terminaison de l'algorithme de `puissance_rapide` ne pose cependant pas trop de problème. En fait, nous pouvons tout simplement utiliser le variant de boucle `i`.

Vérifions sur la simulation de `puissance_rapide(2, 10)` donc pour `x=2` et `n=10` :

Tour de boucle	variable `res`	variable `acc`	variable `i` (**variant**)
entrée	**1**	**2**	**10**
1er	1	4	5
2e	4	16	2
3e	4	256	1
4e (sortie)	**1024**	**65536**	**0**

On constate bien que le variant diminue strictement et *drastiquement* : il est divisé par 2 à chaque étape ! C'est cette division par deux qui réduit grandement le nombre d'étapes de calcul, par rapport à la version «lente» dont le variant ne décroît que de un à chaque étape. La terminaison de la boucle est donc vérifiée ici : un entier naturel strictement positif que l'on divise par 2 - en division entière - à chaque étape finira par atteindre 0.

Pour ce qui concerne la correction, c'est nettement plus complexe et cela demande un peu d'imagination. Au final, voici ce que nous proposons comme invariant de boucle.

Candidat invariant de boucle : $res = \frac{x^n}{acc^i}$

Vérifions cet invariant sur notre simulation :

Tour de boucle	variable `res`	variable `acc`	variable `i`	**invariant** $res = \frac{x^n}{acc^i}$
entrée	**1**	**2**	**10**	$1 = \frac{2^{10}}{2^{10}}$ (Vrai)
1er	1	4	5	$1 = \frac{2^{10}}{4^5}$ (Vrai)
2e	4	16	2	$4 = \frac{2^{10}}{16^2}$ (Vrai)
3e	4	256	1	$4 = \frac{2^{10}}{256^1}$ (Vrai)
4e (sortie)	**1024**	**65536**	**0**	$1024 = \frac{2^{10}}{65536^0}$ (Vrai)

Bien sûr, cette simulation ne suffit pas à démontrer que cet invariant est correct, mais c'est déjà un bon indice. Et si l'on fait l'hypothèse que le variant de boucle est le bon, alors puisque l'on sait qu'en fin de boucle le variant est 0 on a une preuve que la valeur de `res` en sortie de boucle est bien la puissance x^n.

Nous retiendrons ici la règle suivante :

> Plus une fonction est efficace (au sens algorithmique), plus il est difficile d'en certifier la correction ainsi que la terminaison.

En pratique, on commence donc toujours par une solution la plus simple possible (du point de vue de la correction et de la terminaison) sans trop se soucier de l'efficacité. Une fois cette solution simple bien testée, on cherche des solutions plus efficaces. La version simple peut être alors utilisée pour tester la (ou les) version(s) efficace(s).

Donc un jeu de test particulièrement adapté au cas de `puissance_rapide` est le suivant :

```
# Jeu de tests
assert puissance_rapide(2, 5) == puissance(2, 5)
assert puissance_rapide(2, 10) == puissance(2, 10)
assert puissance_rapide(2, 100) == puissance(2, 100)
assert puissance_rapide(2, 1000) == puissance(2, 1000)
assert puissance_rapide(2, 10000) == puissance(2, 10000)
```

4.4 Complément : la récursion

Une question se pose :

> Existe-t-il un autre moyen que les boucles pour effectuer des calculs répétitifs ?

La réponse est oui et cet autre moyen se nomme **la récursion**.

On peut dire que les boucles ont été inventées par et pour les informaticiens. Mais les calculs répétitifs ont aussi intéressé quelques mathématiciens, et ce bien avant que les premiers ordinateurs n'existent.

Les mathématiques s'intéressent le plus souvent au QUOI - ce qui est calculé - mais de temps en temps aussi au COMMENT - comment les calculs sont effectués.

Considérons par exemple deux définitions mathématiques alternatives de la factorielle.

1. Un mathématicien qui s'intéresse plutôt au QUOI écrira sans doute :

$$n! = \prod_{i=1}^{n} i$$

On sait ici ce que l'on calcule : le produit des entiers de 1 à n. Mais on n'a pas de détail concernant la «recette» du calcul, par exemple l'ordre dans lequel les multiplications sont effectuées.

2. Un mathématicien qui s'intéresse plutôt au COMMENT écrira pluôt :

$$n! = \begin{cases} 1 & \text{si } n \leq 1 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

Il est un peu moins clair ici que l'on calcule le produit des entiers de 1 à n mais en revanche on sait exactement comment effectuer ce calcul.

Cette dernière version, dite **récursive**, se traduit très simplement en Python.

```python
def factorielle(n : int) -> int:
    """Précondition : n >= 0
    Retourne la factorielle de n.
    """
    if n <= 1:
        return 1
    else:
        return n * factorielle(n - 1)
```

```python
# Jeu de tests
assert factorielle(0) == 1
assert factorielle(1) == 1
assert factorielle(5) == 120
assert factorielle(6) == 720
```

L'avantage de la version récursive est qu'il y a accord entre la version mathématique récursive et l'implémentation associée en Python. Il existe des outils puissants pour tirer parti de cette proximité.

Tout d'abord, nous disposons d'un **principe d'évaluation par réécriture** qui est plus simple et plus détaillé que nos simulations.

Les deux règles de réécriture utilisées correspondent presque directement à notre définition :

```
[Règle 1] : fact(n)  --> 1  si n <= 1
[Règle 2] : fact(n)  -->   n * fact(n - 1)  sinon
```

On ajoute également une règle `[Mult]` pour la multiplication.

Les réécritures pour `fact(5)` sont alors les suivantes :

```
fact(5)                         [Règle 2]
--> 5 * fact(4)                 [Règle 2]
--> 5 * 4 * fact(3)             [Règle 2]
--> 5 * 4 * 3 * fact(2)         [Règle 2]
--> 5 * 4 * 3 * 2 * fact(1)     [Règle 1]
--> 5 * 4 * 3 * 2 * 1           [Mult]
--> 5 * 4 * 3 * 2               [Mult]
--> 5 * 4 * 6                   [Mult]
--> 5 * 24                      [Mult]
--> 120
```

De plus, il est plus facile de raisonner sur les définitions récursives que sur les boucles. Nous disposons en effet d'un principe fondamental de raisonnement : **le raisonnement par récurrence**. Grâce à ce principe nous pouvons en quelque sorte faire «d'une pierre deux coups» : démontrer la correction *et* garantir la terminaison des calculs.

En guise d'illustration, nous pouvons prouver formellement que `fact(n)` calcule bien le produit des entiers de 1 à n.

Posons la propriété au rang n :

$P(n)$: `fact(n)` calcule le produit des entiers de 1 à n

Les cas de base sont les suivants :

— $P(0)$: `fact(0)` calcule le produit des entiers de 1 à 0.

Par convention le produit de 0 entiers vaut 1 et `fact(0)` retourne 1 donc $P(0)$ est vraie.

— $P(1)$: `fact(1)` calcule le produit des entiers de 1 à 1.

Le produit de l'entier 1 est bien sûr 1, qui est la valeur retournée par `fact(1)`. Donc $P(1)$ est vraie également.

Pour le cas inductif (ou récursif), on suppose que $P(n)$ est vrai pour un entier naturel $n > 1$ (les cas pour $n \leq 1$ sont déjà démontrés). Nous devons montrer sous cette hypothèse dite *hypothèse de récurrence* que $P(n+1)$ est vraie.

— $P(n+1)$: `fact(n + 1)` calcule le produit des entiers de 1 à $n+1$

Si $n > 1$ la règle de réécriture `[Règle 2]` nous dit que la valeur de `fact(n + 1)` est la même que celle de `(n + 1) * fact(n)`. Or par hypothèse de récurrence on sait que $P(n)$ est vraie et donc que `fact(n)` calcule bien le produit des entiers de `1` à `n`. En multipliant par `(n + 1)` on obtient bien pour `fact(n + 1)` le produit des entiers de 1 à $n \times (n+1)$ donc le produit des entiers de 1 à $n+1$. On en déduit que $P(n+1)$ est vraie.

En conclusion, d'après le principe de raisonnement par récurrence, $P(n)$ est vraie pour un entier naturel n quelconque.

Donc notre fonction `fact` définie par récursion :

— termine pour tout entier naturel valeur de `n`
— calcule bien le produit de `1` à `n` donc la factorielle.

L'inconvénient principal de la récursion est spécifique au langage choisi : Python est un langage qui ne favorise par la récursion, en particulier du point de vue de l'efficacité. Python n'élimine par exemple pas les appels récursifs terminaux (contrairement à par exemple Scheme ou Ocaml), ce qui veut dire que chaque appel récursif consomme de la mémoire. Les fameux «dépassements de pile» en sont la conséquence immédiate. De plus, les appels de fonction (récursifs ou non) sont beaucoup plus couteux (toujours en Python) que les tours de boucle. C'est pourquoi dans ce livre nous utiliserons principalement les boucles pour effectuer des calculs répétitifs.

Chapitre 5

Intervalles et chaînes de caractères

Dans ce chapitre, nous allons aborder la manipulation de *données structurées* avec les **intervalles** de type `range` et les **chaînes de caractères** de type `str`. Ce sont des types de données structurées *en séquence* : les éléments contenus sont arrangés de façon séquentielle. Ainsi, les intervalles représentent des séquences de nombres (en général des entiers) et les chaînes de caractères représentent des séquences de caractères. Nous verrons lors du prochain chapitre un type de séquence plus général : les listes.

L'intérêt principal de cette classification est que certaines opérations, en particulier le *principe d'itération*, s'appliquent de la même façon aux différents types de séquence.

5.1 Intervalles

Le type de séquence le plus simple est l'**intervalle d'entiers**, qui correspond au type `range` en Python. La manipulation des intervalles concerne principalement la construction d'intervalle et l'itération sur un intervalle.

5.1.1 Construction d'intervalle

En python, on peut construire un intervalle d'entiers en faisant appel à la fonction `range`, ainsi l'expression :

```
range(m, n)
```

construit l'intervalle des entiers allant de m (inclus) à n (exclu), que l'on noterait $[m; n[$ (ou $[m; n-1]$) en mathématiques.

Par exemple, pour construire l'intervalle $[2; 6[$ des entiers de 2, 3, 4 et 5 (dans cet ordre) on utilise :

```
>>> range(2, 6)
range(2, 6)
```

On voit ici que les intervalles sont *auto-évalués* en Python, on considérera donc `range` comme un type atomique.

On peut utiliser des entiers relatifs, la seule condition à respecter pour `range(m,n)` étant que m soit inférieur à n.

Construisons par exemple l'intervalle $[-4; 3[$ des entiers de -4 (inclus) à 3 (exclu) :

```
>>> range(-4, 3)
range(-4, 3)
```

5.1.2 Itération

Une opération fondamentale disponible pour tous les types de séquence, et donc les intervalles, est l'**itération** avec la boucle `for`.

La syntaxe de cette opération est la suivante :

```
<var> : <type>
for <var> in <sequence>:
     <corps>
```

`<var>` est la *variable d'itération* dont le `<type>` est spécifié en premier lieu.

Le **principe d'interprétation** de la boucle `for` est le suivant :

- le `<corps>` de la boucle est une suite d'instructions qui est exécutée une fois pour chaque élément de la `<sequence>`, selon l'ordre séquentiel de ses éléments.
- dans le `<corps>`, la variable `<var>` est liée à l'élément courant : premier élément au premier tour, deuxième élément au deuxième tour ... jusqu'au dernier tour de boucle avec le dernier élément de la séquence.
- la variable `<var>` n'est plus utilisable après le dernier tour de boucle.

Traduit pour un intervalle `range(m, n)`, ce principe devient :

- le `<corps>` de la boucle est une suite d'instructions qui est exécutée une fois pour chaque entier `m`, `m+1`, ..., jusque `n-1` .
- dans le `<corps>`, la variable `<var>` est liée à l'entier courant : `m` au premier tour, `m+1` au deuxième tour ... jusqu'au dernier tour de boucle avec `n-1`.
- la variable `<var>` ne doit pas être utilisé après le dernier tour de boucle [1].

1. En pratique, Python autorise l'accès à la variable d'itération après la boucle mais c'est indéniablement une pratique *fortement* déconseillée.

Exemple : factorielle par itération

Suivant le principe d'itération, nous pouvons par exemple réécrire la fonction `factorielle` du chapitre 3 de façon plus concise et surtout de façon plus lisible qu'avec une boucle `while`.

```
def factorielle(n : int) -> int:
    """Précondition : n >= 0
    Renvoie la factorielle de n.
    """
    # produit cumulé en résultat
    r : int = 1

    i : int # entier courant
    for i in range(1, n + 1):
        r = r * i

    return r
```

Par exemple :

```
>>> factorielle(5)
120
```

Remarquons que si on veut déclarer le type de la variable d'itération (ce qui est souvent une bonne pratique) alors il est nécessaire de le faire juste au dessus du `for ... in ...`.

Nous pouvons effectuer une simulation de notre boucle `for`. À chaque itération du corps de la boucle, la variable `i` est égale à l'élément courant de l'intervalle. On indique la variable d'itération en premier dans la simulation, car d'une certain façon c'est la première variable modifiée à chaque tour, avant les modifications effetuées par le corps de la boucle.

Effectuons la simulation pour `factorielle(5)` donc pour `n=5` :

tour de boucle	variable `i`	variable `r`
entrée	-	**1**
1er tour	1	1
2e	2	2
3e	3	6
4e	4	24
5e	5	120
sortie	-	**120**

Contrairement aux variables locales - qui sont accessibles dans tout le corps de la fonction - les variables d'itération n'ont de sens que dans le corps de la boucle `for`, et il ne faut pas y accéder en dehors. C'est pour cela que dans la simulation ci-dessus on indique par

un tiret `-` que la variable d'itération `i` n'est pas accessible en entrée de boucle (avant le premier tour) ainsi qu'en sortie (après le dernier tour).

5.2 Chaînes de caractères

Les chaînes de caractères sont très courantes en informatique puisqu'on les utilise pour représenter des données textuelles de nature très variée : noms, adresses, titres de livres, définitions de dictionnaire, séquences d'ADN, etc.

5.2.1 Définition

Une **chaîne de caractères** (*string* en anglais) est une donnée de type `str` contenant des caractères rangés en ordre séquentiel, avec un premier caractère, un deuxième, etc.

Un **caractère** peut être :

— une lettre minuscule 'a', 'b' ... 'z' ou majuscule 'A', 'B', ... 'Z'
— des lettres d'alphabets autres que latins: 'α', 'β', etc.
— des chiffres '0', ..., '9'
— des symboles affichables '$', '%', '&', etc.

La norme *Unicode* prévoit des milliers de caractères différents couvrant à peu près tous les besoins des langues vivantes sur la planète, et même de certaines langues mortes (sumérien, hiéroglyphes Egyptiens, etc.).

Les manipulations courantes sur les chaînes de caractères peuvent être catégorisées de la façon suivante :

— opérations de base : construction, déconstruction et comparaisons
— problèmes de réduction
— problèmes de transformation et de filtrage
— autres problèmes qui ne rentrent pas dans les catégories précédentes.

5.2.2 Opérations de base sur les chaînes

5.2.2.1 Construction

La première question que l'on se pose sur les chaînes de caractères est la suivante :

> Comment créer une chaîne de caractères ?

On peut distinguer les constructions simples par des *expressions atomiques de chaînes* et les constructions complexes par des *expressions de concaténation*.

5.2.2.1.1 Expressions atomiques de chaînes Nous l'avons vu, les chaînes de caractères peuvent être construites directement par une expression atomique correspondant à une suite de caractères encadrée par des guillemets simples (`'`) ou doubles (`"`).

Pour construire la chaîne de caractères :

> Ceci est une chaîne

on peut donc écrire :

```
>>> 'Ceci est une chaîne'
'Ceci est une chaîne'
```

Remarquons au passage que le type d'une chaîne est bien `str`.

```
>>> type('Ceci est une chaîne')
str
```

La chaîne précédente peut être construite de façon équivalente avec des double guillemets :

```
>>> "Ceci est une chaîne"
'Ceci est une chaîne'
```

On remarque ici que Python répond toujours avec des guillemets simples, sauf si la chaîne contient elle-même une guillemet simple.

On a d'ailleurs deux façons principales d'écrire une chaîne contenant une guillemet simple :

Première solution - en encadrant par des guillemets doubles :

```
>>> "l'apostrophe m'interpelle"
"l'apostrophe m'interpelle"
```

Deuxième solution - en utilisant un *antislash* \ (barre oblique inversée) devant la guillemet simple faisant partie de la chaîne :

```
>>> 'l\'apostrophe m\'interpelle'
"l'apostrophe m'interpelle"
```

On remarque dans ce dernier cas que Python «préfère» encadrer par des guillemets doubles, ce qui est effectivement plus lisible.

Remarque : Contrairement à d'autres langages de programmation (comme le C, Java, etc.), le langage Python ne fait pas la différence entre *caractère* et *chaîne de un seul caractère*.

```
>>> 'a'
'a'
```

```
>>> type('a')
str
```

Il existe aussi la **chaîne vide** qui n'est composée d'aucun caractère :

```
>>> ''
''
```

Attention : il ne faut pas confondre les chaînes avec les autres types comme `int` ou `float`

```
>>> type('234')
str
```

```
>>> type(234)
int
```

Nous allons voir notamment que l'opérateur `+` possède une signification bien différente dans le cas des chaînes.

```
>>> 2 + 3
5
```

```
>>> '2' + '3'
'23'
```

5.2.2.1.2 Construction par concaténation Pour construire des chaînes «complexes» à partir de chaînes «plus simples», on utilise le plus souvent l'opérateur `+` qui réalise **la concaténation de deux ou plusieurs chaînes de caractères**.

La signature de cette opérateur est la suivante :

```
str * str  ->  str
```

Par exemple :

```
>>> 'alu' + 'minium'
'aluminium'
```

L'opérateur de concaténation est dit *associatif*, de sorte que pour toutes chaînes c_1, c_2 et c_3 : c_1 `+ ( ` c_2 `+` c_3 `) ==  (` c_1 `+` c_2 `) +` c_3 `==` c_1 `+` c_2 `+` c_3

```
>>> 'ainsi parlait ' + 'Zara' + 'thoustra'
'ainsi parlait Zarathoustra'
```

Un petit point de terminologie. On dit que la chaîne finale `'ainsi parlait Zarathoustra'` est le résultat de la concaténation des trois **sous-chaînes** `'ainsi parlait '`, `'Zara'` et `'thoustra'` (dans cet ordre).

En revanche, contrairement à l'addition numérique, l'opérateur de concaténation *n'est pas* commutatif :

```
>>> 'bon' + 'jour'
'bonjour'
```

```
>>> 'jour' + 'bon'
'jourbon'
```

Une autre propriété importante de l'opérateur de concaténation est qu'il a pour *élément neutre* la chaîne vide, ainsi :

```
>>> '' + 'droite'
'droite'
```

```
>>> 'gauche' + ''
'gauche'
```

Autrement dit, pour toute chaîne c on a les égalités suivantes :

$$c == (c \texttt{ +''}) \texttt{ == (''+ } c)$$

Pour illustrer l'utilisation de l'opérateur de concaténation, considérons le problème de construction suivant. On souhaite définir une fonction `repetition` le but est de faire «bégayer» des chaînes de caractères.

Par exemple :

```
>>> repetition('bla', 3)
'blablabla'
```

```
>>> repetition('zut ! ', 5)
'zut ! zut ! zut ! zut ! zut ! '
```

Une définition pour cette fonction est proposée ci-dessous :

```
def repetition(s : str, n : int) -> str:
    """Précondition : n >= 1
    Retourne la chaîne composée de n répétitions
    successives de la chaîne s.
    """
    # on initialise avec la chaîne vide puisque c'est l'élément neutre
    r : str = ""

    i : int # caractère courant
    for i in range(1, n + 1):
        r = r + s

    return r
```

```
# jeu de tests
assert repetition('bla', 3) == 'blablabla'
                # cf. égalité sur les chaînes un peu plus loin ...

assert repetition('zut ! ', 5) == 'zut ! zut ! zut ! zut ! zut ! '
```

Pour illustrer le fonctionnement de `repetition`, considérons la simulation correspondant à l'appel `repetition('bla', 3)` donc avec `s='bla'` et `n=3`.

tour de boucle	variable `i`	variable `r`
entrée	-	`''`
1er	1	`'bla'`
2e	2	`'blabla'`
3e	3	`'blablabla'`
sortie	-	`'blablabla'`

Remarque : La fonction `repetition` est en fait prédéfinie en Python, sous la forme de l'opérateur `*`.

Par exemple :

```
>>> 'bla' * 3
'blablabla'
```

```
>>> 'zut ! ' * 5
'zut ! zut ! zut ! zut ! zut ! '
```

5.2.2.2 Déconstruction

Maintenant que nous savons comment construire des chaînes, de façon directe ou grâce à une fonction comme `repetition`, découvrons le procédé inverse qui consiste à *déconstruire* (ou décomposer) une chaîne en sous-chaînes ou en caractères (sous-chaînes de 1 caractère).

5.2.2.2.1 Déconstruction en caractères L'opération de déconstruction la plus basique consiste à accéder au `i`-ème caractère d'une chaîne `s` par la syntaxe suivante :

`s[i]`

Chaque caractère d'une chaîne possède un indice unique permettant de le repérer.

Pour la chaîne `'Salut, ça va ?'` les indices sont les suivants :

Caractère	`S`	`a`	`l`	`u`	`t`	` `	`ç`	`a`	` `	`v`	`a`	` `	`?`
Indice	0	1	2	3	4	5	6	7	8	9	10	11	12

Comme c'est souvent le cas en informatique, les indices sont comptés à partir de zéro, ainsi :

- le premier caractère se trouve à l'indice 0, c'est donc le *0-ième* caractère
- le second caractère se trouve à l'indice 1, c'est donc le *1-ième* caractère
- etc.

Confirmons en pratique ces valeurs :

```
>>> ch : str
... ch = 'Salut ça va ?'
```

Remarque : les exemples qui suivent font souvent référence à la variable `ch` définie ci-dessus. Nous n'effectuerons aucune affectation sur cette variable donc son contenu sera toujours la chaîne `'Salut ça va ?'`.

```
>>> ch[0]
'S'
```

```
>>> ch[1]
'a'
```

```
>>> ch[9]
'v'
```

```
>>> ch[12]
'?'
```

Attention : si on accède à un indice au-delà du dernier caractère, une erreur est signalée.

```
>>> ch[13]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
     ...
----> 1 ch[13]

IndexError: string index out of range
```

Exercice - donner le résultat des appels suivants :

- `ch[8]`
- `ch[7]`
- `ch[21]`

Les indices négatifs, cependant, ont une signification particulière :

`s[-i]`

retourne le $(i-1)$-ème caractère en partant de la fin, ainsi :

- s[-1] retourne le dernier caractère de la chaîne,
- s[-2] retourne l'avant-dernier caractère,
- etc.

Complétons notre table des indices :

Caractère	S	a	l	u	t		ç	a		v	a		?
Indice (normal)	0	1	2	3	4	5	6	7	8	9	10	11	12
Indice (inverse)	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> ch[-1]
'?'
```

```
>>> ch[-7]
'ç'
```

```
>>> ch[-13]
'S'
```

Encore une fois, il ne faut pas utiliser d'indice négatif qui tenterait d'accéder avant le premier caractère :

```
>>> ch[-14]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
        ...
----> 1 ch[-14]

IndexError: string index out of range
```

Exercice - donner le résultat des appels suivants :

- `ch[-11]`
- `ch[-9]`
- `ch[-21]`

5.2.2.2.2 Découpage de chaîne Une opération plus générale que l'accès à un caractère par son index consiste à effectuer un **découpage** (*slice* en anglais) d'une chaîne `s` par la syntaxe suivante :

```
s[i:j]
```

qui retourne la sous-chaîne de `s` située entre les indices `i` (inclus) et `j` (non-inclus).

Voici quelques exemples (toujours sur notre chaîne référencée par la variable `ch`) :

```
>>> ch[2:9]
'lut ça '
```

```
>>> ch[9:11]
'va'
```

```
>>> ch[9:13]
'va ?'
```

Exercice - donner le résultat des appels suivants :

- `ch[1:7]`
- `ch[4:13]`
- `ch[7:8]`

Que pensez-vous de la dernière expression ? Peut-on la simplifier ?

Dans le chapitre suivant (sur les listes), nous aborderons des découpages de séquences plus complexes mais dans l'immédiat nous nous limiterons aux découpages simples comme ci-dessus.

5.2.2.3 Comparaison de chaînes

Les séquences comme les chaînes de caractères (ou même les intervalles même si cela représente peu d'intérêt en pratique) peuvent être comparées entre elles, en particulier pour l'**égalité** et l'**inégalité**. Les opérateurs sont les mêmes que pour les autres types de données :

`<chaîne1> == <chaîne2>`

retourne `True` si les deux chaînes sont identiques, c'est-à-dire qu'elles contiennent exactement les mêmes caractères aux mêmes indices, et `False` sinon.

`<chaîne1> != <chaîne2>`

retourne l'inverse, c'est-à-dire :

`not (<chaîne1> == <chaîne2>)`

Par exemple :

```
>>> 'Ceci est une chaîne' == 'Ceci est une autre chaîne'
False
```

```
>>> 'Ceci est une chaîne' != 'Ceci est une autre chaîne'
True
```

Remarque : les lettres minuscules et majuscules sont distinguées dans les comparaisons.

```
>>> 'c' == 'C'
False
```

```
>>> 'Ceci est une chaîne' == 'ceci est une chaîne'
False
```

5.2.2.3.1 Complément : comparateurs d'ordre sur les chaînes de caractères

Au delà de l'égalité et de l'inégalité, on peut comparer des chaînes (et plus généralement des séquences) selon un ordre défini par la norme *Unicode* et qui correspond à peu près à l'*ordre lexicographique* du dictionnaire.

L'opérateur principal est le suivant :

`<chaîne1> < <chaîne2>`

Retourne vrai (`True`) si la `<chaîne1>` est "strictement inférieure" à la `<chaîne2>`

Dans l'ordre lexicographique, le caractère 'a' est par exemple inférieur à 'b' (tout comme 'a' est avant 'b' dans le dictionnaire).

```
>>> 'a' < 'b'
True
```

— Si une chaîne commence par un caractère inférieur au premier caractère d'une seconde chaîne, alors la première chaîne est inférieure à la deuxième. On retrouve ici un ordre équivalent à celui des mots dans un dictionnaire.

```
>>> 'azozo' < 'baba'
True
```

On remarque ici que la chaîne la plus longue est inférieure à l'autre. L'important ici est que le premier caractère `a` est inférieur à `b`. Cet ordre se propage à la chaîne complète.

— Si les deux chaînes commencent par le même caractère, alors la comparaison se propage au deuxième caractère, et ainsi de suite.

```
>>> 'abab' < 'acab'
True
```

```
>>> 'abcab' < 'abcb'
True
```

— Si la première chaîne préfixe la seconde (tous les caractères sont identiques), alors elle est inférieure si elle est de longueur inférieure.

```
>>> 'abcdef' < 'abcdefg'
True
```

— Dans tous les autres cas, la première chaîne n'est pas inférieure.

```
>>> 'abcdef' < 'abcdef'  # égales
False
```

```
>>> 'baba' < 'ac'  # premier caractère plus grand
False
```

Les opérateurs dérivés de comparaisons sont les suivants :

— inférieur ou égal
`<chaîne1> <= <chaîne2>`

Retourne la même valeur que :

`(<chaîne1> < <chaîne2>) or (<chaîne1> == <chaîne2>)`

— supérieur strict
`<chaîne1> > <chaîne2>`

Retourne la même valeur que `<chaîne2> < <chaîne1>`

— supérieur ou égal
`<chaîne1> >= <chaîne2>`

Retourne la même valeur que :

`(<chaîne1> > <chaîne2>) or (<chaîne1> == <chaîne2>)`

5.3 Problèmes sur les chaînes de caractères

Nous allons maintenant étudier différents problèmes que l'on peut avoir à résoudre sur les chaînes de caractères. Certains problèmes se ressemblent, et il est alors possible de les regrouper en classes de problèmes génériques. Nous allons voir des exemples dans les classes suivantes : problèmes de *réduction* et problèmes de *transformation* et de *filtrage*.

Bien entendu tous les problèmes n'appartiennent pas à ces trois classes, et nous verrons des exemples d'autres types de problèmes.

5.3.1 Réductions

Le principe de **réduction** d'une structure de données, comme une séquence, consiste à synthétiser une information «plus simple» en parcourant les éléments contenus dans la structure.

Pour les chaînes de caractères, les problèmes de réduction consistent donc à produire une information «simple» - le plus fréquemment de type `bool` ou `int` - synthétisée à partir du *parcours* (complet ou non) des éléments de la chaîne.

Les fonctions de réduction possèdent une signature de la forme :

— `str -> int` : réduction d'une chaîne vers le type entier
— `str -> bool` : réduction d'une chaîne vers le type booléen

et plus généralement :

— `str * ... -> T` : réduction d'une chaîne vers le type `T` (avec éventuellement des paramètres supplémentaires).

Pour réaliser une réduction de chaîne, on dispose principalement de deux approches complémentaires :

— la réduction par itération des éléments de la chaîne avec `for`, ou
— la réduction par parcours des indices de la chaîne.

5.3.1.1 Réduction par itération

Comme les intervalles et tous les autres types de séquences, on peut parcourir les caractères d'une chaîne par itération avec la boucle `for`.

La syntaxe pour les chaînes se déduit du principe plus général sur les séquences décrit précédemment :

```
<var> : str
for <var> in <chaîne>:
    <corps>
```

Avec le **principe d'interprétation** correspondant :

- le `<corps>` de la boucle est une suite d'instructions qui est exécutée une fois pour chaque caractère de la `<chaîne>`, selon leur ordre d'indice.
- dans le `<corps>`, la variable `<var>` est liée au caractère courant : premier caractère (indice 0) au premier tour, deuxième caractère (indice 1) au deuxième tour ... jusqu'au dernier tour de boucle avec le dernier caractère de la séquence (indice $longueur - 1$).
- la variable `<var>` n'est plus disponible après le dernier tour de boucle.

5.3.1.1.1 Exemple 1 : longueur d'une chaîne de caractères

Comme premier problème de réduction, calculons une information fondamentale sur les chaînes (et des séquences en général) : leur *longueur*.

Définition : la **longueur** d'une chaîne est le nombre de caractères qui la composent.

Par exemple, la chaîne `'Ceci est une chaîne'` possède 19 caractères indicés de 0 à 18, sa longueur est donc 19.

La spécification de la fonction `longueur` est la suivante :

```
def longueur(s : str) -> int:
    """Retourne la longueur de la chaîne s.
    """
```

Par exemple :

```
>>> longueur('Ceci est une chaîne')
19
```

```
>>> longueur('vingt-quatre')
12
```

Cas particulier : la chaîne vide est de longueur 0

```
>>> longueur('')
0
```

Autre cas particulier : un caractère est une chaîne de longueur 1.

```
>>> longueur('a')
1
```

La fonction `longueur` peut être définie de la façon suivante :

```
def longueur(s : str) -> int:
    """Retourne la longueur de la chaîne s.
    """
    # Comptage de la longueur initialement à zéro
    lng : int = 0

    c : str # caractère courant
    for c in s:
        lng = lng + 1  # longueur incrémentée pour chaque caractère

    return lng
```

```
# jeu de tests
assert longueur('Ceci est une chaîne') == 19
assert longueur('vingt-quatre') == 12
assert longueur('') == 0
assert longueur('a') == 1
```

La réduction *longueur de chaîne* est tellement primordiale qu'elle est en fait prédéfinie en Python. Il s'agit de la fonction `len` qui est utilisable sur n'importe quelle séquence.

```
>>> len('Ceci est une chaîne')
19
```

```
>>> len('vingt-quatre')
12
```

```
>>> len('')
0
```

```
>>> len('a')
1
```

Remarque : en pratique, on utilisera toujours la fonction `len` prédéfinie qui est beaucoup plus efficace que `longueur`, la vocation de cette dernière étant essentiellement pédagogique. En effet, `longueur` parcourt la chaîne en entier alors que `len` ne fait quasiment aucun calcul puisque Python maintient en interne la longueur des chaînes (et des séquences en général).

5.3.1.1.2 Exemple 2 : nombre d'occurrences d'un caractère

On a déjà vu qu'un même caractère peut apparaître à plusieurs indices d'une même chaîne.

Par exemple :

```
>>> ch : str
... ch = 'les revenantes'
```

```
>>> ch[1]
'e'
```

```
>>> ch[5]
'e'
```

```
>>> ch[7]
'e'
```

```
>>> ch[12]
'e'
```

On dit qu'il y quatre **occurrences** du caractère `'e'` dans la chaîne ci-dessus : une occurrence à l'indice 1, une autre à l'indice 5, une troisième à l'indice 7 et une dernière à l'indice 12.

Un problème typique concernant les chaînes consiste à définir une fonction `occurrences` qui compte le nombre d'occurrences d'un caractère donné dans une chaîne. Il s'agit d'une réduction vers le type `int`.

Par exemple :

```
>>> occurrences('e', 'les revenantes')
4
```

```
>>> occurrences('t', 'les revenantes')
1
```

```
>>> occurrences('e', 'la disparition')
0
```

La fonction `occurrences` peut être définie de la façon suivante :

```
def occurrences(c : str, s : str) -> int:
    """Précondition : len(c) == 1
    Retourne le nombre d'occurrences du caractère c dans la chaîne s"""

    # Nombre d'occurrences du caractère
    nb : int = 0

    d : str # Caractère courant
    for d in s:
        if d == c:
            nb = nb + 1

    return nb
```

```
# jeu de tests
assert occurrences('e', 'les revenantes') == 4
assert occurrences('t', 'les revenantes') == 1
assert occurrences('e', 'la disparition') == 0
assert occurrences('z', '') == 0
```

5.3.1.1.3 Exemple 3 : présence d'un caractère Un sous-problème très classique du comptage d'occurrences est le test de la présence d'un caractère dans une chaîne. Il s'agit d'une réduction vers le type `bool`.

On peut déduire une solution simple en considérant la propriété suivante :

> Un caractère est présent dans une chaîne si son nombre d'occurrence est strictement positif.

```
def presence(c : str, s : str) -> bool:
    """Précondition : len(c) == 1
    Retourne True si le caractère c est présent dans la chaîne s,
    ou False sinon
    """
    return occurrences(c, s) > 0
```

```
# Jeu de tests
assert presence('e', 'les revenantes') == True
assert presence('e', 'la disparition') == False
```

Cette solution fonctionne mais ne peut satisfaire l'informaticien toujours féru d'efficacité. Considérons en effet l'exemple ci-dessous :

```
>>> presence('a', 'abcdefghijklmnopqrstuvwxyz')
True
```

La réponse est bien sûr `True` mais pour l'obtenir, la fonction `occurrences` (appelée par `presence`) a parcouru l'ensemble de la chaîne `'abcdefghijklmnopqrstuvwxyz'` pour compter les occurrences de `'a'`, soit 26 comparaisons.

On aimerait ici une solution «directe» au problème de présence qui s'arrête dès que le caractère cherché est rencontré. Dans le pire des cas, si le caractère recherché n'est pas présent, alors on effectuera autant de comparaisons que dans la version qui compte les occurrences, mais dans les autres cas on peut gagner de précieuses nanosecondes de temps de calcul !

Pour arrêter le test de présence dès que l'on a trouvé notre caractère, nous allons effectuer une *sortie anticipée* de la fonction avec `return`.

```
def presence(c : str, s : str) -> bool:
    """ ... cf. ci-dessus ...
    """
    d : str # Caractère courant
    for d in s:
        if d == c:
            return True # c'est gagné, on sort de la fonction
                        # (et donc de la boucle).

    return False # si on a parcouru toute la chaîne s,
                 # on sait que c n'est pas présent.
```

```
# jeu de tests
assert presence('e', 'les revenantes') == True
assert presence('e', 'la disparition') == False
assert presence ('z', '') == False
```

Effectuons une simulation pour `presence('e', 'les revenantes')`, donc pour `c='e'` et `s='les revenantes'` afin de constater le gain obtenu :

Tour de boucle	variable `d`
entrée	-
1er	`l`
2e	`e`
sortie (anticipée)	-

Exercice : comparer la simulation précédente avec celle de `occurrences('e', 'les revenantes')`

5.3.1.2 Réduction par parcours des indices

Dans certains cas, le parcours des chaînes par itération sur les caractères qui la composent ne permet pas de résoudre simplement le problème posé. Dans ces cas-là, on peut utiliser un parcours basé sur les indices des caractères dans les chaînes.

Le problème sans doute le plus simple dans cette catégorie est une variante du test de présence. On souhaite définir une fonction `recherche` qui recherche un caractère dans une chaîne, et retourne l'indice de la première occurrence de ce caractère s'il est présent. Si le caractère est absent, alors la fonction retourne la valeur `None`.

Par exemple :

```
>>> recherche('e', 'les revenantes')
1
```

```
>>> recherche('n', 'les revenantes')
8
```

```
>>> recherche('e', 'la disparition')
```

Dans ce dernier cas, Python ne produit aucun affichage. C'est le signe que la valeur `None` a été retournée. Vérifions ce fait :

```
>>> recherche('e', 'la disparition') == None
True
```

Cette caractéristique particulière fait de la fonction `recherche` un *fonction partielle*. Prenons un peu de temps pour préciser ce concept important.

Définition : une **fonction partielle** est une fonction qui ne renvoie pas toujours de résultat.

Dans le cas précis de la fonction `recherche` :

- on retourne un résultat de type `int` si dans `recherche(c, s)` la chaîne `s` possède au moins une occurrence du caractère `c`,
- on ne retourne pas de résultat - donc on retourne `None` - si le caractère `c` n'est pas présent dans `s`.

La signature correspondante est :

```
str * str -> Optional[int]
```

que l'on peut interpréter par :

> Une fonction partielle qui prend deux chaînes de caractères en paramètres, et retourne soit un résultat entier soit pas de résultat.

Dans le cas général, une fonction partielle retournant un type `T` possède dans sa signature le type de retour `Option[T]`. Nous verrons d'autres exemples de fonctions partielles dans ce chapitre et les suivants.

Remarque : pour pouvoir utiliser le type `Optional[T]` il faudra l'importer depuis le module `typing`, par exemple de la façon suivante :

```
from typing import Optional
```

La définition proposée pour `recherche` est la suivante :

```
def recherche(c : str, s : str) -> Optional[int]:
    """Précondition :: len(c) == 1
    Retourne l'indice de la première occurrence du caractère c dans
    la chaîne s, ou None si le caractère n'est pas présent.
    """
    # Indice courant
    i : int = 0  # on commence par l'indice
                 # du premier caractère

    while i < len(s):  # on "regarde" les indices
                       # de 0 (premier caractère)
                       # à len(s) - 1 (dernier caractère)
        if s[i] == c:
            return i  # si c est présent, on retourne directement
                      # l'indice courant
        else:
            i = i + 1 # sinon on passe à l'indice suivant

    return None   # si on a parcouru tous les indices,
```

```
                    # alors le caractère n'est pas présent,
                    # donc on retourne  "pas de résultat".
```

```
# Jeu de tests
assert recherche('e', 'les revenantes') == 1
assert recherche('n', 'les revenantes') == 8
assert recherche('e', 'la disparition') == None
```

Puisque l'on connaît maintenant les itérations sur les intervalles, il est possible de proposer une définition plus concise de la fonction `recherche` en itérant sur l'intervalle `range(0, len(s))`. Cet intervalle contient tous les entiers allant de `0` (inclus) à `len(s)` (exclus), ce qui correspond à tous les indices des caractères de la chaîne `s`.

On peut donc proposer une définition plus concise de la fonction `recherche` :

```
def recherche(c : str, s : str) -> Optional[int]:
    """ ... cf. ci-dessus ...
    """
    i : int
    for i in range(0,len(s)):
        if s[i] == c:
            return i

    return None
```

Analyse d'une valeur optionnelle

Lorsque l'on veut vérifier si une valeur `<val>` de type `Optional[T]` pour un type `T` donné, on écrira un test de la forme suivante :

```
if <val> is None:
  # cas 1 : pas de valeur
  ...
else:
  # cas 2 : valeur de type T
  ...
```

On remarque l'utilisation du comparateur `is` plutôt que `==`. Ce comparateur est beaucoup plus restrictif car le contenu des valeurs comparées ne sont pas analysée. Techniquement, `x is y` vaue `True` uniquement si les deux objets `x` et `y` sont stockés à la même adresse mémoire, on parle alors d'*égalité référentiel*. Pour le `None` cela marche très bien puisqu'il n'existe qu'un seul objet `None` dans l'interprète Python. Dans ce livre, c'est la seule utilisation de `is` que nous ferons, ce qui permettra de bien marquer les endroits où l'on utilise des fonctions partielles.

La définition ci-dessous montre un cas d'utilisation possible de la fonction `recherche` :

```
def presence_bis(c : str, s : str) -> bool:
    """Précondition : len(c) == 1
    Retourne True si et seulement c est présetant dans s.
    """
```

```
    if recherche(c, s) is None:
        return False
    else:
        return True
```

```
# Jeu de tests
assert presence_bis('e', 'les revenantes') == True
assert presence_bis('e', 'la disparition') == False
```

5.3.2 Transformations et filtrages

De nombreux problèmes sur les chaînes de caractères consistent à analyser une chaîne en entrée pour produire une autre chaîne en sortie.

La signature de base correspondant à ce type d'analyse est la suivante :

```
str -> str
```

(avec bien sûr la possibilité d'avoir d'autres paramètres en entrée).

Les grands classiques de ce type d'analyse sont :

- les *transformations* qui consistent à «modifier» les caractères d'une chaîne individuellement
- les *filtrages* qui synthétisent une sous-chaîne à partir d'une chaîne, selon un prédicat donné
- des combinaisons plus ou moins complexes des deux précédents.

Exemple de transformation : substitution

Une **transformation** de chaîne consiste à appliquer une même opération à chacun des éléments d'une chaîne de caractères. Le résultat produit est donc une chaîne de même longueur que la chaîne initiale.

Prenons l'exemple de la substitution de caractère, permettant notamment la création de codages simples.

La spécification proposée est la suivante :

```
def substitution(c : str, d : str, s : str) -> str:
    """Précondition : (len(c) == 1) and (len(d) == 1)
    Retourne la chaîne résultant de la substitution du caractère c par
    le caractère d dans la chaîne s.
    """
```

Par exemple :

```
>>> substitution('e', 'z', 'ceci est un code tres secret')
'czci zst un codz trzs szcrzt'
```

```
>>> substitution('z', 'e', "ceci n'est pas un tres bon code secret")
"ceci n'est pas un tres bon code secret"
```

Voici une définition possible pour la fonction `substitution` :

```
def substitution(c : str, d : str, s : str) -> str:
    """ ... cf. ci-dessus ...
    """
    # Chaîne résultat
    r : str = ''

    e : str  # Caractère courant
    for e in s:
        if e == c:
            r = r + d  # on substitue le caractere e par d
        else:
            r = r + e # on garde le caractere e

    return r
```

```
# jeu de tests
assert substitution('e', 'z', 'ceci est un code tres secret') \
       == 'czci zst un codz trzs szcrzt'
assert substitution('z', 'e', "ceci n'est pas un tres bon code secret") \
       == "ceci n'est pas un tres bon code secret"

# Remarque : l'anti-slash \  permet de continuer sur la ligne suivante.
```

Exemple de filtrage : suppression

La **filtrage** d'une chaîne consiste à reproduire une chaîne en supprimant certains de ses caractères. La **condition de filtrage** qui décide si un caractère doit être retenu - on dit que le caractère *passe le filtre* - ou au contraire supprimé - on dit que le caractère *ne passe pas le filtre* - peut être arbitrairement complexe.

La condition de filtrage la plus simple est sans doute l'égalité avec un caractère donné, ce qui entraîne la suppression de toutes les occurrences de ce caractère dans la chaîne initiale pour produire la chaîne filtrée.

La spécification correspondante est la suivante :

```
def suppression(c : str, s : str) -> str:
    """Précondition : len(c) == 1
    Retourne la sous-chaîne de s dans laquelle toutes
    les occurrences du caractère c ont été supprimées.
    """
```

Par exemple :

```
>>> suppression('e', 'les revenantes')
'ls rvnants'
```

```
>>> suppression('e', 'la disparition')
'la disparition'
```

```
>>> suppression('z', '')
''
```

Voici la solution proposée :

```
def suppression(c : str, s : str) -> str:
    """ ... cf. ci-dessus ..."""

    # Chaîne résultat
    r : str = ''

    d : str # Caractère courant
    for d in s:
        if d != c:
            r = r + d  # ne pas supprimer, donc ajouter au résultat

        # sinon ne rien faire (supprimer)
        # car on a trouvé une occurrence de c

    return r
```

```
# Jeu de tests
assert suppression('e', 'les revenantes') == 'ls rvnants'
assert suppression('e', 'la disparition') == 'la disparition'
assert suppression('z', '') == ''
```

5.3.3 Exemples de problèmes plus complexes

Pour terminer, intéressons-nous aux problèmes qui sortent du cadre de notre classification. Cette dernière est utile car de nombreux problèmes correspondent soit à des constructions, des réductions, des transformations ou des filtrages. Mais il existe bien sûr d'autres problèmes qui «résistent» à cette classification. Ces problèmes sont en général de nature plus complexe.

Exemple 1 : inversion

En guise d'illustration, considérons le problème d'inversion de chaîne de caractère. Le problème n'est en fait pas très complexe : les indices des caractères de la chaîne inversée sont simplement inversés :

— le premier caractère de la chaîne initiale devient le dernier caractère de la chaîne inversée
— le deuxième caractère ... devient l'avant-dernier ...
— ... etc ...
— l'avant-dernier caractère ... devient le deuxième caractère ...
— le dernier caractère de la chaîne initiale devient le premier caractère de la chaîne inversée

La spécification de la fonction `inversion` qui doit résoudre ce problème est la suivante :

```
def inversion(s : str) -> str:
    """Retourne la chaîne s inversée.
    """
```

Par exemple :

```
>>> inversion('abcd')
'dcba'
```

```
>>> inversion('a man a plan a canal panama')
'amanap lanac a nalp a nam a'
```

```
>>> inversion('')
''
```

Remarquons que même si la fonction `inversion` possède une signature `str -> str` elle ne réalise pas directement une transformation ou un filtrage. En fait on pourrait parler de *pliage* (en anglais *fold* ou *folding*) qui est une généralisation du principe de réduction, mais cela nous emmènerait un peu trop loin dans notre classification.

Même si on ne peut classifier simplement ce problème, la définition de la fonction `inversion` reste tout de même assez simple.

```
def inversion(s : str) -> str:
    """Retourne la chaîne s inversée."""

    # Chaîne inversée résultat
    r : str = ''

    c : str # Caractère courant
    for c in s:
        r = c + r # le caractère courant c est placé
                  # au début de la nouvelle chaîne en construction

    return r
```

```
# Jeu de tests
assert inversion('abcd') == 'dcba'
assert inversion('A man, a plan, a canal : Panama') \
       == 'amanaP : lanac a ,nalp a ,nam A'
assert inversion('') == ''
```

Pour comprendre le principe d'inversion, effectuons la simulation de l'exemple `inversion('abcd')` donc pour `s='abcd'`:

tour de boucle	variable `c`	variable `r`
entrée	-	`''`
1er	`'a'`	`'a'`
2e	`'b'`	`'ba'`
3e	`'c'`	`'cba'`
4e	`'d'`	`'dcba'`
sortie	-	*`'dcba'`*

Exemple 2 : entrelacement

Le deuxième problème qui nous intéresse concerne l'entrelacement de deux chaînes de caractères.

Le résultat est une nouvelle chaîne composée de la façon suivante :

- son premier caractère est le premier caractère de la première chaîne de départ,
- son second caractère est le premier caractère de la seconde chaîne,
- son troisième caractère est le second caractère de la première chaîne,
- son quatrième caractère est le second caractère de la seconde chaîne,
- etc.

Par exemple :

```
>>> entrelacement('ace', 'bdf')
'abcdef'
```

Lorsque tous les caractères d'une des deux chaînes ont été reconstruits, on recopie directement les caractères restant dans l'autre chaîne.

Par exemple :

```
>>> entrelacement('aceghi', 'bdf')
'abcdefghi'
```

```
>>> entrelacement('ace', 'bdfghi')
'abcdefghi'
```

```
>>> entrelacement('abc', '')
'abc'
```

```
>>> entrelacement('', 'abc')
'abc'
```

La spécification de cette fonction est donc la suivante :

```
def entrelacement(s1 : str, s2 : str) -> str:
    """Renvoie la chaîne constituée par l'entrelacement
```

```
        des caractères des chaînes s1 et s2.
    """
```

Ce n'est clairement pas une fonction d'une des catégories vues précédemment car pour construire la chaîne résultat, nous devons analyser deux chaînes fournies en paramètres et non une seule. De ce fait, la boucle `for` n'est pas très adaptée puisqu'elle ne permet d'itérer qu'une seule chaîne et non deux simultanément comme nous devons le faire ici. Nous allons donc utiliser un parcours des chaînes par les indices de caractères, avec un compteur et une boucle `while`.

La définition proposée est la suivante :

```
def entrelacement(s1 : str, s2 : str) -> str:
    """ ... cf. ci-dessus ...
    """
    # Indice pour parcourir les deux chaînes
    i : int = 0

    # Chaîne résultat
    r : str = ''

    while (i < len(s1)) and (i < len(s2)):
        r = r + s1[i] + s2[i]
        i = i + 1

    if i < len(s1):
        r = r + s1[i:len(s1)]
    elif i < len(s2):
        r = r + s2[i:len(s2)]

    return r
```

```
# Jeu de tests
assert entrelacement('ace', 'bdf') == 'abcdef'
assert entrelacement('aceghi', 'bdf') == 'abcdefghi'
assert entrelacement('ace', 'bdfghi') == 'abcdefghi'
assert entrelacement('abc', '') == 'abc'
assert entrelacement('', 'abc') == 'abc'
assert entrelacement('', '') == ''
```

Pour bien comprendre le fonctionnement de cette fonction non-triviale, regardons la simulation de la boucle pour `entrelacement('ace', 'bdfghi')` donc pour `s1='ace'` et `s2='bdfghi'`:

tour de boucle	variable `r`	variable `i`
entrée	`''`	0
1er	`'ab'`	1

tour de boucle	variable `r`	variable `i`
2e	`'abcd'`	2
3e	`'abcdef'`	3
sortie	`'abcdef'`	3

Après le 3ème tour de boucle la variable `i` vaut 3 donc la condition `i < len(s1)` est fausse puisque `len(s1) == 3` donc la condition de boucle est fausse et on sort de la boucle.

En revanche, puisque `len(s2) == 6` on exécute la branche `elif`. Comme `s2[3:6] == 'ghi'` la valeur de la variable `r` après cette étape est la suivante :

`r == 'abcdefghi'`

C'est finalement la valeur retournée par la fonction.

```
>>> entrelacement('ace', 'bdfghi')
'abcdefghi'
```

Chapitre 6

Listes

Nous avons introduit dans le chapitre précédent deux types de *séquence* :

- les intervalles de type `range` qui sont des séquences d'entiers,
- les chaînes de caractères de type `str` qui sont des séquences de caractères.

Dans ce chapitre, nous introduisons un type de séquence plus général - *les listes* - qui peuvent contenir des éléments d'autres types que simplement des entiers ou des caractères.

6.1 Définition et opérations de base

Définition : Soit `T` un type (ex.: `int`, `bool`, ou tout autre expression de type). Une **liste** de type `List[T]` est une séquence dont tous les éléments sont du même type `T`.

Pour pouvoir déclarer le type des liste, nous devons au préalable effectuer l'import suivant :

```
from typing import List
```

6.1.1 Construction explicite

Une **expression atomique de liste** de type `List[T]`, pour un type `T` donné, est soit :

- la *liste vide* notée `[]`
- une liste spécifique d'éléments notée `[`e_0 `,` e_1 `,` $\ldots$ `,` e_{n-1} `]` où chacun des e_i est une expression quelconque de type `T`.

Par exemple :

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

une liste de type `List[int]`.

```
>>> [1+1, 3.2, 2*2, 11 / 2]
[2, 3.2, 4, 5.5]
```

une liste de type `List[float]` car elle mélange des entiers et des flottants.

```
>>> ['chat', 'ours', 'pomme']
['chat', 'ours', 'pomme']
```

une liste de type `List[str]`, ou encore :

```
>>> [True, False, True]
[True, False, True]
```

une liste de type `List[bool]`

Pour chaque liste ainsi créée, il faut donc remplacer la *variable de type* T par une expression décrivant un type spécifique: `int`, `float`, `bool`, `str` (ou bien des combinaisons de types plus complexes comme celles que nous étudions dans le chapitre suivant).

Le cas de la liste vide est un petit peu particulier puisque cette dernière est compatible avec tous les types d'élément. L'expression `[]` peut être vue comme une liste d'entiers de type `List[int]`, comme une liste de chaînes de type `List[str]`, etc. En fait `[]` est de type `List[T]` pour *n'importe quel* type `T`.

Remarque : dans ce livre, nous manipulerons uniquement des *listes homogènes* qui ne contiennent que des éléments d'un même type `T` donné. En interne, l'interprète Python ne fait pas la différence entre listes homogènes et listes hétérogènes, c'est-à-dire contenant des éléments de types indéterminés.

6.1.2 Longueur de liste

Le nombre d'éléments d'une liste est fini, mais quelconque, et correspond à la **longueur de la liste**.

- la liste vide `[]` n'a pas d'élément et est donc de longueur 0
- une liste spécifique `[`e_0 `,` e_1 `,` $\ldots$ `,` e_{n-1} `]` est de longueur n

Comme pour les chaînes de caractères (et toutes les séquences en général), on peut utiliser la fonction prédéfinie `len` de Python pour s'enquérir de la longueur d'une liste. La signature de `len` pour les listes est la suivante :

```
List[T] -> int
```

Par exemple :

la liste `[1, 2, 3, 4, 5]` est de longueur 5.

```
>>> len([1, 2, 3, 4, 5])
5
```

Et `[True,False]` est de longueur 2.

```
>>> len([True, False])
2
```

Et bien sûr on peut vérifier que la liste vide est bien de longueur 0.

```
>>> len([])
0
```

6.1.3 Egalité et inégalité

Les opérateurs d'égalité `==` et d'inégalité `!=` sont également disponibles pour les listes.

Soients `l1` et `l2` deux listes de *même type* `List[T]`,

`l1 == l2`

vaut `True` si `l1` et `l2` sont de la même taille `n` et pour `i` entre `0` et `n-1` alors :

`l1[i] == l2[i]`

dans tous les autres cas, les listes `l1` et `l2` sont considérées comme inégales et la valeur `False` est retournée.

En complément,`l1 != l2` retourne la même valeur que : `not (l1 == l2)`.

Par exemple :

```
>>> [1, 2, 3, 4, 5] == [1, 2, 3, 4, 5]
True
```

```
>>> [1, 2, 3, 4, 5] != [1, 2, 3, 4, 5]
False
```

```
>>> [1, 2, 3, -4, 5] == [1, 2, 3, 4, 5]
False
```

```
>>> [1, 2, 3, -4, 5] != [1, 2, 3, 4, 5]
True
```

```
>>> [1, 2, 3, 4, 5] ==  [1, 2, 3, 4]
False
```

```
>>> ["bla", "bli", "blo"] == ['bla', 'bli', 'blo']
True
```

```
>>> ['bla', 'Bli', 'blo' ] == ['bla', 'bli', 'blo']
False
```

Dans ce dernier exemple, les deux listes sont inégales car il n'est pas vrai que `'Bli' == 'bli'`.

Important : retenons que l'on ne compare que des listes contenant des éléments du même type. Comparer deux listes de types différents (par exemple une liste de type `List[int]` avec une autre de type `List[bool]`) ne veut rien dire puisque l'on sait déjà

par le type que les listes sont différentes. Python nous autorise tout de même à écrire ce qui ne veut rien dire, restons donc vigilant !

Remarque : les comparateurs d'ordre `<`, `<=`, `>` et `>=` décrits pour les chaînes de caractères sont également disponibles pour les séquences en général, donc les listes.

6.1.4 Construction par concaténation

Comme pour les chaînes de caractères, on peut concaténer plusieurs listes entre elles en utilisant l'opérateur `+` dont la signature, pour les listes, est la suivante :

```
List[T] * List[T] -> List[T]
```

Attention : pour être sûr que la liste résultat est bien du type `List[T]`, on ne peut concaténer que des listes dont les éléments sont du même type `T`.

Ainsi, on peut concaténer les listes `[1, 2, 3, 4, 5]` et `[6, 7, 8]` toutes deux de type `List[int]`.

```
>>> [1, 2, 3, 4, 5] + [6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

et on obtient bien une liste concaténée du même type `List[int]`.

En revanche, on ne peut pas concaténer les listes `[1, 2, 3, 4, 5]` et `['chat', 'ours', 'pomme']`, par exemple, puisque la première est de type `List[int]` et la seconde du type `List[str]`. En effet, si on effectuait cette dernière concaténation, il ne serait pas possible de définir le type de la liste résultat : il s'agirait d'une liste hétérogène qu'il serait difficile de manipuler.

La terminologie est similaire aux chaînes : les listes `[1, 2, 3, 4, 5]` et `[6, 7, 8]` sont dites *sous-listes* de la liste concaténée `[1, 2, 3, 4, 5, 6, 7, 8]`

Comme pour les chaînes la concaténation sur les listes possèdent un *élément neutre* : la liste vide `[]`.

Par exemple :

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

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

```
>>> [] + [True, False]
[True, False]
```

Nous remarquons que la liste vide est compatible pour tout type de liste, ce qui n'est pas étonnant puisqu'elle ne contient aucun élément, aucun risque de construire une liste hétérogène.

De façon générale, pour toute liste `l` on a les égalités suivantes :

```
l + [] == [] + l == l
```

6.1.5 Construction par ajout en fin de liste

Considérons la chaîne de caractères `'Les chevau'`. Pour corriger notre grossière erreur de français, nous pouvons créer une chaîne en ajoutant un `'x'` à la fin de notre chaîne erronée, de la façon suivante :

```
>>> 'Les chevau' + 'x'
'Les chevaux'
```

On peut faire de même pour les listes en concaténant une liste avec une liste de un seul élément.

Par exemple :

```
>>> [1, 2, 3, 4, 5] + [6]
[1, 2, 3, 4, 5, 6]
```

Il faut savoir, cependant, que l'ajout en fin de liste de cette façon n'est pas du tout efficace : il faut reconstruire intégralement la liste résultat. Ainsi pour ajouter l'élément 6 en fin de liste, nous avons reconstruit une *nouvelle* liste de 6 éléments.

En pratique, on utilise donc une autre solution qui consiste à invoquer ce que l'on appelle une **méthode** - dans le cas présent la méthode `append` - qui ajoute directement un élément dans une liste sans effectuer de reconstruction.

La syntaxe utilisée est la suivante :

```
<liste>.append(<element>)
```

où `<liste>` est une liste de type `list[T]` et `<élément>` une expression de type `T`.

Important : la méthode `append` est spécifique aux listes et ne s'applique pas aux autres séquences - c'est le critère qui différencie les fonctions (comme `len` utilisable sur tous les types de séquences) - et les méthodes. D'autre part, cette opération ne retourne rien (elle agit comme une *instruction* mais retourne effectivement `None`) et modifie directement la liste.

Pour comprendre ces subtilités, considérons la variable suivante :

```
l : List[int]
l = [1, 2, 3, 4, 5]
```

On a placé dans la variable `l` la liste `[1 ,2, 3, 4, 5]`.

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

On peut désormais effectuer par exemple des concaténations en utilisant la variable `l` :

```
>>> l + [6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8]
```

Mais il est important de se rappeler que la concaténation construit une nouvelle liste résultat, laissant la liste référencée par la variable `l` inchangée.

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

En revanche, la méthode `append` d'ajout en fin opère *sans* reconstruction d'une liste résultat et modifie directement la liste.

```
>>> l.append(6)
```

Comme il s'agit d'une instruction (qui retourne `None`), l'interprète Python ne produit aucun affichage ici (de façon similaire à une affectation).

Mais si on demande maintenant la valeur de `l` alors celle-ci a été modifiée.

```
>>> l
[1, 2, 3, 4, 5, 6]
```

Puisque l'on peut ainsi les modifier directement sans les reconstruire, on dit des listes qu'elles sont **mutables**. En comparaison, les chaînes de caractères sont dites **immutables** car on ne peut pas les modifier sans les reconstruire.

A titre d'illustration, nous allons définir une fonction `liste_pairs` permettant de construire la liste des entiers pairs dans l'intervalle $[1, n]$.

Par exemple :

```
>>> liste_pairs(3)
[2]
```

```
>>> liste_pairs(5)
[2, 4]
```

```
>>> liste_pairs(10)
[2, 4, 6, 8, 10]
```

```
>>> liste_pairs(11)
[2, 4, 6, 8, 10]
```

Voici une solution pour ce problème :

```
from typing import List  # on rappelle que cet import est nécessaire

def liste_pairs(n : int) -> List[int]:
    """Précondition : 1 <= n
    Retourne la liste des entiers pairs dans l'intervalle [1,n].
    """
    # la liste résultat, initialement vide
    l : List[int] = []

    i : int # Entier courant
    for i in range(1, n + 1):
```

```
        if i % 2 == 0:
            l.append(i)  # ajout en fin, directement dans L
        # sinon : ne rien faire

    return l
```

```
# Jeu de tests
assert liste_pairs(3) == [2]
assert liste_pairs(5) == [2, 4]
assert liste_pairs(10) == [2, 4, 6, 8, 10]
assert liste_pairs(11) == [2, 4, 6, 8, 10]
assert liste_pairs(20) == [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Remarque : par convention, nous utiliserons des identifiants commençant par un `l` pour les listes, le plus souvent `l`, `l1`, `l2`, etc.

Effectuons la simulation de `liste_pairs(5)`, donc pour `n=5` :

Tour de boucle	variable `n`	variable `l`
entrée	-	`[]`
1er	1	`[]`
2e	2	`[2]`
3e	3	`[2]`
4e	4	`[2, 4]`
5e	5	`[2, 4]`
sortie	-	`[2, 4]`

6.1.6 Accès aux éléments

Comme pour les chaînes de caractères, les indices des éléments d'une liste `l` de type `List[T]` sont numérotés de 0, pour son premier élément, à `len(l)-1`, pour son dernier élément.

Pour $0 \leq i \leq$ `len(l)-1`, l'expression `l[i]` représente l'élément de `l` de type `T` et situé à l'indice `i`. On dit que `l[i]` est l'élément de `l` d'indice i.

Dans la liste `l=['aa','bb','cc','dd','ee']` de type `List[str]` et de longueur 5 :

- l'élément `'aa'` est à l'indice 0 donc `l[0]` vaut `'aa'`,
- l'élément `'bb'` est à l'indice 1 donc `l[1]` vaut `'bb'`,
- l'élément `'cc'` est à l'indice 2 donc `l[2]` vaut `'cc'`,
- l'élément `'dd'` est à l'indice 3 donc `l[3]` vaut `'dd'`,
- l'élément `'ee'` est à l'indice 4 donc `l[4]` vaut `'ee'`.

On peut synthétiser ces informations par un **table des indices**, de la façon suivante :

Elément	'aa'	'bb'	'cc'	'dd'	'ee'
Indice	0	1	2	3	4

Remarque : il existe une différence importante entre l'accès à un caractère dans une chaîne et l'accès à un élément d'une liste. Si par exemple `s` est une chaîne de type `str` et `l` une liste de type `List[T]` (par exemple `List[int]`), toutes deux avec au moins $i+1$ éléments, alors :

— `s[i]` est de type `str` donc du même type que la chaîne `s`
— `l[i]` est de type `T` donc d'un type *différent* de la liste `L`

Ceci est dû au fait que Python n'introduit pas de type *caractère* spécifique, contraiement à d'autres langages de programmation comme Java par exemple.

Les éléments sont également accessibles par des indices négatifs, ainsi :

Elément	'aa'	'bb'	'cc'	'dd'	'ee'
Indice positif	0	1	2	3	4
Indice négatif	-5	-4	-3	-2	-1

Exercice : donner le résultat des expressions suivantes :

— `l[-3]`
— `l[-1]`
— `l[-6]`

6.2 Découpages de listes

L'opération de découpage a été vue pour les chaînes et s'applique à toutes les séquences, donc également aux listes.

Le découpage d'une liste `l` de type `List[T]` permet de construire une nouvelle liste également de type `List[T]` et composée des éléments de `l` situés entre deux indices.

6.2.1 Découpage simple

Pour $0 \leq i \leq$ `len(l)-1` et $1 \leq j \leq$ `len(l)`, l'expression :

`l[i:j]`

renvoie la liste des éléments de `l` situés entre les indices i inclus et j non-inclus (ou entre i et $j-1$ tous deux inclus).

Les nombreux exemples de découpage qui suivent se feront sur une liste de chaînes de caractères référencée par la variable `lcomptine` :

```
lcomptine : List[str]
lcomptine = ['am', 'stram', 'gram', 'pic', 'pic', 'col', 'gram']
```

La représentation avec indices de cette liste est la suivante :

Elément	'am'	'stram'	'gram'	'pic'	'pic'	'col'	'gram'
Indice	0	1	2	3	4	5	6

```
>>> lcomptine[1:3]
['stram', 'gram']
```

```
>>> lcomptine[3:4]
['pic']
```

Les expressions de découpage reconstruisent les listes. La liste de départ référencée par la variable `lcomptine` est donc restée inchangée.

```
>>> lcomptine
['am', 'stram', 'gram', 'pic', 'pic', 'col', 'gram']
```

Remarque : si $i \geq j$ alors `l[i:j]` renvoie la liste vide.

```
>>> lcomptine[3:3]
[]
```

```
>>> lcomptine[5:3]
[]
```

Remarque : pour une liste `l` de type `List[T]` ne pas confondre

- `l[i]` qui renvoie un élément de type `T`
- `l[i:i+1]` qui renvoie une liste de type `List[T]` de longueur 1.

Par exemple :

```
>>> lcomptine[3]
'pic'
```

```
>>> lcomptine[3:4]
['pic']
```

6.2.2 Premiers et derniers éléments

L'expression `l[:j]` renvoie la liste des éléments situés avant la position $j-1$, y compris l'élément en position $j-1$. C'est donc l'équivalent de `l[0:j]`.

De façon complémentaire, l'expression `l[i:]` renvoie la liste des éléments situés après la position i, y compris l'élément en position i. C'est donc l'équivalent de `l[i:len(l)]`.

```
>>> lcomptine[:4]
['am', 'stram', 'gram', 'pic']
```

```
>>> lcomptine[0:4]
['am', 'stram', 'gram', 'pic']
```

```
>>> lcomptine[3:]
['pic', 'pic', 'col', 'gram']
```

```
>>> lcomptine[3:len(Comptine)]
['pic', 'pic', 'col', 'gram']
```

Cas particulier : reconstruction de la liste de départ.

On peut bien sûr reconstruire complètement la liste de départ.

```
>>> lcomptine[0:len(lcomptine)]   # autrement dit:  lcomptine[0:7]
['am', 'stram', 'gram', 'pic', 'pic', 'col', 'gram']
```

En fait, il existe un raccourci d'écriture pour ce dernier cas :

Pour reconstruire une liste `l` à l'identique on écrit `l[:]` qui est équivalent à `l[0:len(l)]`.

Ainsi :

```
>>> lcomptine[:]
['am', 'stram', 'gram', 'pic', 'pic', 'col', 'gram']
```

Cette opération est en fait plus utile qu'il n'y parait. Puisque `append` modifie directement les listes en mémoire, la possibilité de copier une liste permet est parfois nécessaire. Nous donnerons bien sûr quelques exemples par la suite.

6.2.3 Découpage selon des entiers quelconques

Pour tous entiers naturels i et j et pour tout entier relatif k :

- si $i >$ `len(l)-1` ou si $j = 0$ alors `l[i:j]` renvoie la liste vide ;
- si $i \leq$ `len(l)-1` et si $j >$ `len(l)` alors `l[i:j]` renvoie la même liste que `l[i:]` ;
- si $j \leq$ `len(l)-1` alors `l[i:-j]` renvoie la même liste que `l[i:len(l)-j]` ;
- si $j >$ `len(l)-1` alors `l[i:-j]` renvoie la liste vide ;
- si $i \leq$ `len(l)` alors `l[-i:k]` renvoie la même liste que `l[len(l)-i:k]` ;
- si $i >$ `len(l)` alors `l[-i:k]` renvoie la même liste que `l[0:k]`.

La représentation avec indices négatifs de notre `lcomptine` est la suivante :

Elément	'am'	'stram'	'gram'	'pic'	'pic'	'col'	'gram'
Indices positifs	0	1	2	3	4	5	6
Indices négatifs	-7	-6	-5	-4	-3	-2	-1

Par exemple :

— `lcomptine[-4:-1]` renvoie la même liste que `lcomptine[3:-1]` et donc que `lcomptine[3:6]`.

```
>>> lcomptine[-4:-1]
['pic', 'pic', 'col']
```

```
>>> lcomptine[-4:-1] == lcomptine[3:-1] == lcomptine[3:6]
True
```

— `lcomptine[-6:-2]` renvoie la même liste que `lcomptine[1:-2]` et donc que `lcomptine[1:5]`.

```
>>> lcomptine[-6:-2]
['stram', 'gram', 'pic', 'pic']
```

```
>>> lcomptine[-6:-2] == lcomptine[1:-2] == lcomptine[1:5]
True
```

Exercice : selon les mêmes principes, donner le résultat de l'évaluation des expressions suivantes :

— `lcomptine[-5:-3]`
— `lcomptine[-4:-4]`

6.2.4 Découpage avec un pas positif

Le découpage avec pas positif permet de «sauter» d'élément en élément dans la liste initiale avec un certain pas.

Pour $0 \leq i \leq j$ et k entier naturel non nul, `l[i:j:k]` renvoie la liste des éléments de `l` situés aux positions i, $i+k$, $i+2k\ldots$ entre les positions i et $j-1$ comprises.

```
>>> lcomptine[1:5:2]
['stram', 'pic']
```

Cas particulier : `l[i:j:1]` renvoie la même liste que `l[i:j]`

```
>>> lcomptine[2:6:1]
['gram', 'pic', 'pic', 'col']
```

Exercice : en utilisant un découpage avec pas positif :

— retourner la liste des entiers pairs à partir de la liste `[1, 2, 3, 4, 5, 6, 7, 8, 9]`.
— même question pour les impairs.

6.2.5 Découpage avec un pas négatif

On peut aussi inverser le sens du découpage en utilisant un pas négatif.

Pour $0 \leq i \leq j$ et k entier naturel non nul, `l[j:i:-k]` renvoie la liste des éléments de `l` situés aux positions j, $j-k$, $j-2k\ldots$ entre les positions $i+1$ et j comprises.

```
>>> lcomptine[5:2:-2]
['col', 'pic']
```

Cas particuliers :

- `l[j:i:-1]` renvoie la liste `l[i+1:j+1]` inversée ;
- `l[j::-1]` renvoine la liste `l[0:j+1]` inversée ;
- `l[::-1]` renvoie la liste `l` inversée ;

```
>>> lcomptine[6:2:-1]
['gram', 'col', 'pic', 'pic']
```

```
>>> lcomptine[5:0:-1]
['col', 'pic', 'pic', 'gram', 'stram']
```

```
>>> lcomptine[5::-1]
['col', 'pic', 'pic', 'gram', 'stram', 'am']
```

```
>>> lcomptine[::-1]
['gram', 'col', 'pic', 'pic', 'gram', 'stram', 'am']
```

Exercice : Donner des découpages de `lcomptine` avec pas négatif pour obtenir les résultats suivants :

- `['col', 'pic', 'stram']`
- `['gram', 'pic', 'gram', 'am']`
- `['pic', 'pic', 'gram', 'stram']`

Remarque : toutes ces opérations de découpe sont disponibles, au-delà des listes, à tous les types de séquence, notamment les chaînes de caractères.

6.3 Problèmes sur les listes.

Nous allons utiliser la même classification de problèmes sur les listes que celle proposée pour les chaînes de caractères:

- réductions de listes
- transformations de listes,
- filtrages de listes,
- autres problèmes sur les listes (en général plus complexes) qui n'entrent pas dans les catégories précédentes.

6.3.1 Réductions de listes

Les problèmes de réduction consistent à analyser un par un les éléments d'une liste pour en déduire une information d'un type plus «simple» : `int`, `float`, `bool`, `str`, etc.

Exemple 1 : somme des éléments d'une liste

Un grand classique des réductions de liste concerne le calcul de la somme des éléments d'une liste.

La spécification de la fonction `somme_liste` est la suivante :

```
def somme_liste(l : List[float]) -> float:
    """Retourne la somme des éléments de la liste L.
    """
```

Par exemple :

```
>>> somme_liste([1, 2, 3, 4, 5])
15
```

```
>>> somme_liste([1.1, 3.3, 5.5])
9.9
```

```
>>> somme_liste([])
0
```

Une définition simple de cette fonction est proposée ci-dessous.

```
def somme_liste(l : List[float]) -> float:
    """Retourne la somme des éléments de la liste L.
    """
    # Somme cumulée des éléments
    s : float = 0

    n : float # Elément courant
    for n in l:
        s = s + n

    return s
```

```
# Jeu de tests
assert somme_liste([1, 2, 3, 4, 5]) == 15
assert somme_liste([1.1, 3.3, 5.5]) == 9.9
assert somme_liste([]) == 0
```

Puisque c'est notre première itération sur une liste, effectuons la simulation de `somme_liste([1, 2, 3, 4, 5])` donc pour `l=[1, 2, 3, 4, 5]` :

Tour de boucle	variable `n`	variable `s`
entrée	-	0
1er	1	1
2e	2	3
3e	3	6
4e	4	10
5e	5	15

Tour de boucle	variable `n`	variable `s`
sortie	-	15

Exemple 2 : longueur de liste

Considérons un second exemple de réduction avec la fonction `longueur` qui calcule le nombre d'éléments d'une liste.

```
from typing import List, TypeVar
T = TypeVar('T')  # déclaration d'une variable de type

def longueur(l : List[T]) -> int:
    """Retourne la longueur de la liste L."""

    # Longueur à retourner
    long : int = 0  # initialement à zéro

    e : T  # Elément courant
    for e in l:
        long = long + 1

    return long
```

Il s'agit ici d'une *réduction* vers le type `int`, ce qui correspond à la signature suivante :

```
list[T] -> int
```

Le `T` dans cette signature (et également dans les déclarations de variable comme pour la variable `e` ci-dessus) joue un rôle très important. Elle indique que la fonction `longueur` est *générique* (on dit aussi *polymorphe*) : elle accepte en entrée une liste de type `List[T]` pour n'importe quel `T`. Le calcul de la longueur d'une liste est en effet indépendant de la nature des éléments de la liste de départ, seul leur nombre compte. Nous avons expliqué précédemment que le `T` ici jouait le rôle d'une *variable de type*. Pour introduire une telle variable, nous devons importer le symbole `TypeVar` du module `typing` et nous devons déclarer la variable de type de la façon suivante :

```
T = TypeVar('T')
```

Par convention, nous déclarons les variables de type en début de programme, juste après les imports du module `typing`.

Important : la variable de type `T` doit être remplacée par un type spécifique lorsque l'on *applique* la fonction.

Par exemple :

```
>>> longueur([1, 2, 3, 4, 5])
5
```

Ici, on applique la fonction sur une liste de type `List[int]` donc la variable `T` est remplacée par l'expression de type `int`, autrement dit on applique `T=int`. Bien sûr, on peut l'appliquer sur une liste d'un type différent, par exemple `list[str]` avec `T=str`, `list[list[float]]` avec `T=list[float]`, etc.

```
>>> longueur(['bla', 'bla', 'bla'])  # List[T] avec T=str
3
```

```
>>> longueur([[2.3, 2.4], [3.3, 3.1, 4.3]])  # List[T] avec T=List[float]
2
```

Complétons notre définition avec un jeu de tests.

```
# Jeu de tests
assert longueur([1, 2, 3, 4, 5]) == 5
assert longueur(['bla', 'bla', 'bla']) == 3
assert longueur([]) == 0
```

Remarque : en pratique on utilisera bien sûr comme pour les chaînes de caractères la fonction prédéfinie `len` qui est beaucoup plus efficace puisqu'elle ne nécessite pas de recompter le nombre d'éléments de la liste.

6.3.2 Transformations de listes : le schéma `map`

Le principe du schéma `map` de transformation de liste est de produire une liste résultat consistant en l'application d'une fonction unaire (à un seul argument) à chaque élément de la liste de départ.

Plus formellement, soit `l` une liste de type `List[T]` de longueur n :

`[` e_0 `,` e_1 `, ... ,` e_{n-1} `]`

ainsi qu'une fonction `f` de signature `T -> U`.

L'objectif est de construire la liste :

`[` $f(e_0)$ `,` $f(e_1)$ `, ... ,` $f(e_{n-1})$ `]` de type `List[U]` et de longueur n également.

Une transformation possède dans le cas général une signature de la forme :

`list[T] -> list[U]` que l'on peut lire comme :

> Une transformation d'une liste de `T` vers une liste de `U`.

Exemple 1 : liste des carrés

Pour illustrer les transformations de listes, considérons la spécification suivante :

```
def liste_carres(l : List[float]) -> List[float]:
    """retourne la liste des éléments de L élevés au carré.
    """
```

Ici, avec la signature `List[float] -> List[float]` on a donc :

> une transformation d'une liste de nombres vers une liste de nombres.

Par rapport au schéma général on a donc ici, en quelque sorte : `T = U = float`.

Voici quelques exemples :

```
>>> liste_carres([1, 2, 3, 4, 5])
[1, 4, 9, 16, 25]
```

```
>>> liste_carres([2, 4, 8, 16])
[4, 16, 64, 256]
```

```
>>> liste_carres([])
[]
```

Avant de donner une définition complète de la fonction `liste_carres`, explicitons la fonction f du schéma général pour le cas qui nous intéresse ici. Il s'agit bien sûr de définir une fonction d'élévation d'un entier au carré, ce qui est trivial.

```
def carre(x : float) -> float:
    """Retourne le nombre x élevé au carré.
    """
    return x * x

# Jeu de tests
assert carre(0) == 0
assert carre(1) == 1
assert carre(2) == 4
assert carre(3.2) == 3.2 * 3.2
assert carre(16) == 256
```

Nous pouvons maintenant définir proprement notre fonction `liste_carres`.

```
def liste_carres(l : List[float]) -> List[float]:
    """Retourne la liste des éléments de L élevés au carré.
    """
    # liste résultat
    lr : List[float] = []  # initialement vide

    x : float # élément courant
    for x in l:
        lr.append(carre(x)) # ajoute le carré en fin de résultat

    return lr
```

```
# Jeu de tests
assert liste_carres([1, 2, 3, 4, 5]) == [1, 4, 9, 16, 25]
assert liste_carres([2, 4, 8, 16]) == [4, 16, 64, 256]
assert liste_carres([]) == []
```

En guise d'illustration, effectuons la simulation de `liste_carres([1, 2, 3, 4, 5])` :

Tour de boucle	variable `x`	variable `lr`
entrée	-	`[]`
1er	1	`[1]`
2e	2	`[1, 4]`
3e	3	`[1, 4, 9]`
4e	4	`[1, 4, 9, 16]`
5e	5	`[1, 4, 9, 16, 25]`
sortie	-	`[1, 4, 9, 16, 25]`

Nous constatons qu'une transformation consiste en fait en une reconstruction via la méthode `append` d'une liste transformée (référencée par la variable `lr` dans notre définition ci-dessus) à partir de la liste initiale (paramètre `l` ci-dessus).

Remarque : dans cet exercice nous avons défini une fonction `carre` explicite pour bien illustrer le schéma de transformation, mais en pratique une fonction aussi simple ne nécessite pas vraiment de définition dédiée, l'expression `x * x` est déjà bien explicite.

Exemple 2 : liste des longueurs de chaînes

Pour notre second exemple, l'objectif est d'effectuer une transformation d'une liste de chaînes de caractères vers une liste d'entiers correspondants aux longueurs de ces mêmes chaînes. Il s'agit donc d'une transformation de signature :

```
List[str] -> List[int]
```

La spécification complète de la fonction est la suivante :

```
def liste_longueurs(l : List[str]) -> List[int]:
    """Retourne la liste des longueurs des chaînes éléments de L.
    """
```

Par rapport au schéma général de transformation de `List[T]` vers `List[U]]` on a ici `T=str` et `U=int`.

Par exemple :

```
>>> liste_longueurs(['e', 'ee', 'eee', 'eeee'])
[1, 2, 3, 4]
```

```
>>> liste_longueurs(['un', 'deux', 'trois', 'quatre'])
[2, 4, 5, 6]
```

```
>>> liste_longueurs([])
[]
```

La fonction f du schéma général doit ici être de signature `str -> int` et elle correspond d'après le problème à la fonction `len` qui retourne la longueur d'une chaîne de caractères

(ou plus généralement d'une séquence).

On obtient donc la définition suivante :

```
def liste_longueurs(l : List[str]) -> List[int]:
    """Retourne la liste des longueurs des chaînes éléments de L.
    """
    # list résultat
    lr : List[int] = []  # initialement vide

    s : str # Elément courant
    for s in l:
        lr.append(len(s)) # ajoute la longueur en fin de résultat


    return lr
```

```
# Jeu de tests
assert liste_longueurs(['e', 'ee', 'eee', 'eeee']) \
       == [1, 2, 3, 4]
assert liste_longueurs(['un', 'deux', 'trois', 'quatre']) \
       == [2, 4, 5, 6]
assert liste_longueurs([]) == []
```

6.3.3 Filtrages de listes : le schéma `filter`

Le principe du schéma `filter` de transformation de liste est de produire une liste résultat consistant à filtrer les éléments d'une liste initiale en fonction d'un *prédicat unaire* (fonction à un argument et qui retourne un booléen). On obtient donc la liste de départ avec certains éléments supprimés, ceux pour lesquels le prédicat est faux. La liste filtrée résultat est donc de longueur potentiellement plus courte que la liste de départ.

Plus formellement, soit `l` une liste de type `List[T]` de longueur n :

`[` a_0 , a_1 , ... , a_{n-1} `]`

ainsi qu'un prédicat `p` de signature `T -> bool`.

L'objectif est de construire la liste également de type `List[T]` :

`[` b_0 , b_1 , ... , b_{m-1} `]` de longueur $m \leq n$ avec :

- pour chaque b_j il existe un unique a_i tels que $b_j = a_i$ et $p(a_i)$ vaut `True` avec $j \leq i$
- et si $b_j = a_i$ et $b'_j = a'_i$ avec $i < i'$ alors $j < j'$ (préservation de l'ordre).

... ouf ! mathématiquement c'est assez difficile à exprimer ... mais retenons simplement :

- tous les b_k éléments de la liste résultat sont des a_i de la liste de départ tels que $p(a_i)$ vaut `True`
- l'ordre séquentiel des éléments dans la liste de départ est préservé

Un filtrage possède donc dans le cas général une signature de la forme :

`list[T] -> list[T]`

que l'on peut lire comme :

> un filtrage d'une liste de `T`.

Exemple 1 : liste des entiers pairs

Pour notre premier exemple de filtrage, considérons la spécification suivante :

```
def liste_pairs(l : List[int]) -> List[int]:
    """Retourne la liste des entiers pairs éléments de L.
    """
```

Ici il s'agit d'un filtrage d'une liste de `int`. Pour faire apparaître explicitement le prédicat p du schéma général de filtrage, on définit ci-dessous un prédicat permettant de tester la parité d'un entier.

```
def est_pair(n : int) -> bool:
    """Renvoie True si l'entier n est pair, False sinon.
    """
    return n % 2 == 0
```

```
# Jeu de test
assert est_pair(0) == True
assert est_pair(1) == False
assert est_pair(2) == True
assert est_pair(92) == True
assert est_pair(37) == False
```

Ce prédicat est bien de signature `int -> bool` permettant un filtrage d'entiers.

On peut donc compléter la définition de la fonction `liste_pairs` :

```
def liste_pairs(l : List[int]) -> List[int]:
    """Retourne la liste des entiers pairs éléments de L.
    """
    # Liste filtrée en résultat
    lr : list[int] = []  # initialement vide

    n : int # Elément courant
    for n in l:
        if est_pair(n):
            lr.append(n)
        # sinon on ne fait rien

    return lr
```

```
# Jeu de tests
assert liste_pairs([4, 7, 10, 11, 14]) == [4, 10, 14]
assert liste_pairs([232, 111, 424, 92]) == [232, 424, 92]
assert liste_pairs([51, 37, 5]) == []
assert liste_pairs([]) == []
```

En guise d'illustration du filtrage effectuons la simulation de `liste_pairs([4, 7, 10, 11, 14])` c'est-à-dire pour `l=[4, 7, 10, 11, 14]` :

Tour de boucle	variable `n`	variable `lr`
entrée	-	`[]`
1er	4	`[4]`
2e	7	`[4]`
3e	10	`[4, 10]`
4e	11	`[4, 10]`
5e	14	`[4, 10, 14]`
sortie	-	`[4, 10, 14]`

Exercice : définir la fonction `liste_impairs` retournant, à partir d'une liste `l` d'entiers naturels en paramètre, la liste des éléments impairs de `l`.

Exemple 2 : liste des supérieurs à un nombre

Comme nous avons un peu de pratique maintenant, abordons un problème légèrement plus complexe. Considérons la définition de fonction suivante :

```
def liste_superieurs(l : List[float], x : float) -> List[float]:
    """Renvoie la liste des nombres éléments de l supérieurs
    au nombre x."""

    # Liste résultat
    lr = []

    y : Number # Nombre courant
    for y in l:
        if y > x:
            lr.append(y)
        # sinon ne rien faire

    return lr
```

```
# Jeu de tests
assert liste_superieurs([11, 27, 8, 44, 39, 26], 26) == [27, 44, 39]
assert liste_superieurs([11, 26, 8, 4, 9], 26) == []
assert liste_superieurs([11.3, 26.4, 8.9, 4.12, 9.7], 4.11) \
```

```
       == [11.3, 26.4, 8.9, 4.12, 9.7]
assert liste_superieurs([], 0) == []
```

Exercice : effectuer la simulation de `liste_superieurs([11, 27, 8, 44, 39, 26], 26)`.

Avec la définition complète et le jeu de tests ci-dessus, nous comprenons bien le rôle de la fonction : elle filtre dans la liste initiale `l` les nombres qui sont supérieurs strictement au paramètre `x` de type `float`.

On constate sur le jeu de tests que la fonction s'applique tant aux listes d'entiers qu'aux listes de flottants, il s'agit donc d'un filtrage de nombres.

Complément : définitions internes

On aimerait, pour illustrer le fait que `liste_superieurs` est bien une fonction de filtrage, déterminer plus précisément notre prédicat unaire p du schéma général sur cette fonction.

Le problème ici est que la comparaison $y > x$ effectuée dans le corps de la boucle nécessite deux arguments et non un seul. On constate cependant que la valeur du paramètre `x` est toujours la même dans tout le corps de la fonction, c'est une propriété essentielle des paramètres de fonction.

L'idée est donc de définir le prédicat unaire *à l'intérieur* de la fonction `liste_superieurs` : cela s'appelle une **définition interne**.

Voici une nouvelle définition de `liste_superieurs`, plus proche du schéma général de filtrage :

```
def liste_superieurs(l : List[float], x : float) -> List[float]:
    """ ... cf. ci-dessus ... """

    # Voici la définition interne :
    def superieur_a_x(z : float) -> bool:
        return z > x

    # et maintenant le reste du corps de la fonction principale
    # (inchangé)

    # Liste résultat
    lr = []

    y : Number # Nombre courant
    for y in l:
        if y > x:
            lr.append(y)
        # sinon ne rien faire
```

```
    return lr
```

Remarque : en pratique la version avec définition interne est un peu moins concise et on préfèrera la première. Mais elle permet de bien mettre en lumière le schéma de filtrage. Elle illustre de plus une capacité intéressante du langage Python : la possibilité de définir et manipuler des fonctions à l'intérieur des fonctions.

6.3.4 Problèmes plus complexes

Si de nombreux problèmes correspondent à des réductions, transformations, filtrages ou combinaisons de ces derniers, il existe bien sûr des problèmes (en général plus complexes) qui sortent de cette classification.

Exemple 1 : liste des sommes cumulées

Le calcul de la liste des sommes cumulées est un exemple intéressant car il ne peut être résolu en considérant les éléments de la liste indépendemment des autres.

La spécification du problème est la suivante :

```
def liste_sommes_cumulees(l : List[float]) -> List[float]:
    """Retourne la liste des sommes cumulées de l.
    """
```

Quelques exemples permettent de comprendre le problème posé ici :

```
>>> liste_sommes_cumulees([])
[]
```

```
>>> liste_sommes_cumulees([42])
[42]
```

```
>>> liste_sommes_cumulees([1, 2, 3, 4, 5])
[1, 3, 6, 10, 15]
```

```
>>> liste_sommes_cumulees([10, 10, 10, 10, 10, 10, 10, 10, 10, 10])
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
```

Ce problème n'est clairement pas un problème de réduction, et malgré la signature :

```
List[float] -> List[float]
```

il ne s'agit pas non plus d'un problème de transformation - puisque l'on n'applique pas la même fonction à chaque élément de la liste de départ - ou un problème de filtrage - car les éléments de la liste résultat ne sont pas (forcément) présents dans la liste de départ. Mais il est important de savoir classer notre problème et donc de le confronter à notre classification.

Dans ce cas précis, il n'y a pas non plus de difficulté insurmontable pour résoudre ce problème.

Une définition possible est la suivante :

```
def liste_sommes_cumulees(l : List[float]) -> List[float]:
    """Retourne la liste des sommes cumulées de l.
    """

    # Liste des sommes partielles
    ls = []

    # Somme cumulée
    s = 0   #  somme cumulée

    n : float # Elément courant)
    for n in l:
        s = s + n
        ls.append(s)

    return ls
```

```
# Jeu de tests
assert liste_sommes_cumulees([1, 2, 3, 4, 5]) == [1, 3, 6, 10, 15]
assert liste_sommes_cumulees([10, 10, 10, 10, 10, 10, 10, 10, 10, 10]) \
       == [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
assert liste_sommes_cumulees([]) == []
assert liste_sommes_cumulees([42]) == [42]
```

Ce qui rend cette définition légèrement plus complexe que les transformations et filtrages est qu'il existe une dépendance entre le travail effectué sur l'élément courant et celui effectué sur les éléments précédents. Ici, la dépendance est la somme cumulée des éléments précédents donc la valeur référencée par la variable `s`.

Exercice : proposer une définition de `liste_sommes_cumulees` qui ne nécessite pas la variable locale `s` pour la somme cumulée. Quelle définition trouvez-vous la plus simple à comprendre ?

Exemple 2 : interclassement de deux listes

Jusqu'à présent, nous avons essentiellement utilisé le principe d'itération sur les listes avec une boucle `for` qui est effectivement utilisable la plupart du temps. Cependant il faut parfois revenir à des boucles moins concises mais plus facilement contrôlables avec des variables locales et des boucles `while`.

Considérons le problème de l'interclassement de deux listes.

La spécification proposée pour ce problème est la suivante :

```
def interclassement(l1 : List[T], L2 : List[T]) -> List[T]:
    """Précondition : les listes l1 et l2 sont triées
    renvoie la liste triée obtenue en interclassant de l1 et l2."""
```

Un point important dans la spécification est que l'on considère en précondition que les deux listes `l1` et `l2` soient triées (pour l'ordre du comparateur $<$ sur les éléments) et que leurs éléments sont du même type `T`. Il faut faire attention de bien respecter ces contraintes dans les appels à `interclassement`. En revanche, les longueurs de listes ne sont pas contraintes : `l1` et `l2` peuvent être de longueurs différentes.

Par exemple :

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

```
>>> interclassement([11, 17, 18, 19, 25], [2, 3, 7, 18, 20])
[2, 3, 7, 11, 17, 18, 18, 19, 20, 25]
```

```
>>> interclassement(['ab', 'ac', 'ba'], ['aaa', 'aca', 'acb', 'baa' ])
['aaa', 'ab', 'ac', 'aca', 'acb', 'ba', 'baa']
```

```
>>> interclassement(['aa', 'bb', 'cc', 'dd'], ['aa', 'bb', 'cc', 'dd'])
['aa', 'aa', 'bb', 'bb', 'cc', 'cc', 'dd', 'dd']
```

```
>>> interclassement([], ['aa', 'bb', 'cc'])
['aa', 'bb', 'cc']
```

```
>>> interclassement(['aa', 'bb', 'cc'], [])
['aa', 'bb', 'cc']
```

On constate dans tous les exemples que les listes de départ sont bien toutes triées, et qu'en résultat on obtient bien une liste interclassée également triée (pour les chaînes, il faut peut-être se rappeler de la relation d'ordre décrite au chapitre 5).

Pour résoudre notre problème ici, il va falloir réfléchir un peu plus que pour les problèmes précédents. En effet, dans un premier temps on doit essayer de catégoriser ce problème comme une réduction, une transformation ou un filtrage. Mais malheureusement ici on peut rapidement se convaincre que le problème est plus complexe. Remarquons qu'il s'agit déjà d'une information importante !

D'un point de vue méthodologique, dans ce genre de situation on n'a pas vraiment d'autre solution que de prendre quelques feuilles de papier, un crayon, une gomme et de réfléchir à une «recette» permettant de résoudre le problème d'interclassement en s'inspirant des exemples ci-dessus.

Après quelques temps de réflexion, voici quelques éléments de réponse :

Brouillon :

— il va falloir parcourir "simultanément" les listes `l1` et `l2` et construire au fur et à mesure une liste résultat - disons `lr`

Pour cela, la boucle `for` n'est pas utile (elle ne permet de parcourir qu'une seule liste à la fois). Nous allons donc utiliser un parcours par indices grâce à deux variables : l'une -

disons `i1` permettant de parcourir les indices de `l1` et l'autre - disons `i2` les indices de `l2`. Initialement `i1` et `i2` référencent le premier élément de chaque liste : donc l'indice 0.

- à chaque étape du parcours, il faut comparer l'élément de la première liste (donc `l1[i1]`) avec l'élément courant de la seconde liste (donc `l2[i2]`).
  - si `l1[i1]` est inférieur (ou égal) à `l2[i2]` alors on ajoute au résultat `lr` l'élément `l1[i1]` qui est le plus petit. On passe donc à l'élément suivant en incrémentant la variable `i1`.
  - sinon, c'est que `l2[i2]` est inférieur (strictement) à `l1[i1]` et dans ce cas on ajoute au résultat `lr` l'élément `l2[i2]` qui est le plus petit. On passe donc à l'élément suivant en incrémentant la variable `i2`.
- on s'arrête dès que l'on a traité :
  - soit le dernier élément de `l1` et dans ce cas `i1 == len(l1)`
  - soit le dernier élément de `l2` et dans ce cas `i2 == len(l2)`
- mais ce n'est pas terminé car les listes sont potentiellement de longueurs différentes :
  - s'il reste des éléments dans `l1` (donc `i1 < len(l1)`) alors on doit ajouter à `lr` les éléments de `l1[i1:]`
  - sinon il reste peut-être des éléments dans `l2` à partir de l'indice `i2`, on ajoute donc `l2[i2:]` à `lr`
- arrivé à ce stade, la variable `lr` référence bien l'interclassement de `l1` et `l2` et donc on en retourne la valeur.

Cette description informelle mais systématique d'une solution à notre problème s'appelle un **algorithme** (ou une ébauche d'algorithme). C'est ce genre de description que l'on doit produire au brouillon pour un tel problème non-trivial.

Une fois que l'algorithme a été élucidé, nous pouvons en produire une version plus formelle dans le langage Python.

```
def interclassement(l1 : List[T], l2 : List[T]) -> List[T]:
    """Précondition : les listes l1 et l2 sont triées
    renvoie la liste triée obtenue en interclassant de l1 et l2.
    """
    # Indice de parcours de la liste l1
    i1 : int = 0
    # Indice de parcours de la liste l2
    i2 : int = 0

    # Liste résultat
    lr : List[T] = [] # liste résultat de l'interclassement

    while (i1 < len(l1)) and (i2< len(l2)):
        if l1[i1] <= l2[i2]:
            lr.append(l1[i1])  # on choisit l'élément courant de l1
            i1 = i1 + 1
        else:
```

```
            lr.append(l2[i2])  # on choisit l'élément courant de l2
            i2 = i2 + 1

    if i1 < len(l1):  # ne pas oublier les éléments restant dans l1
        lr = lr + l1[i1:]
    else:  # ou bien ceux restant éventuellement dans l2
        lr = lr + l2[i2:]

    return lr
```

Une simulation de la boucle `while` de la fonction semble importante ici. Prenons par exemple l'appel :

`interclassement([11, 17, 18, 19, 25], [2, 3, 7, 18, 20])`

donc pour `l1=[11, 17, 18, 19, 25]` et `l2=[2, 3, 7, 18, 20]`.

Tour de boucle	variable `lr`	variable `i1`	variable `i2`
entrée	`[]`	0	0
1er	`[2]`	0	1
2e	`[2, 3]`	0	2
3e	`[2, 3, 7]`	0	3
4e	`[2, 3, 7, 11]`	1	3
5e	`[2, 3, 7, 11, 17]`	2	3
6e	`[2, 3, 7, 11, 17, 18]`	3	3
7e	`[2, 3, 7, 11, 17, 18, 18]`	3	4
8e	`[2, 3, 7, 11, 17, 18, 19]`	4	4
9e (sortie)	`[2, 3, 7, 11, 17, 18, 19, 20]`	4	5

Après le 9ème tour de boucle, la variable `i2` vaut 5, or `len(l2) == 5` donc on sort de la boucle.

Dans l'alternative qui suit la boucle, la condition `i1 < len(l1)` vaut `True` puisque `i1` vaut `4` en sortie de boucle, et `len(l1)==5`.

On effectue donc l'affectation :

`lr = lr + l1[i1:]`

Et comme `l1[4:]` vaut `[25]` on obtient finalement dans `lr` la valeur :

`[2, 3, 7, 11, 17, 18, 19, 20] + [25] == [2, 3, 7, 11, 17, 18, 19, 20, 25]`

C'est cette dernière valeur (à droite) qui est retournée par la fonction `interclassement`. Il s'agit bien du résultat attendu.

Pour cette fonction, il est clair que notre jeu de tests devient primordial.

```
# Jeu de tests
assert interclassement([1, 3, 5, 7, 9], [2, 4, 6, 8]) \
```

```
       == [1, 2, 3, 4, 5, 6, 7, 8, 9]

assert interclassement([11, 17, 18, 19, 25], [2, 3, 7, 18, 20]) \
       == [2, 3, 7, 11, 17, 18, 18, 19, 20, 25]

assert interclassement(['ab', 'ac', 'ba']
                       , ['aaa', 'aca', 'acb', 'baa' ]) \
       == ['aaa', 'ab', 'ac', 'aca', 'acb', 'ba', 'baa']

assert interclassement(['aa', 'bb', 'cc', 'dd']
                       , ['aa', 'bb', 'cc', 'dd']) \
       == ['aa', 'aa', 'bb', 'bb', 'cc', 'cc', 'dd', 'dd']

assert interclassement([], ['aa', 'bb', 'cc']) == ['aa', 'bb', 'cc']
assert interclassement(['aa', 'bb', 'cc'], []) == ['aa', 'bb', 'cc']
```

Chapitre 7

N-uplets

Dans ce chapitre, nous abordons les structures de n-uplets qui permettent de combiner au sein d'une même donnée des informations de différents types. En mathématiques, cela correspond aux produits cartésiens : les couples, les triplets, etc.

7.1 Construction

Une **expression atomique de n-uplet**, pour $n \geq 2$ de type `Tuple[`T_1`,`T_2`,...,`T_n`]` est de la forme :

(e_1 , e_2 , ... , e_n)

où chacun des e_i est une expression quelconque de type T_i.

Par exemple :

```
>>> (1, True, 'blion')
(1, True, 'blion')
```

Il s'agit d'un n-uplet de 3 éléments, donc un triplet, dont le type est `Tuple[int, bool, str]`.

```
>>> (1, 'deux', 3.0, 'quatre', 5)
(1, 'deux', 3.0, 'quatre', 5)
```

Ici, il s'agit d'un quintuplet de type `Tuple[int, str, float, str, int]`.

```
>>> (0, True, 2.0, 'trois', [4, 5], (6, 7.3))
(0, True, 2.0, 'trois', [4, 5], (6, 7.3))
```

Ce dernier exemple est un peu plus complexe. Il s'agit d'un 6-uplet (donc avec 6 éléménts) avec :

— le premier élément de type `int`

— le second élément de type `bool`
— le troisième élément de type `float`
— le quatrième élément de type `str`
— le cinquième élément de type `List[int]`
— le sixième élément de type `Tuple[int, float]`

Donc le 6-uplet est lui-même de type :

```
Tuple[int, bool, float, str, List[int], Tuple[int, float]]
```

Comme pour les autres types structurés, il faut importer le symbole `Tuple` depuis le module `typing`, par exemple avec l'instruction suivante :

```
from typing import Tuple
```

7.2 Déconstruction

La *déconstruction* d'un n-uplet consiste à récupérer les différents éléments qu'il contient.

7.2.1 Déconstruction dans des variables

La première possibilité de déconstruction consiste à stocker les différents éléments d'un n-uplet dans des variables dites **variables de déconstruction**.

Une **instruction de déconstruction d'un n-uplet** t de type `Tuple[`T_1`,`T_2`,...,`T_n`]` s'écrit de la façon suivante :

$v_1, v_2, \ldots, v_n = t$

avec :

— v_1 est une variable de déconstruction de type T_1
— v_2 est une variable de déconstruction de type T_2
— ...
— v_n est une variable de déconstruction de type T_n

Remarque : comme on connaît le type du n-uplet t, il n'est pas toujours nécessaire de déclarer le type des variables de déconstruction v_1, v_2, ..., v_n. On peut bien sûr tout de même effectuer cette déclaration pour améliorer la lisibilité du programme.

Considérons l'exemple suivant :

```
t : Tuple[int, bool, str] = (1, True, 'blion')

n, b, s = t # déconstruction
```

D'après le type de la variable `t` on sait que :

— la variable `n` est de type `int` et contient la valeur `1`

```
>>> n
1
```

— la variable `b` est de type `bool` et contient la valeur `True`

```
>>> b
True
```

— la variable `s` est de type `str` et contient la chaîne `"blion"`

```
>>> s
'blion'
```

Il n'est donc pas obligatoire de déclarer le type des variables de déconstruction, mais il peut être utile de le faire pour améliorer la lecture des programmes. Dans ce cas, il faut séparer la déclaration du type et l'initialisation.

```
t : Tuple[int, bool, str] = (1, True, 'blion')

n : int  # explication optionnelle
b : bool
s : str
n, b, s = t # déconstruction
```

Il arrive assez souvent que lors de la décomposition certaines variables ne soient pas nécessaires à la suite du calcul. Dans ce cas, on peut utiliser la variable spéciale `_` (souligné ou *underscore*) en remplacement. Par exemple si dans le triplet `t` nous ne souhaitons que le booléen, on pourra écrire :

```
t : Tuple[int, bool, str]
t = (1, True, 'blion')

b : bool
_, b, _ = t
```

Bien sûr, par la suite on n'aura accès qu'à la variable `b`.

```
>>> b
True
```

Il est également possible de récupérer la valeur de `_` mais dans ce cas il serait judicieux d'utiliser un nom de variable explicite, puisque le message envoyé par `_` est clair : on n'a pas besoin de l'élément correspondant.

7.2.2 Déconstruction par indice

Une autre façon de récupérer les différents éléments d'un n-uplet est d'y accéder par indice. Ainsi pour un n-uplet `t` de type `tuple[`T_0`,`T_1`,...,`T_{n-1}`]`, l'expression :

```
t[i]
```

récupère le `i`-ème élément, du type T_i, dans le n-uplet.

Par exemple :

```
>>> t
(1, True, 'blion')
```

```
>>> t[0]
1
```

```
>>> t[1]
True
```

```
>>> t[2]
'blion'
```

Remarque : on peut facilement confondre listes et n-uplets ici puisque cette méthode d'accès est commune. En fait, en Python les *tuples* sont des sortes de listes immutables, c'est-à-dire non modifiable directement en mémoire, et cette proximité fait qu'il est assez complexe de comprendre quand utiliser l'un ou l'autre. Dans ce livre, et c'est une bonne pratique Python en général, nous utilisons exclusivement les n-uplets pour représenter des produits cartésiens.

En conséquence, les listes et les n-uplets deviennent très différents, ainsi :

- une liste de type `List[T]` contient des éléments qui sont tous du même type `T`, et la longueur de cette liste est arbitraire (et peut même changer via `append`)
- un n-uplet de type `Tuple[`T_0`,`T_1`,...,`T_{n-1}`]` possède exactement n éléments pour un $n \geq 2$ fixé et qui ne changera pas, de plus chaque élément est potentiellement d'un type différent, clairement indiqué dans le type du n-uplet.

Dans ce livre, nous utiliserons presque exclusivement la décomposition par variables qui nous semble source de moins de confusion que l'accès par indice. En vous perfectionnant en Python, vous pourrez utiliser la décomposition par indice pour les n-uplets, et vous pourrez découvrir que l'on peut décomposer les listes en variables aussi ! Mais ne brûlons pas les étapes...

7.3 Utilisation des n-uplets

Les n-uplets servent à regrouper un nombre fixé d'éléments qui, une fois regroupés, forment ce que l'on nommera une *entité*.

Dans ce chapitre, nous prendrons essentiellement deux exemples :

- une entité *point du plan* qui regroupe 2 éléments tous deux de type `float`. Un point du plan est donc de type `Tuple[float, float]`.
- une entité *enregistrement de personne* (dans une base de données) qui regroupe 4 éléments : un nom et un prénom de type `str`, un age de type `int` et un statut marital (marié ou non) de type `bool`. Une personne est donc de type `Tuple[str, str, int, bool]`.

7.3.1 n-uplets en paramètres de fonctions

Un n-uplet, une fois construit, peut être passé en argument d'une fonction ou retourné en valeur de retour. Considérons tout d'abord le premier cas.

Exemple 1 : points dans le plan

On se pose le problème suivant :

> Etant donné deux points dans le plan, représentés par des couples de coordonnées, caculer la distance euclidienne séparant les deux points. On suppose ici que les points sont à coordonnées entières.

Rappel :

La **distance** entre deux points $p_1 = (x_1, y_1)$ et $p_2 = (x_2, y_2)$ est :

$$d(p_1, p_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Cette formule se traduit presque littéralement en Python :

```
from typing import Tuple # pour les déclarations de type
import math # pour math.sqrt
```

```
def distance(p1 : Tuple[float, float], p2 : Tuple[float, float]) -> float:
    """Retourne la distance entre les points p1 et p2.
    """
    x1 : float # abscisse de P1
    y1 : float # ordonnée de P1
    x1, y1 = p1

    x2 : float # abscisse de P2
    y2 : float # ordonnée de P2
    x2, y2 = p2

    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)
```

```
# Jeu de tests
assert distance( (0, 0), (1, 1) ) == math.sqrt(2)
assert distance( (2, 2), (2, 2) ) == 0.0
```

Remarque : ici les variables `x1, y1, x2, y2` sont déclarées car nous souhaitons préciser leur rôle : abscisse ou ordonnée.

Il est possible de simplifier un peu les signatures de fonctions sur les n-uplets en déclarant un **alias de type**. Le principe est de donner un nom court (une forme d'abbréviation) pour un type composé complexe.

```
Point = Tuple[float, float]
```

Ici on a déclaré un alias nommé `Point` pour le type `Tuple[float, float]`. Donc pour tout exercice sur les points du plan, on peut utiliser `Point` comme identifiant de type plutôt que `Tuple[float, float]`. Mais on se rappelle que c'est simplement un raccourci d'écriture permettant d'améliorer la lisibilité des programmes. Il faut cependant se rappeler du «véritable» type des points du plan est bien `Tuple[float, float]`.

Avec cette déclaration du type `Point`, nous pouvons rendre plus lisible la spécification de la fonction de calcul de distance :

```
def distance(p1 : Point, p2 : Point) -> float:
    """Retourne la distance entre les points p1 et p2.
    """
```

Exemple 2 : enregistrement de personnes

Pour l'entité *personne* nous introduisons l'alias de type suivante :

```
Personne = Tuple[str, str, int, bool]
```

Considérons par exemple la fonction suivante :

```
def est_majeure(p : Personne) -> bool:
    """Renvoie True si la personne est majeure, ou False sinon.
    """
    nom, pre, age, mar = p
    return age >= 18
```

```
# jeu de tests
assert est_majeure(('Itik', 'Paul', 17, False)) == False
assert est_majeure(('Unfor', 'Marcelle', 79, True)) == True
```

Pour retrouver dans la définition ci-dessous le type des variables de déconstruction, il est nécessaire d'effectuer une petite gymnastique mentale.

- le type de `p` est `Personne`
- puisque `Personne` est un alias de `tuple[str, str, int, bool]` on en déduit que le «vrai» type de `p` est en fait `tuple[str, str, int, bool]`
- on en déduit :
  - la variable `nom` est de type `str`
  - la variable `pre` est de type `str`
  - la variable `age` est de type `int`
  - la variable `mar` est de type `bool`

Cependant, nous n'avons besoin que de la variable `age` donc une version plus satisfaisante de la fonction est la suivante :

```
def est_majeure(p : Personne) -> bool:
    """Renvoie True si la personne est majeure, ou False sinon."""
```

```
    _, _, age, _ = p
    return age >= 18
```

7.3.2 n-uplets en valeur de retour

Nous avons vu deux fonctions qui prennent des n-uplets en paramètre. On peut bien sûr, de façon complémentaire, retourner des n-uplets en valeur de retour de fonction.

Exemple 1 : somme et moyenne

On propose ci-dessous une définition de la fonction `somme_et_moyenne` qui, étant donné une liste `l` de nombres, retourne un couple formé de respectivement la somme et la moyenne des éléments de `l`.

```
from typing import List, Tuple # on a aussi besoin des listes

def somme_et_moyenne(l : List[float]) -> Tuple[float, float]:
    """Précondition : len(l) > 0
    Retourne un couple formé de la somme et de la moyenne
    des éléments de la liste l.
    """
    # Somme
    s : float = 0

    x : float
    for x in l:
        s = s + x

    return (s, (s / len(l)))
```

```
# Jeu de tests
assert somme_et_moyenne([1, 2, 3, 4, 5]) == (15, 3.0)
assert somme_et_moyenne([-1.2, 0.0, 1.2]) == (0.0, 0.0)
```

Exemple 2 : mariage d'une personne

Voici comme second exemple une définition de la fonction `mariage` qui prend en paramètre une personne et retourne la même personne avec son statut martial validé.

```
def mariage(p : Personne) -> Personne:
    """Retourne la personne avec un statut marital validé.
    """
    nom, pre, age, _ = p
    return (nom, pre, age, True)
```

```
# Jeu de tests
assert mariage(('Itik', 'Paul', 17, False)) == ('Itik', 'Paul', 17, True)
assert mariage(('Unfor', 'Marcelle', 79, True)) \
       == ('Unfor', 'Marcelle', 79, True)
```

Question : quels sont les types respectifs des variables de déconstruction de n-uplet ? Expliquer comment déduire cette information.

7.3.3 Listes de n-uplets

De nombreux problèmes nécessitent de considérer des *listes de n-uplets*. On étudie donc ici le mélange du concept de n-uplet vu dans ce chapitre et les listes du chapitre précédent. Nous considérons respectivement des exemples de réduction, de transformation et de filtrage de listes de n-uplets.

7.3.3.1 Schéma de réduction

Le schéma de réduction de liste consiste, rappelons-le, à synthétiser une information plus simple à partir d'une liste d'éléments. On considère ici que les éléments sont des n-uplets, donc la signature générale est la suivante :

`List[Tuple[`T_1`,`T_2`,...,`T_N`]] ->` U

où `U` est un type «plus simple» que celui de la liste de départ.

7.3.3.1.1 Exemple : âge moyen d'un groupe de personnes Un base de données simple peut être représentée par une liste de n-uplets. Par exemple, un groupe de personnes peut être représenté par une information de type `List[Personne]` (ou `List[Tuple[str, str, int, bool]]` si l'on explicite l'alias `Personne`).

Considérons par exemple l'extraction de l'âge moyen des personnes d'un groupe, soit la fonction `age_moyen` définie de la façon suivante :

```
def age_moyen(l : List[Personne]) -> float:
    """Précondition : len(l) > 0
    Renvoie l'age moyen des personne enregistrées dans la liste l.
    """
    # Age cumulé
    age_total : int = 0

    pers : Personne
    for pers in l:
        age : int # optionnel, mais peut-être utile à rappeler
        _, _, age, _ = pers

        age_total = age_total + age
```

```
    return age_total / len(l)
```

```
# Jeu de tests

BD : List[Personne]  # Base de donnée
BD = [('Itik', 'Paul', 17, False),
      ('Unfor', 'Marcelle', 79, True),
      ('Laveur', 'Gaston', 38, False),
      ('Potteuse', 'Henriette', 24, True),
      ('Ltar', 'Gibra', 13, False),
      ('Diaprem', 'Amar', 22, True)]

assert age_moyen(BD)  == (17 + 79 + 38 + 24 +13 + 22) / 6
assert age_moyen([('Un', 'Una', 1, True)])  == 1.0
```

7.3.3.2 boucles d'itérations sur les listes de n-uplets

Il existe une variante des **boucles d'itérations sur les listes de n-uplets** qui permet de simplifier les écritures.

La syntaxe de cette variant du `for` est la suivante :

```
for (v1, v2, ..., vn) in <liste>:
    <instruction_du_corps_1>
    <instruction_du_corps_2>
    ...
```

avec :

- `v1, v2, ..., vn` des variables de déconstructions de n-uplet
- `<liste>` une expression de type `list[tuple[T1, T2, ..., Tn]]`

Remarque : encore une fois, il n'est pas obligatoire de déclarer le type des variables, qui peut être déduit. Mais on peut tout de même effectuer cette déclaration.

Il s'agit d'un raccourci d'écriture pour l'itération suivante :

```
p : tuple[T1, T2, ..., Tn]   (élément courant)
for p in <liste>:
    v1, v2, ..., vn = p
    <instruction_du_corps_1>
    <instruction_du_corps_2>
    ...
```

En guise d'illustration, on peut s'inspirer de ce raccourci d'écriture pour proposer une définition plus concise de la fonction `age_moyen`.

```
def age_moyen(l : List[Personne]) -> float:
    """ ... cf. ci-dessus ...
```

```
    """
    # Age cummulé
    age_total : int = 0

    age : int
    for (_, _, age, _) in l:
        age_total = age_total + age

    return age_total / len(l)
```

7.3.3.3 Transformations

Les transformations de listes s'appliquent bien sûr également aux listes de n-uplets. En guise d'exemple, on considère l'extraction des noms d'une base de données de personnes.

```
def extraction_noms(l : List[Personne]) -> List[str]:
    """Retourne la liste des noms des personnes de l.
    """
    # Liste résultat
    lr : List[str] = []

    for (nom, _, _, _) in l:
        lr.append(nom)

    return lr
```

```
# Jeu de tests

# BD : list[Personne]  (base de donnée, cf. ci-dessus)

assert extraction_noms(BD) \
    == ['Itik', 'Unfor', 'Laveur', 'Potteuse', 'Ltar', 'Diaprem']
assert extraction_noms([('Un', 'Una', 1, True)]) == ['Un']
assert extraction_noms([]) == []
```

Remarque : ici on n'a pas indiqué le type pour la variable `nom`, on laisse le programmeur le déduire du type n-uplet `Personne`.

7.3.3.4 Filtrages

Pour illustrer le filtrage des listes de n-uplets, on considère le sélection des personnes majeures dans la base de données.

```
def personnes_majeures(l : List[Personne]) -> List[Personne]:
    """Retourne la liste des personnes majeures dans la base l.
    """
```

```
    # Liste résultat
    lr : List[Personne] = []

    p : Personne
    for p in l:
        if est_majeure(p):
            lr.append(p)

    return lr
```

```
# Jeu de tests

# BD : list[Personne]  (base de donnée, cf. ci-dessus)

assert personnes_majeures(BD) == [('Unfor', 'Marcelle', 79, True),
                                  ('Laveur', 'Gaston', 38, False),
                                  ('Potteuse', 'Henriette', 24, True),
                                  ('Diaprem', 'Amar', 22, True)]

assert personnes_majeures([('Un', 'Una', 1, True)]) == []
assert personnes_majeures([]) == []
```

Chapitre 8

Compréhensions de listes

Nous avons vu au chapitre 6 des schémas de manipulation de listes que l'on rencontre fréquemment en pratique, notamment :

1. le schéma de construction : lorsque l'on construit une liste de façon algorithmique à partir d'une donnée «plus simple» qu'une liste, comme un entier ou un intervalle d'entiers,
2. le schéma de transformation: lorsque l'on transforme une liste en appliquant à chaque élément une fonction unaire donnée,
3. le schéma de filtrage : lorsque l'on filtre les éléments d'une liste pour un prédicat donné,
4. le schéma de réduction : lorsque l'on synthétise une information à partir des éléments d'une liste.

Les **expressions de compréhension** de Python, que l'on étudie dans ce chapitre, proposent une syntaxe concise permettant de généraliser, et de combiner, les trois premiers schémas. Nous verrons également, en guise de complément, la notion de **fonctionnelle** (ou fonctions dites d'*ordre supérieur*) qui offre une alternative pour généraliser les schémas de manipulation y compris les réductions.

8.1 Schéma de construction

De nombreux problèmes pratiques nécessitent de construire une liste de façon algorithmique (élément par élément) à partir d'une donnée qui n'est pas une liste, et que l'on peut raisonnablement qualifier de «plus simple» (même si ce critère est quelque peu objectif).

8.1.1 Principes de base

Pour illustrer les principes de base du schéma de construction, considérons la fonction `naturels` qui à partir d'un entier naturel `n` non-nul retourne la liste des éléments successifs dans l'intervalle $[1; n]$.

```
from typing import List

def naturels(n : int) -> List[int]:
    """Précondition: n >= 1
    Retourne la liste des entiers de 1 à n.
    """
    # Liste résultat
    lr : List[int] = []
    k : int
    for k in range(1, n + 1):
        lr.append(k)

    return lr
```

```
# Jeu de tests
assert naturels(5) == [1, 2, 3, 4, 5]
assert naturels(1) == [1]
assert naturels(0) == []
```

On exploite dans la définition ci-dessous le *schéma de construction* d'une liste à partir d'une autre séquence, ici l'intervalle d'entiers `range(1, n + 1)` pour un `n` donné.

Cette construction algorithmique peut être décrite de façon très concise en utilisant une **expression de compréhension simple**.

Par exemple :

```
>>> [k for k in range(1, 6)]
[1, 2, 3, 4, 5]
```

L'expression ci-dessous signifie presque littéralement :

> Construction de la liste des `k` pour `k` dans l'intervalle $[1; 6[$

Si on veut une liste composée des entiers successifs dans l'intervalle clos $[3; 9]$, on pourra écrire :

```
>>> [k for k in range(3, 10)]
[3, 4, 5, 6, 7, 8, 9]
```

8.1.2 Syntaxe et principe d'interprétation

Plus généralement, la syntaxe d'une compréhension simple est la suivante :

`[ <elem> for <var> in <seq> ]`

où :

— `<var>` est une **variable de compréhension**
— `<elem>` est une expression appliquée aux valeurs successives de la variable `<var>`
— `<seq>` est une expression retournant une séquence, notamment : `range`, `str` ou `List`.

Principe d'interprétation :

L'expression de compréhension ci-dessus signifie :

> Construction de la liste des `<elem>` pour `<var>` dans la séquence `<seq>`

Plus précisément, la liste construite est composée de la façon suivante :

— le premier élément est la valeur de l'expression `<elem>` dans laquelle :
  — la variable `<var>` a pour valeur le premier élément de `<seq>`
— le second élément est la valeur de l'expression `<elem>` dans laquelle :
  — la variable `<var>` a pour valeur le deuxième élément de `<seq>`
— ...
— le dernier élément est la valeur de l'expression `<elem>` dans laquelle :
  — la variable `<var>` a pour valeur le dernier élément de `<seq>`

Remarque : il n'est pas obligatoire de déclarer le type de la variable de compréhension `<var>`. En effet, le type de cette variable est le même que celui des éléments de la séquence, il est donc en général facile de le déduire du type de la séquence `<seq>`.

Dans l'exemple :

```
[k for k in range(3, 10)]
```

le type de `k` déduit est `int` puisque la séquence est de type `range` (intervalle d'entiers).

Il est toutefois possible de déclarer le type de la variable de compréhension sur la ligne précédent l'expression de compréhension, de la façon suivante :

```
k : int
[k for k in range(3, 10)]
```

Nous pouvons maintenant proposer une troisième définition pour la fonction `naturels`, cette fois-ci de façon particulièrement concise.

```
def naturels(n : int) -> List[int]:
    """ ... cf. ci-dessus ..."""
    return [k for k in range(1, n + 1)]
```

La traduction presque littérale du code Python est la suivante :

> `naturels(n)` retourne la liste des `k` pour `k` dans l'intervalle `[1;n+1[`

On voit ici un intérêt majeur des compréhensions : la description de la solution en Python est très proche de la spécification du problème. On dit que les compréhensions ont un caractère *déclaratif*.

8.1.3 Expressions complexes dans les compréhensions

Dans la syntaxe des compréhensions, l'expression `<elem>` peut être aussi complexe que l'on veut. On doit juste faire attention à bien utiliser la variable de compréhension (dans le cas contraire, tous les éléments construits ont la même valeur).

Illustrons ceci sur la construction d'une liste de multiples.

```
def multiples(k : int, n : int) -> List[int]:
    """Précondition : n >= 1
    Retourne la liste des n premiers entiers naturels
    non-nuls multiples de k.
    """
    return [k*j for j in range(1, n + 1)]
```

```
# Jeu de tests
assert multiples(2, 5) == [2, 4, 6, 8, 10]
assert multiples(3, 10) == [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
assert multiples(0, 5) == [0, 0, 0, 0, 0]
assert multiples(5, 0) == []
```

Encore une fois la traduction littérale est intéressante :

> `multiples(k, n)` retourne la liste des `k*j` pour `j` dans l'intervalle `[1;n+1[`

Exercice : réécrire des versions avec `for` puis `while` de la fonction `multiples`.

8.1.4 Constructions à partir de chaînes de caractères

En plus des intervalles d'entiers, nous avons également vu les chaînes de caractères qui sont également des séquences. Il est donc possible de construire des listes par compréhension à partir des chaînes.

Par exemple :

```
>>> [c for c in 'Bonjour Madame']
['B', 'o', 'n', 'j', 'o', 'u', 'r', ' ', 'M', 'a', 'd', 'a', 'm', 'e']
```

De façon littérale, l'expression de compréhension ci-dessus :

> construit la liste des `c` pour `c` un caractère dans la chaîne `'Bonjour Madame'`

8.2 Schéma de transformation

Le schéma de transformation, ou schéma `map`, consiste comme nous l'avons vu dans les précédents chapitres à appliquer une fonction unaire à chacun des éléments d'une liste pour reconstruire une liste transformée.

8.2.1 Construction à partir d'une liste

Nous l'avons déjà vu dans de nombreux exemples, les listes résultats des transformations sont systématiquement *reconstruites*. On peut donc interpréter une transformation de façon alternative comme la construction d'une liste transformée à partir d'une liste. Puisque les listes Python sont des séquences, ce point de vue alternatif permet d'exploiter les expressions de compréhensions.

Considérons comme premier exemple la liste des `n` premiers carrés, avec la spécification suivante :

```
def liste_carres(l : List[int]) -> List[int]:
    """Retourne la liste des carrés des éléments de la liste l.
    """
```

En nous basant sur la fonction suivante :

```
def carre(k : int) -> int:
    """Retourne le carré de k.
    """
    return k * k
```

Il est très facile de proposer une définition de `liste_carres` en exploitant la boucle `for` ou même le `while`. Cependant, les compréhensions nous offrent une solution beaucoup plus concise. Il n'est même pas nécessaire de définir une fonction, une simple expression suffit. Par exemple, si l'on veut construire la liste des carrés des entiers de 1 à 5 on peut simplement écrire :

```
>>> [carre(k) for k in [1, 2, 3, 4, 5]]
[1, 4, 9, 16, 25]
```

Et bien sûr la fonction `carre` ne semble pas vraiment nécessaire. Nous pouvons écrire plus directement :

```
>>> [k*k for k in [1, 2, 3, 4, 5]]
[1, 4, 9, 16, 25]
```

Littéralement, on a écrit en Python :

> la liste des carrés de `k` pour `k` dans la liste `[1, 2, 3, 4, 5]`

On peut donc redéfinir la fonction `liste_carres` en une seule ligne de code :

```
def liste_carres(L):
    """ ... cf. ci-dessus ...
    """
    return [k*k for k in L]
```

```
# Jeu de tests
assert liste_carres([1, 2, 3, 4, 5]) == [1, 4, 9, 16, 25]
assert liste_carres([10, 100, 1000]) == [100, 10000, 1000000]
assert liste_carres([]) == []
```

Prenons un second exemple illustratif issu du chapitre 6 sur les listes : la fonction `liste_longueurs` qui prend en entrée une liste de chaînes de caractères et qui en sortie construit la liste des longueurs de ces chaînes (avec le fonction `len`). Cette fonction peut également être définie une une seule ligne de code avec une compréhension :

```
def liste_longueurs(l : List[str]) -> List[int]:
    """ ... cf. chapitre 6 ... """
    return [len(s) for s in l]
```

```
# Jeu de tests
assert liste_longueurs(['un', 'deux', 'trois', 'quatre', 'cinq']) \
       == [2, 4, 5, 6, 4]
assert liste_longueurs(['', '.', '..', '...']) == [0, 1, 2, 3]
assert liste_longueurs([]) == []
```

Littéralement :

> `liste_longueurs(L)` retourne la liste des longueurs de `s` pour `s` une chaîne de caractère dans `L`.

8.2.2 Complément : la fonctionnelle map

Une approche alternative, issue de ce que l'on appelle la *programmation fonctionnelle*, consiste à définir une fonction générique, appelée une **fonctionnelle** (ou fonction d'ordre supérieur). Pour les transformations, cette fonction se nomme classiquement `map` et peut être définie de la façon suivante :

```
from typing import List, Callable, TypeVar

# Variables de type
T = TypeVar('T')
U = TypeVar('U')

def map(f : Callable[[T], U], l : List[T]) -> List[U]:
    """Retourne la transformation de la liste L par la fonction f.
    """
    return [f(e) for e in l]
```

```
# Jeu de tests
assert map(carre, [1, 2, 3, 4, 5]) == [1, 4, 9, 16, 25]
assert map(len, ['un', 'deux', 'trois', 'quatre']) == [2, 4, 5, 6]
```

Dans la définition de la fonctionnelle `map`, la principale difficulté concerne la signature de la fonction, que l'on rappelle ci-dessous :

```
(T -> U) * List[T] -> List[U]
```

Cette signature, avec `T` et `U` des variables de type, s'interprète de la façon suivante :

Dans `map(f, L)` :

- le premier paramètre `f` est de type `T -> U`, c'est-à-dire une fonction unaire qui transforme une donnée de type `T` en une donnée de type `U`.
- le second paramètre `l` est une liste de type `List[T]`
- la valeur de retour de `map` est de type `List[U]`.

Par exemple, dans l'expression :

```
map(carre, [1, 2, 3, 4, 5])
```

- le premier argument `carre` pour `f` est de type `int -> int` (donc `T=int` et `U=int`)
- le second argument `[1, 2, 3, 4, 5]` pour `l` est de type `List[int]` (donc `List[T]` pour `T=int`)
- la valeur de retour est `[1, 4, 9, 16, 25]` de type `List[int]` (donc `list[U]` pour `U=int`)

De même, dans l'expression :

```
map(len, ['un', 'deux', 'trois', 'quatre'])
```

- le premier argument `len` pour `f` est de type `str -> int` (donc `T=str` et `U=int`)
- le second argument `['un', 'deux', 'trois', 'quatre']` pour `l` est de type `List[str]` (donc `list[T]` pour `T=str`)
- la valeur de retour est `[2, 4, 5, 6]` de type `List[int]` (donc `List[U]` pour `U=int`)

Le fait que l'on ait utilisé une compréhension pour définir la fonctionnelle `map` illustre le fait que les compréhensions généralisent le principe des transformation (et donc le `map`). Mais on peut bien sûr se passer des compréhensions pour définir `map`.

Exercice : redéfinir `map` en utilisant une boucle `for` plutôt qu'une compréhension.

8.3 Schéma de filtrage

Le schéma de filtrage des listes peut également profiter des expressions de compréhension. Nous rappelons qu'il s'agit, à partir d'une liste `l`, de construire la sous-liste des éléments

de `l` qui vérifient un prédicat donné. Encore une fois, on peut voir le filtrage comme une construction de liste à partir d'une autre liste. Cependant, la différence est que cette construction est conditionnée : on ne retient pas forcément tous les éléments de la liste de départ.

Considérons les deux prédicats suivants.

```
def est_positif(n : int) -> bool:
    """Retourne True si l'entier n est (strictement) positif, False sinon.
    """
    return n > 0
```

```
# Jeu de tests
assert est_positif(1) == True
assert est_positif(-1) == False
assert est_positif(0) == False
```

```
def est_pair(n : int) -> bool:
    """Retourne True si l'entier n est pair, False sinon.
    """
    return (n % 2) == 0
```

```
# Jeu de tests
assert est_pair(1) == False
assert est_pair(2) == True
assert est_pair(3) == False
```

Le filtrage pour le premier prédicat consiste à ne conserver que les éléments positifs d'une liste d'entiers, avec la spécification suivante :

```
def liste_positifs(l : List[int]) -> List[int]:
    """Retourne la sous-liste des entiers positifs de l.
    """
```

Pour le second prédicat on obtient la sous-liste des entiers pairs, avec la spécification suivante :

```
def liste_pairs(l : List[int]) -> List[int]:
    """Retourne la sous-liste des entiers pairs de L.
    """
```

Les compréhensions permettent également d'exprimer ce type de filtrage, en ajoutant une *condition de compréhension*.

```
>>> [n for n in [1, -1, 2, -2, 3, -3, -4] if est_positif(n)]
[1, 2, 3]
```

Littéralement, l'expression ci-dessus signifie :

> la liste des `n` pour `n` dans la liste `[1, -1, 2, -2, 3, -3, -4]` mais *uniquement si* `n` est positif.

Pour les entiers pairs, on peut sur le même principe écrire :

```
>>> [n for n in [1, 2, 3, 4, 5, 6, 7] if est_pair(n)]
[2, 4, 6]
```

Cette fois-ci l'interprétation est la suivante :

> la liste des `n` pour `n` dans la liste `[1, 2, 3, 4, 5, 6, 7]` *uniquement si* `n` est pair.

Nous pouvons ainsi proposer des définitions très concises pour les fonctions de filtrage, en nous passant pour l'occasion des prédicats explicites.

```
def liste_positifs(l : List[int]) -> List[int]:
    """Retourne la sous-liste des entiers positifs de l.
    """
    return [n for n in l if n > 0]
```

```
# Jeu de tests
assert liste_positifs([1, -1, 2, -2, 3, -3, -4]) == [1, 2, 3]
assert liste_positifs([1, 2, 3]) == [1, 2, 3]
assert liste_positifs([-1, -2, -3]) == []
```

Et pour les entiers pairs :

```
def liste_pairs(l : List[int]) -> List[int]:
    """Retourne la sous-liste des entiers pairs de L.
    """
    return [n for n in l if n % 2 == 0]
```

```
# Jeu de tests
assert liste_pairs([1, 2, 3, 4, 5, 6, 7]) == [2, 4, 6]
assert liste_pairs([2, 4, 6]) == [2, 4, 6]
assert liste_pairs([1, 3, 5, 7]) == []
```

8.3.1 Compréhensions avec filtrage : syntaxe et interprétation

La syntaxe des compréhensions peut être complétée par une condition :

```
[ <elem> for <var> in <seq> if <condition> ]
```

où :

- `<var>` est la **variable de compréhension**
- `<elem>` est l'expression appliquée aux valeurs successives de la variable `<var>`,
- `<seq>` est l'expression de séquence sur laquelle porte la compréhension,
- `<condition>` est **la condition de compréhension** : une expression booléenne portant sur la variable de compréhension.

Principe d'interprétation :

La liste résultat correspond à la construction par compréhension décrite précédemment, mais filtrée pour ne retenir que les éléments pour lesquels `<condition>` vaut `True`.

On peut remarquer dès maintenant que la syntaxe des compréhensions permet de composer les transformations et les filtrages. Considérons les exemples ci-dessous :

```
>>> [k*k for k in [1, 2, 3, 4, 5] if k % 2 == 0]
[4, 16]
```

Ici on a construit une liste de carrés, mais seulement pour les nombres pairs de la liste de départ.

```
>>> [10 / x for x in [2, 3, 0, 1, 0, 4] if x != 0]
[5.0, 3.3333333333333335, 10.0, 2.5]
```

Ici, on a divisé 10 par chaque élément `x` de la liste en entrée, cependant en faisant attention de ne pas effectuer de division par 0.

8.3.2 Complément : la fonctionnelle filter

Tout comme pour `map` il est possible d'écrire une *fonctionnelle de filtrage*, qui se nomme traditionnellement `filter`. Cette fonctionnelle est également facile à définir à partir des compréhensions.

```
def filter(predicat : Callable[[T], bool], l : List[T]) -> List[T]:
    """Retourne la sous-liste des éléments de l filtrés par predicat.
    """
    return [e for e in l if predicat(e)]
```

```
# Jeu de tests
assert filter(est_positif, [1, -1, 2, -2, 3, -3, -4]) == [1, 2, 3]
assert filter(est_pair, [1, 2, 3, 4, 5, 6, 7]) == [2, 4, 6]
```

Dans la définition de la fonctionnelle `filter`, la principale difficulté concerne encore une fois la signature de la fonction, qui est la suivante :

```
(T -> bool) * List[T] -> List[T]
```

Cette signature, avec `T` une variable de type, s'interprète de la façon suivante.

Dans `filter(predicat, l)` :

- le premier paramètre `predicat` est de type `T -> bool`, c'est-à-dire une fonction unaire à valeur booléenne,
- le second paramètre `l` est une liste de type `List[T]`
- la valeur de retour de `filter` est de type `List[T]`.

Par exemple, dans l'expression :

```
filter(est_positif, [1, -1, 2, -2, 3, -3, -4])
```

— le premier argument `est_positif` pour `predicat` est de type `int -> bool` (donc `T=int`)
— le second argument `[1, -1, 2, -2, 3, -3, -4]` pour `l` est de type `List[int]` (donc `list[T]` pour `T=int`)
— la valeur de retour est `[1, 2, 3]` de type `List[int]` (donc `List[T]` pour `T=int`)

Exercice : redéfinir `filter` en utilisant une boucle `for` plutôt qu'une compréhension.

8.4 Plus loin avec les compréhensions

Dans cette section, nous abordons certains aspects plus avancés des expressions de compréhensions.

8.4.1 Compréhensions de listes à partir d'autres séquences

Pour commencer, nous avons pour l'instant considérés uniquement des compréhensions prenant en entrée des listes. Or, il est possible d'exploiter tout type de séquence.

Par exemple, nous pouvons prendre un intervalle en entrée, plutôt qu'une liste :

```
>>> [k*k for k in range(1, 11) if est_pair(k)]
[4, 16, 36, 64, 100]
```

On peut aussi, bien sûr, prendre une chaîne de caractère pour obtenir une liste de caractères :

```
>>> [c for c in "Bonjour Tout Le Monde" if c > "a"]
['o', 'n', 'j', 'o', 'u', 'r', 'o', 'u', 't', 'e', 'o', 'n', 'd', 'e']
```

La condition `c > "a"` est une «bidouille» (peu élégante) permettant d'enlever les majuscules de la chaîne en entrée, car les caractères majuscules sont «plus petits» que les minuscules dans la spécification Unicode. On remarque que l'obtient bien une liste (de caractères) en sortie car il s'agit bien d'une *compréhension de liste* ici, le résultat produit est toujours une liste.

8.4.2 Compréhensions sur les n-uplets

Les expressions de compréhensions s'appliquent également aux listes de n-uplets.

Pour cela, on utilise une syntaxe proche des variables de déconstruction de n-uplets dans les boucles d'itération.

Considérons par exemple la fonction suivante :

```
from typing import List, Tuple

def liste_prix(l : List[Tuple[str, float]]) -> List[float]:
    """Retourne la liste des prix des produits de l.
    """
    lr : List[float] = []

    for (produit, prix) in l:
        lr.append(prix)

    return lr
```

```
# Jeu de tests
assert liste_prix([('afone 6', 999.90), ('ralaxy 6', 712.5), ('hbc two', 399.5)]) \
       == [999.90, 712.5, 399.5]

assert liste_prix([]) == []
```

Le premier cas de test peut être reproduit par l'expression de compréhension suivante :

```
>>> [prix for (produit, prix)
          in [('afone 6', 999.90), ('ralaxy 6', 712.5), ('hbc two', 399.5)]]
[999.9, 712.5, 399.5]
```

La syntaxe utilisée est la suivante :

```
[ <elem> for (<var1>, <var2>, ..., <varN>) in <seq>  ...]
```

La différence avec les compréhensions sur des éléments simples et non des n-uplets est que pour chaque calcul de `<elem>` l'ensemble des variables `<var1>, <var2>, ..., <varN>` est disponible. Et il est bien sûr d'effectuer un filtrage avec une conditoin portant sur ces variables avec `if`.

Nous pouvons en déduire une version plus concise de notre fonction `liste_prix`.

```
def liste_prix(l : List[Tuple[str, float]]) -> List[float]:
    """Retourne la liste des prix des produits de l.
    """
    return [prix for (produit, prix) in l]
```

Pour illustrer l'utilisation des conditions, nous pouvons par exemple exprimer une *requête* sur notre liste de produits, ici pour lister les produits dépassant un certain prix (ici *500* euros) :

```
>>> [produit for (produit, prix)
              in [('afone 6', 999.90), ('ralaxy 6', 712.5), ('hbc two', 399.5)]
              if prix > 500.0]
['afone 6', 'ralaxy 6']
```

8.4.3 Compréhensions multiples

Les boucles imbriquées permettent de combiner plusieurs constructions. Considérons en guise d'exemple le problème de la construction des couples d'entiers naturels (i, j) sur l'intervalle $[1; n]$ tels que $i \leq j$.

On peut utiliser au choix une boucle `while` ou une boucle `for`, mais nous privilégions la seconde solution.

```
def liste_couples(n : int) -> List[Tuple[int, int]]:
    """Précondition: n >= 0
    Retourne la liste des couples (i,j) sur l'intervalle [1;n].
    """

    # Liste résultat
    lr : List[Tuple[int, int]] = []

    i : int
    for i in range(1, n + 1):
        # j : int
        for j in range(i, n + 1):
            lr.append( (i, j) )

    return lr
```

```
# Jeu de tests
assert liste_couples(0) == []
assert liste_couples(1) == [(1, 1)]
assert liste_couples(2) == [(1, 1), (1, 2), (2, 2)]
assert liste_couples(3) == [(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

Il est possible de retrouver les exemples ci-dessus avec une **compréhension multiple**.

Par exemple, pour le cas `n=2` :

```
>>> [(i, j) for i in range(1, 3) for j in range(i, 3)]
[(1, 1), (1, 2), (2, 2)]
```

Littéralement :

> la liste des couples `(i,j)` pour `i` dans l'intervalle `[1;3[` et `j` dans l'intervalle `[i;3[` (pour chaque `i`)

Ou encore pour le cas `n=3` :

```
>>> [(i, j) for i in range(1, 4) for j in range(i, 4)]
[(1, 1), (1, 2), (1, 3), (2, 2), (2, 3), (3, 3)]
```

8.4.3.1 Syntaxe et principe d'interprétation des compréhensions multiples

La syntaxe des compréhensions multiples est la suivante :

`[ <elem> for <var1> in <seq1> for <var2> in <seq2>  ... ]`

où :

- `<elem>` est l'expression appliquée aux valeurs successives des variables `<var1>`, `<var2>` , etc.
- `<var1>` est la **première variable de compréhension**
- `<seq1>` est l'expression de séquence sur laquelle porte la première compréhension,
- `<var2>` est la **seconde variable de compréhension**
- `<seq2>` est l'expression de séquence sur laquelle porte la seconde compréhension,
- ...

Principe d'interprétation :

La liste construite par une telle compréhension correspond au produit cartésien des éléments de `<seq1>`, `<seq2>`, etc.

- le premier élément est la valeur de l'expression `<elem>` avec:
  - la variable `<var1>` a pour valeur le premier élément de `<seq1>`
  - la variable `<var2>` a pour valeur le premier élément de `<seq2>`
  - ...
- le second élément est la valeur de l'expression `<elem>` avec:
  - la variable `<var1>` a pour valeur le premier élément de `<seq1>`
  - la variable `<var2>` a pour valeur le deuxième élément de `<seq2>`
  - ...
- le troisième élément est la valeur de l'expression `<elem>` avec:
  - la variable `<var1>` a pour valeur le premier élément de `<seq1>`
  - la variable `<var2>` a pour valeur le troisième élément de `<seq2>`
  - ...
- ...
- le ?-ième élément est la valeur de l'expression `<elem>` avec:
  - la variable `<var1>` a pour valeur le deuxième élément de `<seq1>`
  - la variable `<var2>` a pour valeur le premier élément de `<seq2>`
  - ...
- ...
- l'avant dernier élément est la valeur de l'expression `<elem>` avec:
  - la variable `<var1>` a pour valeur le dernier élément de `<seq1>`
  - la variable `<var2>` a pour valeur l'avant-dernier élément de `<seq2>`
- le dernier élément est la valeur de l'expression `<elem>` avec:
  - la variable `<var1>` a pour valeur le dernier élément de `<seq1>`
  - la variable `<var2>` a pour valeur le dernier élément de `<seq2>`

On déduit de ce principe une nouvelle définition plus concise de notre fonction.

```
def liste_couples(n : int) -> List[Tuple[int, int]]:
    """ ... cf. ci-dessus ... """
    return [(i, j) for i in range(1, n + 1)
                   for j in range(i, n + 1)]
```

Il est intéressant de noter que, comme on le voit dans l'exemple ci-dessus, l'expression de séquence `<seq2>` peut contenir la variable `<var1>`. Plus généralement, l'expression de séquence `<seq i>` peut utiliser toutes les variables de compréhensions `<var1>`, `<var2>`, ..., `<var i-1>`.

Remarque : la présentation des compréhensions sur plusieurs lignes n'est pas obligatoire, mais cela améliore grandement la lecture de ces expressions.

8.4.3.2 Compréhensions multiples avec filtrage

On peut bien sûr compléter les expressions de compréhensions multiples par des conditions de compréhension.

La syntaxe générale des compréhensions de listes est donc la suivante :

```
[ <elem> for <var1> in <seq1> if <cond1>
         for <var2> in <seq2> if <cond2> ... ]
```

Essayons tout de suite un exemple :

```
>>> [(i, j) for i in range(1, 7) if est_pair(i)
             for j in range(i, 7) if est_pair(j)]
[(2, 2), (2, 4), (2, 6), (4, 4), (4, 6), (6, 6)]
```

8.4.3.2.1 Exemple : liste des couples divisibles Voici en guise d'exemple une définition de la fonction `liste_couples_divisibles` qui retourne la liste des couples (i, j) dans l'intervalle $[1; n]$ et tels que j divise i.

```
def liste_couples_divisibles(n : int) -> List[Tuple[int, int]]:
    """Précondition : n >= 1
    Retourne la liste des couples (i,j) dans l'intervalle
    [1;n] tels que j divise i.
    """
    return [(i,j) for i in range(1, n+1)
                  for j in range(1, i+1) if i % j == 0]
```

```
# Jeu de tests
assert liste_couples_divisibles(1) == [(1, 1)]
assert liste_couples_divisibles(2) == [(1, 1), (2, 1), (2, 2)]
assert liste_couples_divisibles(4) \
       == [(1, 1), (2, 1), (2, 2), (3, 1), (3, 3), (4, 1), (4, 2), (4, 4)]
```

Exercice : proposer une définition de la fonction ci-dessus *sans* utiliser de compréhension.

8.5 Complément : le schéma de réduction

Le dernier schéma classique de manipulation des listes consiste à synthétiser une information «simple» à partir des éléments d'une liste. Les compréhensions ici ne sont pas utiles puisqu'il ne s'agit pas de reconstruire une liste. En revanche, on peut généraliser ce schéma par une fonctionnelle.

La somme des éléments d'un liste de nombres est un exemple classique de réduction.

```
def somme_liste(l : List[float]) -> float:
    """Retourne la somme des éléments de la liste l.
    """
    s: int
    s = 0

    n : float
    for n in l:
        s = s + n

    return s
```

```
# Jeu de tests
assert somme_liste([1, 2, 3, 4, 5]) == 15
assert somme_liste([1]) == 1
assert somme_liste([]) == 0
```

On peut de la même façon effectuer un calcul de produit :

```
def produit_liste(l : List[float]) -> float:
    """Retourne le produit des éléments de la liste l.
    """
    s: int
    s = 1

    n : float
    for n in l:
        s = s * n

    return s
```

```
# Jeu de tests
assert produit_liste([1, 2, 3, 4, 5]) == 120
assert produit_liste([1]) == 1
assert produit_liste([]) == 1
```

Les deux fonctions se ressemblent beaucoup. Dans les deux cas on a utilisé un opérateur binaire (addition puis multiplication) ainsi que leur élément neutre (0 puis 1). Plus généralement, on peut considérer une fonction de fusion et une valeur initiale. Ceci permet la définition d'une **fonctionnelle de réduction**, traditionnellement nommée

`reduce` (et parfois `fold` pour «pliage»), que l'on peut définir de la façon suivante :

```
def reduce(op : Callable[[T, U], U], en : U, l : List[T]) -> U:
    """Retourne la réduction de la liste l selon pour l'opérateur binaire
    op associatif avec en comment élément neutre."""

    reduc : U
    reduc = en

    e : alpha
    for e in l:
        reduc = op(e, reduc)

    return reduc
```

Commençons par un cas simple : la somme des éléments d'une liste par réduction avec l'addition et son élément neutre : 0.

```
def plus(a : float, b : float) -> float:
    """Retourne l'addition de a et b."""
    return a + b
```

```
>>> reduce(plus, 0, [1, 2, 3, 4, 5])
15
```

```
>>> reduce(plus, 0, [])
0
```

Dans le même genre d'idée on peut calculer le produit des éléments d'une liste.

```
def fois(a : float, b : float) -> float:
    """Retourne le produit de a et b."""
    return a * b
```

```
>>> reduce(fois, 1, [1, 2, 3, 4, 5])
120
```

```
>>> reduce(fois, 1, [])
1
```

On peut même calculer la longueur d'une liste par réduction, même si c'est un peu tiré par les cheveux (et bien moins efficace que d'utiliser `len`).

```
def incr_un(n : int,m : int) -> int:
    """Retourne toujours `m+1` quelque soit l'argument `n`."""
    return m + 1
```

```
>>> reduce(incr_un, 0, [1, 2, 3, 4, 5])
5
```

```
>>> reduce(incr_un, 0, [])
0
```

Python fournit également une notation spécifique, appelée `lambda`, permettant de définir des fonctions simples de façon *anonyme*.

```
>>> reduce(lambda a,b: a + b, 0, [1, 2, 3, 4, 5])
15
```

```
>>> reduce(lambda a,b: a * b, 1, [1, 2, 3, 4, 5])
120
```

```
>>> reduce(lambda n,m: m + 1, 0, [1, 2, 3, 4, 5])
5
```

Dans la définition de la fonctionnelle `reduce`, la principale difficulté concerne à nouveau la signature de la fonction, qui est la suivante :

```
(T * U -> U) * U * List[T] -> U
```

Cette signature s'interprète de la façon suivante.

Dans `reduce(op, en, L)` :

- le premier paramètre `op` est de type `T * U -> U` qui correspond à une fonction binaire combinant une donnée de type `T` et une donnée de type `U` pour produire une donnée de type `U`.
- le second paramètre `en` est de type `U` qui correspond à un élément neutre «à droite» pour `op`
- le troisième paramètre `l` est une liste de type `List[T]`
- la valeur de retour de `reduce` est de type `U`.

Par exemple, dans l'expression :

```
reduce(plus, 0, [1, 2, 3, 4, 5])
```

- le premier argument `plus` pour `op` est de type `int * int -> int` (donc `T=int` et `U=int`)
- le deuxième argument `0` pour `en` est de type `int` (donc `U` pour `U=int`) . Il s'agit de l'élément neutre de `plus`.
- le troisième argument `[1, 2, 3, 4, 5]` pour `l` est de type `List[int]` (donc `List[T]` pour `T=int`)
- la valeur de retour est `15` de type `List[int]` (donc `U` pour `U=int`)

Question : pouvez-vous trouver un exemple de réduction avec `T` et `U` différents ? (idée : prendre une liste de chaînes de caractères en entrée, et renvoyer une entier après réduction)

Pour les transformations et filtrages, dans la plupart des situations on adopte les compréhensions qui sont particulièrement concises. Cependant, la généralisation des réductions et d'autres types de traitement passent nécessairement par des fonctionnelles. Les modules `itertools` et `functools` de la bibliothèque standard de Python offrent un certain nombre de fonctionnalités relatives aux fonctionnelles.

Chapitre 9

Ensembles

La notion d'*ensemble* est fondamentale en mathématique. De fait, l'essentiel des constructions mathématiques sont basées sur la *théorie des ensembles*. Nous verrons dans ce chapitre que les ensembles ont également un intérêt pratique en informatique, et sont supportés nativement par Python.

9.1 Définition et opérations de base

En restant à un niveau essentiellement informatique, posons-nous la question :

> Qu'est-ce-qu'un ensemble ?

D'après Wikipedia (cf. http://fr.wikipedia.org/wiki/Ensemble) :

> Un **ensemble** désigne *intuitivement* une collection d'objets (les **éléments** de l'ensemble).

En ce qui concerne Python, nous adopterons la définition suivante.

Définition : un **ensemble** de type `Set[T]`, où `T` est une *variable de type*, est une collection d'**éléments** de type `T` et tous *distincts* (au sens de l'égalité de Python `==`)

Pour des raisons techniques, le type `T` des éléments d'un ensemble doit être *immutable* (ou *hashable* selon la terminologie Python). A l'exception des listes et des ensembles eux-mêmes, tous les types que nous avons vus dans les chapitres précédents répondent à cette contrainte. On retiendra donc :

> Aucun type liste ou ensemble ne peut apparaître dans le T de `Set[T]`.

Nous reviendrons un peu plus tard sur cette notion de type mutable vs. immutable.

9.1.1 Construction explicite d'un ensemble

Pour construire un ensemble, Python propose une syntaxe inspirée de la notation mathématique usuelle.

L'expression `{`e_1`,`e_2`,...,` e_n`}` où tous les e_1, ..., e_n sont des expressions de type `T`.

construit un ensemble de type `Set[T]` dont les éléments sont les valeurs des expressions e_1, ..., e_n.

Par exemple, pour construire l'ensemble $\{2, 5, 9\}$, c'est-à-dire l'ensemble contenant les éléments entiers 2, 5 et 9, on peur par exemple utiliser l'expression suivante :

```
>>> {2, 3+2, 9}
{9, 2, 5}
```

Ici, l'ensemble construit est de type `Set[int]`. On remarque la valeur 5 comme élément de l'ensemble construit, qui correspond à l'évaluation de l'expression `3+2`.

La construction de l'**ensemble vide** qui se note $\emptyset$ en mathématiques se note de façon spécifique en Python :

```
>>> set()
set()
```

Attention : Pour des raisons historiques , la notation `{}` qui semblerait adaptée ici est en fait réservée aux dictionnaires vides (cf. le prochain chapitre).

On peut bien sûr construire des ensembles contenant des éléments autres que des nombres, comme par exemple un ensemble de caractères :

```
>>> {'e', 'u', 'a'}
{'u', 'e', 'a'}
```

Le type de cet ensemble est `Set[str]`.

On observe dans les exemples ci-dessus un aspect important des ensembles : leurs éléments ne sont pas ordonnés. Plus précisément, les éléments sont ordonnés (car en informatique tout est bien ordonné), mais de façon arbitraire. Ainsi, contrairement aux intervalles d'entiers, aux chaînes de caractères et aux listes, les ensembles ne sont pas des séquences. Cela signifie également qu'il n'existe pas de notion d'indice pour les ensembles, et donc pas de notion associée de déconstruction.

Le second aspect fondamental des ensembles est que les éléments qui les composent doivent être tous distincts. Ainsi, si l'on essaye de répéter des éléments, comme dans l'expression ci-dessous :

```
>>> { 'e', 'a', 'a', 'u', 'e' }
{'u', 'e', 'a'}
```

Python ne retient que les éléments distincts. Donc, contrairement aux listes, on a *au plus* une occurrence de chaque élément dans un ensemble. C'est sans doute la propriété

la plus importante des ensembles, puisqu'il s'agit de la principale fonctionnalité des ensembles en informatique : éliminer efficacement les doublons.

Finalement, la dernière propriété importante est que les éléments d'un ensemble de type `Set[T]` doivent *tous* être du même type `T`. Les ensemble que l'on considère doivent êtres *homogènes*, comme pour les listes. Mais en l'absence de contrôle de type, Python ne vérifie pas cette propriété, c'est donc au programmeur qu'incombe la responsabilité de garantir que les éléments contenus dans un ensemble de type `Set[T]` soient bien tous du *même* type `T`.

9.1.2 Test d'appartenance

L'opération élémentaire des ensembles est le *test d'appartenance*, qui en mathématique s'écrit $e \in E$ pour signifier l'appartenance (ou non) de l'élément e à l'ensemble E.

En Python, ceci se traduit par une **expression d'appartenance** de type `bool` qui s'écrit :

```
<elem> in <ensemble>
```

où :

— `<elem>` est une expression de type `T`
— `<ensemble>` est une expression de type `Set[T]`

Le **principe d'évaluation** associé est simple. La valeur du test d'appartenance est :

— `True` si la valeur de `<elem>` appartient bien à l'ensemble `<ensemble>`
— `False` sinon

Par exemple :

```
>>> 2 in {2, 5, 9}
True
```

qui traduit le fait que mathématiquement, l'énoncé $2 \in \{2, 5, 9\}$ est vrai.

```
>>> 3 in {2, 5, 9}
False
```

```
>>> 'a' in {'a', 'e', 'u'}
True
```

```
>>> 'o' in {'a', 'e', 'u'}
False
```

Si l'on désire tester la *non-appartenance* d'un élément à un ensemble, on peut utiliser :

```
not (<elem> in <ensemble>)
```

Mais comme en mathématique on peut écrire $e \notin E$ pour énoncer que l'élément e n'appartient pas à l'ensemble E, il existe également une syntaxe spécifique en Python :

```
<elem> not in <ensemble>
```

Par exemple :

```
>>> 2 not in {2, 5, 9}
False
```

qui traduit le fait que mathématiquement l'énoncé $2 \notin \{2, 5, 9\}$ est faux.

```
>>> 3 not in {2, 5, 9}
True
```

L'intérêt majeur du test d'appartenance (ou de non-appartenance) sur les ensembles en Python est qu'il est d'une grande efficacité.

Si l'on compare aux listes, il faut (dans le pire cas) de l'ordre de n tests d'égalité pour vérifier si un élement appartient à une liste de longueur n alors qu'il faut dans la plupart des cas un nombre constrant (et réduit) d'opérations pour effectuer un test d'appartenance dans un ensemble de taille même très importante

9.1.3 Ajout d'un élement

De façon similaire aux listes, l'**ajout d'un élément dans un ensemble** est une opération critique qui doit être effectuée de la manière la plus efficace possible. On utilise pour cela la méthode `add` selon la syntaxe :

```
<ensemble>.add(<elem>)
```

où `<ensemble>` est un ensemble de type `Set[T]` et `<elem>` une expression de type `T`.

Le **principe d'interprétation** associé est la modification de l'`<ensemble>` pour ajouter l'`<element>`. Cette modification se fait par *effet de bord* c'est-à-dire directement en mémoire, sans reconstruction d'un ensemble résultat. En conséquence, l'opération `add` retourne la valeur `None`.

Pour observer ces phénomènes, considérons la définition de variable suivante :

```
from typing import Set

mon_ensemble : Set[str]
mon_ensemble = {'a', 'e', 'u'}
```

En premier lieu, remarquons l'importation explicite du symbole `Set` depuis le module `typing`.

Demandons à Python la valeur de la variable `mon_ensemble` :

```
>>> mon_ensemble
{'a', 'e', 'u'}
```

Ajoutons maintenant un élément :

```
>>> mon_ensemble.add('o')
```

Ici on observe que la méthode `add` ne retourne rien (en fait juste `None` que l'interprète n'affiche pas). En revanche, l'ensemble référencé par la variable `mon_ensemble` a été modifié directement en mémoire, comme l'atteste l'exemple ci-dessous :

```
>>> mon_ensemble
{'a', 'e', 'o', 'u'}
```

Que se passe-t-il maintenant si l'on souhaite ajouter «un autre» `o` à l'ensemble ?

```
>>> mon_ensemble.add('o')
```

```
>>> mon_ensemble
{'a', 'e', 'o', 'u'}
```

Bien sûr ! L'ensemble est inchanté puisque les doublons sont proscrits. C'est tout l'intérêt de l'opération `add` qui n'ajoute l'élément que si il n'est pas déjà présent dans l'ensemble.

En pratique, on utilise la méthode `add` pour (re)construire des ensembles dans des fonctions dont le type de retour est de la forme `Set[T]`.

En guise d'illustration, donnons une définition de la fonction `presences` qui étant donnée une liste `l` d'entiers retourne l'ensemble des entiers qui apparaissent au moins une fois dans `l`.

```
from typing import Set, List

def presences(l : List[int]) -> Set[int]:
    """Retourne l'ensemble des entiers qui apparaissent
    au moins une fois dans l."""

    # Ensemble résultat
    ens : Set[int] = set()  # Initialement vide

    n : int
    for n in l:
        ens.add(n)

    return ens
```

```
# Jeu de tests
assert presences([9, 11, 11, 2, 5, 9, 2, 2, 1, 3]) == {1, 2, 3, 5, 9, 11}
assert presences([1, 2, 3, 4]) == {1, 2, 3, 4}
assert presences([2, 2, 2, 2]) == {2}
assert presences([]) == set()
```

On voit bien avec cette fonction que les éléments répétés dans les listes ne «comptent» qu'une seule fois dans l'ensemble retourné.

9.1.4 Complément : retrait d'un élement

L'opération complémentaire de **retrait d'un élément dans un ensemble** est également disponible. Si le retrait d'un élément dans un liste est coûteux, il s'agit d'une opération très efficace sur les ensembles. On utilise dans ce cas la méthode `remove` selon la syntaxe :

```
<ensemble>.remove(<elem>)
```

où `<ensemble>` est un ensemble de type `Set[T]` et `<elem>` une expression de type `T`.

Le **principe d'interprétation** associé est la modification de l'`<ensemble>` pour enlever l'`<element>`. Cette modification se fait par *effet de bord* c'est-à-dire directement en mémoire, commme pour l'opération `add` (et la valeur retournée est `None`). Si l'élément n'est pas déjà présent dans l'ensemble, l'opération `remove` reste sans effet.

Pour observer ces phénomènes, reprenons la variable `mon_ensemble` définie ci-dessus :

```
>>> mon_ensemble
{'a', 'e', 'o', 'u'}
```

Extrayons maintenant la lettre `'e'` de l'ensemble.

```
>>> mon_ensemble.remove('e')
```

Encore une fois, Python n'affiche rien, ce qui traduit le fait que le retrait a été effectué en mémoire et qu'aucun ensemble n'a été reconstruit. Regardons maintenant l'ensemble référencé par la variable `mon_ensemble` :

```
>>> mon_ensemble
{'a', 'o', 'u'}
```

L'élément a bien été retiré de l'ensemble.

Essayons maintenant de le retirer «une deuxième fois».

```
>>> mon_ensemble.remove('e')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
    ...
----> 1 mon_ensemble.remove('e')

KeyError: 'e'
```

Une erreur est signalée dans ce cas, donc on retriendra que l'opération `remove` ne doit être utilisée que lorsque l'on est sûr que l'élément est bien présent, par exemple en utilisant le test d'appartenance.

Dans le reste de cet ouvrage, nous n'utiliserons pas l'opération `remove`. Nous procéderons systématiquement par (re)construction d'un ensemble avec l'opération `add`.

9.1.5 Itération sur les ensembles

Puisque leurs éléments sont ordonnés de façon arbitraire et non séquentielle, les ensembles ne sont pas des séquences. Cependant, la boucle `for` d'itération reste disponible pour les ensembles. La différence fondamentale est que contrairement aux séquences, l'ordre de parcours de l'ensemble est lui-même arbitraire. La seule propriété que l'on peut exploiter du parcours est que tous les éléments de l'ensemble on été visités exactement une fois.

La syntaxe des itérations sur les ensemble est la même que pour les séquences :

```
<var> : T
for <var> in <ensemble>:
     <corps>
```

où :

- `<ensemble>` est une expression d'ensemble de type `Set[T]`
- `<var>` est la *variable d'itération d'élément* de type `T`
- `<corps>` est une suite d'instructions

Le **principe d'interprétation** est le suivant :

- le `<corps>` de la boucle est une suite d'instructions qui est exécutée une fois pour chaque élément de l'`<ensemble>`, selon un ordre arbitraire.
- pour chaque tour de boucle (donc pour chaque élément de l'ensemble), dans le `<corps>` de la boucle la variable `<var>` est liée à l'élément concerné.
- la variable `<var>` n'est plus utilisable après le dernier tour de boucle.

Illustrons ce principe d'itération sur le problème très simple du calcul de la somme des éléments d'un ensemble d'entiers.

```python
def somme_ensemble(ens : Set[int]) -> int:
    """Renvoie la somme des éléments de ens.
    """
    # Somme
    s : int = 0

    n : int # Elément courant
    for n in ens:
        s = s + n

    return s
```

```python
# Jeu de tests
assert somme_ensemble({1, 3, 9, 24, 5}) == 42
assert somme_ensemble({1, 3, 1, 9, 3, 24, 5, 24, 5}) == 42
assert somme_ensemble({-2, -1, 0, 1, 2}) == 0
assert somme_ensemble(set()) == 0
```

Le deuxième cas de test ci-dessus un particulièrement intéressant. On voit bien que

la somme 42 obtenue pour l'ensemble $\{1, 3, 1, 9, 3, 24, 5, 24, 5\}$ est la même que celle obtenue pour l'ensemble $\{1, 3, 9, 24, 5\}$. La raison est évidente : il s'agit exactement des mêmes ensembles puisque les répétitions sont éliminées des ensembles. On peut bien sûr confirmer cela par de simples interactions :

```
>>> {1, 3, 9, 24, 5}
{1, 3, 5, 9, 24}
```

```
>>> {1, 3, 1, 9, 3, 24, 5, 24, 5}
{1, 3, 5, 9, 24}
```

En fait, les deux ensembles sont égaux au sens de l'*égalité ensembliste*, ce que l'on peut vérifier en Python :

```
>>> {1, 3, 9, 24, 5} == {1, 3, 1, 9, 3, 24, 5, 24, 5}
True
```

Mais cette notion requiert quelques explications, alors enchaînons ...

9.1.6 Egalité ensembliste et notion de sous-ensemble

La notion de **sous-ensemble** est fondamentale en théorie des ensembles. Mathématiquement, on dirait qu'un ensemble ens_1 est sous-ensemble d'un ensemble ens_2, ce que l'on note : $ens_1 \subseteq ens_2$ si et seulement si $\forall e \in ens_1, e \in ens_2$

Par exemple :

- l'énoncé $\{2, 5\} \subseteq \{2, 5, 9\}$ est vrai
- l'énoncé $\{5\} \subseteq \{2, 5, 9\}$ est vrai
- l'énoncé $\emptyset \subseteq \{2, 5, 9\}$ est vrai
- mais l'énoncé $\{2, 8\} \subseteq \{2, 5, 9\}$ est faux

On peut définir la relation $\subseteq$ en Python, de la façon suivante :

```
from typing import Set, TypeVar
T = TypeVar('T')

def est_sous_ensemble(ens1 : Set[T], ens2 : Set[T]) -> bool:
    """Renvoie True si ens1 est sous-ensemble de ens2, False Sinon.
    """
    e : T
    for e in ens1:
        if e not in ens2:
            return False
    return True
```

Remarque : cettte fonction accepte deux ensembles de type `Set[T]` quel que soit le type `T` concerné, on introduit donc la variable de type `T` dans la signature.

```
# Jeu de tests
assert est_sous_ensemble({2, 5}, {2, 5, 9}) == True
```

```
assert est_sous_ensemble({5}, {2, 5, 9}) == True
assert est_sous_ensemble(set(), {2, 5, 9}) == True
assert est_sous_ensemble({2, 8}, {2, 5, 9}) == False
```

Cette relation *sous-ensemble* correspond en fait à l'odre naturel sur les ensembles, et donc en Python plutôt que d'écrire :

```
est_sous_ensemble(ens1, ens2)
```

On peut utiliser de façon équivalente (et préférée) l'expression suivante :

```
ens1 <= ens2
```

Par exemple :

```
>>> {2, 5} <= {2, 5, 9}
True
```

```
>>> {5} <= {2, 5, 9}
True
```

```
>>> set() <= {2, 5, 9}
True
```

```
>>> {2, 8} <= {2, 5, 9}
False
```

Les autres comparateurs correspondent à leur équivalent mathématique :

- `ens1 < ens2` correspond au comparateur *sous-ensemble strict* $ens_1 \subset ens_2$
- `ens1 >= ens2` correspond au comparateur *sur-ensemble* $ens_1 \supseteq ens_2$
- `ens1 > ens2` correspond au comparateur *sur-ensemble strict* $ens_1 \supset ens_2$

En mathématiques, deux ensembles ens_1 et ens_2 sont égaux si et seulement ils contiennent exactement les mêmes éléments, c'est-à-dire :

- $\forall e_1 \in ens_1, e_1 \in ens_2$, et
- $\forall e_2 \in ens_2, e_2 \in ens_1$

Par exemple :

- $\{5, 2, 9\}$ est égal à $\{2, 5, 9\}$
- mais $\{5, 2\}$ n'est pas égal à $\{2, 5, 9\}$
- et $\{5, 2, 9, 8\}$ n'est pas égal à $\{2, 5, 9\}$

Autrement dit, les deux ensembles sont égaux si et seulement si $ens_1 \subseteq ens_2$ *et* $ens_2 \subseteq ens_1$.

On peut donc directement traduire cette dernière définition de la **relation d'égalité sur les ensembles** en Python :

```
def ensembles_egaux(ens1 : Set[T], ens2 : Set[T]) -> bool:
    """Renvoie True si ens1 et ens2 sont égaux, False Sinon.
    """
    return est_sous_ensemble(ens1, ens2) and est_sous_ensemble(ens2, ens1)
```

```
# Jeu de tests
assert ensembles_egaux({5, 2, 9}, {2, 5, 9}) == True
assert ensembles_egaux({5, 2}, {2, 5, 9}) == False
assert ensembles_egaux({5, 2, 9, 8}, {2, 5, 9}) == False
```

Sans suprise, l'opérateur d'égalité sur les ensembles est prédéfini en Python, de sorte que :

```
ensembles_egaux(ens1, ens2)
```

s'écrit plus simplement (et de façon préférée) :

```
ens1 == ens2
```

De même, le test d'inégalité s'écrit tout simplement :

```
ens1 != ens2
```

Par exemple :

```
>>> {5, 2, 9} == {2, 5, 9}
True
```

```
>>> {5, 2, 9} != {2, 5, 9}
False
```

```
>>> {5, 2} == {2, 5, 9}
False
```

```
>>> {5, 2} != {2, 5, 9}
True
```

9.2 Opérations ensemblistes

Pour terminer notre introduction aux ensembles, nous allons présenter les **opérations ensemblistes** les plus fondamentales : l'union, l'intersection et la différence entre deux ensembles. Ceci nous permettra à la fois de réviser ces notions mathématiques fondamentales, mais également de mettre en œuvre les opérations de base sur les ensembles que nous avons introduites dans les sections précédentes.

9.2.1 Union

L'union ensembliste fait sans doute partie des opérations les plus fréquemment employées dans la vie de tous les jours :

> *«Tu n'oublieras pas de ranger tes chaussettes et tes culottes dans ton tiroir de commode»*

En mathématiques, l'**union** entre deux ensembles ens_1 et ens_2 se note: $ens_1 \cup ens_2$. La propriété fondamentale de l'union est que si un élément $e \in ens_1 \cup ens_2$ alors :

— e appartient à ens_1 (ex.: *«e est une chaussette»*)
— **ou** e appartient à ens_2 (ex.: *«e est une culotte»*)

Remarque : il est possible que l'élément e appartienne simultanément à ens_1 et ens_2 (une *«chaussette-culotte»*?) et on dit alors qu'il est dans leur *intersection*, nous y reviendrons.

Par exemple :

— $\{2,5,9\} \cup \{1,3,4,8\} = \{1,2,3,4,5,8,9\}$
— $\{1,2,5,9\} \cup \{1,3,4,8\} = \{1,2,3,4,5,8,9\}$

Dans ce deuxième exemple l'élément 1 est présent dans les deux ensembles dont on forme l'union.

D'un point de vue informatique, l'union $ens_1 \cup ens_2$ consiste à composer un nouvel ensemble regroupant tous les éléments de ens_1 et tous les éléments de ens_2. Ceci se traduit très simplement en Python.

```python
def union(ens1 : Set[T], ens2 : Set[T]) -> Set[T]:
    """Retourne l'union des ensembles ens1 et ens2.
    """
    # Ensemble résultat
    ens : Set[T] = set()

    e1 : T  # élément de ens1
    for e1 in ens1:
        ens.add(e1)

    e2 : T # élément de ens2
    for e2 in ens2:
        ens.add(e2)

    return ens
```

```python
# Jeu de test
assert union({'a','e','u'}, {'o', 'i'}) == {'a', 'e', 'i', 'o', 'u'}
assert union({2, 5, 9}, {1, 3, 4, 8}) == {1, 2, 3, 4, 5, 8, 9}
assert union({1, 2, 5, 9}, {1, 3, 4, 8}) == {1, 2, 3, 4, 5, 8, 9}
assert union({1, 2, 5, 9}, set()) == {1, 2, 5, 9}
assert union(set(), {1, 2, 5, 9}) == {1, 2, 5, 9}
```

L'union ensembliste est une opération fondamentale, et ce n'est donc pas étonnant de la trouver prédéfinie en Python. Ainsi :

```
union(ens1, ens2)
```

s'écrit plus simplement (et de façon préférée) :

```
ens1 | ens2
```

Voici quelques exemples d'utilisation de cet opérateur d'union ensembliste :

```
>>> {'a','e','u'} | {'o', 'i'}
{'a', 'e', 'i', 'o', 'u'}
```

```
>>> {2, 5, 9} | {1, 3, 4, 8}
{1, 2, 3, 4, 5, 8, 9}
```

```
>>> {1, 2, 5, 9} | {1, 3, 4, 8}
{1, 2, 3, 4, 5, 8, 9}
```

9.2.2 Intersection

L'intersection ensembliste est également une opération courante dans la vie de tous les jours.

> *«Tu mettras tes chaussettes rouges et à pois dans le tiroir du haut»*

On parle ici de l'intersection des ensembles *«chaussettes rouges»* et *«chaussettes à pois»*. En mathématiques, l'**intersection** entre deux ensembles ens_1 et ens_2 se note : $ens_1 \cap ens_2$.

La propriété fondamentale de l'intersection est que si un élément $e \in ens_1 \cap ens_2$ alors :

— e appartient à ens_1 (ex.: *«e est une chaussette rouge»*)
— **et** e appartient à ens_2 (ex.: *«e est une chaussette à pois»*)

Par exemple :

— $\{2,5,9\} \cap \{1,2,3,4,5,8\} = \{2,5\}$
— $\{2,5,9\} \cap \{2,3,4,8\} = \{2\}$
— $\{2,5,9\} \cup \{1,3,4,8\} = \emptyset$

D'un point de vue informatique, l'intersection $ens_1 \cap ens_2$ consiste à composer un nouvel ensemble en prenant tous les éléments de ens_1 qui sont également présent dans ens_2.

Remarque : on pourrait de façon symétrique prendre tous les éléments de ens_2 qui sont également dans ens_1.

En Python, cet algorithme (ou son symétrique) s'exprime très simplement.

```
def intersection(ens1 : Set[T], ens2 : Set[T]) -> Set[T]:
    """Retourne l'intersection des ensembles ens1 et ens2.
    """
    # Ensemble résultat
    ens : Set[T] = set()

    e1 : T  # Elément de ens1
    for e1 in ens1:
        if e1 in ens2:
            ens.add(e1)
```

```
    return ens

# Jeu de tests
assert intersection({'a', 'i', 'o'}, {'u', 'i', 'o', 'y'}) == {'o', 'i'}
assert intersection({2, 5, 9}, {1, 2, 3, 4, 5, 8}) == {2, 5}
assert intersection({2, 5, 9}, {2, 3, 4, 8}) == {2}
assert intersection({2, 5, 9}, {1, 3, 4, 8}) == set()
assert intersection({2, 5, 9}, set()) == set()
assert intersection(set(), {2, 5, 9}) == set()
```

On ne s'étonnera pas du fait que l'intersection ensembliste est également prédéfinie en Python.

Ainsi :

```
intersection(ens1, ens2)
```

s'écrit plus simplement (et de façon préférée) :

```
ens1 & ens2
```

Par exemple :

```
>>> {'a', 'i', 'o'} & {'u', 'i', 'o', 'y'}
{'i', 'o'}
```

```
>>> {2, 5, 9} & {1, 2, 3, 4, 5, 8}
{2, 5}
```

```
>>> {2, 5, 9} & {2, 3, 4, 8}
{2}
```

```
>>> {2, 5, 9} & {1, 3, 4, 8}
set()
```

9.2.3 Différence

La différence ensembliste est la dernière opération que nous allons reconstruire dans ce chapitre.

> *«Il faudra trier les chaussettes sans trous, et jeter les autres»*

La **différence** entre deux ensembles ens_1 et ens_2 se note traditionnellement $ens_1 \setminus ens_2$ et consiste simplement à prendre les éléments de ens_1 (ex.: *«toutes les chaussettes»*) et à en «retirer» les éléments qui sont également dans ens_2 (ex.: *«les chaussettes trouées»*).

Par exemple :

— $\{2,5,9\} \setminus \{2,3,4,8\} = \{5,9\}$ (on a juste «retiré» l'élément 2)
— $\{2,5,9\} \setminus \{2,3,5,8\} = \{9\}$ (on a «retiré» les éléments 2 et 5)
— $\{2,5,9\} \setminus \{2,3,5,8,9\} = \emptyset$ (on a «retiré» les éléments 2, 5 et 9)

On peut à nouveau traduire ce principe en Python :

```
def difference(ens1 : Set[T], ens2 : Set[T]) -> Set[T]:
    """Retourne la difference des ensembles E1 et E2.
    """
    # Ensemble résultat
    ens : Set[T] = set()

    e1 : T
    for e1 in ens1:
        if e1 not in ens2:
            ens.add(e1)

    return ens
```

```
# Jeu de tests
assert difference({'a', 'i', 'o', 'y'}, {'u', 'i', 'o'}) == {'a', 'y'}
assert difference({ 2, 5, 9 }, {2, 3, 4, 8}) == {5, 9}
assert difference({ 2, 5, 9 }, {2, 3, 5, 8}) == {9}
assert difference({ 2, 5, 9 }, {2, 3, 5, 8, 9}) == set()
assert difference({ 2, 5, 9 }, {2, 5, 9}) == set()
assert difference({2, 5, 9}, set()) == {2, 5, 9}
assert difference(set(), {2, 5, 9}) == set()
```

L'opérateur de différence est proche dans les principe de la soustraction, ce qui explique que :

`difference(ens1, ens2)`

s'écrit plus simplement (et à nouveau de façon préférée) :

`ens1 - ens2`

Par exemple :

```
>>> {'a', 'i', 'o', 'y'} - {'u', 'i', 'o'}
{'a', 'y'}
```

```
>>> { 2, 5, 9 } - {2, 3, 4, 8}
{5, 9}
```

```
>>> { 2, 5, 9 } - {2, 3, 5, 8}
{9}
```

```
>>> { 2, 5, 9 } - {2, 3, 5, 8, 9}
set()
```

Chapitre 10

Dictionnaires

Les dictionnaires sont sans doute, avec les listes, les structures de données les plus couramment utilisées en Python.

10.1 Définition et opérations de base

Un dictionnaire Python exprime une relation entre un ensemble de *clés* - dites *clés de recherche* - et un ensemble de *valeurs associées*. Cette relation est mathématiquement une *application* qui impose qu'à chaque clé corresponde exactement une valeur. On appelle *association* le couple formé d'une clé et de sa valeur.

Définition : un **dictionnaire** de type `Dict[K,V]`, où `K` et `V` sont des *variables de type*, est un ensemble d'assocations entre :

— une **clé** (ou *clé de recherche*) de type `K`
— une **valeur** de type `V` associée à la clé

On remarque la proximité entre la notion d'ensemble et celle de dictionnaire. En effet, les clés du dictionnaires forment un ensemble au sens de ce que nous avons présenté dans le chapitre précédent, nous y reviendrons.

Important : puisque les clés d'un dictionnaire forment un ensemble, le type `K` de `Dict[K,V]` doit être un type valide pour les ensembles, c'est à dire immutable (ou *hashable* dans la terminologie Python). En particulier, on ne pourra référencer de type liste, ensemble ou dictionnaire dans le type des clés, car ce sont des structures de données mutables. En pratique, on prendra souvent des chaînes de caractères, des nombres ou des n-uplets composés de chaînes, de nombres et de booléens. Il n'y a pas de telle restriction sur le type `V` des valeurs.

10.1.1 Construction explicite d'un dictionnaire

La construction explicite d'un dictionnaire consiste à énumérer l'ensemble des associations clé/valeur.

L'expression `{`k_1`:`v_1`,`k_2`:`v_2`,...,` k_n`:`v_n`}`

où

- tous les k_1, ..., k_n sont des expressions pour les clés de type `K` (*hashable*).
- tous les v_1, ..., v_n sont des expressions pour les valeurs associées de type `V`.

exprime la construction explicite d'un dictionnaire `d` de type `Dict[K,V]`.

Pour illustrer cette syntaxe, voic un exemple de construction d'un dictionnaire (très partiel et approximatif) de constantes mathématiques :

```
{'pi' : 3.14159265, 'sqrt2' : 1.41421356237, 'phi' : 1.6180339887 }
```

Le type de ce dictionnaire est `Dict[str,float]` c'est-à-dire une association entre :

- des noms de constantes mathématiques représentées par des chaînes de caractères pour les clés
- et leur valeur numérique sous la forme d'un flottant.

Il est aussi possible de construire un **dictionnaire vide** avec l'expression suivante :

```
>>> dict()
{}
```

On remarque que Python affiche `{}` pour le dictionnaire vide, qui est effectivement une expression permise. Cependant, nous allons privilégier l'expression `dict()` qui a le mérite d'être plus explicite sur le fait que l'on crée un dictionnaire, et surtout pour éviter toute ambiguïté avec l'ensemble vide que l'on doit écrire `set()` (alors que `{}` semblerait également adéquat puisque c'est une écriture mathématique usuelle).

Remarque : il serait possible d'«imiter» un dictionnaire Python de type `Dict[K,V]` avec une liste de type `List[Tuple[K,V]]`. Mais en pratique on ne le fait pas car cette «imitation» de dictionnaire est bien moins efficace.

Avant de passer à la suite, nous allons conserver notre dictionnaire des constantes en l'affectant à une variable permettant de le référencer dans les exemples.

```
from typing import Dict

constantes : Dict[str,float]
constantes =  {'pi' : 3.14159265,
               'sqrt2' : 1.41421356237,
               'phi' : 1.6180339887 }
```

Nous remarquons l'import explicite du symbole `Dict` depuis le module `typing`.

10.1.2 Accès aux valeurs

Lorsque nous utilisons un dictionnaire commun, le cas d'utilisation typique est de rechercher une définition à partir d'un mot. Transposé en termes de dictionnaire Python, ce cas d'utilisation consiste à recherche une valeur à partir de sa clé. La syntaxe de l'*accès à une valeur* à partir de sa clé s'écrit :

```
<dictionnaire>[<clé>]
```

où

- `<dictionnaire>` est une expression de type `Dict[K,V]`
- `<clé>` est une expression de type `K`

Le principe d'évaluation est le suivant :

- si la `<clé>` est présente dans le `<dictionnaire>`, alors la valeur associée de type `V` est retournée
- sinon, si la `<clé>` n'est pas présente, alors une erreur est signalée par l'interprète Python.

Remarque : il faudra savoir *à l'avance* si une clé est présente ou non dans un dictionnaire, ce que nous pourrons tester explicitement (cf. section suivante).

Exploitons cette syntaxe d'accès aux valeurs pour récupérer la valeur numérique associée à la constante de nom `'pi'` dans notre dictionnaire.

```
>>> constantes['pi']
3.14159265
```

Ici, la variable `Constantes` référence un dictionnaire de type `Dict[str,float]` et la chaîne `'pi'` (qui est bien de type `str`) fait partie de l'ensemble des clés de ce dictionnaire. On obtient donc la valeur associée de type `float`.

En revanche, l'expression suivante conduit au signalement d'une erreur.

```
>>> constantes['e']
```

```
KeyError                                  Traceback (most recent call last)
   ...
----> 1 Constantes['e']

KeyError: 'e'
```

De façon similaire au test d'appartenance d'un élément dans un ensemble, le grand intérêt de l'opération d'accès à une valeur à partir d'une clé dans un dictionnaire est sa grande efficacité. Dans la plupart des cas pratiques, cette opération s'effectue en un nombre d'étapes constant (et réduit), indépendemment de la taille du dictionnaire (dans une certaine mesure).

10.1.3 Test d'appartenance d'une clé

Nous l'avons vu ci-dessus, avant de pouvoir accéder à une valeur associée à une clé dans un dictionnaire, il faut préalablement vérifier si cette clé est bien présente. Si l'on construit explicitement le dictionnaire alors on peut déduire cette information simplement, mais si le dictionnaire est construit de façon algorithmique (nous verrons une telle construction dans la suite du chapitre) alors il est nécessaire de pouvoir tester l'appartenance d'une clé à un dictionnaire.

La syntaxe employée est la suivante :

```
<clé> in  <dictionnaire>
```

où

- `<dictionnaire>` est une expression de type `Dict[K,V]`
- `<clé>` est une expression de type `K`

Le principe d'évaluation associé est simple :

- la valeur `True` est retournée si la `<clé>` est présente dans le `<dictionnaire>`
- sinon (si la clé n'est pas présente), alors la valeur `False` est retournée.

De façon complémentaire, pour tester si une clé n'est pas présente dans un dictionnaire alors on utilise la syntaxe suivante :

```
<clé> not in  <dictionnaire>
```

Par exemple :

```
>>> 'pi' in constantes
True
```

```
>>> 'pi' not in constantes
False
```

```
>>> 'e' in constantes
False
```

```
>>> 'e' not in constantes
True
```

Remarque : le test d'appartenance d'une clé à un dictionnaire revient à tester l'appartenance de cette clé à l'ensemble des clés du dictionnaire, il s'agit donc d'une opération très efficace.

10.1.4 Ajout d'une association

Pour ajouter ou remplacer une association dans un dictionnaire, on utilise la syntaxe suivante :

```
<dictionnaire>[<clé>] = <valeur>
```

où

— `<dictionnaire>` est une expression de dictionnaire de type `Dict[K,V]`
— `<clé>` est une expression de type `K`
— `<valeur>` est une expression de type `V`

Le principe d'interprétation associé est le suivant :

— si la `<clé>` n'appartient pas initialement au `<dictionnaire>` alors l'assocation `<clé>:<valeur>` est ajoutée au dictionnaire par *effet de bord* donc directement en mémoire
— si en revanche la `<clé>` appartient déjà au `<dictionnaire>` alors l'association préexistante pour `<clé>` est remplacée par la nouvelle association `<clé>:<valeur>`

Remarque : l'ajout/remplacement est une instruction, aucun valeur n'est donc retournée.

Rappelons le contenu de notre dictionnaire de `Constantes` :

```
>>> constantes
{'pi': 3.14159265, 'sqrt2': 1.41421356237, 'phi': 1.6180339887}
```

Nous allons commencer par ajouter une association pour la constante mathématique `'e'` :

```
>>> constantes['e'] = 2.71828182
```

On constate bien ici, de par l'absence d'affichage, que la modification est effectuée directement en mémoire et qu'aucune valeur significative n'est retournée par l'instruction d'ajout.

Vérifions que la modification en mémoire a bien été effectuée :

```
>>> constantes
{'pi': 3.14159265,
 'sqrt2': 1.41421356237,
 'phi': 1.6180339887,
 'e': 2.71828182}
```

Désormais, il est possible d'accéder à la valeur de la nouvelle constante :

```
>>> constantes['e']
2.71828182
```

Aucune erreur n'est maintenant signalée, puisque le test d'appartenance est valide :

```
>>> 'e' in constantes
True
```

La valeur de π enregistrée est un peu imprécise :

```
>>> constantes['pi']
3.14159265
```

Effectuons un remplacement de cette association pour ajouter quelques décimales à l'approximation :

```
>>> constantes['pi'] = 3.141592653589793
```

Pour constater le changement en mémoire, regardons le nouveau contenu du dictionnaire :

```
>>> constantes
{'pi': 3.141592653589793,
 'sqrt2': 1.41421356237,
 'phi': 1.6180339887,
 'e': 2.71828182}
```

La valeur associée à la clé `'pi'` a bien été remplacée.

Exemple : comptage d'occurrences

L'instruction d'ajout/remplacement dans un dictionnaire nous permet d'effectuer des constructions algorithmiques. Un cas d'utilisation typique des dictionnaires consiste à compter les occurrences des éléments d'une liste.

Considérons par exemple la liste suivante :

```
['b', 'c', 'b', 'c', 'j', 'd', 'b', 'j', 'a']
```

Dans cette liste (de type `List[str]`) le caractère `'b'` est par exemple répété quatre fois, et le `j` deux fois, etc. Notre objectif est de définir une fonction `compte_occurrences` qui étant donnée une liste `L` construit le dictionnaire des comptes d'occurrences associées.

Pour notre liste ci-dessus, nous obtenons le dictionnaire suivant :

```
>>> compte_occurrences(['b', 'c', 'b', 'c', 'j', 'd', 'b', 'j', 'a'])
{'a': 1, 'b': 3, 'c': 2, 'd': 1, 'j': 2}
```

Remarque : nous constatons que l'ordre entre les associations dans un dictionnaire semble arbitraire. C'est bien sûr le cas puisque l'on a défini un dictionnaire comme un ensemble d'associations, et donc, tout comme pour les ensembles, l'ordre des éléments (ici les assocations) est arbitraire [1].

La définition proposée pour la fonction de comptage d'occurrences est la suivante :

```
from typing import List, Dict

def compte_occurrences(l : List[str]) -> Dict[str,int]:
    """Retourne le compte des occurrences des éléments
    de la liste l."""

    # Dictionnaire résultat
    dico = dict()  # le dictionnaire résultat
```

1. Depuis Python 3.7, l'ordre n'est plus arbitraire et respecte en fait l'ordre d'insertion des associations. Cependant, exploiter cette propriété n'est pas une bonne idée : on risque de rendre le code incompatible avec les versions précédentes de Python. De plus, il existe une variante spécifique des dictionnaires dans la bibliothèque standard : avec le type `OrderedDict`.

```
    e : str
    for e in l:
        if e not in dico: # première occurrence
            dico[e] = 1
        else:  # k-ième occurrence, k > 1
            dico[e] = dico[e] + 1

    return dico
```

```
# Jeu de tests
assert compte_occurrences(['b', 'c', 'e', 'b', 'c',
                           'j', 'd', 'b', 'j', 'a', 'b']) \
       == {'b' : 4, 'c': 2, 'e': 1, 'j': 2, 'd': 1, 'a':1 }

assert compte_occurrences(['a', 'a', 'a', 'a', 'a']) == {'a' : 5}
assert compte_occurrences([]) == dict()
```

10.1.5 Complément : retrait d'une assocation

De façon symétrique à l'ajout, il est possible de supprimer une association d'un dictionnaire. On utilise pour cela la syntaxe suivante :

`del <dictionnaire>[<clé>]`

où

- `<dictionnaire>` est une expression de dictionnaire de type `Dict[K,V]`
- `<clé>` est une expression de type `K`

Le principe d'interprétation associé est le suivant :

- si la `<clé>` appartient au `<dictionnaire>` alors l'association correspondante est supprimée par *effet de bord* donc directement en mémoire
- sinon (si la `<clé>` n'appartient pas au `<dictionnaire>`) alors une erreur est signalée par l'interprète Python.

Encore une fois, il faut savoir à l'avance qu'une clé appartient à un dictionnaire avant de supprimer l'association correspondante.

Rappelons la valeur de notre dictionnaire de constantes mathématiques (après ajout de l'association pour la clé `'e'` et le remplacement de la valeur associée à la clé `'pi'`) :

```
>>> constantes
{'pi': 3.141592653589793,
 'sqrt2': 1.41421356237,
 'phi': 1.6180339887,
 'e': 2.71828182}
```

Supprimons maintenant l'association pour la constante de nom `'sqrt2'` :

```
>>> del constantes['sqrt2']
```

Aucun affichage n'est produit, signe que les effets de l'instruction se passent en mémoire. Regardons donc les conséquences de ces effets sur le dictionnaire de constantes :

```
>>> constantes
{'pi': 3.141592653589793, 'phi': 1.6180339887, 'e': 2.71828182}
```

L'association pour `'sqrt2'` a bien été retirée du dictionnaire. Nous pouvons constater cette suppression en essayant d'effectuer la même suppression :

```
>>> del constantes['sqrt2']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
   ...
----> 1 del Constantes['sqrt2']

KeyError: 'sqrt2'
```

Python nous signale bien que la clé (*key* en anglais) est manquante dans le dictionnaire.

10.2 Itération sur les dictionnaires

Après avoir construit un dictionnaire, une opération très courante consiste à le parcourir intégralement. Il n'est donc pas étonnant que les concepteurs du langage Python aient prévu d'étendre la boucle `for` d'itération pour répondre à ce besoin.

En fait, il existe deux types de boucles d'itérations sur les dictionnaires :

— l'itération sur les clés du dictionnaire,
— l'itération sur les associations clé/valeur.

10.2.1 Itération sur les clés

Le cas d'utilisation le plus typique sur les dictionnaire consiste à en parcourir les clés. En effet, il est assez rare d'avoir besoin de toutes les valeurs stockées dans le dictionnaire lors du parcours.

La **syntaxe de l'itération sur les clés** d'un dictionnaire est la suivante :

```
<var> : K
for <var> in <dictionnaire>:
    <corps>
```

où :

— `<dictionnaire>` est une expression de dictionnaire de type `Dict[K,V]`

— `<var>` est la *variable d'itération de clé* de type `K`
— `<corps>` est une suite d'instructions

Le **principe d'interprétation** est le suivant :

— le `<corps>` de la boucle est une suite d'instructions qui est exécutée une fois pour chaque clé du `<dictionnaire>`, selon un ordre arbitraire.
— pour chaque tour de boucle (donc chaque clé du dictionnaire), dans le `<corps>` de la boucle la variable `<var>` est liée à la clé concernée.
— la variable `<var>` n'est plus utilisable après le dernier tour de boucle.

Illustrons ce principe d'itération en reconstruisant explicitement l'ensemble des clés d'un dictionnaire. La fonction `ensemble_des_cles` prend en paramètre un dictionnaire de type `Dict[K,V]` et retourne un ensemble de type `Set[K]`.

Par exemple :

```
>>> ensemble_des_cles({'a':1, 'z':26, 't':20, 'q':17})
{'a', 'q', 't', 'z'}
```

```
>>> ensemble_des_cles(Constantes)
{'e', 'phi', 'pi'}
```

Une définition utilisant une itération sur les clés est proposée ci-dessous :

```
from typing import Dict, Set, TypeVar
K = TypeVar('K')
V = TypeVar('V')

def ensemble_des_cles(dico : Dict[K, V]) -> Set[K]:
    """Renvoie l'ensemble des clés du dictionnaire dico.
    """
    # Ensemble résultat
    ens : Set[K] = set()

    cle : K
    for cle in dico:
        ens.add(cle)

    return ens
```

Remarque : puisque la fonction est générique, applicable à n'importe quel dictionnaire, on a introduit des variables de type `K` et `V`.

```
# Jeu de tests
assert ensemble_des_cles({'a':1, 'z':26, 't':20, 'q':17}) \
       == {'a', 'q', 't', 'z'}

assert ensemble_des_cles({'e': 2.71828182,
                          'pi': 3.141592653589793,
                          'phi': 1.6180339887}) == {'e', 'pi', 'phi'}
```

```
assert ensemble_des_cles(dict()) == set()
```

10.2.2 Itération sur les associations

Dans certains cas, on peut s'intéresser au parcours les associations clé/valeur dans un dictionnaire plutôt que juste les clés. Pour cela, on peut utiliser la syntaxe suivante :

```
<cvar> : K
<vvar> : V
for (<cvar>, <vvar>)  in <dictonnaire>.items():
     <corps>
```

où :

- `<dictionnaire>` est une expression de dictionnaire de type `Dict[K,V]`
- `<cvar>` est la *variable d'itération de clé* de type `K`
- `<vvar>` est la *variable d'itération de valeur* de type `V`
- `<corps>` est une suite d'instructions

Le **principe d'interprétation** est le suivant :

- le `<corps>` de la boucle est une suite d'instructions qui est exécutée une fois pour chaque association clé/valeur du `<dictionnaire>`, selon un ordre arbitraire.
- pour chaque tour de boucle (donc chaque association clé/valeur du dictionnaire), dans le `<corps>` de la boucle la variable `<cvar>` est liée à la clé concernée et la variable `<vvar>` à la valeur associée.
- les variables `<cvar>` et `<vvar>` ne sont plus utilisables après le dernier tour de boucle.

Illustrons ce principe d'itération en reconstruisant explicitement la liste des valeurs d'un dictionnaire. On utilise ici une liste car une même valeur peut bien sûr être répétée dans le dictionnaire, seules les clés forment un ensemble sans répétition. La fonction `liste_des_associations` prend en paramètre un dictionnaire de type `Dict[K,V]` et retourne une liste de type `List[Tuple[K,V]]`.

Par exemple :

```
>>> liste_des_associations({'a':1, 'b':2, 'c':2, 'd':3})
[('a', 1), ('b', 2), ('c', 2), ('d', 3)]
```

On constate ici deux informations importantes :

1. la valeur 2 est répétée deux fois dans le dictionnaire,
2. l'ordre des valeurs dans la liste résultat est arbitraire.

La définition proposée est la suivante :

```
from typing import Dict, List, Tuple, TypeVar
K = TypeVar('K')
V = TypeVar('V')
```

```
def liste_des_associations(dico : Dict[K,V]) -> List[Tuple[K,V]]:
    """Renvoie la liste des associations du dictionnaire dico.
    """
    # Liste résultat
    l : List[Tuple[K,V]] = []

    cle : K
    val : V
    for (cle, val) in dico.items():
        l.append((cle, val))

    return l
```

```
# Jeu de tests
### Alerte : test douteux
assert liste_des_associations({'a':1, 'b':2, 'c':2, 'd':3}) \
    == [('a', 1), ('b', 2), ('c', 2), ('d', 3)]
### Fin d'alerte
assert len(liste_des_associations({'a':1, 'b':2, 'c':2, 'd':3})) == 4
assert liste_des_associations(dict()) == []
```

Remarque : le premier cas de test est très «fragile » car il dépend de l'ordre dans lequel les associations du dictionnaire sont parcourues. D'un ordinateur à un autre, et d'une version de Python à une autre, cet ordre pourrait changer.

Le résultat `[('d', 3), ('c', 2), ('a', 1), ('b', 2)]` serait en effet tout aussi acceptable, comme toute permutation des valeurs.

Le second test est moins fin mais beaucoup plus solide, et le fait que la liste résultat possède 4 éléments est une information intéressante à tester. On gardera en tête que la récupération des valeurs d'un dictionnaire sous la forme d'une liste est une opération non-recommandée car *fixant* un ordre arbitraire.

Il est important de garder en tête que le principe d'itération sur les clés permet également de reconstruire les valeurs. Pour illustrer ce point, voici une seconde définition de la fonction `liste_des_associations` :

```
def liste_des_associations(dico : Dict[K,V]) -> List[Tuple[K,V]]:
    """Renvoie la liste des associations du dictionnaire d.
    """

    # Liste résultat
    l : List[Tuple[K,V]] = []

    cle : K
    for cle in dico:
        l.append( (cle, dico[cle]) )
```

```
    return l
```

D'un point de vue stylistique, on peut trouver l'itération sur les associations un peu plus lisible et elle est de fait légèrement plus efficace (on n'a pas besoin de «chercher» la valeur).

Chapitre 11

Itérables et compréhensions

Les constructions par compréhension d'ensembles et de dictionnaires (sans oublier les listes) forment le fil conducteur de ce chapitre.

Pour ce chapitre, nous aurons besoin du module `typing` avec les déclarations suivantes :

```
from typing import Set, List, Dict, Tuple, TypeVar
K = TypeVar('K') # Variable de type pour les clés de dictionnaire
V = TypeVar('V') # Variable de type pour les valeur de dictionnaire
T = TypeVar('T') # Variable de type pour les éléments de listes
                 # ou d'ensembles
```

11.1 Notion d'itérable

Dans les compréhensions du chapitre 8, nous avons construit des listes à partir d'autres séquences : intervalles d'entiers, chaînes de caractères ou listes. Les compréhensions se généralisent en fait en Python aux structures de données dites **itérables**. Parmi les itérables prédéfinis on trouve notamment :

— les séquences : intervalles d'entiers, chaînes de caractères et listes
— les ensembles
— les dictionnaires (itérables sur les clés ou les associations)

Remarque : il est possible de programmer ses propres structures itérables, et nous reviendrons sur ce point en fin d'ouvrage.

La syntaxe des compréhensions simples de listes peut donc être généralisée de la façon suivante :

```
[ <elem> for <var> in <iterable> ]
```

où :

— `<var>` est une **variable de compréhension**,

— `<elem>` est une expression appliquée aux valeurs successives de la variable `<var>`
— `<iterable>` est une expression retournant une structure de donnée itérable : intervalle, chaîne, liste, ensemble ou dictionnaire.

Principe d'interprétation :

L'expression de compréhension ci-dessus signifie :

> Construit la liste des `<elem>` pour `<var>` dans `<iterable>`

Plus précisément, la liste construite est composée de la façon suivante :

— le premier élément est la valeur de l'expression `<elem>` dans laquelle la variable `<var>` a pour valeur le premier élément itéré de `<iterable>`
— le second élément est la valeur de l'expression `<elem>` dans laquelle la variable `<var>` a pour valeur le deuxième élément itéré de `<iterable>`
— ...
— le dernier élément est la valeur de l'expression `<elem>` dans laquelle la variable `<var>` a pour valeur le dernier élément itéré de `<iterable>`

Remarque : les compréhensions conditionnées et multiples se généralisent bien sûr de la même façon aux itérables.

En guise d'exemple, considérons la construction d'une liste à partir d'un ensemble :

```
>>> [k*k for k in {2, 4, 8, 16, 32, 64}]
[1024, 4096, 4, 16, 64, 256]
```

Ici on a construit une liste de type `List[int]` à partir d'un ensemble de type de `Set[int]`.

Remarque : ce dernier exemple nous rappelle que l'ordre d'itération dans un ensemble est arbitraire. Il est donc en pratique assez peu fréquent de convertir un ensemble en liste, mais il est possible que cette conversion soit nécessaire pour pouvoir ensuite appliquer une fonction prenant une liste et non un ensemble en paramètre.

Il faut faire particulièrement attention à l'écriture de tests pour de tels calculs. Par exemple, l'assertion suivante s'évalue en `True` :

```
>>> [k*k for k in {2, 4, 8, 16, 32, 64}] == [1024, 4096, 4, 16, 64, 256]
True
```

Mais toute permutation devrait être possible, donc il n'y a aucune raison que l'assertion suivante soit elle fausse alors que la précédente était «vraie» :

```
>>> [k*k for k in {2, 4, 8, 16, 32, 64}] == [4, 16, 64, 256, 1024, 4096]
False
```

Autrement dit, l'égalité n'est pas un bon test pour des listes construites à partir d'itérables non-ordonnés.

De façon similaire, on peut reconstruire la liste des valeurs stockées dans un dictionnaire. Cette conversion peut être utile par exemple si l'on souhaite étudier les répétitions des valeurs dans un dictionnaire.

Reprenons pour cela la fonction `liste_des_associations` du dernier chapitre, que l'on peut maintenant définir de façon très concise.

```
def liste_des_associations(dico : Dict[K, V]) -> List[Tuple[K,V]]:
    """Renvoie la liste des associations du dico.
    """
    return [(k, v) for (k, v) in dico.items()]
           # ou [(v, dico[v]) for k in dico]
```

Par exemple :

```
>>> liste_des_associations({'a':1, 'b':2, 'c':2, 'd':3})
[('a', 1), ('b', 2), ('c', 2), ('d', 3)]
```

Si l'ordre des éléments dans la liste résultat est arbitraire, une information intéressante est que la valeur 2 est répétée alors que les valeurs 1 et 3 sont uniques. Mais se pose le problème du test d'une telle fonction. Puisque l'ordre est arbitraire, le test suivant, quoique accepté, n'est pas satisfaisant :

```
# mauvais test
assert liste_des_associations({'a':1, 'b':2, 'c':2, 'd':3}) \
       == [('a', 1), ('b', 2), ('c', 2), ('d', 3)]
```

On peut dire qu'il s'agit d'un «coup de chance». Un test bien plus intéressant et robuste consiste à vérifier que l'on n'a pas perdu d'information. En anticipant un peu sur la suite de ce chapitre, voici un test satisfaisant :

```
# bon test  (cf. suite du chapitre pour les compréhensions de dictionnaire)
assert {k:v for (k,v)
                in liste_des_associations({'a':1, 'b':2, 'c':2, 'd':3})} \
       == {'a':1, 'b':2, 'c':2, 'd':3}
```

Ce test sera toujours vérifié, même si l'ordre d'énumération des dictionnaires change (ce qui se produit dans les versions Python précédent Python 3.7).

11.2 Compréhensions d'ensembles

Les *compréhensions d'ensembles*, également appelées *constructions d'ensembles par compréhension*, permettent de construire des ensembles complexes à partir d'*itérables* : autres ensembles, séquences, ou dictionnaires.

11.2.1 Compréhensions simples

Prenons l'exemple simple mais très utile de la construction d'un ensemble d'éléments sans répétition apparaissant dans une liste.

Une première possibilité consiste à exploiter la boucle `for` d'itération sur les séquences.

```
def elements(l : List[T]) -> Set[T]:
    """Retourne l'ensemble des éléments apparaissant
    dans la liste l
    ."""
    # Ensemble résultat
    ens : Set[T] = set()

    e : T
    for e in l:
        ens.add(e)

    return ens
```

```
# Jeu de tests
assert elements([1, 3, 5, 5, 7, 9, 1, 11, 13, 13]) \
       == {1, 3, 5, 7, 9, 11, 13}
assert elements([1, 1, 1, 1, 1]) == {1}
assert elements([]) == set()
```

Il est possible d'effectuer cette construction de façon beaucoup plus concise avec une **compréhension simple d'ensemble**.

La syntaxe proposée par le langage Python est la suivante :

`{ <elem> for <var> in <iterable> }`

où :

- `<var>` est une **variable de compréhension**
- `<elem>` est une expression appliquée aux valeurs successives de la variable `<var>`
- `<iterable>` est une expression retournant une structure itérable.

Principe d'interprétation :

L'expression de compréhension ci-dessus construit l'ensemble contenant les éléments suivant (sans les doublons éventuels) :

- la valeur de l'expression `<elem>` dans laquelle la variable `<var>` a pour valeur le premier élément itéré de `<iterable>`
- la valeur de l'expression `<elem>` dans laquelle la variable `<var>` a pour valeur le deuxième élément itéré de `<iterable>`
- ...
- la valeur de l'expression `<elem>` dans laquelle la variable `<var>` a pour valeur le dernier élément itéré de `<iterable>`

Remarques:

— l'ordre des éléments dans l'ensemble résultat est, comme pour tout ensemble, tout à fait arbitraire et probablement différent de l'ordre d'occurrence de ces mêmes éléments dans l'itérable d'origine.
— comme pour les compréhensions de listes, il n'est pas nécessaire de déclarer le type de la variable de compréhension `<var>` car celui-ci peut être déduit du type de `<iterable>` (il s'agit du type de ses éléments).

Illustrons cette syntaxe en reproduisant le premier cas de test de la fonction `elements_for` ci-dessus à l'aide d'une expression de compréhension :

```
>>> {k for k in [1, 3, 5, 5, 7, 9, 1, 11, 13, 13]}
{1, 3, 5, 7, 9, 11, 13}
```

Ceci conduit à une définition alternative particulièrement concise de la fonction :

```
def elements(l : List[T]) -> Set[T]:
    """ ... cf. ci-dessus ..."""
    return {e for e in l}
```

On peut bien sûr également construire des ensembles à partir de chaînes de caractères :

```
>>> {c for c in 'abracadabra'}
{'a', 'b', 'c', 'd', 'r'}
```

Nous récupérons naturellement les caractères qui composent la chaîne, mais dans un ordre arbitraire et sans répétition.

Tout comme pour le cas des listes, l'expression `<elem>` qui décrit les éléments de l'ensemble construit par compréhension peut être une expression complexe. Par exemple, on peut reprendre l'expression ci-dessus mais en récupérant cette-fois le rang du caractère dans l'alphabet :

```
>>> { ord(c) - ord('a') + 1 for c in 'abracadabra' }
{1, 2, 3, 4, 18}
```

La lettre `r` est par exemple la $18^{\text{ème}}$ lettre de l'alphabet (et `a` la $1^{\text{ère}}$, etc.).

11.2.2 Compréhensions avec filtrage

En mathématiques, la théorie des ensembles introduit la notion de *construction d'un ensemble par compréhension*. La notation usuelle de cette construction est la suivante :

$$\{x \in ens \mid P(x)\}$$

où *ens* est un ensemble et P est une propriété (ou prédicat)

Cette construction décrit l'ensemble composé des seuls éléments x de *ens* pour lesquels la propriété P est vérifiée pour x.

Par exemple, l'ensemble des entiers pairs peut se noter : $\{k \in \mathbb{N} \mid k \mod 2 = 0\}$

En Python, on retrouve presque littéralement cette notation de construction d'ensemble par compréhension :

```
{ <elem> for <var> in <iterable> if <condition> }
```

où :

- `<var>` est une **variable de compréhension**
- `<elem>` est une expression appliquée aux valeurs successives de la variable `<var>`
- `<iterable>` est la structure de donnée itérée
- `<condition>` est **la condition de compréhension**

Principe d'interprétation :

L'expression ci-dessus :

> construit l'ensemble des `<elem>` pour les `<var>` de `<iterable>` qui vérifient `<condition>`.

Par exemple, voici l'expression permettant de construire l'ensemble des entiers pairs dans l'intervalle $[1; 10]$:

```
>>> { k for k in range(1, 11) if k % 2 == 0 }
{2, 4, 6, 8, 10}
```

La traduction mathématique de cettte expression est très similaire puisqu'elle peut se noter : $\{k \in [1; 11[\mid k \mod 2 = 0\}$.

Remarque : c'est sans doute cette proximité avec le concept mathématique d'*ensemble défini par compréhension* qui a conduit à l'adoption de cette terminologie en Python.

Exemple : personnes célibataires

Dans le chapitre sur les n-uplets (chapitre 7) nous avons manipulé des bases de données représentées par des listes de personnes, en utilisant l'alias de type suivant :

```
Personne = Tuple[str, str, int, bool]
```

Prenons la base de données suivante :

```
BD : List[Personne]  # Base de donnée
BD = [('Itik', 'Paul', 17, True),
      ('Unfor', 'Marcelle', 79, False),
      ('Laveur', 'Gaston', 38, True),
      ('Potteuse', 'Henriette', 24, True),
```

```
      ('Ltar', 'Gibra', 13, False),
      ('Diaprem', 'Amar', 22, False)]
```

On souhaite donner une définition – bien sûr par compréhension – de la fonction `nom_celibataires` qui retourne l'ensemble des noms des personnes célibataires dans la base.

Pour notre base de données exemple, l'expression suivante calcule cet ensemble :

```
>>> { nom for (nom, _, _, marie) in BD if not marie }
{'Diaprem', 'Ltar', 'Unfor'}
```

On rappelle l'utilisation de la variable «spéciale» `_` (*underscore*) pour remplacer les variables non-nécessaires au calcul demandé. On en déduit une définition très simple de la fonction voulue :

```
def nom_celibataires(personnes : List[Personne]) -> Set[str]:
    """Retourne l'ensemble des noms des Personnes
     célibataires.
    """
    return { nom for (nom, _, _, marie) in personnes if not marie }
```

```
# Jeu de tests
assert nom_celibataires(BD) == {'Diaprem', 'Ltar', 'Unfor'}
assert nom_celibataires([('Aimar', 'Jean', 39, True)]) == set()
assert nom_celibataires([('Aiplumar', 'Jeanne', 39, False)]) \
       == {'Aiplumar'}
```

Si l'on souhaite identifier plus précisément les personnes célibataires, on peut retourner le prénom en complément du nom, par exemple :

```
>>> { (prenom, nom) for (nom, prenom, _, marie) in BD if not marie }
{('Amar', 'Diaprem'), ('Gibra', 'Ltar'), ('Marcelle', 'Unfor')}
```

On peut en déduire la définition de fonction suivante :

```
def celibataires(personnes : List[Personne]) -> Set[Tuple[str, str]]:
    """Retourne l'ensemble des prénoms/noms des Personnes
    célibataires.
    """
    return { (prenom, nom) for (nom, prenom, _, marie) in personnes
                               if not marie }
```

```
# Jeu de tests
assert celibataires(BD) == {('Amar', 'Diaprem'),
                            ('Gibra', 'Ltar'),
                            ('Marcelle', 'Unfor')}

assert celibataires([('Aimar', 'Jean', 39, True)]) == set()
assert celibataires([('Aiplumar', 'Jeanne', 39, False)]) \
```

```
       == {('Jeanne', 'Aiplumar')}
```

11.2.3 Complément : typage des éléments d'un ensemble

Le type de retour de la fonction `celibataires` ci-dessus est `Set[Tuple[str, str]]`. Les n-uplets peuvent donc être éléments d'ensembles. En revanche, comme expliqué dans le chapitre 10 (sur les ensembles) aucun type mutable comme `Set`, `List` ou `Dict` ne doit apparaître dans le type des éléments d'un ensemble. Essayons tout de même de braver cet interdit avec l'expression suivante :

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

Ici, on essaye de construire un ensemble composé de deux éléments : la liste `[1, 2, 3]` et la liste `[4, 5]`. On s'attend donc à ce que l'expression ci-dessus ait le type `Set[List[int]]`. Ici, le type des listes apparaît bien comme type des éléments de l'ensemble.

Regardons ce qu'indique l'interprète Python si on lui soumet cette expression :

```
>>> { [1, 2, 3],  [4, 5] }
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
   ...
----> 1 { [1, 2, 3],  [4, 5] }

TypeError: unhashable type: 'list'
```

Python indique que le type `list` (ici pour nous `List[int]`) n'est pas «hachable». Cette notion fait référence à la technique - dite de *hachage* - utilisé par les développeurs de Python pour programmer efficacement les ensembles (ainsi que les clés des dictionnaires). Du point de vue de notre livre, cela correspond aux types des structures de données que l'on peut modifier directement en mémoire : les listes (avec la méthode `append`), les ensembles (avec les méthodes `add` et `remove`) ainsi que les dictionnaires (avec l'affectation `D[k] = v` et la méthode `del`).

11.3 Compréhensions de dictionnaires

Les dictionnaires peuvent également être construits par compréhension, en indiquant ce que sont leurs clés et les valeurs associées.

11.3.1 Compréhensions simples

La syntaxe des compréhensions simples de dictionnaires est la suivante :

```
{<cle>:<valeur> for <var> in <iterable>}
```

où :

— `<cle>` est une expression pour la clé contenant éventuellement une ou plusieurs occurrences de `<var>`
— `<valeur>` est une expression pour la valeur contenant éventuellement une ou plusieurs occurences de `<var>`
— `<var>` est la **variable de compréhension**
— `<iterable>` est une expression retournant la structure itérée pour construire le dictionnaire.

Principe d'interprétation :

L'expression ci-dessus construit le dictionnaire formé :

— d'une première association `<cle>:<valeur>` où dans chaque expression la variable `<var>` a pour valeur le premier élément itéré de `<iterable>`
— d'une deuxième association `<cle>:<valeur>` où dans chaque expression la variable `<var>` a pour valeur le deuxième élément itéré de `<iterable>`
— ...
— d'une dernière association `<cle>:<valeur>` où dans chaque expression la variable `<var>` a pour valeur le dernier élément itéré de `<iterable>`

11.3.1.1 Construction à partir d'une liste de n-uplets

Un cas typique d'utilisation de compréhension simple de dictionnaire consiste à construire un dictionnaire de type `Dict[K,V]` à partir d'une liste de type `List[Tuple[K,V]]`.

Voici un exemple d'une telle construction :

```
>>> { nom:age for (nom, age) in [('dupont',41)
                                 ,('dupond',27)
                                 , ('dupons',31)]}
{'dupond': 27, 'dupons': 31, 'dupont': 41}
```

L'avantage du passage au dictionnaire est qu'il est maintenant possible de rechercher l'age d'un personne à partir de son nom de façon très efficace, ce qui n'est pas le cas avec la liste de couples.

Il faut tout de même faire attention avec cette construction, car si le premier élément du couple est répété, alors uniquement la dernière association sera retenue :

```
>>> { nom:age for (nom, age) in [('dupont',41), ('dupond',27),
                                 ('dupont',13) , ('dupons',31)]}
{'dupond': 27, 'dupons': 31, 'dupont': 13}
```

Ici uniquement la deuxième association pour `dupont` est retenue.

Dans de nombreux cas pratiques, on ne part pas directement d'une liste de couples mais plus généralement d'une liste de n-uplets avec $n > 2$.

Considérons le problème de la construction d'un dictionnaire associant des noms de personnes avec leur age depuis une base de donnée du type `List[Personne]`.

La définition proposée pour la fonction `ages_dict` est la suivante :

```
def ages_dict(personnes : List[Personne]) -> Dict[str,int]:
    """Retourne le dictionnaire associant le nom des Personnes à leur age.
    """
    return { nom:age for (nom, _, age, _) in personnes }
```

```
# Jeu de tests

# BD : list[Personne]  (cf. ci-dessus)

assert ages_dict(BD) == { 'Diaprem': 22,
                          'Ltar': 13,
                          'Laveur': 38,
                          'Itik': 17,
                          'Unfor': 79,
                          'Potteuse': 24 }
```

Remarque : les principes sont similaires si l'on construit le dictionnaire non pas à partir d'une liste mais d'un ensemble.

11.3.1.2 Construction à partir d'un autre dictionnaire

Il est fréquent de devoir reconstruire un dictionnaire à partir d'un autre dictionnaire. Pour illustrer ce point, nous reprenons notre principe de base de personnes mais représentée cette fois-ci par un dictionnaire associant un numéro unique de personne avec un enregistrement de type `Personne` :

```
DBD : Dict[int,Personne] # Base de donnée
DBD = { 331:('Itik', 'Paul', 17, True),
        282:('Unfor', 'Marcelle', 79, False),
        4417:('Laveur', 'Gaston', 38, True),
        14:('Potteuse', 'Henriette', 24, True),
        22215:('Ltar', 'Gibra', 13, False),
        146:('Diaprem', 'Amar', 22, False) }
```

Pour rechercher efficacement une personne dans cette base, il faut en connaître le numéro unique. Par exemple :

```
>>> DBD[14]
('Potteuse', 'Henriette', 24, True)
```

L'accès par numéro unique est souvent nécessaire notamment pour résoudre le problème des homonymes. Mais d'un point de vue pratique, il est fréquent de travailler sur une

base d'index permettant de trouver le numéro unique d'une personne à partir d'autres critères, notamment le nom de la personne. Voici un exemple d'une telle base d'index :

```
# DBDIndex : dict[str:int]  (base d'index)
DBDIndex = { nom:uniq for (uniq, (nom, _, _, _) ) in DBD.items() }
```

Important : on exploite ici l'itération sur les associations du dictionnaire `BD` (avec la méthode `items`) et non uniquement sur ses clés. C'est en fait l'unique moyen pour accéder directement dans la compréhension aux informations des personnes par déconstruction des n-uplets. L'itération sur les associations d'un dictionnaire avec la méthode `items` est expliquée dans le chapitre 11.

Regardons la valeur de la variable `DBDIndex` :

```
>>> DBDIndex
{'Diaprem': 146,
 'Itik': 331,
 'Laveur': 4417,
 'Ltar': 22215,
 'Potteuse': 14,
 'Unfor': 282}
```

Remarque : cette base d'index ne fonctionne que si les noms des personnes sont tous différents. On pourrait utiliser une base d'index de type `Dict[Tuple[str,str],int]` avec des couples `(nom, prenom)` comme clés, mais là encore on n'évite pas le problème des homonymes. En fait, la solution passe par le type `Dict[str,Set[int]]` où l'on associerait chaque nom de la base à l'ensemble des numéros uniques associés. Retenons que concevoir un système de base de données n'est pas toujours facile.

Avec cette base d'index, on peut maintenant accéder efficacement aux informations d'une personne particulière à partir du son nom :

```
>>> DBD[DBDIndex['Unfor']]
('Unfor', 'Marcelle', 79, False)
```

```
>>> DBD[DBDIndex['Ltar']]
('Ltar', 'Gibra', 13, False)
```

11.3.1.3 Construction différenciée pour les clés et les valeurs

Dans les constructions de dictionnaires, il est parfois utile de différencier la construction des clés de celle des valeurs associées.

Pour illustrer cette notion, nous nous intéressons à la construction d'un dictionnaire mettant en correspondance les chiffres de 0 à 9 et leur représentation textuelle.

Effectuons une première tentative avec l'expression suivante :

```
>>> { numero:mot for numero in range(0, 3)
                     for mot in  ['zero', 'un', 'deux']}
{0: 'deux', 1: 'deux', 2: 'deux'}
```

Ici, on a utilisé une compréhension multiple mais on n'obtient pas le résultat désiré car les itérations sont imbriquées.

En effet, on a construit ici un dictionnaire an ajoutant une-à-une :

- les assocations avec la clé `numero=0` :
  - l'assocation `0:'zero'`
  - l'assocation `0:'un'` qui remplace `0:'zero'` (car à une clé peut correspondre au plus une valeur)
  - l'assocation `0:'deux'` qui remplace `0:'un'`
- les assocations avec la clé `numero=1` :
  - l'assocation `1:'zero'`
  - l'assocation `1:'un'` qui remplace `1:'zero'`
  - l'assocation `1:'deux'` qui remplace `1:'un'`
- les assocations avec la clé`numero=2` :
  - l'assocation `2:'zero'`
  - l'assocation `2:'un'` qui remplace `2:'zero'`
  - l'assocation `2:'deux'` qui remplace `2:'un'`

Finalement, il ne nous reste qu'une seule association pour chaque clé, à chaque fois avec le dernier élément de la liste de valeurs : `deux`.

On peut traduire la compréhension ci-dessus sous la forme d'une fonction utilisant des boucles `for` d'itération.

```
>>> exemple_numeros_mots()
{0: 'deux', 1: 'deux', 2: 'deux'}
```

Pour résoudre notre problème, nous avons besoin de parcourir *simultanément* l'intervalle des clés et la liste des valeurs.Pour cela, on peut utiliser la primitive `zip` qui construit un itérable à partir de deux itérables en permettant leur itération simultanée.

Illustrons l'utilisation de cette primitive `zip` :

```
>>> { numero:mot for (numero, mot)
                 in zip(range(0, 3),['zero', 'un', 'deux' ]) }
{0: 'zero', 1: 'un', 2: 'deux'}
```

Ici on obtient bien une itération simultanée sur l'intervalle pour les clés le la liste spécifiée pour les valeurs.

La syntaxe générale pour les compréhensions différenciées clé/valeur est la suivante :

```
{<cle>:<valeur> for (<cvar>,<vvar>) in zip(<citerable>, <viterable>) }
```

où :

- `<cle>` est une expression pour la clé contenant éventuellement une ou plusieurs occurrences de `<cvar>`
- `<valeur>` est une expression pour la valeur contenant éventuellement une ou plusieurs occurences de `<vvar>`
- `<cvar>` est la *variable de compréhension pour les clés*
- `<vvar>` est la *variable de compréhension pour les valeurs*

— `<citerable>` est une expression retournant la structure itérée pour les clés.
— `<viterable>` est une expression retournant la structure itérée pour les valeurs.

Principe d'interprétation :

L'expression ci-dessus construit le dictionnaire formé :

— d'une première association `<cle>:<valeur>` où dans chaque expression les variables `(<cvar>,<vvar>)` ont pour valeur le premier couple itéré de `(<citerable>,<viterable>)`
— d'une deuxième association `<cle>:<valeur>` où dans chaque expression les variables `(<cvar>,<vvar>)` ont pour valeur le deuxième couple itéré de `(<citerable>,<viterable>)`
— ...
— d'une dernière association `<cle>:<valeur>` où dans chaque expression les variables `(<cvar>,<vvar>)` ont pour valeur le dernier couple itéré de `(<citerable>,<viterable>)`

Remarque : le dernier couple itéré pour `(<citerable>,<viterable>)` correspond au dernier élément itéré de l'un ou l'autre des deux itérables.

Pour illustrer la remarque ci-dessus, considérons l'expression suivante :

```
>>> { numero:mot for (numero, mot)
                 in zip(range(0, 3), ['zero', 'un',  'deux', 'trois' ]) }
{0: 'zero', 1: 'un', 2: 'deux'}
```

Ici, même si la liste contient plus d'éléments que l'intervalle $[0; 3[$, le dernier couple itéré est `(2,'deux')` puisque l'itération des clés s'arrête à l'entier 2. On ne pourrait de toute façon pas construire de couple suivant puisque la valeur `'trois'` n'aurait pas de clé correspondante.

De façon symétrique :

```
>>> { numero:mot for (numero, mot)
                 in zip(range(0, 10),['zero', 'un', 'deux', 'trois' ]) }
{0: 'zero', 1: 'un', 2: 'deux', 3: 'trois'}
```

Dans ce cas, le dernier couple itéré est `(3,'trois')` car la chaîne '`trois`' est le dernier élément de la liste itérée pour les valeurs. En effet les éléments suivants dans l'intervalle $[4; 10[$ n'ont pas pas de valeur associée.

11.3.2 Compréhensions avec filtrage

Les compréhensions de dictionnaires peuvent être contraintes par une *condition de compréhension*.

La syntaxe utilisée est alors la suivante :

```
{<cle>:<valeur> for <var> in <iterable> if <condition>}
```

où :

- `<cle>` est une expression pour la clé contenant éventuellement une ou plusieurs occurrences de `<var>`
- `<valeur>` est une expression pour la valeur contenant éventuellement une ou plusieurs occurences de `<var>`
- `<var>` est la **variable de compréhension**
- `<iterable>` est une expression retournant la structure itérée pour construire le dictionnaire.
- `<condition>` est **la condition de compréhension**

Principe d'interprétation :

La construction est la même que pour les constructions simples, mais en conservant uniquement les associations clé/valeur qui vérifie la condition de compréhension.

11.3.2.1 Filtrage sur les clés

Le fitrage d'un dictionnaire consiste souvent à utiliser les clés comme critère de filtrage.

Considérons par exemple le dictionnaire suivant :

```
chiffres : Dict[int,str]
chiffres = { n:s for (n,s) in zip(range(0, 10),
                                  ['zero', 'un', 'deux','trois','quatre','cinq',
                                   'six','sept','huit', 'neuf' ]) }
```

```
>>> chiffres
{0: 'zero',
 1: 'un',
 2: 'deux',
 3: 'trois',
 4: 'quatre',
 5: 'cinq',
 6: 'six',
 7: 'sept',
 8: 'huit',
 9: 'neuf'}
```

Pour filtrer le dictionnaire `chiffres` en ne conservant que les associations pour les entiers pairs, on peut utiliser la compréhension suivante :

```
>>> { n:chiffres[n] for n in chiffres if n % 2 == 0 }
{0: 'zero', 2: 'deux', 4: 'quatre', 6: 'six', 8: 'huit'}
```

Ici on construit un dictionnaire mais on peut aussi par exemple construire un ensemble :

```
>>> { chiffres[n] for n in chiffres if n % 2 == 0 }
{'deux', 'huit', 'quatre', 'six', 'zero'}
```

La même construction peut être obtenue en filtrant les associations plutôt que juste les clés :

```
>>> { chiffre for (n,chiffre) in chiffres.items() if n % 2 == 0}
{'deux', 'huit', 'quatre', 'six', 'zero'}
```

Choisir l'une ou l'autre des solutions est principalement une affaire de goût. Le parcours sur les assocations est légèrement plus efficace puisque l'on n'a pas besoin d'effectuer de recherche supplémentaire dans le dictionnaire de départ.

Finalement, on peut bien sûr effectuer un travail similaire en filtrant les chiffres impairs.

```
>>> { n:chiffres[n] for n in chiffres if n % 2 == 1 }
{1: 'un', 3: 'trois', 5: 'cinq', 7: 'sept', 9: 'neuf'}
```

```
>>> { chiffres[n] for n in chiffres if n % 2 == 1 }
{'cinq', 'neuf', 'sept', 'trois', 'un'}
```

```
>>> { chiffre for (n, chiffre) in chiffres.items() if n % 2 == 1 }
{'cinq', 'neuf', 'sept', 'trois', 'un'}
```

11.3.2.2 Filtrage sur les valeurs

En complément du filtrage sur les clés, il est parfois utile de construire un dictionnaire en filtrant un autre dictionnaire par rapport à ses valeurs.

Le plus souvent, ce besoin apparaît lorsque les valeurs sont des n-uplets. Dans ce cas, on a souvent besoin de l'itération sur les associations. Afin d'illustrer ce point, considérons à nouveau notre base de données de personnes :

```
DBD : Dict[int,Personne] # Base de donnée
DBD = { 331:('Itik', 'Paul', 17, True),
        282:('Unfor', 'Marcelle', 79, False),
        4417:('Laveur', 'Gaston', 38, True),
        14:('Potteuse', 'Henriette', 22, True),
        22215:('Ltar', 'Gibra', 13, False),
        146:('Diaprem', 'Amar', 22, False) }
```

Construisons à partir de cette base un dictionnaire associant les noms des personnes à leur age mais uniquement pour les personnes mariées. L'expression suivante effectue cette construction :

```
>>> { nom:age for (_, (nom, _, age, marie) ) in DBD.items() if marie }
{'Itik': 17, 'Laveur': 38, 'Potteuse': 22}
```

Si on veut plutôt les nom et numéros des personnes célibataires de plus de 20 ans, on pourra écrire :

```
>>> { nom:num for (num, (nom, _, age, marie) ) in DBD.items()
                  if (age >= 20) and (not marie) }
{'Diaprem': 146, 'Unfor': 282}
```

Du fait de son intérêt pratique, on retiendra la syntaxe des compréhensions de dictionnaires à partir d'un dictionnaire avec filtrage sur les valeurs quand ces dernières sont des n-uplets.

```
{ <cle>:<valeur> for (<cvar>, (<vvar1>, <vvar2>, ..., <vvarN>))
                 in <dictionnaire>.items() if <condition> }
```

où :

- `<cle>` est une expression pour les clés, référençant un ou plusieurs variables parmi `<cvar>`, `<vvar1>`, `<vvar2>`, ..., `<vvarN>`
- `<valeur>` est une expression pour les valeurs, référençant un ou plusieurs variables parmi `<cvar>`, `<vvar1>`, `<vvar2>`, ..., `<vvarN>`
- `<cvar>` est la *variable de compréhension pour les clés*
- `<vvar1>`, `<vvar2>`, ..., `<vvarN>` sont les *variables de compréhension pour les n-uplets valeur*
- `<dictionnaire>` est une expression correpondant à un dictionnaire dont les valeurs sont des n-uplets.
- `<condition>` est la *condition de compréhension* pouvant porter sur les variables `<cvar>`, `<vvar1>`, `<vvar2>`, ..., `<vvarN>`

On a ici un petit langage permettant d'effectuer des requêtes assez complexes dans notre base de donnée. Ce principe n'est d'ailleurs pas limité aux dictionnaires. On peut par exemple récupérer l'ensemble des noms des personnes de plus de 30 ans :

```
>>> { nom for (_, (nom, _, age, _)) in DBD.items()
              if age >= 30 }
{'Laveur', 'Unfor'}
```

11.4 Synthèse sur les compréhensions

A ce stade, nous avons vu de nombreuses constructions par compréhensions pour les listes, les ensembles et les dictionnaires. Nous avons présenté les compréhensions en fonction de la structure de données qu'elles engendraient :

- compréhension de liste pour construire une liste à partir d'un itérable. Si l'itérable est une séquence alors l'ordre des éléments itérés est conservé.
- compréhension d'ensemble pour construire un ensemble à partir d'un itérable. Les doublons sont supprimés et l'ordre de l'itérable n'est pas conservé.
- compréhension de dictionnaire pour construire un dictionnaire à partir d'un itérable unique. Les clés sont sans doublon mais les valeurs peuvent être répétées, et l'ordre de l'itérable n'est pas conservé.

- compréhension de dictionnaire pour construire un dictionnaire à partir d'un itérable pour les clés et un itérable pour les valeurs en utilisant la primitive `zip`. Les clés sont sans double et les ordres des itérables ne sont pas conservés.

Ces quatre possibilités sont à croiser avec les différents itérables que nous avons étudiés :

- les séquences : intervalles d'entiers, chaînes de caractères et listes
- les ensembles
- les dictionnaires itérés sur leurs clés
- les dictionnaires itérés sur leurs associations (avec la méthode `items`).

De plus, nous avons vu deux types de compréhension :

- sans filtrage des éléments itérés
- avec filtrage des éléments itérés par une condition de compréhension
  - pour les dictionnaire, le filtrage peut être opéré sur les clés uniquement, sur les valeurs uniquement ou sur les associations clé/valeur.

Nous avons également vu, en particulier sur les listes, que les compréhensions pouvaient être combinées avec des clauses `for` multiples. On obtient donc une combinatoire des possibilités assez importante correspondant à une gamme assez large de problèmes pouvant être résolus de façon concise avec des compréhensions.

Il faut tout de même garder à l'esprit que malgré l'éventail de possibilités offertes, de nombreux problèmes pratiques ne peuvent être résolus directement par compréhension. Pour les listes, ce sont notamment les problèmes qui ne sont pas des compositions de transformations et filtrages. Pour les dictionnaires, on peut citer par exemple la construction d'un dictionnaire de fréquences.

Chapitre 12

Procédures et tableaux

Dans ce livre, nous insistons beaucoup sur le fait que les programmes sont plus simples à concevoir, à programmer et (surtout) à tester ou à mettre au point si les fonctions que nous mettons en œuvre sont de véritables au sens mathématique du terme. L'idée est qu'une fonction renvoie toujours la même valeur si on lui passe les mêmes arguments. La difficulté principale concerne les structures mutables comme les listes, les ensembles et les dictionnaires. En effet, si une fonction reçoit une telle structure en argument, alors elle peut potentiellement la modifier directement en mémoire, par exemple avec le `.append` des listes. Dans ce chapitre, nous abordons la notion de *procédure* qui exploite cette possibilité. L'inconvénient principal est que notre «fonction» au sens de Python n'est plus une fonction au sens mathématique du terme. Les programmes utilisant de telles procédures sont plus difficile à tester (puisque la valeur retournée, qui est souvent `None`, ne dit pas tout) et à mettre au point. Le principal intérêt concerne l'efficacité avec la possibilité de mettre en œuvre des algorithmes dits *en place*, c'est-à-dire sans reconstruction (parfois couteuse) des données. Dans la terminologie usuelle de la programmation on parle d'*algorithmique de tableaux* (sur les listes Python !) avec une opération caractéristique : le *remplacement* d'un élement de liste à un indice donné.

12.1 Notion de procédure

Définition : Une **procédure** est une fonction (Python) qui produit **un effet de bord** (ou simplement *effet*), c'est à dire que le calcul que fait la fonction produit autre chose que simplement une valeur de retour. On distingue deux types principaux d'effets de bord: les **entrées/sorties** et les **modifications en place**.

Nous ne nous étendrons pas sur les entrées/sorties ici, nous avons déjà vu dans les premiers chapitres comment la primitive `print` pouvait être utilisée, avec comme effet de bord des affichages sur la sortie standard. Nous avons également utilisé, dans des exercices, la lecture (entrée) et l'écriture (sortie) de fichiers textes.

Les modifications en place sont les opérations primitives qui permettent d'altérer les

structures de données mutables directement en mémoire, de façon localisée donc sans remplacer entièrement la structure. Nous avons vu un certain nombre de ces opérations :

— `l.append(e)` qui ajoute un élément `e` à la fin de la liste `l` (cf. chapitre 6)
— `ens.add(e)` qui ajoute un élément `e` à l'ensemble `ens`, si c'est élément n'était pas déjà présent (également le retrait d'un élément, cf. chapitre 9)
— `d[k] = e` qui crée une association de la clé `k` à la valeur `v`, ou de mettre à jour une association existant pour la clé `k`, en «écrasant» l'ancienne valeur (cf. chapitre 10)

Nous avons insisté sur la «dangerosité» potentielle de ces opérations, qui effectuent des modifications directement en mémoire.

Nous avons donc utilisé ces opérations avec de grandes précautions, en nous assurant que les fonctions Python que nous écrivions étaient bien des fonctions au sens mathématique du terme. Nous avons donc fait attention à ce que les modifications en place ne soient appliqués que sur des structures (listes, ensembles ou dictionnaires) intégralement construites dans les fonctions. Par exemple, la plupart des fonctions qui renvoient une liste commencent par initialiser une liste résultat vide, puis utilisent `append` pour «remplir» la liste élément par élément. Ainsi les modifications sont locales à la fonction, et n'impliquent pas les structures mutables passées en argument. Nous avons toujours évité de modifier les paramètres d'une fonction et n'avons jamais appliqué `l.append(e)` ou `d[k] = e` sur un `l` ou un `d` correspondant à un paramètre.

Nous allons maintenant relâcher cette contraite en nous focalisant sur les listes. L'objectif est de pouvoir mettre œuvre des *algorithmes de tableaux*, qui s'accompagnent souvent d'un gain en efficacité, si on les compare aux algorithmes de listes basés sur des fonctions mathématiques.

Commençons par un algorithme de liste, basé sur une fonction mathématique :

```
from typing import List

def ajout_zero_fonction(l : List[int]) -> None:
    """Retourne la liste l suivie de l'entier 0
    """
    return l + [0]
```

Cette fonction est extrêmement simple et elle est bien «mathématique» : peu importe la liste en entrée, on obtiendra bien une *nouvelle* liste avec les éléments de `l` suivis d'un 0. Ce que l'on peut vérifier par des tests.

```
# Jeu de tests
assert ajout_zero_fonction([1, 2, 3]) == [1, 2, 3, 0]
assert ajout_zero_fonction([]) == [0]
```

Un autre façon de résoudre ce «problème» est de passer par une procédure. L'idée est de modifier directement `l` en utilisant `append`. La définition est la suivante :

```
def ajout_zero_proc(l : List[int]) -> None:
    """***Procédure***
```

```
    Ajoute l'entier 0 à la fin de l.
    """
    l.append(0)
```

Même si en Python on parle toujours de «fonction», nous sommes bel et bien ici en présence d'une procédure et non d'une fonction au sens mathématique du terme, ce que nous indiquons clairement dans la documentation. Tout d'abord, cette «fonction» ne retourne pas de valeur intéressante, juste la valeur `None`, ce qui est un signe distinctif des procédures. Donc le seul test possible est de comparer la valeur de retour avec `None`, ce qui est loin d'être suffisant. En anticipant un peu la suite du chapitre, pour tester une telle procédure il faut pouvoir observer les effets de bord produits par la procédure. Pour cela, nous déclarons une variable de test et l'initialisons avec la liste que nous souhaitons manipuler directement en mémoire. On peut ensuite écrire des assertions intéressantes en observant les manipulations en place effectuées sur cette liste.

```
# Jeu de test
liste_ex : List[int] = [1, 2, 3]  # la liste d'exemple pour le test
assert ajout_zero_proc(liste_ex) == None # la valeur retournée est testée
assert liste_ex == [1, 2, 3, 0]  # on observe l'effet sur la liste
```

Ceci illustre la principale difficulté du test pour les procédures : la nécessité d'observer les effets, et non uniquement de vérifier les valeurs calculées. En pratique, les effets et les calculs se mélangent dans les fonctions Python. Que penser par exemple de cette troisième variante ?

```
def ajout_zero_mix(l : List[int]) -> List[int]:
    """????"""
    l.append(0)
    return l
```

S'agit-il d'une procédure, d'une fonction ? Pour le programmeur Python c'est bien une fonction, mais ce n'est clairement pas une fonction au sens mathématique du terme, un effet de bord est bien présent !

```
>>> liste_ex : List[int] = [1, 2, 3]
```

```
>>> ajout_zero_mix(liste_ex)
[1, 2, 3, 0]
```

On peut croire qu'on a bien une fonction ici, mais ce n'est certainement pas le cas puisque la liste a été changé également en mémoire :

```
>>> liste_ex
[1, 2, 3, 0]
```

Notre ancienne liste est perdue ! Comparons avec la version véritablement fonctionnelle :

```
>>> liste_ex : List[int] = [1, 2, 3]
```

```
>>> ajout_zero_fonction(liste_ex)
[1, 2, 3, 0]
```

On a le même résultat en retour, mais cette fois-ci aucun effet de bord ne s'est produit, ce que nous pouvons vérifier :

```
>>> liste_ex
[1, 2, 3]
```

Cette fois-ci *ouf* ! Nous n'avons pas perdu la liste de départ. On peut se demander l'intérêt de la version *en place* par rapport à la version fonctionnelle qui ne perd jamais d'information. La réponse est simple : *l'efficacité*. Là où `ajout_zero_fonction` doit reconstruire intégralement la liste résultat, les procédures `ajout_zero_proc` ou `ajout_zero_mix` effectuent une unique opération mémoire. Par exemple, si la liste passée en argument est de longueur 1000, la version fonctionnelle effectuera globalement un millier d'opérations mémoire, contre une seule pour les procédures puisqu'aucune reconstruction n'est nécessaire.

Une autre caractéristique des procédures est qu'elles ne retournent pas forcément les même valeurs si on les appelle avec les mêmes arguments (lorsqu'elles retournent autre chose que `None`). Un exemple assez parlant est celui de la génération aléatoire non-déterministe, en utilisant le module `random`.

```
import random

def liste_aleatoire(n : int) -> List[int]:
    """***Procédure***
    Retourne une liste d'entiers aléatoires (entre 0 et 100)
    de longueur n.
    """
    return [int(random.uniform(0, 100)) for _ in range(n)]
```

Essayons de générer une liste de taille 5 :

```
>>> liste_aleatoire(5)
[42, 27, 94, 97, 69]
```

Si nous appliquons une nouvelle fois la même procédure avec le même valeur 5 pour `n`, la probabilité d'obtenir le même résultat en retour est extrêmement faible (car la distribution est uniforme) :

```
>>> liste_aleatoire(5)
[69, 30, 65, 52, 77]
```

```
>>> liste_aleatoire(5)
[35, 70, 92, 7, 81]
```

On voit donc assez clairement qu'il ne s'agit pas d'une fonction au sens mathématique du terme. Cependant, les effets de cette procédure sont intégralement capturés par la valeur retournée. Donc pour observer les effets de la fonction, nous devons nous baser sur la valeur de retour. Une première observation importante est que la longueur de la liste produite est bien celle spécifiée. Mais il faudrait aussi vérifier que les entiers sont bien compris dans l'intervalle $[0; 100]$ tel qu'expliqué dans la documentation. Pour cela, on peut utiliser un prédicat de vérification auxiliaire.

```
def dans_intervalle(l : List[int], a : int, b : int) -> bool:
    """Vérifie que les éléments de la liste l sont bien
    dans l'intervalle [a;b].
    """
    k : int
    for k in l:
        if k < a or k > b:
            return False
    return True
```

On en déduit le jeu de test pour la procédure `liste_aleatoire`.

```
# Jeu de tests
assert len(liste_aleatoire(5)) == 5    # test de la longueur
assert len(liste_aleatoire(10)) == 10
assert dans_intervalle(liste_aleatoire(30), 0 , 100)
assert dans_intervalle(liste_aleatoire(50), 0 , 100)
```

Encore une fois, on peut voir que le test de procédure est plus complexe que le test des fonctions mathématiques, du fait de la réflexion nécessaire sur l'observation des effets de bord, soit sur les structures mutables en entrées (comme pour `ajout_zero_proc`) ou sur le lien entre la procédure et les résultats produits (comme pour `liste_aleatoire`).

12.2 Instruction de remplacement

Pour la suite de ce chapitre, nous allons nous concentrer sur la modification en place des listes, avec l'introduction d'une autre possibilité de modifier les listes directement en mémoire : l'instruction de **remplacement** en place à un indice donné [1].

La syntaxe du remplacement est : `l[i] = e`.

avec `l` de type `List[T]`, `i` de type `int` et `e` de type `T`

Le **principe d'interprétation de l'instruction de remplacement** est le suivant :

- la valeur de l'expression `e` est calculée
- la valeur de l'expression `i` est également calculée
- si cette valeur correspond à un indice *correct* dans la liste `l` (c'est-à-dire une valeur entre `0` et `len(l)-1`), alors la valeur du `i`-ème élément de la liste est *remplacée directement en mémoire* par la valeur de `e`.
- sinon, si l'indice est incorrect, une erreur est signalée.

Puisque cette opération est une instruction et non une expression, aucune valeur n'est retournée ici.

1. L'instruction de remplacement est souvent appelée une affectation (à un indice), ce qui est à notre avis un peu trompeur car l'affectation porte sur les variables et non sur les structures de données (mutables). Les affectations de variables sont présentées au chapitre 2.

Remarque : cette opération est très similaire à l'opération `d[k] = e` pour les dictionnaires, en considérant que les «clés» d'une liste sont ses indices.

Illustrons ce principe d'interprétation assez complexe.

```
>>> liste_ex : List[int] = [1, 2, 3]
>>> liste_ex[1] = 0
```

Avec cette opération, on s'attend à ce que l'élément à l'indice 1 soit modifié, vérifions ce fait :

```
>>> liste_ex
[1, 0, 3]
```

Illustrons également le fait que des expressions composées peuvent être utilisées à la fois pour l'indice et pour la valeur de remplacement.

```
>>> liste_ex[42-42] = 42 % 10
>>> liste_ex
[2, 0, 3]
```

Il faut noter ici que le calcul de l'indice est de l'arithmétique entière, qui peut être particulièrement complexe. Il s'agit d'une source très fréquente de *bug* dans les programmes. Souvent, ces erreurs s'observent par des valeurs d'indice incorrects. Heureusement, Python est un langage dit *sûr*, qui vérifie les calculs d'indices à l'exécution (d'où l'importance des tests). Observons cette vérification :

```
>>> liste_ex[42] = 12
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
  ...
----> 1 liste_ex[42] = 12

IndexError: list assignment index out of range
```

L'interprète nous signale ici que l'index d'affectation est en dehors de l'intervalle autorisé.

12.3 Algorithmes de tableaux

L'instruction de remplacement à un indice est principalement utilisée pour ce que l'on appelle *l'algorithmique de tableaux*, que l'on pourrait dans ce livre plus précisément caractériser comme les algorithmes qui modifient les listes *en place*.

12.3.1 Un premier exemple : le renversement

Pour illustrer les principes de base de cette algorithmique, nous allons considérer un problème simple : le «renversement» des listes. Commençons par une solution fonctionnelle :

```
from typing import List, TypeVar
T = TypeVar("T")

def renversement(l : List[T]) -> List[T]:
    """Renvoie la liste renversée de l."""
    return [e for e in l[::-1]]
```

```
# Jeu de tests
assert renversement([1, 2, 3, 4, 5]) == [5, 4, 3, 2, 1]
assert renversement(["un", "deux", "trois"]) == ["trois", "deux", "un"]
assert renversement([]) == []
```

Grâce aux compréhensions (et au découpage en «sens arrière» `L[::-1]`) notre fonction est vraiment concise et toute à fait opérationnelle. On remarque à ce propos la simplicité d'écriture des tests.

Pour la version *procédurale*, avec l'objectif de renverser la liste en place, c'est un peu plus complexe. On commence par une opération qui permet d'échanger deux éléments dans une liste.

```
def echanger(l : List[T], i : int, j : int) -> None:
    """***Procédure***
    Précondition : len(l) > 0
    Précondition : 0 <= i < len(l)
    Précondition : 0 <= j < len(l)
    Echanger en place les éléments situés aux indices i et j
    dans la liste.
    """
    temp : T = l[i]  # variable temporaire
    l[i] = l[j]
    l[j] = temp
```

Cette procédure est simple, mais nécessite tout de même un certain nombre de précautions (et donc de préconditions), illustrant encore le fait que le raisonnement sur les indices de listes n'est souvent pas trivial. On utilise ici une variable (donc une case mémoire) *temporaire* pour éviter de perdre de l'information. En Python «expert», l'échange peut s'écrire de façon plus concise avec `l[i],l[j] = l[j],l[i]` mais ce genre de «multi-affectation» à la sémantique complexe (et pour certains douteuse) ne devrait pas être trop employée en pratique.

Pour le jeu de test, il faut prendre en compte l'observation des effets et donc partir d'une liste externe, un peu comme pour `ajout_zero_proc`. Voici une possibilité :

```
# Jeu de tests
liste_echanger : List[str] = ["a", "b", "c", "d", "e"]
assert echanger(liste_echanger, 1, 3) == None
assert liste_echanger == ["a", "d", "c", "b", "e"]
assert echanger(liste_echanger, 0, 4) == None
assert liste_echanger == ["e", "d", "c", "b", "a"]
```

```
assert echange(liste_echanger, 2, 2) == None
assert liste_echanger == ["e", "d", "c", "b", "a"]
```

Nous n'avons pas encore abordé la question du renversement proprement dite et la solution procédurale semple déjà bien plus complexe que la version fonctionnelle ! Mais nous avons maintenant avec `echange` la pièce essentielle du *puzzle*. Voici notre proposition pour le renversement.

```
def renverser(l : List[T]) -> List[T]:
    """***Procédure***
    Renverse la liste l en place.
    """
    for k in range(0, len(l) // 2):
        echanger(l, k, len(l) - k - 1)
    return l
```

Une remarque importante est que plus que de parler de *renversement*, nous avons préféré indiquer un verbe, ici *renverser* pour bien expliquer qu'il s'agit d'une procédure.

Le principe ici est de parcourir avec la variable d'itération `k` les indices de la liste de `0` à la position se situant au milieu de la liste. A chaque tour on effectue l'échange entre l'élément à la position `k` et l'élément à cette même position mais en partant de la fin.

Pour le jeu de test, encore une fois, nous devons observer les effets de la fonction, car son type de retour est trompeur puisqu'il s'agit bien d'une procédure : la liste `l` passée en argument est bel et bien modifiée *en place*.

```
# Jeu de tests
liste_renv : List[str] = ["a", "b", "c", "d", "e"]
assert renverser(liste_renv) == ["e", "d", "c", "b", "a"]
assert liste_renv == ["e", "d", "c" , "b", "a"]
liste_renv2 : List[str] = ["a", "b", "d", "e"]
assert renverser(liste_renv2) == ["e", "d", "b", "a"]
assert liste_renv2 == ["e", "d", "b", "a"]
```

La procédure `renverser` est pleine de pièges : notamment les bornes pour l'échange ne sont pas si faciles à calculer. Le gain en efficacité est indéniable puisque l'on n'a pas à construire de liste résultat. En revanche, il faut effectuer à peu près `len(l) // 2` opérations d'échanges, constre `len(l)` opération `.append` pour la version fonctionnelle, le gain n'est pas aussi important que pour l'ajout en fin.

12.3.2 Etude de cas 1 : le tri en place par sélection

Les algorithmes de tri sont des exemples canoniques d'algorithmes de tableaux. Ils permettent de mettre en pratique le *calcul en place* effectué par les procédures que nous étudions dans ce chapitre. Il s'agit aussi de problèmes intéressants pour discuter de la notion de *complexité asymptotique*.

Le principe du tri d'une liste est simple, et fait référence à un ordre total souvent appelé

ordre naturel des éléments contenus dans la liste. L'exemple le plus simple est celui des listes d'entiers selon l'ordre naturel des nombres entiers avec le comparateur `<=`.

En mathématiques, une *fonction de tri* calculera, à partir d'une liste `l` en entrée, une «nouvelle» liste résultat contenant les mêmes éléments que `l` (y compris les répétitions) mais «rangés» en ordre croissant. Par exemple si la liste `l` est `[2, 9, 5, 1, 6]` la liste résultat sera `[1, 2, 5, 6, 9]`. Mathématiquement parlant le résultat est une *permutation triée* de la liste. Les répétitions sont possibles, ainsi par exemple à partir de `[2, 9, 5, 9, 1, 2, 6]` on obtiendra la liste triée `[1, 2, 2, 5, 6, 9, 9]`.

Dans ce chapitre, cependant, nous nous intéressons plutôt aux *procédures de tri* qui produisent un résultat similaire, mais en effectuant des modifications en place sur la liste de départ pour la trier. On y gagne souvent en efficacité car il n'y a pas à reconstruire de liste, mais en revanche il faut être conscient que l'on a «perdu» la liste de départ, non-triée. Il existe un certain nombres d'algorithmes de tri, car il s'agit d'une opération très fréquemment employée et dont l'efficacité est parfois critique. Il n'est pas étonnant que de nombreuses solutions au problème ont été proposées. On peut citer, pour les plus simples : le *tri par sélection*, le *tri par insertion* ou le *tri à bulles* ; et pour les tris un peu plus complexes et souvent plus efficaces : le *tri-fusion*, le *tri par tas*, le *tri rapide* (ou *quicksort*) ou encore le plus récent *Timsort* qui a été développé pour Python et qui de plus en plus utilisé aujourd'hui dans les langages de programmation.

Principes du tri par sélection

Le tri pas sélection est un algorithme de tri assez simple, surtout si l'on considère les modifications en place. Comme tous les algorithmes de tri «simples», le principe est de considérer le processus de tri à deux niveaux :

— au premier niveau, on s'intéresse à un élément de la liste en particulier, identifié par son indice. Le plus souvent on procède «de gauche à droite» donc en partant du premier élément à l'indice 0 et pour arriver au dernier élément à l'indice `len(l)-1` pour une liste `l` donnée. Pour le tri en place, ce n'est pas vraiment l'élément qui compte (il faudra tout de même faire quelque chose avec sa valeur, notamment ne pas la perdre) mais plutôt *l'indice courant*.

— au second niveau, au cœur de l'algorithme, le principe va être d'effectuer un certain nombre de *comparaisons* à partir de l'indice courant (du premier niveau), et en particulier comparer la valeur l'élément actuellement à cet indice avec d'autres éléments situés, le plus généralement, *après* (donc à des indices supérieurs). Bien sûr, ces comparaisons pourront entraîner des modifications en place pour effectivement trier la liste (ou le tableau, dans d'autres langages de programmation).

Les algorithmes de tri qui se basent sur cette approche, comme le tri par sélection que nous allons étudier mais également le tri par insertion, le tri à bulles et bien d'autres encore, effectuent, dans le pire cas, de l'ordre de n^2 opérations (de comparaison) pour trier une liste de taille n. Ce n'est pas étonnant puisqu'il y a n indices courants à considérer au premier niveau, et qu'au pire cas il faudra comparer l'élément d'indice courant avec tous les autres éléments. Cependant, on peut montrer que le problème du

tri peut être résolu, toujours au pire cas et sans autre information que l'ordre naturel sur les éléments à trier avec un nombre de comparaisons de l'ordre de $n.logn$. Pour cela, on doit utiliser un principe de *dichotomie* (et souvent des structures de données auxiliaires parfois assez complexes) qui dépasse le cadre de ce chapitre et de cet ouvrage en général. Il faudra aborder l'apprentissage de l'algorithmique en tant que tel pour approfondir ces questions de *complexité*.

Pour le tri par sélection, le cœur de l'algorithme consiste au remplacement de l'élément d'indice courant par le *minimum* des éléments qui le suivent, c'est-à-dire les éléments dont les indices sont supérieurs à l'indice courant. Pour ce remplacement, on exploite le principe de l'échange et donc la procédure `echanger` définie précédemment. Voici une définition de la procédure réalisant cet algorithme de sélection (du minimum).

```
def selectionner_min(l : List[T], indice_courant : int) -> None:
    """***Procédure***
    Précondition : 0 <= indice_courant < len(l)
    Sélectionne le minimum des éléments suivant l'indice courant
    dans la liste l, plaçant ce dernier à l'indice courant.
    """
    j : int # indice du prochain minimum <<potentiel>>
    for j in range(indice_courant + 1, len(l)):
        if l[j] < l[indice_courant]:
            echanger(l, indice_courant, j)
```

Nous allons maintenant "jouer" un peu avec cette procédure. Partons de la liste suivante :

```
>>> liste_sel : List[int] = [5, 3, 4, 2, 1]
```

Prenons l'indice 0 comme courant, avec l'élément de valeur 5, et appelons notre procédure.

```
>>> selectionner_min(liste_sel, 0)
```

Bien sûr, rien n'est retournée c'est une procédure qui retourne `None` mais nous pouvons observer les modifications effectuées en place sur la liste.

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

Le minimum des éléments qui suivent l'indice courant était l'élément 1 (précédemment à l'indice 4) et qui se retrouve donc comme prévu à l'indice courant. A partir de maintenant, il ne faut bien sûr plus toucher au 1 et donc à l'indice 0. On considère donc l'indice 1 comme courant.

```
>>> selectionner_min(liste_sel, 1)
>>> liste_sel
[1, 2, 5, 4, 3]
```

Le «nouveau» minimum pour l'indice 1 est bien le 2 et on se retrouve avec une liste décomposée en deux parties :

— de l'indice 0 à l'indice courant les éléments sont triés puisqu'on a placé les minimums «le plus à gauche possible» à chaque sélection.

— pour le reste de la liste (pour les indices supérieurs à l'indice courant), on obtient une permutation des éléments non-encore sélectionnés comme minimum. L'ordre précis de ces éléments correspond aux échanges effectués, et peut être considéré comme parfaitement arbitraire : toute permutation est «correcte».

Si on continue ce processus en augmentant à chaque étape l'indice courant, on obtiendra bien sûr une liste complètement triée au final.

```
>>> selectionner_min(liste_sel, 2)
>>> liste_sel
[1, 2, 3, 5, 4]
```

```
>>> selectionner_min(liste_sel, 3)
>>> liste_sel
[1, 2, 3, 4, 5]
```

A ce stade, la liste est triée. Il est inutile d'aller chercher une nouvelle fois un minimum pour l'indice courant 4 puisqu'il n'existe pas d'indice suivant. Donc pour une liste `l` le «dernier» indice courant à considérer est `len(l)-2` (même s'il ne serait pas très couteux d'aller jusque `len(l)-1`).

Exercice : déduire des quelques interactions un jeu de tests pour la procédure `selectionner_min`.

Nous pouvons maintenant en déduire la procédure de tri principale, qui devient très simple puisque nous avons extrait le cœur de l'algorithme.

```
def trier_par_selection(l : List[T]) -> List[T]:
    """***Procédure***
    Modifie en place la liste l pour en trier les éléments
    en ordre croissant. Retourne cette même liste.
    """
    indice_courant : int
    for indice_courant in range(0, len(l) - 1):
        selectionner_min(l, indice_courant)
    return l
```

La fonction semble fonctionner :

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

Et puisque Python propose un ordre naturel pour la plupart des types de données, on peut par exemple trier des chaînes de caractères (en ordre lexicographique des caratères Unicode) :

```
>>> trier_par_selection(["a","man","a","plan","a","canal","panama"])
['a', 'a', 'a', 'canal', 'man', 'panama', 'plan']
```

Il nous reste à définir un jeu de tests pour notre tri par sélection. Nous avons deux difficultés, la première étant que nous considérons un tri et qu'ainsi nous devons tester

que l'opération réalisée est bien un tri, et la seconde étant qu'il s'agit d'une procédure de tri et qu'il faut donc en observer les effets, au delà de la valeur de retour.

Pour la première difficulté, la solution passe par un algorithme de vérification de tri, qui se décompose en deux parties : (1) que la liste triée est une permutation de la liste de départ (en supposant des permutations qui préservent les répétitions d'éléménts), et (2) que les éléments sont finalement ordonnés de façon croissante. Nous donnons ci-dessous une solution simple (et de nature fonctionnelle) pour ces deux problèmes. Pour les permutations (généralisées aux répétitions), nous commençons par une petite fonction auxiliaire permettant de retourner le nombre d'occurrences d'un élement dans une liste.

```
def occurrences(a : T, l : List[T]) -> int:
    """Retourne le nombre d'occurrences de l'élément a dans la liste l.
    """
    nb : int = 0
    b : T
    for b in l:
        if b == a:
            nb = nb + 1
    return nb
```

```
# Jeu de tests
assert occurrences(2, [3, 2, 5, 9, 2, 2]) == 3
assert occurrences(1, [3, 2, 5, 9, 2, 2]) == 0
assert occurrences("bla", ["bli", "bla", "blo", "blu"]) == 1
```

On peut en déduire une fonction de vérification pour les permutations (avec répétitions).

```
def verif_permutation(l : List[T], lp : List[T]) -> bool:
    """Retourne True si et seulement lp est une permutation
    (avec répétitions) de la liste l.
    """
    # vérification 1 : même nombre d'éléments
    if len(lp) != len(l):
        return False

    # vérification 2 : mêmes éléments et nombres d'occurrences
    for e in l:
        if occurrences(e, lp) != occurrences(e, l):
            return False

    return True
```

```
# Jeu de tests
assert verif_permutation([3, 6, 1, 2, 5, 4], [1, 2, 3, 4, 5, 6])\
       == True
assert verif_permutation([3, 6, 1, 2, 5, 4], [1, 2, 3, 4, 4, 5, 6])\
       == False
```

```
assert verif_permutation([3, 6, 1, 2, 5, 4], [1, 2, 3, 4, 5, 7])\
       == False
```

Et pour la croissance, le vérification est assez simple :

```
def verif_croissance(l : List[T]) -> bool:
    """Retourne True si et seulement si la liste l est triée
    en ordre croissant.
    """
    # i : indice courant
    for i in range(0, len(l)-1):
        if l[i] > l[i+1]:
            return False
    return True
```

```
# Jeu de tests
assert verif_croissance([3, 6, 1, 2, 5, 4]) == False
assert verif_croissance([1, 2, 3, 4, 5, 6]) == True
assert verif_croissance([1, 2, 3, 4, 4, 5, 6]) == True
```

Nous pouvons maintenant proposer un jeu de test pour la procédure `trier_par_selection` ainsi que pour n'importe quelle fonction et procédure de tri.

```
# Jeu de tests (pour trier_par_selection)
liste_trisel : List[int] = [3, 6, 1, 2, 5, 4]
liste_trisel_sauvegarde : List[int] = liste_trisel[:]
assert trier_par_selection(liste_trisel) == [1, 2, 3, 4, 5, 6]
assert verif_permutation(liste_trisel_sauvegarde, liste_trisel)
assert verif_croissance(liste_trisel)
```

On remarquera, dans ce jeu de tests, l'importance de la *sauvegarde* de la valeur initiale de la liste à trier. En effet, on ne pourrait pas vérifier la permutation si cette sauvegarde n'était pas effectuée. Encore une fois, on peut voir que la problématique de test en style *procédural* est bien moins évidente que pour le style *fonctionnel*. Nous pouvons également remarquer que la vérification elle-même est définie en style *fonctionnel*. Comprendre la problématique du tri (ou de toute autre problématique non-triviale), c'est d'abord en comprendre le sens *fonctionnel*, en tout cas c'est le point de vue très affirmé des auteurs de ce livre.

12.3.3 Etude de cas 2 : les matrices

Un autre exemple emblématique des algorithmes de tableaux concerne le *calcul matriciel*. On se rappelle que les **matrices** sont des objet mathématiques organisant des données (appelées **coefficients**) en deux dimensions, par ligne et colonnes. Par exemple:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

est une matrice de taille 3x3 (3 lignes et 3 colonnes) dont les coefficients sont des entiers. Les matrices ont un rôle fondamental en algèbre linéaire pour représenter les transformations dans les espaces vectoriels. Elles sont par exemple largement exploitées dans le monde du jeu vidéo pour décrire entre autre les transformations 3D.

12.3.3.1 Exemple : la transposition

Il n'existe généralement pas de type spécifique pour les matrices dans les langages de programmation usuels (c'est le cas dans les langages spécifiquement dédiés aux calculs numériques comme *NumPy*[2].) et la manière la plus commune de les représenter est d'utiliser une liste de listes: les éléments de la représentation d'une matrice sont les *lignes* de la matrice, dans l'ordre.

Par exemple la matrice suivante :

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

peut être représentée par la liste de liste : `[[1, 2, 3],[4, 5, 6],[7, 8, 9]]`

en ajoutant des retours chariot apres chaque ligne, la correspondance est évidente :

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

Avant de continuer, nous définissons un alias de type pour les matrices entières.

```
Matrice = List[List[int]]
```

En plus du type des matrices entières, nous devrions imposer une contrainte supplémentaire : que les lignes des matrices sont bien toutes de la même longueur. Cette information n'est en général pas caractérisée par les systèmes de type des langages de programmation courants. Plus généralement, on devrait pouvoir garantir la préservation des *dimensions* des matrices pendant les calculs. Pour le reste du chapitre nous considérerons que ces contraintes de dimensions sont respectées, mais en pratique il faudrait pouvoir les tester.

Si l'on veut accéder à l'unique élément situé à la fois dans la ligne numéro `i` et dans la colonne numéro `j` de la matrice, appelé *élément d'indice* (i, j), ce que l'on écrira `m[i][j]` pour une matrice `m` représentée avec des listes de listes. Cette expression, équivalente à `(m[i])[j]` accède d'abord aec `m[i]` à l'élément d'indice `i` de `m`, qui est une ligne de la matrice, et donc une liste. Dans cette liste `m[i]` nous récupérons l'élément d'indice `j` qui sera bien l'élément d'indice (i, j).

Une opération classique des matrices est la *transposition*, ici avec l'objectif de l'effectuer *en place*. Le principe consiste à échanger le coefficient d'indice (i, j) avec le coefficient d'indice (j, i), pour tous les i et j dans les dimensions considérées. On notera que les éléments de la *diagonale* de la matrice (ceux de ligne i et de la colonne i pour tout i)

2. La bibliothèque *NumPy* pour Python permet le calcul matriciel numérique avec une emphase sur l'efficacité, cf. https://numpy.org/

restent constants par transposition. On notera aussi que si la matrice initiale possède `n` lignes et `m` colonnes, sa transposée aura `m` lignes et `n` colonnes. Bien sûr, notre objectif est de réaliser cette transposition *en place*, c'est-à-dire en modifiant la matrice directement en mémoire. Par un principe d'échange entre les éléments (i, j) et (j, i) de la matrice, il est assez simple de résoudre ce problème.

```
def transposer(m : Matrice) -> Matrice:
    """***Procédure***
    Tanspose la matrice m
    """
    i : int
    for i in range(len(m)):
        j : int
        for j in range(i):
            # échanger (i,j) avec (j,i)
            temp : int = m[i][j]
            m[i][j] = m[j][i]
            m[j][i] = temp
    return m
```

Nous avons choisi de retourner la matrice transposée, mais il ne faut pas oublier que c'est la même matrice que celle passée en argument, et qui a été modifiée directement en mémoire. Ceci est important pour élaborer le jeu de test.

```
# Jeu de tests
mat_trans : Matrice
mat_trans = [[1, 2, 3],
             [4, 5, 6],
             [7, 8, 9]]

assert transposer(mat_trans) == [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
assert(mat_trans) == [[1, 4, 7],
                      [2, 5, 8],
                      [3, 6, 9]]
```

Exercice : définir la fonction `transposee` qui, à partir d'une matrice `m`, retourne une *nouvelle* matrice qui en est la transposée. Comparer le jeu de tests avec celui défini pour la procédure `transposer`. Quelle version est selon-vous la plus efficace ?

12.3.3.2 Exemple : le jeu de la vie

De nombreux jeux vidéos se basent sur une représentation matricielle du «monde». C'est le cas par exemple pour les jeux de plateaux simulés sur les ordinateurs (les échecs, les dames), ou encore les jeux de nombres comme le sudoku, le très ludique 2048, etc. Les images en deux dimensions sont également des matrices de *pixels*. Dans cette section nous allons implémenter le *jeu de la vie*[3] du mathématicien John H. Conway (1937-1920) qui n'est pas un jeu au sens litéral du nom, mais qui reste très amusant à expérimenter.

3. cf. https://fr.wikipedia.org/wiki/Jeu_de_la_vie

Le jeu de la vie est ce que l'on appelle un *automate cellulaire* dont le fonctionnement est très simple. On considère une grille en deux dimensions que l'on représentera par une matrice carrée. Chaque élément de la grille/matrice est ce que l'on appelle une *cellule*. Dans le jeu de la vie de base chaque cellule est soit vivante, soit morte. Pour représenter l'état d'une cellule, nous allons utiliser des caractères :

— le caractère `'.'` pour une cellule morte (ou l'absence de cellule, nous ne faisaons pas de distinction pour l'instant),
— et le caractère `'O'` pour une cellule vivante.

Pour simplifier l'écriture des types nous allons utiliser l'alias suivant :

```
Grille = List[List[str]]
# les éléments sont des caractères représentant l'état d'une cellule
```

On supposera dans la suite la contrainte supplémentaire que la grille est une matrice carrée. Voici un exemple d'une grille pour le jeu de la vie :

```
oscil : Grille = [['.', 'O', '.', '.'],
                  ['.', '.', 'O', 'O'],
                  ['O', 'O', '.', '.'],
                  ['.', '.', 'O', '.']]
```

Cette grille est de dimension 4 (matrice de 4 lignes et 4 colonnes), et elle contient 6 cellules vivantes.

Pour obtenir la génération $k+1$ de cellules à partir de la génération k, les règles du jeu de la vie sont les suivantes :

— une cellule morte ou vide `'.'` nait (ou renait ?) à la génération $k+1$ si elle possède *exactement* 3 voisines vivantes à la génération k,
— une cellule vivante `'O'` ne survit à la génération $k+1$ que si elle a 2 ou 3 voisines vivantes à la génération k.

Les *voisines* d'une cellule donnée sont les 6 cellules (au maximum) qui l'entourent. On ne considère pas bien sûr qu'il soit possible de sortir de la grille.

12.3.3.2.1 Modèle fonctionnel du jeu de la vie

Pour commencer, nous allons définir une fonction `voisines_vivantes` qui compte le nombre de cellules vivantes dans une grille autour d'une cellule donnée, repérée par ses coordonnées dans la matrice. Elle utilisera elle-même une petite fonction permettant d'«évaluer» une cellule.

```
def ceval(g : Grille, i : int, j : int) -> int:
    """Retourne la valeur de la cellule à la ligne i et la colonne j
    dans la grille. Si les coordonnées sont incorrectes la
    valeur 0 est retournée.
    """
    if 0 <= i < len(g) and 0 <= j < len(g)and g[i][j] == 'O':
        return 1  # seules les cellules vivantes comptent
    else:
```

```
        return 0  # tout le reste ne compte pas
                  # (cellules mortes, extérieur de la grille)
```

```
# Jeu de tests
assert ceval(oscil, 0, 0) == 0
assert ceval(oscil, 2, 1) == 1
assert ceval(oscil, -1, 3) == 0
```

La définition de la fonction de comptage devient simple :

```
def voisines_vivantes(g : Grille, i : int, j : int) -> int:
    """Précondition : 0 <= i < len(g)
    Précondition : 0 <= j < len(g)
    Retourne le nombre de cellules vivantes autour de la cellule
    située ligne i et colonne j.
    """
    return \
      ceval(g, i-1, j-1) + ceval(g ,i, j-1) + ceval(g, i+1, j-1)  \
    + ceval(g, i-1, j)                      + ceval(g, i+1, j)  \
    + ceval(g, i-1, j+1) + ceval(g, i, j+1) + ceval(g, i+1, j+1)
```

```
# Jeu de tests
assert voisines_vivantes(oscil1, 0, 0) == 1
assert voisines_vivantes(oscil1, 1, 3) == 1
assert voisines_vivantes(oscil1, 2, 3) == 3
assert voisines_vivantes(oscil1, 2, 2) == 4
```

A partir de cela, nous pouvons assez simplement définir une version fonctionnelle pour le passage d'une génération à l'autre.

```
def evolution(g : Grille) -> Grille:
    """Précondition : len(g) > 0
    Retourne la génération suivante de `g` dans le jeu de la vie.
    """
    # La génération suivante, pour l'instant vide
    # et de même dimension que la grille g
    gs : Grille = [['.' for _ in g] for _ in g]

    i : int # numéro de ligne
    for i in range(len(g)):
        j : int # numéro de colonne
        for j in range(len(g[i])):
            # Règle 1 : renaissance d'une cellule morte
            if g[i][j] == '.' \
               and voisines_vivantes(g, i, j) == 3:
                gs[i][j] = 'O'
            # Règle 2 : survie d'une cellule vivante
            elif g[i][j] == 'O' \
```

```
                and voisines_vivantes(g, i, j) in {2, 3}:
                gs[i][j] = 'O'
            # dans tous les autres cas c'est une cellule morte
            # et la grille gs n'a pas besoin d'être modifiée

    return gs
```

Exceptionnellement nous n'allons pas effectuer de test, mais simplement «jouer» avec notre jeu de la vie.

```
>>> evolution(oscil)
[['.', '.', 'O', '.'],
 ['O', '.', 'O', '.'],
 ['.', 'O', '.', 'O'],
 ['.', 'O', '.', '.']]
```

```
>>> evolution(evolution(oscil))
[['.', 'O', '.', '.'],
 ['.', '.', 'O', 'O'],
 ['O', 'O', '.', '.'],
 ['.', '.', 'O', '.']]
```

Ce jeu de la vie est intéressant car il s'agit d'un oscillateur de fréquence 4, c'est-à-dire que la 4ème génération est identique à la première, ce que nous pouvons vérifier de la façon suivante :

```
>>> evolution(evolution(evolution(evolution(oscil)))) == oscil
True
```

12.3.3.2.2 Le jeu de la vie en place Le jeu de la vie *en place* est intéressant car il s'accompagne d'une «petite» difficulté intéressante, qui montre que le passage du fonctionnel au procédural n'est pas toujours évident. Notre objectif dans cette section est de calculer la génération $k+1$ du jeu de la vie en modifiant directement la grille de la génération k, donc sans la reconstruire comme avec `gen_suivante`. Le problème ici, caractéristique des programmes qui transforment un état comme le jeu de la vie, est que les informations portées par la génération k ne doivent pas être perdues pendant la phase d'évolution.

Prenons un exemple simple avec 9 cellules :

```
small : Grille = [['O', '.', 'O'],
                  ['O', 'O', '.'],
                  ['.', '.', '.']]
```

La première cellule en $(0, 0)$ est vivante et restera vivante puisqu'elle a deux voisines vivantes. Il n'y a donc pas de modification en place à effectuer. La cellule à sa droite en $(0, 1)$ est morte, mais doit renaître pendant l'évolution car elle à exactement trois voisines. Effectuons cette modification en place dès maintenant.

```
>>> small[0][1] = 'O'
```

```
>>> small
[['O', 'O', 'O'],
 ['O', 'O', '.'],
 ['.', '.', '.']]
```

Considérons maintenant la cellule vivante en `(0,2)`. Si l'on regarde la génération initiale (valeur initiale de la variable `small`), elle devrait mourrir lors de l'évolution car elle ne possédait initialement qu'une seule voisine vivante. Cependant, la modification que nous avons effectuée en place pour la cellule $(0, 1)$ a ajouté une cellule vivante aux voisines. Les deux générations se mélangent de façon problématique: la génération suivante dépend d'elle même et non uniquement de la génération précédente, ce qui correspond à ce que l'on appelle plus généralement un *problème de causalité*.

Une façon élégante de résoudre ce problème, tout en maintenant une solution en place, est de rendre notre représentation un peu plus complexe. En plus des cellules mortes `'.'` ou vivantes `'O'` nous allons introduire deux états intermédiaires :

— les cellules *mourantes* `'#'` qui devront disparaître à la génération $k+1$, mais qui sont *encore* vivante à la génération k,
— les cellules *naissantes* `'o'` qui sont mortes à la génération k mais qui devront renaître à la génération $k+1$.

En se basant sur cette nouvelle représentation, le passage d'une génération à la suivante se déroulera en deux temps :

1. dans la phase d'*évolution* les cellules meurent (avec `'#'`), (re)naissent (avec 'o') ou ne changent pas d'état
2. dans la phase de *nettoyage* les cellules mourantes naissent effectivement (avec `'O'`) et les cellules mourantes finissent en `'.'` (mais en gardant l'espoir de renaître dans de futures générations)

Le changement de représentation nous impose de reprendre notre fonction d'évaluation :

```
def ceval2(g : Grille, i : int, j : int) -> int:
    """cf. ceval ci-dessus."""
    if 0 <= i < len(g) and 0 <= j < len(g)and g[i][j] in {'O', '#'}:
        return 1  # seules les cellules vivantes comptent
    else:
        return 0  # tout le reste ne compte pas
                  # (cellules mortes, extérieur de la grille)
```

Pour notre jeu de tests, vérifions de façon exhaustive que le comportement de la fonction initiale ne change pas sur notre exemple d'oscillateur (ce qui nous donne un exemple intéressant de compréhension d'ensemble).

```
# Jeu de tests
assert {ceval2(oscil, i, j) == ceval(oscil, i, j)
```

```
        for i in range(len(oscil)) for j in range(len(oscil))} \
       == {True}
```

Il faut aussi modifier `voisines_vivantes` pour qu'elle utilise cette nouvelle version.

```
def voisines_vivantes2(g : Grille, i : int, j : int) -> int:
    """ cf. voisines_vivantes ci-dessus
    """
    return \
      ceval2(g, i-1, j-1) + ceval2(g ,i, j-1) + ceval2(g, i+1, j-1)  \
    + ceval2(g, i-1, j)                       + ceval2(g, i+1, j)  \
    + ceval2(g, i-1, j+1) + ceval2(g, i, j+1) + ceval2(g, i+1, j+1)
```

Nous pouvons maintenant définir la première phase d'évolution, qui est assez proche de la fonction `evolution` mais qui effectue des modifications en place. Pour bien marquer ce fait nous choisissons de retourner `None` et non directement la grille modifiée.

```
def evoluer(g : Grille) -> None:
    """***Procédure***
    Précondition : len(g) > 0
    Modifie `g` pour produire la génération suivante dans le jeu de la vie.
    """
    i : int # numéro de ligne
    for i in range(len(g)):
        j : int # numéro de colonne
        for j in range(len(g[i])):
            # Règle 1 : renaissance d'une cellule morte
            if g[i][j] == '.'\
               and voisines_vivantes2(g, i, j) == 3:
                g[i][j] = 'o'  # Naissance en prévision seulement
                # sinon la cellule reste morte, rien à faire
            # Règle 2 <<modifiée>> : décès d'une cellule vivante
            elif g[i][j] == 'O'\
                 and voisines_vivantes(g, i, j) not in {2, 3}:
                g[i][j] = '#' # décès en prévision
                # sinon la cellue reste vivante
```

Par rapport à la version fonctionnelle, les modifications se font donc en place avec les nouveaux états *mourantes* et *naissantes* mais surtout nous avons modifié la deuxième règle pour que le décès soit pris en compte plutôt que la survie.

Regardons ce que produit cette procédure sur l'exemple d'oscillateur.

```
>>> evoluer(oscil)
>>> oscil
[['.', '#', 'o', '.'],
 ['o', '.', 'O', '#'],
 ['#', 'O', '.', 'o'],
 ['.', 'o', '#', '.']]
```

Ici on obtient un état qui est à mi-chemin des deux générations, et qu'il nous faut maintenant «nettoyer» pour que le cycle de la vie se poursuive : les mourantes meurent et les naissantes naissent.

```
def nettoyer(g : Grille) -> None:
    """***Procédure***
    Précondition : len(g) > 0
    Nettoye `g` pour obtenir après évolution la génération suivante
    dans le jeu de la vie.
    """
    i : int # numéro de ligne
    for i in range(len(g)):
        j : int # numéro de colonne
        for j in range(len(g[i])):
            # Naissance
            if g[i][j] == "o":
                g[i][j] = "O"
            elif g[i][j] == "#":
                g[i][j] = '.'
            # sinon on n'a rien à faire
```

Effectuons le nettoyage de notre oscillateur pour la deuxième génération :

```
>>> nettoyer(oscil)
>>> oscil
[['.', '.', 'O', '.'],
 ['O', '.', 'O', '.'],
 ['.', 'O', '.', 'O'],
 ['.', 'O', '.', '.']]
```

Passons à la troisième génération maintenant :

```
>>> evoluer(oscil)
>>> nettoyer(oscil)
>>> oscil
[['.', 'O', '.', '.'],
 ['.', '.', 'O', 'O'],
 ['O', 'O', '.', '.'],
 ['.', '.', 'O', '.']]
```

Et finalement la quatrième génération :

```
>>> evoluer(oscil)
>>> nettoyer(oscil)
>>> oscil
[['.', '.', 'O', '.'],
 ['O', '.', 'O', '.'],
 ['.', 'O', '.', 'O'],
 ['.', 'O', '.', '.']]
```

Cette génération devrait être la même que la génération de départ, malheureusement nous sommes en style procédural, nous avons perdu la génération de départ ! Ce qui nous conduit tout naturellement à la problématique de test.

Finalement, la meilleure façon de tester la version *en place* n'est-t-elle pas d'utiliser la version fonctionnelle ? C'est l'idée sous-jascente du jeu de tests suivant :

```
# Jeu de test
oscil : Grille = [['.', 'O', '.', '.'],
                  ['.', '.', 'O', 'O'],
                  ['O', 'O', '.', '.'],
                  ['.', '.', 'O', '.']]
# Effectuons une copie de la grille initiale
oscil_init : Grille = [[e for e in ligne] for ligne in oscil]
# génération 1
assert evoluer(oscil) == None
assert nettoyer(oscil) == None
assert oscil == evolution(oscil_init)
# génération 2
assert evoluer(oscil) == None
assert nettoyer(oscil) == None
assert oscil == evolution(evolution(oscil_init))
# génération 3
assert evoluer(oscil) == None
assert nettoyer(oscil) == None
assert oscil == evolution(evolution(evolution(oscil_init)))
# génération 4
assert evoluer(oscil) == None
assert nettoyer(oscil) == None
assert oscil == oscil_init
```

Ce test n'est bien sûr par complet, mais il montre de façon exhaustive que notre oscillateur est bien un oscillateur de fréquence 4. En comparant une dernière fois la version fonctionnelle et la version procédurale, nous pouvons remarquer que :

— la version fonctionnelle est conceptuellement plus simple, avec une représentation plus simple et sans problème possible de causalité. De plus, les tests sont simples à réaliser puisque l'on n'a pas besoin de réfléchir à l'observation éventuelle d'effets de bord. En revanche, à chaque calcul d'évolution il faut recopier intégralement la grille en entrée. Ceci permet, cependant, de ne pas avoir à la sauvegarder.

— la version procédurale, avec modifications en place, est plus complexe, et nécessite un changement de représentation pour éviter un problème de causalité que l'on rencontre souvent dans ce style de programmation. Les tests sont plus complexes à écrire car il faut penser à observer les effets de bord, et aussi à sauvegarder les informations utiles aux tests, comme les générations précédentes du jeu de la vie. Le principal gain est qu'il n'est plus nécessaire de reconstruire la grile. En revanche, nous devons désormais effectuer deux passages complexes dans la matrice : d'abord l'évolution et ensuite le nettoyage.

Pour le jeu de la vie, on peut penser que pour les grilles de très grandes tailles (disons, à vue de nez, de l'ordre de centaines de milliers pour la dimension) la version procédurale, malgré ses deux passages, sera plus efficace que la version fonctionnelle car les allocations mémoires sont relativement coûteuses. Pour les plus petites grilles, la version fonctionnelle l'emporte pas sa simplicité.

Dans tous les cas, nous avons illustré l'intérêt de développer un modèle fonctionnel du problème, *y compris* si l'on vise une plus grande efficacité avec des procédures et des modifications en place. Nous aurons en effet l'outil le plus intéressant pour les tests : l'utilisation du modèle fonctionnel plus simple et parfois moins efficace, pour tester le modèle procédurale parfois plus efficace mais invariablement plus complexe.

Chapitre 13

Vers les objets

Ce dernier chapitre constitue une ouverture vers la notion de *paradigme de programmation* et en particulier la notion de *programmation objet*. Il s'agit également d'en découvrir un peu plus sur ce langage particulièrement flexible qu'est Python, un très bon exemple de ce que l'on appelle un *langage multi-paradigmes*.

13.1 La programmation orientée objet (POO)

Dans les années 90, les paradigmes pourtant bien installés de programmation procédurale impérative et de programmation fonctionnelle récursive ont été presque «balayés» (heureusement temporairement) par le concept d'**objet**. Le constat de départ était globalement une critique de l'approche procédurale : les effets de bord incontrôlés posent d'importants problèmes de fiabilité. D'une certaine façon, le constat est donc le même que celui de la programmation fonctionnelle : il faut confiner les effets de bord. La solution proposée, en revanche, est radicalement différente pour ce qui concerne les objets. La principale idée est de rapprocher les données – donc les variables – et les traitements sur ces données – donc les procédures – au sein d'une entité unique appelée *Objet*. Le bénéfice est immédiat : on peut contrôler finement et confiner les effets de bords. Les langages popularisés dans les années 90 sont des *langages orientée objet*. Ce sont notammment le C++ (dès le début des années 90) et Java (à partir de 1995). Python a également été conçu à l'origine - Python 1.0 a été diffusé en 1994 - comme un langage objet. Voici le descriptif proposé sur la *foire aux questions* (FAQ) de Python : Python est un langage de programmation orienté-objet interprété et interactif. Cependant, Python est souvent décrit et utilisé aujourd'hui comme un langage *multi-paradigmes* qui permet de mélanger les différents styles de programmation. Nous avons largement exploité cette caractéristique en Python puisque les objets définis par l'utilisateur n'arrivent qu'au dernier chapitre !

13.2 Objets du monde réel

Le concept d'*objet* utilisé dans le cadre de la programmation a pour origine le langage *Simula-67* (oui, 1967 !) dont l'objectif était de faciliter l'écriture de programmes de simulation. Dans un programme de simulation, l'idée est de modéliser une situation du monde réel, par exemple un système de transport, et de simuler son comportement par un programme informatique. Pour illustrer ce point de vue, considérons la modélisation d'un tel objet du monde réel : le feu de signalisation.

Un **feu tricolore de signalisation** fournit une information de couleur permettant de réguler la circulation notamment à un carrefour. En France, ces informations de couleur sont les suivantes :

- la couleur *rouge* ferme la circulation
- la couleur *vert* ouvre la circulation
- la couleur *orange* est une couleur transitoire du *vert* au *rouge*. La circulation est ouverte uniquement aux véhicules qui ne peuvent s'arrêter sans danger avant le *rouge*.

Dans le cadre d'une simulation, la couleur correspond à une donnée que l'on stockera donc dans une variable. Le comportement du feu se résume à une notion de *changement de couleur*.

Une traduction en langage Python très fidèle à cette description est donnée ci-dessous.

```
class FeuTricolore:
    """Représentation d'un feu tricolore de circulation."""

    def __init__(self):
        """Constructeur pour feu tricolore initialement au rouge."""
        self._couleur = 'rouge'  # attribut privé pour la couleur, de type str

    def couleur(self):
        """ self -> str
        Accesseur pour la couleur du feu."""
        return self._couleur

    def change(self):
        """ self -> NoneType
        Change le feu (cycles vert-orange-rouge)."""
        if self.couleur() == 'vert':
            self._couleur = 'orange'
        elif self.couleur() == 'orange':
            self._couleur = 'rouge'
        elif self.couleur() == 'rouge':
            self._couleur = 'vert'
```

```
        else:
            raise Exception('Mauvaise couleur')
```

Un objet feu tricolore est une entité spécifique : dans un réseau de transport sont placés plusieurs feu tricolores, donc plusieurs objets, chacun dans un *état* particulier : *rouge*, *vert* ou *bleu*. Du point de vue de la programmation, l'objectif est de décrire les *caractéristiques communues* de ces nombreux feux tricolores. En programmation orientée objet la description de ces caractéristiques communes se nomme un *classe*. En Python, les classes sont introduites par l'intermédiaire du mot-clé `class`.

La classe `FeuTricolore` définie ci-dessus représente les propriétés communes des objets représentant un feu tricolore, notamment :

— la construction d'un nouvel objet par l'intermédiaire du *constructeur* `__init__` de Python. Par exemple, chaque nouveau feu tricolore est initialisé à la couleur rouge.
— les *attributs privés* qui sont des variables représentant les données spécifiques à l'objet, qui ne sont normalement pas accessibles depuis l'extérieur. Dans l'exemple, l'attribut privé se nomme `self._couleur` et contient une chaîne de caractère représentant la couleur.
— les *méthodes* qui sont des sortes de fonctions spécifiques permettant de manipuler les objets. Dans l'exemple, on a une méthode `couleur` qui permet de «lire» la couleur actuelle du feu ainsi qu'une méthode `change` pour faire cycler la couleur du feu.

Pour construire un nouvel objet – donc ici un feu tricolore particulier – on effectue une sorte d'appel de la classe.

```
# feu1 : FeuTricolore
feu1 = FeuTricolore()
```

On a construit ici un nouvel objet de la classe `FeuTricolore`. Nous avons stocké cet objet en mémoire en l'associant à une variable `feu1`.

D'un point de vue terminologique, on dit que l'objet référencé par la variable `feu1` est une *instance* de la classe `FeuTricolore`. Vérifions ce fait :

```
>>> isinstance(feu1, FeuTricolore)
True
```

Une *invocation de méthode* consiste à appeler une méthode définie dans la classe sur un objet. La syntaxe générale est proche des appels de fonctions :

```
<objet>.<methode>(<argument1>, ..., <argumentN>)
```

La différence avec une fonction est que le «véritable» premier argument de la fonction est en fait `<objet>` – on dit parfois qu'il s'agit de l'*argument implicite* de l'appel – et ne viennent ensuite que les arguments d'appel : `<argument1>, ..., <argumentN>`.

Par exemple, on peut lire la couleur du feu en invoquant la méthode `couleur` sur l'objet référencé par `feu1` :

```
>>> feu1.couleur()
'rouge'
```

Ici, l'argument implicite est l'objet référencé par `feu1` et il n'y a aucun autre argument d'appel. Dans la définition de la méthode `couleur`, cet argument implicite est rendu explicite par l'identifiant `self` qui signifie : *l'objet lui-même*. Par soucis de simplicité, on dira que le type de `self` est également `self`, qui est un type compatible avec le type de la classe, ici `FeuTricolore`.

Au passage, nous confirmons bien ici que le feu est initialement de couleur rouge.

Construisons maintenant un deuxième feu tricolore :

```
# feu2 : FeuTricolore
feu2 = FeuTricolore()
```

Un point important ici est que ce nouveau feu est indépendent du précédent. Ils ont certes des caractéristiques communes – notamment les méthodes que l'on peut invoquer – mais leur état est distinct. Essayons de vérifier ce fait.

Pour l'instant, les deux feu sont dans un état semblable : leur couleur est rouge.

```
>>> feu2.couleur()
'rouge'
```

Essayons de modifier la couleur d'un des feux.

```
>>> feu1.change()
```

Ici Python ne retourne rien, ce qui est le signe d'un effet de bord non-confiné. Quelque chose a donc du se passer en mémoire.

Pour `feu2` rien ne s'est produit :

```
>>> feu2.couleur()
'rouge'
```

En revanche, pour `feu1` la couleur a été modifiée :

```
>>> feu1.couleur()
'vert'
```

Les deux feux sont dans des couleurs différentes, on constate donc bien que leurs états sont indépendants. C'est fort heureux puisque sinon les feux d'un réseau de transport devraient tous êtres synchronisés !

On peut bien sûr encore modifier les feux :

```
>>> feu1.change()
```

```
>>> feu1.couleur()
'orange'
```

```
>>> feu2.couleur()
'rouge'
```

```
>>> feu2.change()
```

```
>>> feu1.couleur()
'orange'
```

```
>>> feu2.couleur()
'vert'
```

Ici il y a clairement de nombreux effets de bord : les couleurs sont modifiées. En revanche, ces modifications ne se font pas arbitrairement, il est nécessaire d'utiliser la méthode `change` de la classe `FeuTricolore`. Donc il n'est pas possible (en tout cas pas simplement) de rendre l'état du feu incohérent.

13.3 Types de données utilisateur

Au delà de la modélisation objet illustrée précédemment, l'autre intérêt de la définition de classe concerne la création de types de données définis par le programmeur.

Dans ce livre, nous avons jusqu'à présent uniquement utilisé des types de données prédéfinis en Python :

- les types simples `bool`, `int`, etc.
- les séquences `range`, `str` et `list`[α]
- les n-uplets `tuples`[$\alpha_1, \alpha_2, \ldots, \alpha_n$]
- les ensembles `set`[α]
- les dictionnaires `dict`[α:β]

Ces types sont en fait implémentés par des classes Python. On peut vérifier ce fait par la primitive `isinstance` :

```
>>> isinstance(1, int)
True
```

```
>>> isinstance([1, 2, 3], list)
True
```

Lorsque l'on crée une nouvelle classe, on définit par la même occasion un nouveau type de données. Par exemple, les objets feux tricolores sont tous des objets instances de la même classe `FeuTricolore`. Et les variables qui les référencent, notamment `feu1` et `feu2` dans nos précédents exemples sont du type `FeuTricolore`.

Dans le reste de cette section, nous exploitons cette caractéristique pour définir de nouveaux types Python.

13.3.1 Enregistrements

Dans les chapitre sur les n-uplets, nous avons souvent défini des *alias de type* permettant de faciliter les écritures. Considérons à nouveau l'alias de type `Personne` que nous avons utilisé dans le chapitre 7.

```
# type Personne = tuple[str, str, int, bool]
```

Plutôt que de travailler directement avec des n-uplets de type `Personne`, il est possible de définir un nouveau type `Personne` *au sens du langage Python* par l'intermédiaire d'une classe éponyme.

```
class Personne:
    """ Représentation d'une personne dans une base de données."""

    def __init__(self, nom, prenom, age, marie):
        """Construit une personne."""
        # attributs privés
        self._nom = nom  # type str
        self._prenom = prenom  # type str
        self._age = age  # type int
        self._statue_marital = marie  # type bool

    def nom(self):
        """ self -> str
        Accesseur pour le nom."""
        return self._nom

    def prenom(self):
        """ self -> str
        Accesseur pour le prénom."""
        return self._prenom

    def age(self):
        """ self -> str
        Accesseur pour l'age."""
        return self._age

    def est_marie(self):
        """ self -> bool
        Retourne True si la personne est mariée,
        ou False sinon."""
        return self._statut_marital

    def anniversaire(self):
        """ self -> NoneType
        Incrémente l'age."""
        self._age = self._age + 1
```

```
    def mariage(self):
        """ self -> NoneType
        Marie la personne."""
        self._statut_marital = True
```

En complément des accesseurs pour les quatre attributs des objets personnes (nom, prénom, age et statut marital), on a ajouté des méthodes pour modifier l'age (à leur date d'anniversaire) ainsi que le statut marital.

Voici un exemple d'interaction avec une personne :

```
# pers : Personne
pers = Personne('Yoda', 'Yoda', 700, False)
```

```
>>> pers.nom()
'Yoda'
```

```
>>> pers.age()
700
```

```
>>> pers.anniversaire()
```

```
>>> pers.age()
701
```

L'avantage par rapport aux alias de type pour les n-uplets est que l'on n'a plus besoin de se rappeler quelle est la position des informations dans le n-uplet. Par exemple, pour retrouver l'âge d'une personne `p` on n'a pas besoin de se souvenir qu'il s'agit du 3ème élément d'un quadruplet, mais simplement que l'âge s'obtient par l'écriture `p.age`. De plus, on a ajouté la possibilité de modifier de façon contrôlée certains attributs des personnes, notamment leur age et leur statut marital.

13.3.2 Types numériques

Python est un langage de programmation très largement diffusé et utilisé notamment dans le domaine scientifique. Une caractéristique du langage semble avoir joué un rôle important dans ce succès : la possibilité de définir de nouveaux types numériques.

Pour illustrer ce point, nous allons définir un nouveau type pour les rationnels.

```
class Ratio:
    def __init__(self, n, m):
        """ Construit un rationnel."""
        if m == 0:
            raise ZeroDivisionError()

        self._quo = n  # quotient du rationnel, type int
        self._div = m   # diviseur du rationnel, type int
```

```
        self._normalisation()

    def _normalisation(self):
        """ self -> NoneType
        Normalise le rationnel."""

        # 1) calcul du pgcd

        # p : int
        p = self._quo
        # q : int
        q = self._div

        while q > 0:
            p, q = (q, p % q)

        # 2) normalisation

        self._quo = self._quo // p
        self._div = self._div // p

    def __mul__(self, r):
        """ self * Ratio -> Ratio
        Multiplication par le rationnel r."""
        return Ratio(self._quo * r._quo, self._div * r._div)

    def __floordiv__(self, r):
        """ self * Ratio -> Ratio
        Division par le rationnel r."""
        return Ratio(self._quo * r._div, self._div * r._quo)

    def __add__(self, r):
        """ self * Ratio -> Ratio
        Addition avec le rationnel r."""
        ### en exercice ! ####

    def __sub__(self, r):
        """ self * Ratio -> Ratio
        Soustraction par le rationnel r."""
        ### en exercice ! ###

    def __repr__(self):
        """ self -> str
        Retourne la représentation textuelle du
        rationnel."""
        return "{}/{}".format(self._quo, self._div)
```

Voici quelques exemples de manipulation de rationnels :

```
r1 = Ratio(14, 4)
```

```
>>> r1
7/2
```

```
r2 = Ratio(384, 144)
```

```
>>> r2
8/3
```

```
>>> r1 * r2
28/3
```

```
>>> r1 // r2
21/16
```

13.3.3 Types itérables

Dans le chapitre 12, nous avons introduit la notion d'*itérable*. Un itérable est une structure de donnée que l'on peut parcourir dans le cadre d'une boucle d'itération `for` ou une compréhension de liste, d'ensemble ou de dictionnaire.

Une fonctionnalité très intéressante du langage Python est de permettre, encore une fois par l'intermédiaire d'une classe, la définition d'un nouveau *type itérable*.

Nous allons prendre l'exemple d'un type `CharRange` permettant les itérations sur les intervalles de caractères (en complément de `Range` qui ne permet que les itérations sur les intervalles de nombres).

```
class CharRange:
    def __init__(self, cmin, cmax):
        """ Construit un intervalle de caractères,
        entre le caractère cmin minimal et le caractère
        cmax maximal (et inclus) dans l'intervalle."""
        self._cmin = cmin  # type str (longueur 1)
        self._cmax = cmax   # type str (longueur 1)

        self._current = cmin  # type str (longueur 1)
                              # caractère courant
                              # dans l'itération

    def __iter__(self):
        """ indique qu'il s'agit d'un itérateur. """
        return self

    def __next__(self):
        """ retourne le prochain caractère dans l'itération."""
        prochain = self._current
```

```
        if ord(self._current) == ord(self._cmax) + 1:
            raise StopIteration()
        self._current = chr(ord(self._current) + 1)
        return prochain
```

Les méthodes spéciales `__iter__` et `__next__` permettent de s'intégrer au *protocole d'itération* de Python. Sans entrer dans les détails d'implémentation (qui demandent quelques approfondissements Python), illustrons l'utilisation du type `CharRange` en pratique.

Commençons par une fonction qui retourne une liste formée de caractères itérés dans un intervalle :

```
def liste_de_caracteres(CR):
    """ CharRange -> list[str]
    Retourne la liste des caractères dans l'intervalle CR."""

    # L : list[str]
    L = []  # la liste résultat

    # c : str  (caractère courant)
    for c in CR:
         L.append(c)

    return L
```

```
# Jeu de tests
assert liste_de_caracteres(CharRange('a', 'e')) == ['a', 'b', 'c', 'd', 'e']
assert liste_de_caracteres(CharRange('a', 'a')) == ['a']
assert liste_de_caracteres(CharRange('b', 'a')) == []
```

Dans la fonction `liste_de_caracteres` on a utilisé le paramètre de type `CharRange` comme itérable dans une boucle `for` d'itération.

On peut en fait directement construire les listes de caractères en utilsant des compréhensions de listes :

```
>>> [ c for c in CharRange('a', 'e')]
['a', 'b', 'c', 'd', 'e']
```

```
>>> [ c for c in CharRange('a', 'a')]
['a']
```

```
>>> [ c for c in CharRange('b', 'a')]
[]
```

13.4 Aller plus loin ...

Les exemples précédents permettent de «toucher du doigt» la *force* d'un langage de programmation comme Python : ses possibilités d'extension. Mais pour pleinement exploiter cette richesse, notre cours d'introduction aux concepts élémentaires de programmation n'est pas suffisant.

Les étudiants intéressés ont maintenant des connaissances suffisantes pour aborder un apprentissage plus spécifique du langage Python. Parmi les livres que nous recommandons, citons :

Think Python

un projet de *livre libre* initié par *Allen B. Downey*

Il s'agit d'un très bon complément de notre livre, moins axé sur la résolution de problèmes et l'algorithmique mais qui propose quelques approfondissement concernant le langage Python. Le livre reste proche d'un enseignement universitaire et aborde donc des questions plus générales que simplement lister les possibilités du langage Python. Ce livre est en anglais, disponible en PDF à l'adresse suivante :

https://greenteapress.com/wp/think-python-2e

Une version interactive très intéressante (toujours en anglais) est disponible à l'adresse suivante :

http://interactivepython.org/runestone/static/thinkcspy/toc.html

Il existe également une variante francophone :

https://allen-downey.developpez.com/livres/python/pensez-python

Pour aller ensuite plus loin en Python, de nombreux ouvrages sont disponibles mais le plus souvent uniquement en langue anglaise.

Une exception notable, un livre de niveau avancé traite de Python pour le traitement des données, disponible chez le même éditeur que notre ouvrage.

Python avancé et programmation scientifique - Techniques d'algorithmisation et de construction de programmes compacts et efficaces par *Karczmarczuk Jerzy* aux éditions Ellipses (paru le 01.10.2019)

Enfin, n'hésitez pas à glaner des information sur la page principale du langage :

https://www.python.org/