Eléments de Programmation (en Python)

Sorbonne Université – Licence 1 – 2019/2020

Table des matières

Avant-propos

Cet ouvrage :

Eléments de programmation (en Python)

est le support principal du cours *Eléments de programmation I* proposé aux étudiants de licence première année (premier semestre) à Sorbonne Université (ex. Université Pierre et Marie Curie).

Dispensé à plus de 1700 étudiants en sciences chaque année, ce cours d'introduction à l'informatique utilisait précédemment (de 1999 à 2013) le langage Scheme pour illustrer les concepts du cours. Pour des raisons diverses, il a été décidé de changer de langage support pour adopter Python (version 3) mais la philosophie de ce cours a été préservée.

- le cours n'est pas une énumération des constructions syntaxique du langage de support
- l'accent est mis sur la spécification de problèmes (souvent à thématique scientifique) et leur résolution par la conception d'un algorithme et son implantation dans le langage de support

De ce fait, certaines constructions syntaxiques du langage support ne sont pas abordées (notamment l'essentiel de ce qui concerne la programmation orientée objet). On se concentre sur les constructions fondamentales qui supportent la «pensée» algorithmique : définitions de fonctions, variables et affectation, alternatives, répétitions (boucles), etc. Certains traits avancés du langage Python seront également mis en pratique, notamment les fameuses *compréhensions*. De plus, le sous-ensemble de Python considéré est *statiquement typé*, c'est-à-dire que les erreurs de typage commises par les étudiants ou les enseignants sont signalées.

Cette approche nous semble essentielle dans un cursus universitaire scientifique, où l'algorithmique (et leur programmation pratique) jouera un rôle certain.

Ce cours est donc tout à fait compatible avec un apprentissage plus classique du langage Python 3 à travers ses constructions syntaxiques.

Un ouvrage compagnon :

Eléments de programmation (en Python) – Recueil d'exercices

est également disponible sous les mêmes conditions. A la fin de chaque chapitre du présent ouvrage est proposée une sélection d'exercices corrigés issus de ce recueil compagnon.

MrPython

L'environnement de programmation utilisé par les étudiants dans le cadre de cours est un logiciel libre disponible sur le site suivant : https://github.com/nohtyprm/MrPython

0.1 Le langage Python

Python est un langage de programmation très répandu, utilisé dans le monde scientifique et industriel, et reconnu pour certaines de ses vertus pédagogiques.

D'après wikipedia (en date du 28 août 2014) :

> Python est un langage de programmation objet, multi-paradigme et multi-plateformes. Il favorise la programmation impérative structurée et orientée objet. Il est doté d'un typage dynamique fort, d'une gestion automatique de la mémoire par ramasse-miettes et d'un système de gestion d'exceptions ; il est ainsi similaire à Perl, Ruby, Scheme, Smalltalk et Tcl.
>
> Le langage Python est placé sous une licence libre proche de la licence BSD2 et fonctionne sur la plupart des plates-formes informatiques, des supercalculateurs aux ordinateurs centraux, de Windows à Unix en passant par GNU/Linux, Mac OS, ou encore Android, iOS, et aussi avec Java ou encore .NET. Il est conçu pour optimiser la productivité des programmeurs en offrant des outils de haut niveau et une syntaxe simple à utiliser.
>
> Il est également apprécié par les pédagogues qui y trouvent un langage où la syntaxe, clairement séparée des mécanismes de bas niveau, permet une initiation aisée aux concepts de base de la programmation.

Important : le cours *Eléments de programmation* n'est pas un cours **de** Python mais un cours **en** Python. Le langage est utilisé pour illustrer les concepts et nous ne considérons qu'une partie de ses possibilités. A l'issue de ce cours, il est fortement suggérer d'aller plus loin dans la découverte de ce langage de programmation.

0.2 Plan du cours

- Cours 1 : expressions arithmétiques et définitions de fonctions simples
- Cours 2 : suites d'instructions, variables et alternatives `if`
- Cours 3 : répétitions et boucle `while`
- Cours 4 : plus sur les boucles : correction, terminaison et efficacité
- Cours 5 : intervalles, itérations avec `for` et chaînes de caractères
- Cours 6 : listes
- Cours 7 : n-uplets, plus sur les listes et fonctionnelles
- Cours 8 : compréhensions de listes
- Cours 9 : ensembles et dictionnaires
- Cours 10 : compréhensions d'ensembles et de dictionnaires
- Cours 11 : cours d'ouverture sur les objets

0.3 Contributeurs

Les contributeurs principaux de ce document sont membres de l'équipe pédagogique du cours *Eléments de programmation I* (codifié `LU1NI001`).

- *Frédéric Peschanski*, Maître de conférences en informatique Sorbonne Université / LIP6 (responsable d'édition, contributeur principal)
- *Romain Demangeon*, Maître de conférences en informatique Sorbonne Université / LIP6 (contributeur majeur, responsable du cours)
- *Fabien Tarissan*, Chercheur CNRS (contributeur majeur)
- *Christophe Marsala*, Professeur en informatique Sorbonne Université / LIP6 (contributeur, responsable des supports formation à distance)
- *Antoine Genitrini*, Maître de conférences en informatique Sorbonne Université / LIP6 (contributeur)

— *Maryse Pelletier*, Maître de conférences en informatique Sorbonne Université / LIP6 (contributrice)
— *Clémence Magnien*, Directrice de Recherches en informatique CNRS / Sorbonne Université / LIP6 (contributrice)
— *Choun Tong Lieu*, Maître de conférences en informatique Sorbonne Université / LIP6 (contributeur)
— *Pascal Manoury*, Maître de conférences en informatique Sorbonne Université / LIP6 (contributeur)
— *Mathilde Carpentier*, Maître de conférences en informatique Sorbonne Université (contributrice)
— *Isabelle Mounier*, Maître de conférences en informatique Sorbonne Université / LIP6 (contributrice)
— *Marie-Jeanne Lesot*, Maître de conférences en informatique Sorbonne Université / LIP6 (contributrice, responsable des supports formation à distance)

Nous remercions également l'ensemble des membres de l'équipe pédagogique 1i001 ainsi que l'ensemble des étudiants pour les nombreux retours sur ce cours.

0.4 Licence

1 Premiers pas

1.1 Constituants d'un programme

D'après Wikipedia (en date du 28 août 2014) :

> Un programme informatique est une séquence d'instructions qui spécifie étape par étape les opérations à effectuer pour obtenir un résultat. Il est exprimé sous une forme qui permet de l'utiliser avec une machine comme un ordinateur pour exécuter les instructions. Un programme est la forme électronique et numérique d'un algorithme exprimé dans un langage de programmation - un vocabulaire et des règles de ponctuation destinées à exprimer des programmes.

1.1.1 Exemple de programme

```
def liste_premiers(n):
    """int -> list[int]
       Hypothèse : n >= 0

       Retourne la liste des nombres premiers inférieurs à n."""

    # i_est_premier : bool
    i_est_premier = False  # indicateur de primalité

    # L : list[int]
    L = []   # liste des nombres premiers en résultat

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

        # j : int (candidat diviseur)
        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
```

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

Au-delà du sens exact de ce programme, nous pouvons en extraire les principaux constituants :

— définitions de fonction,
— variables,

— expressions et applications,
— instructions : affectation, alternative, suite d'instructions, boucle.

Remarque: Contrairement à de nombreux autres langages de programmation, le langage Python impose les *indentations* c'est-à-dire les retours de ligne et le nombre d'espaces nécessaires ensuite. Nous reviendrons sur ce point à de nombreuses reprises.

1.1.2 Définition de fonction

La construction :

```
def liste_premiers(n):
    ... etc ...
```

est une **définition de fonction** de nom `liste_premiers` et avec un **paramètre formel** dont le nom est `n`

1.1.3 Variables

L'identifiant `i_est_premier` est un **nom de variable**. Nous verrons les variables au prochain cours.

1.1.4 Expressions

Les constructions telles que `1`, `i-1`, `i % j == 0` sont des **expressions**, ici arithmétiques (`1`, `i-1`, et `i % j`) ou booléennes (`i % j == 0`).

L'écriture :

```
liste_premiers(30)
```

est une expression dite **application de fonction utilisateur**. La fonction appelée se nomme `liste_premiers` et son unique **argument d'appel** est `30`.

1.1.5 Instructions

La construction `i_est_premier = True` est une instruction dite d'**affectation** à la variable de nom `i_est_ premier`.

La construction :

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

est une instruction dite **alternative**. Les alternatives seront vues au cours 2.

La construction :

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

est une instruction dite de **boucle**. Les boucles seront abordées au cours 3.

1.2 Notion d'expression

Nous allons commencer par expliquer 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, c'est à l'ordinateur que l'on s'adresse et non à un autre humain.

Dans ce cours, les expressions sont écrites dans le **langage Python**, mais il existe bien sûr de nombreux autres langages informatiques (C, C++, java, ocaml, etc.).

Les expressions sont 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. En arithmétique, la valeur d'une expression est soit un entier de type `int` ou un flottant de type `float`.

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`.

Par exemple :

```
>>> 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 ci-dessus 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` (entier) mais pour la représentation interne, on dit que l'objet qui représente 42 en mémoire est de la **classe `int`**.

Cette terminologie fait de Python un **langage objet** et nous reviendrons sur ce point, notamment lors du dernier cours d'ouverture sur ce thème.

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.

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` pour l'affichage de l'entier 42.

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 dénoté `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
```

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.

```
>>> 23239287329837298382739284739847394837439487398479283729382392283
23239287329837298382739284739847394837439487398479283729382392283
```

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 cours d'introduction, nous ne nous occuperons pas trop des problèmes liés aux approximations, mais il faut savoir que ces problèmes sont très complexes et sont sources de nombreux *bugs* informatiques, certains célèbres comme le *bug* de la division en flottant du Pentium, cf. http://fr.wikipedia.org/wiki/Bug_de_la_division_du_Pentium.

```
>>> 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 générique* `Number` pour désigner les nombres en général. Nous reviendrons sur ce point dès le prochain cours.

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'
```

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 aborderons lors des prochains cours.

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 cours nous nous limiterons dans nos exemples aux expressions arithmétiques, c'est-à-dire aux expressions usuelles des mathématiques, le langage des calculs simples sur les nombres. Nous aborderons d'autres types d'expressions lors des prochains cours, 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 sans surprise (supplémentaire !) 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 est spécifiée de la façon suivante :

```python
def max(a, b):
  """Number * Number -> Number
     retourne le maximum des nombres a et b."""
```

Par exemple :

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

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

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

- l'**en-tête de la fonction**, `def max(a, b)`, qui donne le *nom de la fonction* (`max`) et de ses *paramètres formels* (`a` et `b`).
- la **signature de la fonction** qui indique ici que le premier paramètre (`a`) est de type `Number`, le second paramètre (`b`) également de type `Number` (l'étoile `*` est le produit cartésien qui sépare les types des paramètres). La signature indique également après la flèche `->` que *la valeur de retour* de la fonction `max` est de type `Number`.
- la **description de la fonction** qui explique le problème résolu par la fonction. Ici, la fonction *«retourne le maximum des nombres a et b»*.

Nous reviendrons longuement dans ce cours 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 en python :

```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**.

> La **carte de référence** résume les fonctions de bibliothèque que vous pouvez utiliser dans le cadre de ce cours. Il s'agit du seul document autorisé lors des épreuves : devoir sur table, TME en solitaire et examen.
>
> **Important** : nous ne distribuons *qu'un seul exemplaire par étudiant* donc prenez soin de votre carte de références et ne la perdez pas !

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 :

```python
import math # bibliothèque mathématique
```

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

```python
# fonction présente dans le module math
def sin(x):
    """Number -> 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.

```python
>>> 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, on retiendra qu'ils sont assez proches de notre propre manière de calculer.

1.3 Définition de fonctions simples

Les fonctions occupent une place centrale en programmation, notamment dans les langages Python et C que vous verrez cette année.

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, en mathématiques on écrirait quelque chose comme ce qui suit :

Le périmètre p d'un rectangle de largeur l et de longueur L est :

$$p = 2 * (l + L)$$

Cette expression arithmétique peut être traduite presque directement en Python, de la manière suivante :

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

    return 2 * (largeur + longueur)
```

Une fois écrite, on peut utiliser la fonction avec quelques exemples:

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

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

De ces exemples d'application de la fonction `perimetre` nous pouvons déduire un **jeu de tests** permettant de 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, longueur)
    int * int -> int
    hypothèse : (longueur >= 0) and (largeur >= 0)
    hypothèse : longueur >= largeur
    retourne le périmètre du rectangle défini par sa largeur et sa longueur.
```

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. l'**en-tête** de la fonction
b. la **signature** de la fonction
c. les **hypothèses** éventuelles pour une bonne application de la fonction
d. 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 cours.

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.

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

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

La ligne

```python
def perimetre(largeur, longueur):
```

se nomme l'**en-tête** 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 de ses *paramètres formels* (ou plus simplement *paramètres* tout court). Pour notre problème de périmètre, les paramètres se nomment naturellement `largeur` et `longueur`.

Le mot-clé `def` de Python introduit la définition d'une fonction.

La partie de la spécification en dessous de l'en-tête se trouve entre des triples guillemets `""" ... """`.

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

Sur la première ligne, directement à proximité des guillemets se trouve la **signature** de la fonction. Cette signature indique :

- les types des paramètres formels, séparés par le symbole `*` (signifiant le *produit cartésien* des types)
- le type de la valeur de retour

Ici, il est indiqué que le premier paramètre formel `largeur` est de type `int`, donc un entier. Le type du deuxième paramètre `longueur` est également `int` (pour simplifier on ne calcule que des périmètres entiers). Le type de la *valeur de retour* de la fonction se situe à droite de la flèche `->` est également de type `int`.

La signature indique donc ici qu'à partir de deux entiers passés en paramètre, la fonction retourne un entier. 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.

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

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 hypothèses sont des expressions logiques Python, c'est-à-dire de type `bool` et s'évaluant donc soit en `True`, soit en `False`. Nous reviendrons sur les expressions logiques avec l'alternative lors du prochain cours.

Finalement, après une ligne vide, 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 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, par exemple :

```
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 cours 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).
- le *fournisseur* qui propose une solution informatique au problème posé par le client (comme l’étudiant qui répond à l’exercice).

Dans ce cours, pour chaque problème posé par le client (l’enseignant), une définition de fonction sera rédigée par le fournisseur (l’étudiant).

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 (par exemple, un enseignant qui corrige un exercice).

Pour assurer cette validation, l’objectif est de proposer un **jeu de tests** sous la forme d’une suite d’expressions d’appel 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 notre cours :

> les expressions d'appel de fonctions doivent respecter la signature et les hypothèses spécifiées par la fonction. Dans le cas contraire, aucune garantie n'est fournie concernant les effets éventuels de cette expression d'appel erronée.

Donc 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'expression d'appel 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'appel ci-dessous :

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

l'erreur commise, qui rend l'expression invalide, est que le premier argument d'appel est négatif alors qu'une hypothèse 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 :

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

est invalide puisqu'elle contredit l'hypothèse 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 hypothèses de la fonction appelée
- couvrir suffisamment de cas permettant à l'*exécutant* d'avoir confiance dans la validité de sa solution.

Bien sûr, c'est bien le client (par exemple l'enseignant) qui décidera finalement si la solution proposée est bien valide ou non (par exemple : en décidant d'une note pour l'exercice).

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.

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

1.4 Exercices corrigés

Exercice (1.1) : Moyenne de trois nombres (corrigé)

Cet exercice a pour but de trouver les paramètres d'une fonction pour résoudre un problème, et d'écrire la spécification et la définition de fonctions très simples.

Question 1

Donner une définition de la fonction `moyenne_trois_nb` qui effectue la moyenne arithmétique de trois nombres.

Dans le jeu de tests, on vérifiera notamment le calcul de moyenne des nombres 3, 6 et -3 puis de -3, 0 et 3 puis de 1.5, 2.5 et 1.0 (on pourra ajouter d'autres tests en complément).

Réponse

```python
def moyenne_trois_nb(a, b, c):
    """
    Number * Number * Number -> float

    retourne la moyenne arithmétique des trois nombres a, b et c.
    """

    return (a + b + c) / 3.0   # remarque : division flottante
```

```python
# Jeu de tests
assert moyenne_trois_nb(3, 6, -3) == 2.0
assert moyenne_trois_nb(3, 0, -3) == 0.0
assert moyenne_trois_nb(1.5, 2.5, 1.0) == 5.0 / 3.0
assert moyenne_trois_nb(3, 6, 3) == 4.0
assert moyenne_trois_nb(1, 2, 3) == 2.0
assert moyenne_trois_nb(1, 1, 1) == 1.0
assert moyenne_trois_nb(3, 6, 3) == 4.0
assert moyenne_trois_nb(1, 2, 3) == 2.0
```

Question 2 : moyenne pondérée

Écrire une définition de la fonction `moyenne_ponderee` qui effectue la moyenne de trois nombres `a`, `b`, `c` avec des coefficients de pondération, respectivement `pa` (pondération en `a`), `pb` et `pc`.

Proposer un jeu de tests comprenant *au moins* trois tests.

Réponse

```python
def moyenne_ponderee(a, b, c, pa, pb, pc):
    """
    Number^6 -> float
    Hypothèse : pa + pb + pc != 0

    retourne la moyenne des trois nombres a, b, c, pondérés respectivement
    par pa (pondération pour a), pb et pc.
    """

    return ((a * pa) + (b * pb) + (c * pc)) / (pa + pb + pc)
```

```python
# Jeu de tests
assert moyenne_ponderee(1, 1, 1, 3, 6, -3) == 1.0
assert moyenne_ponderee(2, 3, 4, 1, 1, 1) == 3.0
assert moyenne_ponderee(1, 0, 4, 2, 1, 2) == 2.0
```

Exercice (1.3) : Calcul d'un prix TTC (corrigé)

Le but de cet exercice est d'effectuer des conversions simples entre des *prix hors-taxe* (HT) et des *prix toutes taxes comprises* (TTC). Ceci permet de s'intéresser au passage d'un problème de la vie courante à une solution informatique.

Question 1

Ecrire une définition de la fonction `prix_ttc` qui calcule le prix toutes taxes comprises (TTC) à partir d'un prix hors taxe (HT) et d'un taux de TVA exprimé en pourcentage, par exemple `20.0` pour une TVA de 20%.

Remarque : La signature de la division `/` est: `Number * Number -> float`.

Par exemple :

```python
>>> prix_ttc(100.0, 20.0)
120.0
```

```python
>>> prix_ttc(100, 0.0)    # remarque : conversion implicite de l'entier 100
                          #            en le flottant équivalent 100.0
100.0
```

```python
>>> prix_ttc(100, 100.0)
200.0
```

```python
>>> prix_ttc(0, 20)
0.0
```

```python
>>> prix_ttc(200, 5.5)
211.0
```

Réponse

```python
def prix_ttc(prix, taux):
    """
    Number * Number -> float
    H: prix >= 0

    retourne le prix TTC correspondant au prix HT 'prix'
    avec un taux de TVA 'taux'
    """
    return prix * (1 + taux / 100.0)
```

```python
# Jeu de tests
assert prix_ttc(100.0, 20.0) == 120.0
assert prix_ttc(100, 0.0) == 100.0
```

```python
assert prix_ttc(100, 100.0) == 200.0
assert prix_ttc(0, 20) == 0.0
assert prix_ttc(200, 5.5) == 211.0
```

Question 2

Donner une définition de la fonction `prix_ht` qui calcule le prix hors taxe à partir du prix toutes taxes comprises et du taux de TVA.

Remarque : votre jeu de tests doit correspondre à celui proposé pour la fonction `prix_ttc`.

Réponse

```python
def prix_ht(prix, taux):
    """
    Number * Number -> float

    retourne le prix HT correspondant au prix TTC 'prix'
    avec un taux de TVA 'taux'
    """
    return prix / (1 + taux / 100.0)
```

```python
# Jeu de tests
assert prix_ht(120, 20) == 100.0
assert prix_ht(100, 0) == 100.0
assert prix_ht(200, 100) == 100.0
assert prix_ht(0, 20) == 0.0
assert prix_ht(211, 5.5) == 200.0
```

Exercice (1.5) : Calcul de fonctions polynômiales (corrigé)

Cet exercice a pour but de définir des fonctions de calcul de polynômes. On souhaite mettre l'accent sur l'efficacité des algorithmes utilisés (minimisation du nombre de multiplications).

Question 1

Après avoir spécifié le problème, écrire un jeu de tests et donner une définition de la fonction `polynomiale` telle que `polynomiale(a, b, c, d, x)` rend la valeur de $ax^3 + bx^2 + cx + d$.

Par exemple :

```python
>>> polynomiale(1, 1, 1, 1, 2)
15
```

```python
>>> polynomiale(1, 1, 1, 1, 3)
40
```

Quel est le nombre de multiplications effectuées par votre définition ? Peut-on faire mieux ? Si oui, proposer une version plus efficace, sinon justifiez.

Réponse

```python
def polynomiale(a, b, c, d, x):
    """Number^5 -> Number

    retourne la valeur de a*x^3 + b*x^2 + c*x + d"""

    return (a*x*x*x + b*x*x + c*x + d)

# remarque : on peut aussi utiliser la fonction puissance :
# return (a*x**3 + b*x**2 + c*x + d)
# mais en considèrant que l'opérateur x**3 effectue lui-même x*x*x
# cela revient à faire le même nombre de multiplications.
```

```python
# Jeu de tests
assert polynomiale(1,1,1,1,2) == 15
assert polynomiale(1,1,1,1,3) == 40
assert polynomiale(2,0,0,0,1) == 2
assert polynomiale(0,3,0,0,1) == 3
assert polynomiale(0,0,4,0,1) == 4
assert polynomiale(1,2,3,4,0) == 4
assert polynomiale(2,3,4,5,1) == 14
```

Il faut 6 multiplications ici.

Une autre définition plus efficace, ne nécessitant que 3 multiplications, est la suivante :

```python
def polynomiale(a, b, c, d, x):
    """
    Number^5 -> Number

    retourne la valeur de a*x^3 + b*x^2 + c*x + d"""

    return ((((a*x + b) * x) + c) * x) + d)
```

```python
# Jeu de tests
assert polynomiale(1,1,1,1,2) == 15
assert polynomiale(1,1,1,1,3) == 40
assert polynomiale(2,0,0,0,1) == 2
assert polynomiale(0,3,0,0,1) == 3
assert polynomiale(0,0,4,0,1) == 4
assert polynomiale(1,2,3,4,0) == 4
assert polynomiale(2,3,4,5,1) == 14
```

Question 2

Après avoir spécifié le problème, écrire un jeu de tests et donner une définition de la fonction `polynomiale_carre` qui rend la valeur de $ax^4 + bx^2 + c$.

Par exemple :

```python
>>> polynomiale_carre(1, 1, 1, 2)
21
```

```
>>> polynomiale_carre(1, 1, 1, 3)
91
```

Quel est le nombre de multiplications effectuées par votre définition ? Pouvez-vous proposer une version plus efficace ?

Réponse

```
def polynomiale_carre(a, b, c, x):
    """Number^4 -> Number

    retourne la valeur de a*x^4 + b*x^2 + c """

    return (a*x*x*x*x + b*x*x + c)

# ou
# return (a*x**4  + b*x**2 + c)
```

```
# Jeu de tests
assert polynomiale_carre(1,1,1,2) == 21
assert polynomiale_carre(1,1,1,3) == 91
assert polynomiale_carre(2,0,0,1) == 2
assert polynomiale_carre(0,3,0,1) == 3
assert polynomiale_carre(2,3,4,0) == 4
assert polynomiale_carre(2,3,4,1) == 9
```

Il faut 6 multiplications ici aussi.

Une autre définition plus efficace (utilisant le schéma de *Hörner*), ne nécessitant plus que 4 multiplications, est la suivante :

```
def polynomiale_carre(a,b,c,x):
    """
    Number^4 -> Number

    retourne la valeur de a*x^4 + b*x^2 + c """

    return (((a*x*x + b) * x*x) + c)
```

```
# Jeu de tests
assert polynomiale_carre(1,1,1,2) == 21
assert polynomiale_carre(1,1,1,3) == 91
assert polynomiale_carre(2,0,0,1) == 2
assert polynomiale_carre(0,3,0,1) == 3
assert polynomiale_carre(2,3,4,0) == 4
assert polynomiale_carre(2,3,4,1) == 9
```

2 Instructions, variables et alternatives

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

Les *principes d'évaluation des expressions* (arithmétiques) vues lors du cours 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 lors du cours précédent est le *retour de fonction* :

```
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 (où se trouve le `return`) avec la valeur de retour calculée précédemment.

Dans ce cours, 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.

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 :

— d'abord on commence par X
— puis il faut faire 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>`.

Intuitivement, il s'agit donc du même principe d'interprétation que celui de la recette de cuisine !

Dans une telle suite d'instructions, l'instruction `return <expression>` n'est alors pas très intéressante, sauf en dernière position, car elle a pour effet d'interrompre l'interprétation pour sortir directement de la fonction (nous exploiterons cette caractéristique par la suite).

Pour compléter notre répertoire d'instructions, nous allons 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
```

Contrairement à ce que l'on pourrait penser, `print` ne produit aucune valeur intéressante. Elle retourne simplement la valeur `None` (de type `NoneType`) qui signifie «absence de valeur» (intéressante).

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**.

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. Dans le cadre de ce cours, nous utilisons également `print` pour illustrer les principes d'interprétation comme celui des suites d'instructions.

Considérons ainsi la fonction suivante :

```
def recette_omelette():
    """ -> NoneType

       affiche la recette de l'omelette et retourne None.
       """

    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.")
```

Le type de retour `NoneType` dans la signature ci-dessus explique également que cette fonction, en terme de calcul, est peu utile puisqu'elle ne renvoie rien. Mais elle reste (relativement) intéressante pour l'utilisateur grâce aux affichages qu'elle réalise.

```
>>> 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.
```

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 donner des indices 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 de 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` :

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

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

    return 2 * (largeur + longueur)
```

```python
# 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 :

```python
>>> 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 cours, nous allons principalement nous intéresser au premier cas, et nous étudierons en détail le second cas lors du prochain cours 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 hypothèses 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 `Number`) qui représentent des longueurs entières ou flottantes
— une hypothèse importante est que les trois paramètres sont des longueurs donc des nombres strictement positifs.
— une hypothèse plus difficile à satisfaire 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.

Remarque : nous verrons que le calcul de l'aire nécessite d'utiliser la division classique (en python, `x` divisé par `y` s'écrit : `x / y`) ainsi que la fonction de calcul d'une racine carrée (en python, racine carrée de `x` s'écrit : `math.sqrt(x)`). Ces deux opérations retournent des valeurs de type `float`, le résultat du calcul de l'aire est donc forcément flottant.

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

```python
def aire_triangle(a,b,c):
    """ Number * Number * Number -> float
        hypothèse : (a>0) and (b>0) and (c>0)
        hypothèse : 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, on doit maintenant *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 cours 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.

Cette formule a été inventée, selon la légende, par *Héron d'Alexandrie* au premier siècle de notre ère. D'après la page Wikipedia (en date du 28 août 2014) dédiée à cet illustre ingénieur :

Formule de Héron

Cette formule permet de calculer l'aire d'un triangle en connaissant la longueur de ses côtés, sans utiliser la hauteur.

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

A partir de la valeur du demi-périmètre $p = \dfrac{a+b+c}{2}$, 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,b,c):
    """ Number * Number * Number -> float
        hypothèse : (a>0) and (b>0) and (c>0)
        hypothèse : 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.
        """

    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 énoncée par Héron,
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 donc besoin d'une case mémoire supplémentaire pour pouvoir stocker ce résultat intermédiaire.

Pour introduire cette case mémoire supplémentaire, nous allons utiliser une **variable** (que l'on dénommera `p` pour garder le même nom que dans l'énoncé de la formule de Héron) pour stocker le résultat du calcul du demi-périmètre.

La définition de la fonction devient alors :

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

```
def aire_triangle(a,b,c):
    """ Number * Number * Number -> float
        hypothèse : (a>0) and (b>0) and (c>0)
        hypothèse : 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.
        """

    # p : float
    p = (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).

Ici, on utilise une variable dans le corps de la fonction. Cette variable, *p*, est **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. Elle ne joue un rôle que local car elle ne nous est utile que pour mémoriser un résultat intermédiaire (la valeur du demi-périmètre) et l'utiliser ensuite plusieurs fois dans l'expression du calcul de l'aire.

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 hypothèse - que 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 :

```python
# 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.

```python
def aire_triangle(a,b,c):
    """ Number * Number * Number -> float
        hypothèse : (a>0) and (b>0) and (c>0)
        hypothèse : (a <= b + c) and (b <= a + c) and (c <= a + b)
                    -- 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.
        """
```

2.2.2 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 quatre principales manipulations liées aux variables sont les suivantes :

1. la **déclaration de variable**
2. l'**initialisation de variable** ou **première affectation**
3. l'**occurrence de variable** au sein d'une expression
4. sa mise à jour ou **réaffectation**.

2.2.2.1 Déclaration de variable Pour pouvoir être utilisée, une variable doit préalablement être déclarée.

La syntaxe utilisée pour une telle déclaration est la suivante :

```
# <var> : <type>
```

où `<var>` est un nom de variable et `<type>` le type du contenu de la variable (`int`, `float`, etc.).

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

```
# p : float
```

On a ici déclaré une variable de nom `p` et de type `float`.

Il s'agit donc, en quelque sorte, de spécifier la signature de la variable.

2.2.2.2 Initialisation de variable Une variable n'existe pas «encore» après sa déclaration. Pour la créer, il faut impérativement préciser son contenu initial.

L'initialisation de la variable se fait juste en dessous de sa déclaration.

La syntaxe pour initaliser une variable est la suivante :

```
# <var> : <type>
<var> = <expression>
```

où `<var>` est un nom de variable dont le contenu doit être de type `<type>` et `<expression>` est une expression dont le type doit également être `<type>`.

La **définition de la variable** est donc la succession de la déclaration et de l'initialisation de cette variable.

Par exemple, dans la fonction `aire_triangle`, nous avons défini la variable `p` de la façon suivante :

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

On a ainsi précisé que l'on déclarait la variable de nom `p`, dont le contenu serait de type `float`, et on a initialisé son contenu avec le **résultat** du calcul de `(a + b + c) / 2`.

On remarque ici que l'expression utilisée pour cette initialisation est une division flottante. Elle est donc bien du même type `float` que celui déclaré pour la variable `p`.

Important : à une exception près (les variables d'itération que nous aborderons au cours 4), *toutes* les variables nécessaires doivent être déclarées *au début* du corps de la fonction : juste en dessous de la spécification et *avant* l'implémentation proprement dite de l'algorithme de calcul.

Ainsi la structure usuelle d'une fonction est la suivante :

```python
def ma_fonction(param1, param2, ...):
    """partie 1 : specification
    ..."""

    # partie 2 : déclaration et initialisation des variables

    # v1 : type1
```

```
v1 = expr1

# v2 : type2
v2 = expr2

...

# vn : typeN
vn = exprN

# partie 3 : algorithme de calcul
instruction1
instruction2
... etc ...
```

2.2.2.3 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 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 n'y a aucune surprise ici.

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` corrigée précédemment, 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.

2.2.2.4 Affectation et réaffectation de variable

La syntaxe :

```
<var> = <expression>
```

(que nous avons par exemple rencontrée dans l'initialisation d'une variable) est appelée une **affectation de variable**.

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 :

```python
p = (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

Il y a deux types différents d'affectation :

— la **première affectation** qui correspond à l'initialisation de la variable.
— une **réaffectation** qui consiste à mettre à jour le contenu de la variable.

Une variable ne possède pas de contenu avant sa première affectation (lors de son initialisation). En revanche, lors d'une réaffectation, on *efface* tout simplement le précédent contenu pour en stocker un nouveau.

Considérons la suite d'instructions suivante :

```python
# n : int
n = 0

# m : int
m = 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.

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

```python
# m : int
m = 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 :

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

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

```
<ipython-input-25-a278aada65a5> in <module>()
      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

Désormais, les deux variables sont initialisées et possèdent donc une valeur, il devient possible de les référencer.

L'étape suivante est une *nouvelle* affectation de la variable `n`. Puisque cette variable possède déjà une valeur, il ne s'agit pas d'une initialisation mais d'une **réaffectation** dans le but de mettre à jour le contenu de la variable.

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 réaffectation `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 : - nous avons d'abord récupéré la valeur de `m`, - puis nous avons calculé `58 + 1`, donc 59, - 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 expression de la forme `v = v + 1` où `v` est une variable (de type `int`) se nomme une **incrémentation** de la variable `v`
— une expression de la forme `v = v - 1` où `v` est une variable (de type `int`) se nomme une **décrémentation** de la variable `v`

2.2.2.5 Complément : règles de base concernant le nommage des variables

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. . .).

Pour déterminer un nom de variable, la convention que nous suivrons 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

Remarques :

— les noms de fonctions suivent les mêmes règles
— les noms de variables contenant des données complexes comme des listes, des ensembles et des dictionnaires commencent par une majuscule afin de les distinguer des variables contenant des données plus simples. Il s'agit d'une convention spécifique à notre cours.

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 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 :

```python
def valeur_absolue(x):
    """ Number -> Number
        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 les **4 espaces** (ou 1 tabulation) 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 :

```python
def valeur_absolue(x):
    """ Number -> Number

        retourne la valeur absolue de x.
        """

    # abs_x : Number
    abs_x = 0    #  stockage de la valeur absolue, le choix de 0 pour
                 # l'initialisation est ici arbitraire

    if x >= 0:
        abs_x = x  # conséquent
    else:
        abs_x = -x # alternant

    return abs_x
```

```python
# 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. 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_absolue2(x):
    """ Number -> Number

        retourne la valeur absolue de x.
        """

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

```
# Jeu de tests
assert valeur_absolue2(3) == 3
assert valeur_absolue2(-3) == 3
assert valeur_absolue2(1.5 - 2.5) == valeur_absolue2(2.5 - 1.5)
assert valeur_absolue2(0) == 0
assert valeur_absolue2(-0) == 0
```

Une question se pose finalement ici :

> Quelle définition est la «meilleure» ?

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.

En pratique, un programmeur aguerri choisira la seconde solution du fait de sa concision, mais nous avons vu avec `aire_triangle` que se passer d'une variable pouvait également représenter une très mauvaise idée.

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 :

opérateur	notation Python	signature
égalité	`==`	`Number * Number -> bool`
inégalité	`!=`	`Number * Number -> bool`
inférieur strict	`<`	`Number * Number -> bool`
inférieur ou égal	`<=`	`Number * Number -> bool`
supérieur strict	`>`	`Number * Number -> bool`
supérieur ou égal	`>=`	`Number * Number -> bool`

— d'expressions composées d'**opérateurs logiques** :

opérateur	notation Python	signature
conjonction (*et logique*)	`and`	`bool * bool -> bool`
disjonction (*ou logique*)	`or`	`bool * bool -> bool`
négation (*non logique*)	`not`	`bool -> bool`

Attention : il est important de distinguer le symbole `=` utilisé dans le cadre d'une affectation de variable et le symbole `==` utilisé pour représenter l'opérateur d'égalité entre deux nombres.

Par exemple :

```python
# n : int
n = 42
```

Ici il s'agit d'une affectation, la variable `n` contient désormais la valeur `42`. Ici, Python n'affiche rien puisqu'une affectation est une instruction et non une expression : elle ne retourne pas de valeur.

```python
>>> n == 42
True
```

Ici, il s'agit d'une expression booléenne pour comparer la valeur de `n` (donc 42) avec l'entier `42`. Les deux valeurs sont égales donc cette expression booléenne s'évalue bien en `True`.

A retenir : ne pas confondre `=` (affectation) et `==` (égalité).

Remarque : des *bugs* célèbres proviennent de la confusion entre ces deux symboles dans le cadre du langage C. Heureusement, cette ambigüité est souvent signalée par l'interprète Python.

```python
# Exemple de mauvaises utilisations (A NE PAS FAIRE !!)

if n = 12:  # affectation !!!
    n == 42 # égalité !
else:
    n == 29  # égalité !


  File "<ipython-input-33-647ab5ba6e35>", line 3
    if n = 12:  # affectation !!!
         ^
SyntaxError: invalid syntax
```

2.3.2.1 Principe d'évaluation de la négation

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

`not <expression>`

où `<expression>` est une expression booléenne.

- on évalue tout d'abord `<expression>`
  - si la valeur obtenue est `True` alors `not <expression>` renvoie `False`
  - si la valeur obtenue est `False` alors `not <expression>` renvoie `True`

Autrement dit, l'opérateur de négation inverse la valeur de vérité de son expression.

Par exemple :

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

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

```python
>>> 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 !

On peut juste remarquer que Python répondra quelque chose… mais qui n'a pas forcément sens.

2.3.2.2 Principe d'évaluation de la conjonction

La plupart des opérateurs binaires fonctionnent de manière «traditionnelle».

Si on note `op` un opérateur, pour évaluer : `<expression1> op <expression2>`

— on évalue tout d'abord `<expression1>` ce qui produit une certaine valeur v_1
— puis on évalue `<expression2>` ce qui produit une certaine valeur v_2
— la valeur finale est le calcul : v_1 `op` v_2.

Par exemple, pour évaluer : `(5 * 9)  +  (7 // 2)`

— on évalue tout d'abord `5 * 9` ce qui produit la valeur 45
— puis on évalue `7 // 2` ce qui produit la valeur 3
— la valeur finale est le calcul : 45 `+` 3 soit 48.

```
>>> (5 * 9) + (7 // 2)
48
```

Les opérateurs binaires arithmétiques fonctionnent tous selon ce même procédé (leur seule différence portant sur le calcul final réalisé qui, lui, est propre à l'opérateur).

Un autre exemple booléen cette fois-ci, pour évaluer : `(5 * 9) >= (7 // 2)`

— on évalue tout d'abord `5 * 9` ce qui produit la valeur 45
— puis on évalue `7 // 2` ce qui produit la valeur 3
— la valeur finale est le calcul : $45 \geq 3$ qui a pour valeur `True`.

Les opérateurs binaires de comparaisons de nombres fonctionnent également selon ce même procédé.

En revanche, le principe de calcul des opérateurs binaires logiques `and` et `or` est, lui, très différent.

Commençons par la conjonction (appelée aussi "*et logique*") : l'opérateur binaire `and`.

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 et 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 ne soit 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 :

```python
def est_divisible(n, m):
    """ int * int -> bool
        Hypothèse : (n >= 0) and (m >= 0)

        retourne True si n est divisible par m, False sinon.
        """

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

```python
# 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`.

Ceci revient à vérifier si le reste de la division entière de `n` par `m` vaut `0`, soit la condition suivante en Python :

```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
```

Ici,

— 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
```

Ici,

— 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'une fonction indiquant si au moins deux nombres parmi trois de ses paramètres sont égaux.

```
def deux_egaux(x,y,z):
    """ Number * Number * Number -> 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
```

2.3.2.4 Notion de prédicat

Les fonctions comme `est_divisible` ou `deux_egaux` définies ci-dessus ont une propriété commune :

- elles **retournent toutes deux un booléen** en fonction de la valeur de leurs paramètres.

De telles fonctions retournant une valeur booléenne sont des **prédicats**.

Une propriété importante est qu'une application de prédicat (par exemple `est_divisible(5, 2)` ou `deux_egaux(1, 2, 1)`) constitue une *expression booléenne*.

Les prédicats combinés à l'aide de conjonctions, disjonctions ou négations permettent de composer des expressions booléennes très complexes.

Pour illustrer ce point, considérons la fonction suivante :

```python
def est_triangle_quelconque(a, b, c):
    """ Number * Number * Number -> bool

        retourne la valeur True si le triangle dont les côtés ont pour
        longueurs a, b et c est un triangle quelconque, c'est-à-dire s'il
        n'est ni isocèle (2 côtés égaux), ni équilatéral (3 côtés égaux).
        """
    return not deux_egaux(a,b,c)
```

```python
# Jeu de tests
assert est_triangle_quelconque(42, 42, 42) == False
assert est_triangle_quelconque(20, 20, 2) == False
assert est_triangle_quelconque(1, 2, 1) == False
assert est_triangle_quelconque(2, 5, 5) == False
assert est_triangle_quelconque(11, 38, 42) == True
```

La fonction `est_triangle_quelconque` est un exemple de prédicat complexe composé à partir d'un autre prédicat (ici `deux_egaux`).

En pratique, on utilise très souvent des applications de prédicats complexes dans les conditions des alternatives.

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 :

```python
def nb_solutions(a,b,c):
    """ Number * Number * Number -> int

        retourne le nombre de solutions de l'équation aX^2 + bX + c = 0
        """

    # discriminant : Number
    discriminant = b*b - 4*a*c

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

```python
# 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.

2.4 Exercices corrigés

Exercice (2.1) : Variables et affectations (corrigé)

Question 1

On considère la suite d'instructions (affectations de variables et expressions simples) ci-dessous.

Après chaque ligne :

— s'il s'agit d'une affectation donner la table des variables (nom / valeur) après interprétation de cette instruction, et
— s'il s'agit d'une initialisation, préciser la déclaration de variable correspondante.
— s'il s'agit d'une expression donner le type de l'expression ainsi que sa valeur.

```
a = 3
```

```
a
```

```
a = 0
```

```
a
```

```
a == 0
```

```
a == 3
```

```
b = 2
```

```
a == b
```

```
b = a
```

```
b
```

```
a == b
```

```
b = 4
```

```
a == b
```

```
(a == 2) == (b == 0)
```

```
c = (a == 0)
```

```
c
```

```
c == (b == 3)
```

```
x
```

Réponse

```
a = 3
```

Initialisation (première affectation) : la déclaration est :

```
# a : int
```

Table des variables :

Variable	a
Valeur	3

```
a
```

Expression type `int` : valeur = `3`

```
a = 0
```

Affectation (mise-à-jour).

Table des variables :

Variable	a
Valeur	0

```
a
```

Expression type `int` : valeur = `0`

```
a == 0
```

Expression type `bool` : valeur = `True`

```
a == 3
```

Expression type `bool` : valeur = `False`

```
b = 2
```

Initialisation (première affectation) : la déclaration est :

```
# b : int
```

Table des variables :

Variable	a	b
Valeur	0	2

```
a == b
```

Expression type `bool` : valeur = `False`

```
b = a
```

Affectation.

Table des variables :

Variable	a	b
Valeur	0	0

```
b
```

Expression type `int` : valeur = `0`

```
a == b
```

Expression type `bool` : valeur = `True`

```
b = 4
```

Affectation.

Table des variables :

Variable	a	b
Valeur	0	4

```
a == b
```

Expression type `bool` : valeur = `False`

```
(a == 2) == (b == 0)
```

Expression type `bool` : valeur = `True`

```
c = (a == 0)
```

Initialisation (première affectation) : la déclaration est :

```
# c : bool
```

Table des variables :

Variable	a	b	c
Valeur	0	4	True

```
c
```

Expression type `bool` : valeur = `True`

```
c == (b == 3)
```

Expression type `bool` : valeur = `False`

```
x
```

Expression : mais **erreur** car variable `x` non-initialisée.

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
   ...
----> 1 x

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

Question 2

D'après-vous, que se passe-t-il si l'on soumet la définition suivante à l'interprète Python (à essayer ensuite en TME) ?

```python
def sante(x):
    """
    Number -> str

    retourne "Bonne santé" si x vaut 37.5, retourne "Malade" sinon.
    """

    if x = 37.5:
        return "Bonne santé"
    else:
        return "Malade"
```

Réponse

L'alternative n'est tout simplement pas correcte car le `if` attend une instruction booléenne et reçoit une instruction d'affectation. C'est la confusion classique entre l'instruction d'affection `=` et l'opérateur de test d'égalité `==`.

Il faut donc écrire en fait :

```python
def sante(x):
    """
    Number -> str

    retourne "Bonne santé" si x vaut 37.5, retourne "Malade" sinon.
    """

    if x == 37.5:
        return "Bonne santé"
    else:
        return "Malade"
```

```python
# Jeu de tests
assert sante(37.5) == "Bonne santé"
assert sante(40) == "Malade"
assert sante(36) == "Malade"
```

Exercice (2.3) : Calcul des mentions (corrigé)

Cet exercice a pour but de faire manipuler des *alternatives* multiples et/ou imbriquées. Il s'agit de calculer la mention correspondant à une note sur 20.

Question 1

Donner une définition en Python de la fonction `mention` qui calcule la mention correspondant à une note (sur 20) donnée en utilisant les intervalles de notes suivants :

— $[0, 10[\rightarrow$ Eliminé,

— $[10, 12[\rightarrow$ Passable,
— $[12, 14[\rightarrow$ AB,
— $[14, 16[\rightarrow$ B,
— $[16, 20] \rightarrow$ TB.

Les mentions seront représentées par des chaînes de caractères encadrées par des guillemets simples. La mention Eliminé sera par exemple représentée par la valeur `'Eliminé'` de type `str`.

Voici quelques exemples d'applications de la fonction `mention` :

```
>>> mention(0)
'Eliminé'
```

```
>>> mention(8)
'Eliminé'
```

```
>>> mention(10)
'Passable'
```

```
>>> mention(12.5)
'AB'
```

```
>>> mention(15)
'B'
```

```
>>> mention(20)
'TB'
```

Remarque : penser à utiliser le mot-clef `elif` pour les alternatives multiples.

Réponse

Voici une première définition dans l'ordre croissant.

```python
def mention(note):
    """
    Number -> str
    Hypothèse : (note>=0) and (note<=20)

    retourne la mention correspondant à la note spécifiée."""

    if note < 10:
        return 'Eliminé'
    elif note < 12:
        return 'Passable'
    elif note < 14:
        return 'AB'
    elif note < 16:
        return 'B'
    else:
        return 'TB'
```

```python
# Jeu de tests
assert mention(0) == 'Eliminé'
assert mention(8) == 'Eliminé'
```

```python
assert mention(10) == 'Passable'
assert mention(12.5) == 'AB'
assert mention(15) == 'B'
assert mention(20) == 'TB'
```

Une autre version dans l'ordre décroissant :

```python
def mention(note):
    """
    Number -> str
    Hypothèse : (note>=0) and (note<=20)

    retourne la mention correspondant à la note spécifiée."""

    if note >= 16:
        return 'TB'
    elif note >= 14:
        return 'B'
    elif note >= 12:
        return 'AB'
    elif note >= 10:
        return 'Passable'
    else:
        return 'Eliminé'
```

```python
# Jeu de tests
assert mention(0) == 'Eliminé'
assert mention(8) == 'Eliminé'
assert mention(10) == 'Passable'
assert mention(12.5) == 'AB'
assert mention(15) == 'B'
assert mention(20) == 'TB'
```

Question 2

Définir à nouveau la fonction `mention`, cette fois-ci en imbriquant les structures conditionnelles.

L'objectif est de minimiser le nombre maximum de tests conditionnels effectués, d'une part, et de minimiser le nombre moyen de tests, d'autre part, en fonction de la répartition des notes, en supposant que la majorité des notes se situent entre 10 et 12.

Réponse

Dans la première réponse à la question 1, on fait 1 test lorsque $note < 10$, 2 tests lorsque $10 \leq note < 12$, 3 tests lorsque $12 \leq note < 14$ et 4 tests lorsque $14 \leq note \leq 20$. C'est statistiquement une version efficace si la majorité des notes sont mauvaises.

Dans la deuxième réponse à la question 1, on fait 4 tests lorsque $note < 12$, 3 tests lorsque $12 \leq note < 14$, 2 tests lorsque $14 \leq note < 16$ et 1 test lorsque $16 \leq note \leq 20$. C'est statistiquement une version efficace si la majorité des notes est bonne.

On va donner une version qui permet d'effectuer moins de tests si on suppose que la majorité des notes est entre 10 et 12 et qui fait moins de tests dans le pire des cas.

```python
def mention(note):
    """
    Number -> str
    Hypothèse : (note>=0) and (note<=20)

    retourne la mention correspondant à la note spécifiée."""

    if note < 12:
        if note < 10:
            return 'Eliminé'
        else:
            return 'Passable'
    elif note < 14:
        return 'AB'
    elif note < 16:
        return 'B'
    else:
        return 'TB'
```

```python
# Jeu de tests
assert mention(0) == 'Eliminé'
assert mention(8) == 'Eliminé'
assert mention(10) == 'Passable'
assert mention(12.5) == 'AB'
assert mention(15) == 'B'
assert mention(20) == 'TB'
```

Avec cette version, l'évaluation ne nécessite que 2 tests lorsque $0 \leq \mathit{note} < 14$ et 3 tests lorsque $14 \leq \mathit{note} \leq 20$.

Exercice (2.4) : Couverture (corrigé)

On cherche dans cet exercice à définir un jeu de tests permettant d'explorer tous les cas de tests d'une fonction.

Question 1

La fonction suivante rend 6 valeurs différentes (une chaîne de caractères) selon l'ordre dans lequel sont donnés ses trois arguments, **différents deux à deux**.

```python
def f(n1, n2, n3):
    """Number * Number * Number -> str
    H: n1 != n2 and n2 != n3 and n3 != n1
    retourne un cas parmi 6 selon les valeurs de n1, n2 et n3."""

    if n1 < n2 and n2 < n3:
        return 'cas 1'
    elif n1 < n3 and n3 < n2:
        return 'cas 2'
```

```python
    elif n2 < n1 and n1 < n3:
        return 'cas 3'
    elif n2 < n3 and n3 < n1:
        return 'cas 4'
    elif n3 < n1 and n1 < n2:
        return 'cas 5'
    else:
        return 'cas 6'
```

Définir un jeu de six tests vérifiant les six cas possibles.

Réponse

```python
# Jeu de tests
assert f(1, 2, 3) == 'cas 1'
assert f(1, 3, 2) == 'cas 2'
assert f(2, 1, 3) == 'cas 3'
assert f(3, 1, 2) == 'cas 4'
assert f(2, 3, 1) == 'cas 5'
assert f(3, 2, 1) == 'cas 6'
```

Question 2

Donner une autre définition de `f` sans utiliser les opérateurs logiques `and` et `or` mais uniquement des alternatives. Vérifier que les résultats obtenus sont identiques à ceux fournis dans le jeu de tests de la question précédente.

Réponse

```python
def f(n1, n2, n3):
    """Number * Number * Number -> str
    H: n1 != n2 and n2 != n3 and n3 != n1

    retourne un cas parmi 6 selon les valeurs de n1, n2 et n3."""

    if n1 < n2:
        if n2 < n3:
            return 'cas 1'
        elif n1 < n3:
            return 'cas 2'
        else:
            return 'cas 5'
    elif n1 < n3:
        return 'cas 3'
    elif n2 < n3:
        return 'cas 4'
    else:
        return 'cas 6'
```

```python
# Jeu de tests
assert f(1, 2, 3) == 'cas 1'
assert f(1, 3, 2) == 'cas 2'
```

```python
assert f(2, 1, 3) == 'cas 3'
assert f(3, 1, 2) == 'cas 4'
assert f(2, 3, 1) == 'cas 5'
assert f(3, 2, 1) == 'cas 6'
```

Exercice (2.6) : Mesures fiables (corrigé)

Dans cet exercice, nous explorons le problème de l'égalité entre flottants qui est le plus souvent considérée *à epsilon près*. Ceci donne l'occasion de réfléchir à la complétude des jeux de tests. De plus, les alternatives multiples sont ici d'une grande utilité.

Question 1

Donner la définition du prédicat `egal_eps` qui teste l'égalité de deux nombres `x1` et `x2` *à epsilon près*, c'est-à-dire si la valeur absolue de leur différence est inférieure à un nombre strictement positif (et supposé petit) `epsilon` également passé en paramètre.

Le jeu de tests proposé doit être le plus complet possible. Il faut notamment tenir compte du maximum de cas de figures possibles : arguments positifs ou négatifs, rapport entre les deux arguments (plus petit ou plus grand), valeur attendue vraie ou fausse.

Remarque : il est possible soit d'utiliser la fonction prédéfinie `abs` pour le calcul de la valeur absolue, soit de redéfinir la fonction `valeur_absolue` vue en cours.

Réponse

```python
def egal_eps(x1, x2, epsilon):
    """ float * float * float -> bool
    Hypothèse: epsilon > 0

    renvoie True quand x1 et x2 sont égaux à epsilon près."""

    return epsilon > abs (x1 - x2)
```

A DISCUTER EN TD Voici comment on peut *dénombrer* un ensemble de cas à tester en tenant compte des trois critères (signes des arguments, rapport des arguments, valeur attendue) :

- — il y a 2 valeurs attendues (`True` et `False`) pour chacune d'elles,
- — il y a 4 possibilités de signes:
  - — 2 possibilités du même signe et alors, il y a pour chacune d'elle 2 possibilités de rapport entre les arguments: le premier est plus petit, le deuxième est plus petit
  - — 2 possibilités de signes différents et alors le rapport est fixé.

Ce qui donne $2 * (2 * 2 + 2) = 12$ cas à envisager.

Par exemple:

```python
# Jeu de tests
assert egal_eps(4,5,1.1) == True
assert egal_eps(5,4,1.1) == True
```

```python
assert egal_eps(-4,-5,1.1) == True
assert egal_eps(-5,-4,1.1) == True
assert egal_eps(3,5,1.1) == False
assert egal_eps(5,3,1.1) == False
assert egal_eps(-3,-5,1.1) == False
assert egal_eps(-5,-3,1.1)== False
assert egal_eps(-1,0,1.1) == True
assert egal_eps(0,-1,1.1) == True
assert egal_eps(-1,1,1.1) == False
assert egal_eps(1,-1,1.1) == False
```

Question 2

Lors de l'utilisation d'un instrument de mesure (par exemple un thermomètre), il est bon de ne pas se fier à une seule mesure. Soit un instrument de mesure qui dispose de trois capteurs fournissant trois valeurs numériques `v1`, `v2` et `v3` censées donner chacune la mesure d'un même phénomène (par exemple une température). On tolère une différence jugée négligeable entre ces valeurs : en d'autres termes, on considère leur égalité à *epsilon près*.

Attention : l'égalité à *epsilon* près n'est pas transitive en général, elle reste en revanche symétrique.

On cherche à déterminer un taux de fiabilité de la mesure en appliquant le principe suivant: - si les trois valeurs sont deux à deux égales à *epsilon* près, le taux est de $\frac{3}{3}$, c'est-à-dire 1; - si deux couples de valeurs seulement sont égales à *epsilon* près, le taux de fiabilité est de $\frac{2}{3}$ (par exemple `v1` $\approx$ `v3` et `v2` $\approx$ `v3`). - sinon le taux de fiabilité est de $\frac{0}{3}$, c'est-à-dire nul (0).

- Combien y a-t-il de façons d'obtenir le taux de fiabilité de 1, de $\frac{2}{3}$, et de 0 ? Donner des exemples.
- Donner une définition de la fonction `fiabilite` qui donne le taux de fiabilité de trois valeurs `v1`, `v2` et `v3` à `epsilon` près. On prendra soin de bien vérifier tous les cas possibles dans le jeu de tests.

Réponse

Il n'y a qu'une seule façon d'obtenir $\frac{3}{3}$ (c'est-à-dire 1) : les trois valeurs sont égales à *epsilon* près deux à deux.

Il y a trois façons d'obtenir $\frac{2}{3}$: soit `v1` est égale à *epsilon* près à `v2` et à `v3` ; soit `v2` est égale à *epsilon* près à `v1` et à `v3` ; soit `v3` est égale à *epsilon* près à `v1` et à `v2`.

Il y a quatre façons d'obtenir 0 : soit les valeurs sont toutes différentes, soit l'une des égalités à *epsilon* près est vérifiée mais aucune des deux autres n'est possible et il y a trois possibilités pour cela : soit `v1` est égale à *epsilon* près à `v2` mais ni `v1` à `v3`, ni `v2` à `v3` ; soit `v2` est égale à *epsilon* près à `v3` mais ni `v1` à `v3`, ni `v1` à `v2` ; soit `v2` est égale à *epsilon* près à `v3` mais ni `v1` à `v3`, ni `v1` à `v2`.

```python
def fiabilite(v1, v2, v3, epsilon):
    """ float * float * float * float -> float
    Hypothèse: epsilon > 0

    Retourne le taux de fiabilité des trois mesures v1, v2 et v3
    à epsilon près."""
```

```python
    if egal_eps(v1, v2, epsilon) and egal_eps(v2 ,v3 , epsilon) \
        and egal_eps(v1, v3, epsilon):
        return 1
    elif ((egal_eps(v1, v2, epsilon) and egal_eps(v2,v3,epsilon))
        or (egal_eps(v1, v2, epsilon) and egal_eps(v1, v3, epsilon))
        or (egal_eps(v1, v3, epsilon) and egal_eps(v2, v3, epsilon))):
        return 2/3
    else:
        return 0
```

```python
# Jeu de tests
assert fiabilite(3,3,3,1.1) == 1
assert fiabilite(3,2,1,1.1) == 2/3
assert fiabilite(1,2,3,1.1) == 2/3
assert fiabilite(1,3,2,1.1) == 2/3
assert fiabilite(1,3,5,1.1) == 0
assert fiabilite(1,2,5,1.1) == 0
assert fiabilite(1,5,2,1.1) == 0
assert fiabilite(5,1,2,1.1) == 0
```

Deuxième Solution : La solution précédente, si elle calcule bien ce que l'on veut, effectue possiblement plusieurs fois un même test, ce que l'on peut éviter en imbriquant les conditionnelles.

```python
def fiabilite(v1, v2, v3, epsilon):
    """ float * float * float * float -> float
    Hypothèse: epsilon > 0

    Retourne le taux de fiabilité des trois mesures v1, v2 et v3 à epsilon près."""

    if egal_eps(v1, v2, epsilon):
        if egal_eps(v1, v3, epsilon):
            if egal_eps(v2, v3, epsilon):
                return 1
            else:
                return 2/3
        else:
            if egal_eps(v2, v3, epsilon):
                return 2/3
            else:
                return 0
    elif egal_eps(v2, v3, epsilon):
        if egal_eps(v1, v3, epsilon):
            return 2/3
        else:
            return 0
    else:
        return 0
```

```python
# Jeu de tests
assert fiabilite(3,3,3,1.1) == 1
```

```python
assert fiabilite(3,2,1,1.1) == 2/3
assert fiabilite(1,2,3,1.1) == 2/3
assert fiabilite(1,3,2,1.1) == 2/3
assert fiabilite(1,3,5,1.1) == 0
assert fiabilite(1,2,5,1.1) == 0
assert fiabilite(1,5,2,1.1) == 0
assert fiabilite(5,1,2,1.1) == 0
```

Troisième Solution : Bien sûr, on peut aussi précalculer nos égalités *à epsilon près*. Cette solution est peut-être plus simple mais elle effectue des calculs parfois inutiles.

```python
def fiabilite(v1, v2, v3, epsilon):
    """ float * float * float * float -> float
    Hypothèse: epsilon > 0

    Retourne le taux de fiabilité des trois mesures v1, v2 et v3 à epsilon près."""

    # egal_v1_v2 : bool
    egal_v1_v2 = egal_eps(v1, v2, epsilon)

    # egal_v2_v3 : bool
    egal_v2_v3 = egal_eps(v2, v3, epsilon)

    # egal_v1_v3 : bool
    egal_v1_v3 = egal_eps(v1, v3, epsilon)

    if egal_v1_v2 and egal_v2_v3 and egal_v1_v3:
        return 1
    elif ((egal_v1_v2 and egal_v2_v3)
        or (egal_v1_v2 and egal_v1_v3)
        or (egal_v1_v3 and egal_v2_v3)):
        return 2/3
    else:
        return 0
```

```python
# Jeu de tests
assert fiabilite(3,3,3,1.1) == 1
assert fiabilite(3,2,1,1.1) == 2/3
assert fiabilite(1,2,3,1.1) == 2/3
assert fiabilite(1,3,2,1.1) == 2/3
assert fiabilite(1,3,5,1.1) == 0
assert fiabilite(1,2,5,1.1) == 0
assert fiabilite(1,5,2,1.1) == 0
assert fiabilite(5,1,2,1.1) == 0
```

3 Répétitions et boucles

Nous avons déjà vu lors des cours 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**.

3.1 Répéter des calculs

Supposons que l'on veuille calculer la somme des 5 premiers entiers naturels. Une solution possible, à partir des éléments de langage que nous connaissons serait de définir la fonction suivante :

```python
def somme_5():
    """ -> int

        retourne la somme des 5 premiers entiers naturels."""

    return 1 + 2 + 3 + 4 + 5
```

```python
# Test
assert somme_5() == 15
```

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 somme des 5 mais des 100 000 premiers entiers naturels par exemple … Cela serait cependant encore formellement possible. Mais si on veut maintenant non plus les 100 000 mais les n premiers entiers naturels, n étant un paramètre de la fonction, là l'approche présentée n'est plus valable.

En fait, il est nécessaire de pouvoir 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 somme, nous aimerions un moyen de décrire le calcul générique :

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n$$

sans fixer à l'avancer la borne n.

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

Nommons `s` la somme ainsi construite étape par étape. Initialement `s` possède la valeur 0. On peut ensuite :

1. ajouter à `s` la valeur du prochain entier naturel 1 : `s` vaut désormais 1
2. ajouter à `s` la valeur du prochain entier naturel 2 : `s` vaut désormais 3

3. ajouter à `s` la valeur du prochain entier naturel 3 : `s` vaut désormais 6
4. ajouter à `s` la valeur du prochain entier naturel 4 : `s` vaut désormais 10
5. ajouter à `s` la valeur du prochain entier naturel 5 : `s` vaut désormais 15

On a alors atteint notre borne $n = 5$ et on peut constater que `s` vaut bien 15, la somme des entiers de 1 à 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 :

- affecter à `s` (la somme) sa valeur courante augmentée de la valeur de la variable `i`
- passer à la valeur suivante de `i` (par une incrémentation `i = 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 cours l'une des plus simples et des plus générales : la boucle `while` qui permet de répéter un bloc d'instructions *tant qu'une certaine condition booléenne donnée est satisfaite*.

Pour reproduire le calcul de la somme des entiers de 1 à n. Nous supposons avoir nos deux variables `s` (pour la somme) et `i` pour l'entier courant. Au départ, la somme cumulée vaut bien sûr `0`, et le premier entier courant est `1`.

Le principe de répétition est alors le suivant :

- répéter tant que `i` <= 5:
  - ajouter à `s` la valeur de l'entier naturel stocké dans `i`.
  - 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 calculer la somme des 5 premiers entiers :

```python
def somme_5_while():
    """ -> int

        retourne la somme des 10 premiers entiers naturels."""

    # i : int
    i = 1 # entier courant

    # s : int
    s = 0 # la somme cumulée

    while i <= 5:
        s = s + i
        i = i + 1
```

```
    return s
```

```
# Jeu de test
assert somme_5_while() == 1 + 2 + 3 + 4 + 5
```

Terminaison : à chaque étape de la boucle, la valeur de `i` est augmentée et se rapproche donc un peu plus de la valeur du nombre de répétitions que nous avons choisie (ici : 5). À la $6^{\text{ème}}$ étape, la valeur de `i` devient telle que la condition `(i<=5)` devient fausse.

Maîtriser l'art de savoir «sortir» d'une boucle est l'une des premières tâches de l'apprenti programmeur. Nous y reviendrons notamment lors du prochain cours d'approfondissement sur les boucles.

3.1.1 Syntaxe du while

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

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

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**.

Notez comme les instructions sont décalées par rapport à l'indentation du `while` (toujours de 4 espaces ou 1 tabulation).

S'il y a d'autres instructions «après» la boucle, on a :

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

Important : l'instruction `<instruction n+1>` ne fait pas partie du corps de la boucle.

3.1.2 Répétitions paramétrées

Dans la fonction `somme_5_while` vue précédemment, le `while` nous a permis d'écrire de façon synthétique une fonction pour calculer la somme des 5 premiers entiers. Par simple modification de la condition d'arrêt, on pourrait également décrire tout le calcul de la somme des 100 000 premiers entiers naturels. Mais il serait fastidieux de devoir proposer une définition de fonction spécifique pour chaque borne du calcul de la somme. Il est donc naturel de généraliser encore un peu plus pour en déduire une fonction qui calcule la somme des n premiers entiers naturels pour une borne n quelconque.

Pour cela, nous allons paramétrer notre fonction de calcul par la borne `n`. Ceci conduit à la spécification suivante :

```
def somme_entiers(n):
    """ int -> int
        hypothèse: n >= 1

        retourne la somme des n premiers entiers naturels.
        """
```

Par exemple :

```
>>> somme_entiers(10)
55
```

```
>>> somme_entiers(100000)
5000050000
```

En ajoutant un paramètre `n` à la fonction et en remplaçant la borne `5` par ce paramètre, nous obtenons une solution simple au calcul général de la somme des entiers de 1 à n.

```
def somme_entiers(n):
    """ int -> int
        hypothèse: n >= 1

        retourne la somme des n premiers entiers naturels."""

    # i : int
    i = 1 # entier courant, en commençant par 1

    # s : int
    s = 0 # la somme cumulée

    while i <= n:
        s = s + i
        i = i + 1

    return s
```

```
# Jeu de tests
assert somme_entiers(5) == 15
assert somme_entiers(10) == 55
assert somme_entiers(100000) == 5000050000
```

Nos tests semblent indiquer que la fonction calcule bien la somme des entiers de 1 à n pour un n arbitrairement grand.

Exercice : donner la définition d'une fonction `produit_entiers` qui calcule le produit des n premiers entiers naturels (non nuls).

3.2 Interprétation des boucles

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

3.2.1 Principe d'interprétation

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

Dans :

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

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

— 2a. 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.

— 2b. Soit la `<condition logique>` s'évalue en `False` et on sort de la boucle pour interpréter l'instruction `<instruction n+1>` (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 `somme_entiers`.

La simulation de boucle conduit à la construction d'une *table de simulation* selon les principes suivants :

— on fixe une valeur pour chaque paramètre de la fonction qui joue un rôle dans la boucle que l'on veut simuler.

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

— si des variables ne sont pas modifiées dans le corps de la boucle, mais jouent un rôle dans ce dernier, alors on fixe également une valeur.

Il n'y a pas de telle variable dans `somme_entiers`.

— on crée un tableau 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 `somme_entiers` la variable `s` est modifiée avant la variable `i`, on obtient donc l'en-tête suivant pour notre table de simulation :

tour de boucle	variable `s`	variable `i`

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

— on remplit la première ligne du tableau en indiquant `entrée` dans la colonne `tour de boucle` et on donne pour chaque variable sa valeur avant d'effectuer le premier tour de boucle.

Pour `somme_entiers` la valeur initiale de `s` est `0` et la valeur de `i` est `1` avant le premier tour de boucle, on obtient donc :

tour de boucle	variable `s`	variable `i`
entrée	0	1

— la seconde ligne et les suivantes indiquent le nombre de tours de boucle effectués (dans la colonne `tour de boucle`) ainsi que la valeur **après** ce tour de boucle pour les variables locales modifiées (dans les autres colonnes).

Pour `somme_entiers` la valeur de `s` après le premier tour est `1` et la valeur de `i` est `2`.

On obtient donc :

tour de boucle	variable `s`	variable `i`
entrée	0	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 `s`	variable `i`
entrée	0	1
1er tour	1	2
2e tour	3	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 `s`	variable `i`
entrée	0	1
1er tour	1	2
2e tour	3	3

tour de boucle	variable `s`	variable `i`
3e tour	6	4
4e tour	10	5
5e tour	15	6

A l'issue de ce cinquième tour, la condition `i <= n` n'est plus vérifiée car `i` vaut 6, la condition de boucle est donc invalidée et cela provoque donc la *sortie de boucle*.

— pour préciser que ce 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.

La simulation complète est donc la suivante :

tour de boucle	variable `s`	variable `i`
entrée	**0**	**1**
1er tour	1	2
2e tour	3	3
3e tour	6	4
4e tour	10	5
5e tour (sortie)	**15**	**6**

À la fin de la simulation, la valeur de `s` est donc 15 qui correspond bien à la somme des entiers de 1 à 5 (plus précisément de 1 à `n` pour `n` valant 5).

À retenir : le principe de construction des simulations de boucle sera souvent utilisé pour expliquer le déroulement des calculs répétitifs.

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 lors du prochain cours).

Cependant, en salle machine on aimerait également pouvoir observer le comportement des programmes sans systématiquement faire des simulations à la main. Pour cela, nous utilisons l'instruction `print` pour *tracer* l'interprétation des programmes et en particulier les boucles.

Voici comment reproduire notre simulation sur `somme_entiers` :

```python
def somme_entiers(n):
    """ int -> int
        Hypothèse: n >= 1

        retourne la somme des n premiers entiers naturels."""

    # i : int
    i = 1 # compteur
```

```python
    # r : int
    s = 0 # somme

    print("=====================")
    print("s en entrée vaut ", s)
    print("i en entrée vaut ", i)
    while i <= n:
        s = s + i
        i = i + 1
        print("------------------")
        print("s après le tour vaut ", s)
        print("i après le tour vaut ", i)
    print("------------------")
    print("sortie")
    print("=====================")

    return s
```

```python
>>> somme_entiers(5)
=====================
s en entrée vaut  0
i en entrée vaut  1
------------------
s après le tour vaut  1
i après le tour vaut  2
------------------
s après le tour vaut  3
i après le tour vaut  3
------------------
s après le tour vaut  6
i après le tour vaut  4
------------------
s après le tour vaut  10
i après le tour vaut  5
------------------
s après le tour vaut  15
i après le tour vaut  6
------------------
sortie
=====================
15
```

Important : dans ce cours, nous n'utiliserons `print` que pendant les travaux sur machine, et uniquement dans le but d'afficher des traces d'exécution.

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) ou d'un produit
— 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

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 :

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

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 $n-1$. 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 :

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

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
        Hypothèse: n >= 0

        retourne la valeur au rang n de la suite U."""

    # u : int
    u = 7  # valeur au rang 0

    # i : int
    i = 0  # initialement rang 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 ou d'un produit

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
        hypothèse: n >= 0

        retourne le n-ième terme de la série  :
                 1 + 1/2 + 1/4 + ... + (1/2)^n"""

    # s : float
    s = 0.0  # la somme vaut 0 initialement
```

```python
    # k : int
    k = 0    # on commence au rang 0

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

    return s
```

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

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

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

```python
>>> serie_s(1000)
2.0
```

Remarque 1 : on peut montrer assez simplement que, lorsque n tend vers l'infini, la série S_n converge effectivement vers 2.

Remarque 2 : ici les calculs se font en flottants (type `float`) et en raison des problèmes d'arrondis, il est assez fastidieux d'effectuer des simulations de boucle. On se limitera donc en général aux simulations pour les calculs exacts sur les entiers.

Le calcul d'un produit est également assez fréquent en mathématiques. Un exemple classique est la fonction factorielle qui correspond au calcul suivant :

$$n! = \prod_{k=1}^{n} k \;\; \text{pour} \;\; n > 0$$

Par exemple :

$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

Remarque : par convention, on définit $0! = 1$

Encore une fois, cette définition mathématique est relativement simple à traduire en Python.

```python
def factorielle(n):
    """int -> int
        hypothese : n > 0

        retourne le produit factoriel n!"""

    # k : int
    k = 1   # on démarre au rang 1
```

```python
    # f : int
    f = 1   # factorielle au rang 1

    while k <= n:
        f = f * k
        k = k + 1

    return f
```

```python
>>> factorielle(5)
120
```

Exercice : proposer un jeu de tests pour la fonction `factorielle`

Exercice : définir une fonction de calcul de la puissance n-ième d'un nombre x en suivant la formule, pour n entier strictement positif :

$$x^n = \prod_{k=1}^{n} x$$

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, les séries ou les produits. Nous en verrons un certain nombre en TD car la résolution de ces problèmes nécessite un peu d'entraînement.

En guise d'illustration, considérons le problème du calcul du PGCD de deux entiers naturels.

La spécification proposée est la suivante :

```python
def pgcd(a,b):
    """ int * int -> int
        Hypothèse: (a >= b) and (b > 0)

        Retourne le plus grand commun diviseur de a et b.
        """
```

Par exemple :

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

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

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

```python
>>> 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,b):
    """ int * int -> int
        Hypothèse: b > 0
        Hypothèse: a >= b

        Retourne le plus grand commun diviseur de a et b.
        """

    # q : int
    q = a   #  le quotient

    # r : int
    r = b   #  le diviseur

    # temp : int
    temp = 0   #  mémoire auxiliaire

    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
```

Exercices : donner la simulation correspondant à `pgcd(56, 42)`. Même question pour `pgcd(4199, 1530)`.

3.3.4 Boucles imbriquées : les couples d'entiers

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 :

```python
def nb_couples_distincts(n):
    """ int -> int
        hypothèse : n >= 0

        retourne le nombre de couples distincts (i,j) d'entiers
                 naturels inférieurs ou égaux à n."""

    # i : int
    i = 0 # premier élément du couple

    # j : int
    j = 0 # second élément du couple

    # compte : int
    compte = 0  #  pour compter les couples

    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
```

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

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

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

Les boucles imbriquées sont assez contraignantes à simuler (il faut faire plusieurs simulations imbriquées !), donc nous limiterons les simulations aux boucles simples.

En revanche, nous pourrons en TME utiliser `print` pour tracer l'exécution des boucles imbriquées.

```
def nb_couples_distincts(n):
    """ int -> int

        retourne le nombre de couples distincts (i,j) d'entiers
                 naturels inférieurs ou égaux à n."""

    # i : int
    i = 0 # premier élément du couple

    # j : int
    j = 0 # second élément du couple

    # compte : int
    compte = 0  #  pour compter les couples

    while i <= n:
        j = 0  # il faut réinitialiser j pour chaque "nouveau" i
        while j <= n:
            if i != j:
                print("couple (",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)
couple ( 0 , 1 )
couple ( 0 , 2 )
couple ( 0 , 3 )
couple ( 1 , 0 )
couple ( 1 , 2 )
couple ( 1 , 3 )
couple ( 2 , 0 )
couple ( 2 , 1 )
couple ( 2 , 3 )
couple ( 3 , 0 )
```

```
couple ( 3 , 1 )
couple ( 3 , 2 )
12
```

3.4 Complément : généraliser les calculs avec 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 :

```python
def suite(n, k, f):
    """int * Number * (Number -> Number) -> Number
    Hypothèse: 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 : `(Number -> Number)`. Cette signature indique que le troisième paramètre `f` doit être une référence à une fonction unaire paramétrée par un nombre et retournant un nombre.

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 :

```python
def suite(n, k, f):
    """ int * Number * (Number -> Number) -> Number
        Hypothèse: n >= 0

        Retourne l'élément de rang n de la suite U définie
        par U_0=k et pour tout n>0, U_n+1 = f(U_n).
        """

    # i : int
    i = 0   # élément courant de rang i (on démarre au rang 0)

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

```python
def suivant_u(x):
    """ Number -> Number

        Retourne l'élément suivant pour la suite U.
        """

    return 2 * x + 3
```

```python
# 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 :

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

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

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

```python
>>> suite(5, 7, suivant_u) == suite_u(5)
True
```

```python
>>> 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.

```python
>>> 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. Même question pour le produit.

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 :

```python
def annulation(a, b, f):
    """ int * int * (int -> float) -> bool
        Hypothèse: 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 `(int -> float)` qui fait de `annulation` une fonctionnelle. Ici, il s'agit naturellement de prendre la fonction f que l'on veut tester.

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
        Hypothèse : 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
        hypothèse : 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 ? Si oui proposer une telle fonction.

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

```python
def annulation(a, b, f):
    """ int * int * (int -> float) -> bool
        Hypothèse: a <= b

        Retourne True si la fonction f  s'annule sur l'intervalle [a;b].
        """

    # x : int
    x = a  # élément courant, au début de l'intervalle

    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
```

```python
# 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
```

Nous reviendrons sur les fonctionnelles lors d'un prochain cours et dans certains exercices de travaux dirigés.

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`.

3.5 Exercices corrigés

Exercice (3.1) : Somme des impairs (corrigé)

Question 1

Donner une définition de la fonction `somme_impairs_inf` telle que `somme_impairs_inf(n)` renvoie la somme des entiers impairs inférieurs ou égaux à `n`.

Par exemple :

```
>>> somme_impairs_inf(1)
1
```

```
>>> somme_impairs_inf(2)
1
```

```
>>> somme_impairs_inf(5)
9
```

Réponse

```python
def somme_impairs_inf(n):
    """ int -> int
    Hypothèse: n >= 0

    renvoie la somme de tous les entiers naturels impairs
            inférieurs ou égaux à n."""

    # s : int
    s = 0 # somme calculée

    # i : int
    i = 1 # impair courant  (1 est le premier impair)

    while i <= n:
        s =  s + i
        i = i + 2

    return s
```

```python
# Jeu de test
assert somme_impairs_inf(0) == 0
assert somme_impairs_inf(1) == 1
assert somme_impairs_inf(2) == 1
assert somme_impairs_inf(3) == 4
assert somme_impairs_inf(4) == 4
assert somme_impairs_inf(5) == 9
assert somme_impairs_inf(8) == 16
```

Question 2

Donner une définition de la fonction `somme_premiers_impairs` telle que `somme_premiers_impairs(n)` renvoie la somme des `n` premiers entiers impairs.

Par exemple :

```
>>> somme_premiers_impairs(1)
1
```

```
>>> somme_premiers_impairs(2)
4
```

```
>>> somme_premiers_impairs(5)
25
```

Remarque : comme les exemples ci-dessus le suggérent, la somme des n premiers impairs vaut n^2 (le démontrer est un bon exercice mathématique). On exploitera cette propriété dans les jeux de tests (rappel : l'opérateur d'élévation à la puissance est `**` en Python).

Réponse

```python
def somme_premiers_impairs(n):
    """ int -> int
    Hypothèse : n > 0

    renvoie la somme des n premiers entiers impairs."""

    # s: int
    s = 0 # somme calculée

    # i : int
    i = 1 # compteur

    # imp : int
    imp = 1  # impair courant (1 est le premier impair)

    while i <= n:
        s = s + imp
        imp = imp + 2
        i = i + 1

    return s
```

```python
# Jeu de tests
assert somme_premiers_impairs(1) == 1 ** 2
assert somme_premiers_impairs(2) == 2 ** 2
assert somme_premiers_impairs(3) == 3 ** 2
assert somme_premiers_impairs(4) == 4 ** 2
assert somme_premiers_impairs(5) == 5 ** 2
assert somme_premiers_impairs(6) == 6 ** 2
assert somme_premiers_impairs(7) == 7 ** 2
assert somme_premiers_impairs(8) == 8 ** 2
assert somme_premiers_impairs(9) == 9 ** 2
```

Question 3

Effectuer la simulation de boucle pour l'application `somme_premiers_impairs(5)` donc pour `n=5`.

Réponse

Tour de boucle	variable `s`	variable `imp`	variable `i`
entrée	**0**	**1**	**1**
1er	1	3	2
2e	4	5	3
3e	9	7	4
4e	16	9	5
5e (sortie)	**25**	**11**	**6**

Remarque : dans le corps de la boucle la première variable affectée est `s`, puis `imp` et enfin `i`. Cela impose l'ordre des colonnes dans la table de simulation (cf. le cours sur le sujet). L'idée est de toujours avoir en gros une seule réponse possible (pour une implémentation donnée bien sûr).

Exercice (3.3) : Fonction mystère (corrigé)

Le but de cet exercice est de réussir à déterminer ce que fait une fonction, sans en connaître ni le nom, ni la spécification mais simplement en étudiant son implémentation.

Soit la fonction «mystère» `f` ci-dessous :

```python
def f(x, y):
    """
    ??? mystère !"""

    # z : ?
    z = 0

    # w : ?
    w = x

    while w <= y:
        z = z + w * w
        w = w + 1

    return z
```

Question 1

Compléter cette définition en donnant la signature de la fonction ainsi que les types à déclarer pour les variables, en supposant que les paramètres `x` et `y` sont entiers.

Réponse

```python
def f(x, y):
    """ int * int -> int

    ??? mystère !"""
```

```python
    # z : int
    z = 0

    # w : int
    w = x

    while w <= y:
        z = z + w * w
        w = w + 1

    return z
```

Question 2

Selon les principes vus en cours, effectuer une *simulation de boucle* correspondant à l'évaluation de l'application :

```python
f(3, 6)
```

donc pour `x=3` et `y=6`.

Quelle est la valeur retournée par cette application ?

Réponse

Tour de boucle	variable `z`	variable `w`
entrée	**0**	**3**
1e	9	4
2e	25	5
3e	50	6
4e (sortie)	**86**	**7**

Remarque : la colonne de la variable `z` est à gauche de la colonne de la variable `w` car *dans le corps de la boucle* l'affectation à `z` précède l'affecation à `w` (il n'existe qu'une table de simulation possible, cf. description des principes de simulation dans le cours).

```python
>>> f(3,6)
86
```

Question 3

En supposant qu'on évalue `mystere(x, y)` avec $x \leq y$. Pour quelle valeur de w, la boucle va-t-elle s'arrêter ?

Réponse

La boucle incrémente la variable `w` à chaque passage. La condition devient donc fausse quand $w > y$, c'est-à-dire lorsque w vaut $y + 1$.

Question 4

Que pensez-vous de l'application `f(5, 3)` ?

Quelle est la valeur retournée ? En déduire une *hypothèse d'appel* pertinente pour la fonction mystère.

Réponse

1. Avant le premier passage dans la boucle, `w` vaut 5 (valeur de `x`), `z` vaut 0 et `y` vaut 3.
2. La condition de la boucle `while` $w \leq y$ est donc immédiatement fausse. On ne fait aucun passage.
3. La valeur de retour est donc 0, valeur initiale de `z`.

Donc il semble pertinent d'ajouter l'hypothèse d'appel suivante :

```
def f(x, y)
"""int * int -> int
   Hypothèse:  x <= y

   ...".
```

Question 5

En déduire une définition complète et plus lisible de cette fonction, en particulier :

— proposer un nom plus pertinent pour la fonction
— renommer les paramètres et expliquer leur rôle dans la description de la fonction
— renommer les variables et expliquer leur rôle dans le corps de la fonction
— proposer un jeu de tests pour valider la fonction.

Réponse

```
def somme_carres(m, n):
    """ int * int -> int
    Hypothèse: m <= n

    retourne la somme des carrés des entiers
    dans l'invervalle [m;n]."""

    # s : int
    s = 0   # la somme des carrés

    # i : int
    i = m   # entier courant dans l'intervalle

    while i <= n:
        s = s + i * i
        i = i + 1

    return s
```

```
# Jeu de tests
assert somme_carres(1, 5) == 55
assert somme_carres(2, 5) == 54
assert somme_carres(3, 5) == 50
assert somme_carres(4, 5) == 41
```

```python
assert somme_carres(5, 5) == 25
assert somme_carres(3, 6) == 86
assert somme_carres(-4, 0) == 30
```

Exercice (3.4) : Nombres premiers (corrigé)

Le but de cet exercice est d'écrire une fonction qui teste si un nombre est premier ou non. Ceci permet d'étudier les alternatives *dans* les boucles ainsi que les sorties anticipées de fonction.

On rappelle qu'un entier naturel n est dit *premier* s'il n'existe aucun entier naturel autre que 1 et n lui-même qui le divise.

Par convention, 1 n'est **pas** un nombre premier.

Question 1

Donner une définition de la fonction `divise` qui, étant donné un entier naturel non nul n et un entier naturel p renvoie `True` si n divise p, `False` sinon.

Par exemple :

```python
>>> divise(1, 4)
True
```

```python
>>> divise(2, 4)
True
```

```python
>>> divise(3,4)
False
```

```python
>>> divise(4,2)
False
```

Réponse

```python
def divise(n, p):
    """ int * int -> bool
    Hypothèse : n > 0 et p >= 0

    renvoie True si et seulement si n divise p."""

    return p % n == 0
```

```python
# Jeu de tests
assert divise(1, 4) == True
assert divise(2, 4) == True
assert divise(3, 4) == False
assert divise(4, 4) == True
assert divise(4, 2) == False
assert divise(17, 123) == False
assert divise(17, 357) == True
assert divise(21, 357) == True
```

Question 2

On se propose de définir la fonction `est_premier` qui, étant donné un entier naturel n, renvoie `True` si n est premier, `False` sinon.

Par exemple :

```
>>> est_premier(0)
False
```

```
>>> est_premier(1)
False
```

```
>>> est_premier(2)
True
```

```
>>> est_premier(17)
True
```

```
>>> est_premier(357)
False
```

Donner *deux* définitions *distinctes* de la fonction `est_premier` :

- une première définition *sans* sortie anticipée de la fonction
- une seconde définition *avec* sortie anticipée

Réponse

Première définition sans sortie anticipée :

```python
def est_premier(n):
    """int -> bool
    Hypothèse: n >= 0

    renvoie True si et seulement si n est premier."""

    if n < 2:
        return False
    else:
        # b : bool
        b=True  # pas de diviseur trouvé  ?

        # i : int
        i = 2  # prochain diviseur potentiel

        while b and (i < n):
            if divise(i, n):
                b = False
            else:
                i = i + 1

        return b
```

```python
# Jeu de tests
assert est_premier(0) == False
assert est_premier(1) == False
assert est_premier(2) == True
assert est_premier(17) == True
assert est_premier(357) == False
```

Deuxième définition avec sortie anticipée :

```python
def est_premier(n):
    """int -> bool
    Hypothèse: n >= 0

    renvoie True si et seulement si n est premier."""

    if n < 2:
        return False
    else:
        # i : int
        i = 2  # prochain diviseur potentiel

        while i < n:
            if divise(i, n):
                return False
            else:
                i = i + 1

        return True
```

```python
# Jeu de tests
assert est_premier(0) == False
assert est_premier(1) == False
assert est_premier(2) == True
assert est_premier(17) == True
assert est_premier(357) == False
```

4 Plus sur les boucles

Dans ce cours, 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 **correctement** au problème posé ?
— est-ce que le calcul effectué par la fonction **termine** ?
— le calcul effectué est-t-il **efficace** ?

Finalement, nous abordons en complément une approche différente pour traiter les calculs répétitifs : la **récursion**.

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 ?

Dans le cadre de problèmes numériques, cette question devient :

> La fonction effectue-t-elle un calcul correct ?

Cette problématique est intéressante à étudier dans le cadre des boucles car c'est la construction principale permettant de décrire des calculs complexes, dont l'étude de correction est le plus souvent non triviale.

Pour illustrer cet aspect nous allons considérer le problème de l'élévation d'un nombre à une puissance entière positive.

Voici une solution candidate :

```
def puissance(x,n):
    """ Number * int -> Number
        Hypothèse : n >= 0

        retourne la valeur de x élevé à la puissance n."""

    # res : Number
    res = 1 # valeur de x^0

    # i : int
    i = 1 # compteur

    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**

La simulation confirme que 2^5 vaut 32, la fonction semble donc à première vue correcte.

On aimerait cependant un argument un peu plus solide qu'un sentiment résultant d'une seule simulation pour des valeurs de paramètres fixées. Cet argument est obtenu en étudiant un *invariant de boucle*.

Définition : invariant de boucle

Un **invariant de boucle** est une expression booléenne devant :

— exprimer une relation intéressante impliquant les variables modifiées dans le corps de la boucle
— être vraie en *entrée* de boucle (avant le premier tour de boucle)
— rester vraie *après chaque tour* de boucle.

La «science» des invariants de boucle peut parfois relever de l'alchimie, mais il est possible d'expliquer un peu la méthodologie mise en œuvre. En particulier, le principe de simulation que nous avons introduit la semaine dernière se révèle être d'une aide précieuse.

Pour trouver un invariant de boucle, il faut :

— 1) 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) une (ou plusieurs) simulation(s)

Pour la puissance, nous avons déjà effectué une simulation et nous allons l'exploiter (en pratique, il en faudrait 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) de la pratique et un peu d'imagination

Rien ne vaut effectivement l'expérience et puis, à un moment ou à un autre, il faudra utiliser quelques neurones, c'est toujours la partie difficile.

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 :

> **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.

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 un cours d'introduction. 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 :

```python
def puissance_modif(x, n):
    """ Number * int -> Number
        Hypothèse : n >= 0

        retourne la valeur de x élevé à la puissance n."""

    # res : Number
```

```
    res = 1 # valeur de x^0

    # i : int
    i = 1 # compteur

    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.

Cela se confirme bien sûr en pratique :

```
>>> puissance_modif(2, 2)
2
```

```
>>> puissance_modif(2, 3)
4
```

```
>>> puissance_modif(2, 4)
8
```

```
>>> puissance_modif(2, 5)
16
```

Donc retenons qu'une légère modification peut bien sûr fausser le résultat du calcul : mais l'invariant de boucle nous aide à comprendre précisément *pourquoi* le calcul n'est pas le bon.

De plus, il est important de remarquer que la correction d'une fonction est 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)
...
```

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 *mesure* que l'on nomme un *variant de boucle*.

Définition : **Variant de boucle**

Un **variant de boucle** est une expression définie sur un ensemble muni d'un *ordre bien fondé* (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.

Pour simplifier un peu la méthode, dans ce cours on s'assurera que le variant de boucle :

— est un entier naturel positif en entrée,
— qui décroît strictement à chaque tour de boucle,
— 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 :

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 cours nous allons en rester à de simples vérifications pour des simulations données, comme ci-dessus.

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 aiment surtout 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 cours précédent la fonction de calcul de l'aire d'un triangle. Notre première définition était la suivante :

```python
def aire_triangle(a,b,c):
    """ Number * Number * Number -> float
        hypothèse : (a>0) and (b>0) and (c>0)
        hypothèse : 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."""

    return math.sqrt(((a + b + c) / 2)
                     * (((a + b + c) / 2) - a)
                     * (((a + b + c) / 2) - b)
                     * (((a + b + c) / 2) - c))
```

Cette définition est clairement loin d'être satisfaisante. Elle est tout d'abord illisible mais surtout le calcul `(a + b + c) / 2` est répété quatre fois . . . c'est trois fois de trop !

Une meilleure solution est de réaliser une factorisation des calculs grâce à une *variable* permettant de stocker le demi-périmètre, conduisant à la définition suivante :

```python
def aire_triangle(a,b,c):
    """ Number * Number * Number -> float
        hypothèse : (a>0) and (b>0) and (c>0)
        hypothèse : 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."""

    # p : float
    p = (a + b + c) / 2   # demi-périmètre

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

Cette définition est nettement plus satisfaisante : elle est beaucoup plus lisible et aucun calcul n'est répété inutilement.

La factorisation est un outil simple mais important pour rendre efficace les calculs. Cela conduit souvent à des définitions plus lisibles et mieux décomposées.

4.3.2 Sortie anticipée

Considérons la fonction suivante :

```python
def plus_petit_diviseur(n):
    """ int -> int
    Hypothèse : n >= 2

    retourne le plus petit diviseur de n (autre que 1)."""

    # d : int
    d = 0 # Diviseur trouvé, 0 pour démarrer (donc pas de diviseur)

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

    # m : int  (candidat diviseur)
    m = 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
```

```python
# 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 :

```
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 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 trouvé un diviseur justement.

Voici la version modifiée de la définition :

```python
def plus_petit_diviseur(n):
    """ int -> int
    Hypothèse : n >= 2

    retourne le plus petit diviseur de n (autre que 1)."""

    # d : int
    d = 0 # Diviseur trouvé, 0 pour démarrer (donc pas de diviseur)

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

    # m : int  (candidat diviseur)
    m = 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
```

```python
# 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 directement de la boucle, mais sortir de la fonction avec le **return** *implique* de sortir de n'importe quelle boucle située à l'intérieur de la fonction.

Ceci conduit à la définition suivante :

```python
def plus_petit_diviseur(n):
    """ int -> int
    Hypothèse : n >= 2

    retourne le plus petit diviseur de n (autre que 1)."""

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

    # m : int  (candidat diviseur)
    m = 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
```

```python
# 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.

En pratique, il faudra être capable de comprendre les deux types de solution : sortie anticipée de boucle ou sortie anticipée de fonction.

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.

```python
def puissance(x,n):
    """ Number * int -> Number
    Hypothèse : n >= 0

    retourne x élevé à la puissance n."""

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

    # i : int
    i = 1 # Compteur

    # nb_mults : int
    nb_mults = 0 # nombre de multiplication(s), initialement 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
10715086071862673209484250490600018105614048117055336074437 50 ...
```

```
>>> 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,

$$x^j \times x^k = x^{j+k}$$

.

Et, 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 :

$$\left[\begin{array}{l} x^4 = x^{\lfloor 4/2 \rfloor} \times x^{\lfloor 4/2 \rfloor} = x^2 \times x^2 \\ x^5 = x^{\lfloor 5/2 \rfloor} \times x^{\lfloor 5/2 \rfloor} \times x = x^2 \times x^2 \times x \end{array} \right.$$

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 :

```
def puissance_rapide(x, n):
    """ Number * int -> Number
    Hypothèse : n >= 0

    retourne x élevé à la puissance n."""
```

```python
    # res : Number
    res = 1   # résultat

    # acc : Number
    acc = x   # accumulateur pour les puissances impaires

    # i : int
    i = n     # variant de boucle (voir plus loin)

    # nb_mults : int
    nb_mults = 0  # compteur de multiplications

    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
```

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

```python
>>> puissance_rapide(2, 10)
Nombre de multiplications = 6
1024
```

```python
>>> puissance_rapide(2, 100)
Nombre de multiplications = 10
1267650600228229401496703205376
```

```python
>>> puissance_rapide(2, 1000)
Nombre de multiplications = 16
10715086071862673209484250490600018105614048117055336074437 50 ...
```

```python
>>> puissance_rapide(2, 10000)
Nombre de multiplications = 19
19950631168807583848837421626835850838234968318861924548520 08 ...
```

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 importe est que ce nombre de multiplication est dominé par le logarithme $\log_2(n)$ qui croît extrêmement lentement, et 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 (remarque : `i % 2 == 0`)	4	5
2e	4 (remarque : `i % 2 == 1`)	16	2
3e	4 (remarque : `i % 2 == 0`)	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.

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 :

```python
# 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 les deux définitions de la factorielle :

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.

Un mathématicien qui s'intéresse plutôt au COMMENT écrira sans doute :

$$n! = \begin{cases} 1 & \text{si } n \leq 1 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

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 procéder au calcul.

Cette version dite **récursive** se traduit très simplement en Python.

```python
def factorielle(n):
    """ int -> int
    Hypothèse : 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 avec récursion et l'implémentation associée en Python.

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
```

Il est également plus facile de raisonner qu'avec les invariants et variants de boucle car avec la récursion on dispose 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 d'induction 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 à notre cours : Python est un langage qui ne favorise par la récursion, en particulier du point de vue de l'efficacité.

Des langages de programmation plus adéquats existent pour apprendre les principes des algorithmes récursifs et de leur implémentation naturelle sous forme de fonctions récursives. On citera en particulier les *langages fonctionnels* Scheme, Ocaml et Haskell. Tous trois méritent d'être découverts en complément des langages impératifs comme Python, C/C++ ou Java.

4.5 Exercices corrigés

Exercice (4.1) : Retour sur la factorielle (corrigé)

Cet exercice reprend la définition de la factorielle vue en cours et étudie les problèmes de correction et de terminaison la concernant.

La définition proposée en cours est rappelée ci-dessous :

```
def factorielle(n):
    """int -> int
        hypothese : n >= 0

        Retourne le produit factoriel n!
        """

    # k : int
    k = 1   # on démarre au rang 1

    # f : int
    f = 1   # factorielle au rang 1

    while k <= n:
        f = f * k
        k = k + 1

    return f
```

Question 1

Proposer un jeu de tests pour valider la fonction `factorielle`.

Réponse

```
# Jeu de tests
assert factorielle(0) == 1
assert factorielle(1) == 1
assert factorielle(2) == 2
assert factorielle(3) == 6
assert factorielle(4) == 24
assert factorielle(5) == 120
assert factorielle(6) == 720
```

Question 2

Effectuer la simulation de boucle pour l'application `factorielle(5)`.

Réponse

Tour de boucle	variable `f`	variable `k`
entrée	**1**	**1**
1er	1	2
2e	2	3
3e	6	4
4e	24	5
5e (sortie)	**120**	**6**

Question 3 : à propos de la correction

Afin de vérifier que la définition proposée ci-dessus réalise correctement le calcul de la factorielle :

— proposer un **invariant de boucle**
— vérifier cet invariant sur la simulation de boucle pour `factorielle(5)`.
— en supposant que cet invariant est vérifié pour n'importe quelle valeur de `n` satisfaisant l'hypothèse `n > 0`, justifier le fait que `factorielle(n)` calcule bien $n!$.

Réponse

Rappel : l'invariant doit être une expression booléenne vraie en entrée de boucle ainsi qu'après chaque tour de boucle (cf. cours).

Après quelques minutes (courtes) de réflexion, si aucun étudiant ne propose …

candidat **invariant de boucle** : $f = (k-1)!$ (ou $f = \prod_{i=1}^{k-1} i$).

Remarque : l'invariant est une expression mathématique, pas une expression python.

Pour vérifier l'invariant sur la simulation, on ajoute simplement une colonne dédiée :

Tour de boucle	variable `f`	variable `k`	**invariant** $f = (k-1)!$
entrée	**1**	**1**	$1 = (1-1)!$ (vrai par convention)
1er	1	2	$1 = (2-1)!$ (vrai)
2e	2	3	$2 = (3-1)!$ (vrai)
3e	6	4	$6 = (4-1)!$ (vrai)
4e	24	5	$24 = (5-1)!$ (vrai)
5e (sortie)	**120**	**6**	$120 = (6-1)!$ (vrai)

On a donc vérifié notre candidat invariant pour `n=5`. On peut montrer que cet invariant de boucle est vérifié quel que soit `n` satisfaisant l'hypothèse `n > 0` mais cela dépasse le cadre du cours.

Si on en fait l'hypothèse alors en sortie de boucle on a $k = n + 1$ donc l'invariant devient $f = (n + 1 - 1)! = n!$. Comme la fonction retourne f elle calcule bien $n!$.

En conclusion notre fonction est correcte.

Question 4 : à propos de la terminaison

Afin d'assurer la terminaison de la fonction `factorielle` :

— proposer un **variant de boucle**
— vérifier ce variant sur la simulation de boucle `factorielle(5)`
— justifier informellement le fait que `factorielle(n)` termine toujours.

Réponse

Rappel : le variant doit être une expression arithmétique sur les entiers naturels

```
- qui décroit strictement après chaque tour de boucle
- qui vaut 0 lorsque la condition de boucle devient fausse (donc en sortie de boucle)
```

Après quelques minutes (courtes) de réflexion, si aucun étudiant ne propose ...

Candidat **variant de boucle** : $n + 1 - k$

Tour de boucle	variable `f`	variable `k`	**variant** $n + 1 - k$
entrée	**1**	**1**	**5**
1er	1	2	4
2e	2	3	3
3e	6	4	2
4e	24	5	1
5e (sortie)	**120**	**6**	0

Donc le variant de boucle est vérifié pour `n=5`.

Relativement informellement, on sait que :

— $n \geq 0$ et $k = 0$ en entrée de boucle donc $n + 1 - k$ est positif.
— le variant $n + 1 - k$ décroit strictement après chaque tour de boucle.
— et la plus petit valeur de k pour laquelle $k > n$ (la condition de boucle est fausse) est $k = n + 1$ donc exactement lorsque le variant $n + 1 - k = 0$.

On peut donc en conclure que la boucle termine pour n'importe quelle valeur de `n` satisfaisant l'hypothèse `n > 0`.

Message à faire passer aux étudiants : l'essentiel à retenir est de savoir vérifier un invariant ou un variant sur des simulations précises, et d'avoir compris la démarche de réflexion. Mais on ne leur demandera pas de preuves détaillées ou de recherche d'invariants ou de variants complexes.

Exercice (4.3) : Fonction mystère II (corrigé)

Le but de cet exercice est de réussir à déterminer ce que calcule une fonction mystère en simulant des boucles imbriquées.

Considérer la fonction «mystère» `f` suivante:

```python
def f(n,m):
    a=n
    b=0
    c=0

    while a>0:
        while a>0:
            a = a-1
            b = b+1
        a = b-1
        b = 0
        c = c+m
    return c
```

Question 1

Compléter la définition ci-dessus avec la signature de la fonction ainsi que les déclarations de variables.

Réponse

```python
def f(n, m):
    """
    int * int -> int
    """

    # a : int
    a = n

    # b : int
    b = 0

    # c : int
    c = 0

    while a > 0:
        while a > 0:
            a = a - 1
            b = b + 1

        a = b - 1
        b = 0
        c = c + m

    return c
```

Question 2

Calculer “à la main” différentes valeurs de `f` (sur de petits entiers).

On explique ici comment effectuer une simulation de boucles imbriquées, avec l’appel de `f(3,4)`.

Les variables `a`, `b` et `c` sont modifiées dans cet ordre à chaque tour de boucle, ce seront donc les colonnes principales de notre simulation.

Dans le cas de deux boucles imbriquées, on distingue la boucle *extérieure* et la boucle *intérieure*.

On construit un tableau où la première colonne indique à quel tour de la boucle extérieure l'on se trouve et la deuxième colonne indique à quel tour de la boucle intérieure l'on se trouve (ou «-») quand on est en dehors de cette boucle). Les valeurs des variables sont celles en fin de tour de boucle comme pour les simulations «simples».

tour de boucle ext.	**tour de boucle int.**	**a**	**b**	**c**
entrée	-	**3**	**0**	**0**
	entrée	**3**	**0**	**0**
	1er	2	1	0
	2e	1	2	0
	3e (sortie)	0	3	0
1er	-	2	0	4
	entrée	**2**	**0**	**4**
	1er	1	1	4
	2e (sortie)	0	2	4
2e	-	1	0	8
	entrée	**1**	**0**	8
	1er (sortie)	0	1	8
3e (sortie)	-	**0**	**0**	**12**

Expliquer comment interpréter cette simulation.

Réponse

```
>>> f(2, 1)
2
```

```
>>> f(3, 10)
30
```

```
>>> f(0, 8)
0
```

```
>>> f(5, 0)
0
```

```
>>> f(6, 7)
42
```

La lecture de la simulation n'est pas difficile :

- l'entrée de la boucle extérieure est comme pour les boucles simples.
- il y a une entrée de boucle intérieure :
  - après l'entrée de la boucle extérieure (ici, il n'y a pas d'affectation entre les deux `while` mais ce n'est bien sûr pas le cas en général)
  - après chaque fin de tour la boucle extérieure
- le tiret de la boucle intérieur correspond au code qui est :

— soit entre les deux `while` (ici vide mais il faut compter l'entrée du `while extérieur`)
— soit entre la fin du *dernier* tour de la boucle intérieure et la prochaine entrée.

Sur ces bases, les étudiants devraient réussir à lire la simulation.

Remarque : insister sur le fait que nous ne leur demanderons pas de faire eux même des simulations imbriquées.

Question 3

Conjecturer ce que calcule `f`.

Que pensez-vous de :

— `f(3, -4)` ?
— `f(-3, 4)` ?

En déduire une définition complète de la fonction, en lui trouvant un nom plus explicite.

Réponse

D'après la simulation précédente, la fonction `f` ajoute la valeur de `m` à `c` autant de fois que l'on réinitialise `a`. Comme on réinitialise `a` en le décrémentant, on conjecture donc que la fonction renvoie **`n` fois `m`** et réalise donc la multiplication de ses entrées.

```
>>> f(3, -4)
-12
```

```
>>> f(-3, 4)
0
```

On trouve donc raisonnable de restreindre la fonction aux entiers naturels.

```
def mult(n, m):
    """ int * int -> int
    Hypothèse : (n >= 0) and (m >= 0)

    renvoie la multiplication de n par m."""

    # a : int
    a = n

    # b : int
    b = 0

    # c : int
    c = 0

    while a > 0:
        while a > 0:
            a = a - 1
            b = b + 1

        a = b - 1
        b = 0
```

```python
        c = c + m

    return c
```

```python
# Jeu de tests
assert mult(3, 4) == 12
assert mult(2, 1) == 2
assert mult(3, 10) == 30
assert mult(8, 0) == 0
assert mult(0, 5) == 0
assert mult(6, 7) == 42
assert mult(9, 99) == 891
```

Question 4

Question subsidiaire : combien d'opérations arithmétiques sont nécessaires pour calculer `f(n, m)` en fonction de `n` et `m` ?

Proposer une version plus efficace de la fonction. Combien d'opérations arithmétiques sont-elles alors nécessaires ?

Réponse

```python
def mult_rapide(n, m):
    """ int * int -> int
    Hypothèse : (n >= 0) and (m >= 0)

    renvoie la multiplication de n par m."""

    return n * m
```

Il faut une seule multiplication, bien sûr !

Exercice (4.5) : Retour sur l'algorithme d'Euclide (corrigé)

Cet exercice reprend la définition de l'algorithme d'Euclide vu en cours et étudie les problèmes de correction et de terminaison la concernant.

La définition proposée en cours est rappelée ci-dessous :

```python
def pgcd(a, b):
    """ int * int -> int
    Hypothèse: b > 0 et a >= b

    retourne le plus grand commun diviseur de a et b."""

    # q : int
    q = a   #  le quotient

    # r : int
    r = b   #  le diviseur
```

```
    # temp : int
    temp = 0   #  variable temporaire

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

    return q
```

Question 1 - Simulation

Effectuer une simuation de `pgcd(54,15)` donc pour `a=54` et `b=15`.

Réponse

Le couple `(54,15)` vérifie bien les hypothèses de la fonction: $54 \geq 15 > 0$.

La variable temporaire `temp` servant uniquement à stocker une valeur pour `r`, on ne reporte pas sa valeur dans le tableau.

Tour de boucle	variable `q`	variable `r`
entrée	**54**	**15**
1er	15	9
2e	9	6
3e	6	3
4e (sortie)	**3**	**0**

```
>>> pgcd(54, 15)
3
```

Question 2 - Correction

On discute maintenant de la correction de la fonction `pgcd`. Le *candidat* invariant de boucle que nous proposons est le suivant :

invariant de boucle : $\mathbf{div}(a, b) = \mathbf{div}(q, r) \wedge (q \geq r \geq 0)$, où $\mathbf{div}(x, y)$ est l'ensemble des diviseurs communs à x et y.

- vérifier cet invariant sur la simulation de `pgcd(54,15)`.
- en supposant que cet invariant est vérifié pour n'importe quelle valeur de `a, b` satisfaisant l'hypothèse $a \geq b > 0$, justifier le fait que `pgcd(a, b)` calcule bien le pgcd de `a` et `b`.

Réponse

Remarque : dans cette question, il faut bien mettre en avant que l'invariant de boucle est une expression logique-mathématique et non une expression Python.

Vérifions l'invariant sur la simulation:

Tour de boucle	variable `q`	variable `r`	$\mathbf{div}(q,r)$	$q \geq r \geq 0$	Invariant
entrée	**54**	**15**	$\mathbf{div}(54,15) = \{1,3\}$	$54 \geq 15 \geq 0$	Vrai
1er	15	9	$\mathbf{div}(15,9) = \{1,3\}$	$15 \geq 9 \geq 0$	Vrai
2e	9	6	$\mathbf{div}(9,6) = \{1,3\}$	$9 \geq 6 \geq 0$	Vrai
3e	6	3	$\mathbf{div}(6,3) = \{1,3\}$	$6 \geq 3 \geq 0$	Vrai
4e (sortie)	**3**	**0**	$\mathbf{div}(3,0) = \{1,3\}$	$3 \geq 0 \geq 0$	Vrai

On a donc vérifié notre candidat invariant pour le couple d'entrée `(54,15)`.

Complément : On peut montrer que cet invariant de boucle est vérifié quels que soient `a,b` satisfaisant les hypothèses `n > 0` (c'est hors programme mais intéressant).

Voilà comment faire:

Supposons `a,b` tels que $a \geq b > 0$.

L'invariant est vrai au début de la fonction. En effet, comme $q = a$ et $b = r$, on a trivialement $\mathbf{div}(q,r) = \mathbf{div}(a,b)$ et l'hypothèse de départ implique $(q \geq r \geq 0)$.

Supposons que l'invariant est vrai au début d'une boucle. On a $\mathbf{div}(a,b) = \mathbf{div}(q,r) \wedge (q \geq r \geq 0)$. La condition de boucle nous indique en plus que $r \neq 0$.

Appelons `qq` et `rr` les valeurs respectives de `q` et `r` en fin de boucle. On a `qq = r` et `rr = q % r`.

Soit d un diviseur de q et r. Trivialement, d divise qq. La définition du modulo nous permet d'écrire $q = r * s + rr$ avec $0 \leq rr < r$. Comme d divise q, d divise la somme $r * s + rr$, et comme d divise $r * s$ par hypothèse, alors d divise rr.

Soit d un diviseur de qq et rr. Trivialement, d divise r. La définition du modulo nous permet d'écrire $q = r * s + rr$ avec $0 \leq rr < r$. Comme d divise rr et d divise $r * s$ (car il divise r), alors d divise q.

On vient de montrer que les diviseurs communs à q et r sont les mêmes que ceux communs à qq et rr. On a supposé que $\mathbf{div}(a,b) = \mathbf{div}(q,r)$, on a montré que $\mathbf{div}(qq,rr) = \mathbf{div}(q,r)$, on en conclut que $\mathbf{div}(a,b) = \mathbf{div}(qq,rr)$.

On a supposé que $q \geq r \geq 0$, et on a, par définition du modulo, $q = r * s + rr$ avec $0 \leq rr < r$. On conclut que $qq = r \geq rr \geq 0$.

Ainsi, on vient de montrer que si on suppose l'invariant vrai au début d'une boucle, il est vrai en fin de boucle.

Fin du complément

Si l'invariant est vrai en sortie de boucle, on a (entre autres) $\mathbf{div}(q,r) = \mathbf{div}(a,b)$. La condition de sortie de boucle nous indique que $r = 0$, ainsi $\mathbf{div}(q,r)$ est égal à l'ensemble des diviseurs de q. Et l'invariant devient $\mathbf{div}(q) = \mathbf{div}(a,b)$, c'est-à-dire "l'ensemble des diviseurs communs à a et b et l'ensemble des diviseurs de q", ce qui équivaut à "q est le plus grand diviseur commun à a et b".

En conclusion notre fonction est correcte.

Question 3 - Terminaison

Discuter de la terminaison de la fonction `pgcd` :

— proposer un **variant de boucle**
— vérifier ce variant sur la simulation de boucle `pgcd(54,15)`
— justifier informellement le fait que `pgcd(a,b)` termine toujours.

Réponse

Le **variant de boucle** de plus évident est : r

On a déjà la valeur de la variable `r` dans la simulation effectuée précédemment :

Tour de boucle	variable `r`
entrée	**15**
1er	9
2e	6
3e	3
4e (sortie)	**0**

Donc le variant de boucle est vérifié pour le couple d'entrées `54,15`.

Montrons que `pgcd(a,b)` termine. On utilise pour cela la deuxième partie de l'invariant de la fonction précédente: on sait qu'à tout moment $q \geq r \geq 0$.

Comparons la valeur du variant au début et à la fin d'une boucle. Appelons `rr` la valeur de `r` en fin de boucle. On sait que `rr = q % r`. On a, par définition du modulo, $q = r * s + rr$ avec $q > rr \geq 0$. On sait par l'invariant que $q \geq r$, on en déduit que s ne peut valoir 0 (sinon on a $rr = q > rr$). L'invariant nous dit aussi que r et q sont positifs, et on en déduit que $rr < r$.

Plus simplement, on peut dire que dans `rr = q % r` si `r` est non nul, alors `rr < r` **par définition du modulo**.

On vient de montrer que r décroit strictement à chaque tour de boucle. Comme la condition de sortie de boucle est $r = 0$, toute exécution finit par sortir de la boucle, et `pgcd` termine.

5 Séquences, intervalles et chaînes de caractères

Nous avons pour l'instant principalement résolu des problèmes numériques en manipulant des données du type `int` ou `float`. Nous allons à partir de ce cours nous éloigner quelque peu des problèmes mathématiques (et de leurs solutions informatiques), pour aborder des problèmes typiquement informatiques.

Pour cela, nous allons commencer à manipuler des **données structurées** avec dans ce cours les *intervalles* de type `range` et les *chaînes de caractères* de type `str`. Ce sont tous deux des types de données structurées *en séquence* : les éléments contenus dans la donnée sont arrangées 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 cours 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 possède le type `range` en Python.

Les manipulations sur les intervalles concernent principalement :

— la construction d'intervalle
— l'itération sur un intervalle

5.1.1 Construction d'intervalle

En python, on peut construire un intervalle d'entiers en faisant appel à `range`.

La syntaxe :

`range(m,n)`

construit l'intervalle des entiers allant de m (inclus) à n (exclu).

La notation mathématique usuelle pour cet intervalle est : $[m; n[$

Remarques :

— le nombre d'éléments dans l'intervalle est égal à $n - m$.
— une notation équivalente est $[m; n - 1]$.

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* de Python, on considérera donc `range` comme un type atomique.

```
>>> type(range(2, 6))
range
```

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 :

```
for <var> in <sequence>:
    <corps>
```

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>` n'est plus utilisable après le dernier tour de boucle.

5.1.2.1 Exemple : somme des entiers par itération

Suivant le principe d'itération, nous pouvons par exemple réécrire la fonction `somme_entier` du cours 3 de façon plus concise et surtout de façon plus lisible qu'avec une boucle `while`.

Rappelons tout d'abord la définition donnée dans le cours 3 sur les boucles.

```
def somme_entier(n):
    """ int -> int
    Hypothèse: n >= 0

    renvoie la somme des n premiers entiers naturels."""
```

```python
    # i : int
    i = 1 # entier courant, en commençant par 1

    # s : int
    s = 0 # somme cumulée en résultat

    while i <= n:
        s = s + i
        i = i + 1

    return s
```

Avec `for` la définition ci-dessus peut être simplifiée de la façon suivante :

```python
def somme_entier(n):
    """ int -> int
    Hypothèse: n >= 0

    renvoie la somme des n premiers entiers naturels."""

    # s : int
    s = 0 # somme cumulée en résultat

    # i : int (entier courant)
    for i in range(1, n+1):
        s = s + i

    return s
```

Par exemple :

```python
>>> somme_entier(10)
55
```

N'oublions pas le jeu de tests.

```python
# jeu de tests
assert somme_entier(0) == 0
assert somme_entier(3) == 6
assert somme_entier(4) == 10
assert somme_entier(5) == 15
assert somme_entier(10) == 55
```

Remarquons qu'il faut déclarer le type de la variable d'itération 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 c'est la première variable modifiée à chaque tour, avant les modifications du corps de la boucle.

Effectuons la simulation pour `somme_entier(5)` donc pour `n=5` :

tour de boucle	variable `i`	variable `s`
entrée	-	**0**
1er tour	1	1
2e	2	3
3e	3	6
4e	4	10
5e	5	15
sortie	-	**15**

Important : contrairement aux variables locales - qui sont accessibles dans tout le corps de la fonction - les variables d'itérations 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).

Exercice : reprendre les fonctions déjà étudiées en cours et en TD/TME et en proposer, lorsque cela est possible, une version avec des itérations sur des intervalles.

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` représentant une *séquence de caractères.*

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, etc.).

Les manipulations courantes sur les chaînes de caractères peuvent être catégorisées de la façon suivante :

— les opérations de base : construction, déconstruction et comparaisons
— les problèmes de réduction
— les problèmes de transformation et de filtrage
— les autres problèmes, en général plus complexes, 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 Comme indiqué lors du premier cours, 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.

Exercice : proposer deux façons différentes de construire une chaîne contenant des guillemets doubles.

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 :

$$c == (c \texttt{ +'') == (''+ } c)$$

Pour illustrer l'utilisation de l'opérateur de concaténation, considérons le problème de construction de chaînes 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, n):
    """ str * int -> str
    Hypothèse : n >= 1

    retourne la chaîne composée de n répétitions
             successives de la chaîne s."""

    # r : str
    r = ''  # on initialise avec la chaîne vide puisque c'est l'élément neutre

    # 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 ?

Nous verrons lors du prochain cours (sur les listes) que les découpages de séquences (donc de chaînes) peuvent être beaucoup plus complexes mais pour l'instant 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 :

```
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. Cette chaîne est donc de longueur 19 soit le nombre de caractères ou encore «l'indice du dernier caractère plus un».

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."""

    # l : int
    l = 0  # comptage de la longueur initialement à zéro

    # c : str   (caractère courant)
    for c in s:
        l = l + 1  # longueur incrémentée pour chaque caractère

    return l
```

```
# 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 aucun calcul puisque Python maintient systématiquement la longueur de la liste.

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, s):
    """str * str -> int
    Hypothèse : len(c) == 1

    retourne le nombre d'occurrences du caractère c dans la chaîne s"""

    # nb : int
    nb = 0  # nombre d'occurrences du caractère

    # 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, s):
    """str * str -> bool
    Hypothèse : 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, s):
    """str * str -> bool
    Hypothèse : len(c) == 1

    Retourne True si le caractère c est présent dans la chaîne s,
             ou False sinon"""

    # 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`.

La spécification est la suivante :

```python
def recherche(c, s):
    """str * str -> int + NoneType
    Hypothèse: len(c) == 1

    retourne l'indice de la première occurrence du caractère c dans s,
             ou None si le caractère est absent."""
```

Par exemple :

```python
>>> recherche('e', 'les revenantes')
1
```

```python
>>> recherche('n', 'les revenantes')
8
```

```python
>>> recherche('e', 'la disparition')
```

Dans ce dernier cas, Python ne produit aucun affichage. C'est le signe que la valeur `None` (unique valeur de type `NoneType`) a été retournée. Vérifions ce fait :

```python
>>> type(recherche('e', 'la disparition'))
NoneType
```

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 -> int + NoneType
```

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 `T + NoneType`. Nous verrons d'autres exemples de fonctions partielles dans ce cours et les suivants.

La définition proposée pour `recherche` est la suivante :

```
def recherche(c, s):
    """str * str -> int + NoneType
    Hypothèse: len(c) == 1

    retourne l'indice de la première occurrence du caractère c dans s,
             ou None si le caractère est absent."""

    # i : int
    i = 0  # indice courant, en commençant 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, s):
    """str * str -> int + NoneType
    Hypothèse: len(c) == 1

    retourne l'indice de la première occurrence du caractère c dans s,
             ou None si le caractère est absent."""

    # i : int
    for i in range(0,len(s)):
```

```
        if s[i] == c:
            return i

    return None
```

```
# Jeu de tests
assert recherche('e', 'les revenantes') == 1
assert recherche('n', 'les revenantes') == 8
assert recherche('e', 'la disparition') == None
```

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.

5.3.2.1 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, d, s):
    """str * str * str -> str
    Hypothèse: (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` :

```python
def substitution(c, d, s):
    """str * str * str -> str
    Hypothèse: (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."""

    # r : str
    r = ''  # chaîne résultat

    # 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
```

```python
# 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.
```

5.3.2.2 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 :

```python
def suppression(c, s):
    """str * str -> str
    Hypothèse: len(c) == 1

    retourne la sous-chaîne de s dans laquelle toutes
             les occurrences du caractère c ont été supprimées."""
```

Par exemple :

```python
>>> suppression('e', 'les revenantes')
'ls rvnants'
```

```
>>> suppression('e', 'la disparition')
'la disparition'
```

```
>>> suppression('z', '')
''
```

Voici la solution proposée :

```
def suppression(c, s):
    """str * str -> str
    Hypothèse: len(c) == 1

    retourne la sous-chaîne de s dans laquelle toutes
             les occurrences du caractère c ont été supprimées."""

    # r : str
    r = ''  # la chaîne résultat

    # 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.

5.3.3.1 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 :

```python
def inversion(s):
    """str -> str

    retourne la chaîne s inversée."""
```

Par exemple :

```python
>>> inversion('abcd')
'dcba'
```

```python
>>> inversion('a man a plan a canal panama')
'amanap lanac a nalp a nam a'
```

```python
>>> 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.

```python
def inversion(s):
    """str -> str

       retourne la chaîne s inversée."""

    # r : str
    r = ''  # la chaîne inversée résultat

    # 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
```

```python
# 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'`

tour de boucle	variable `c`	variable `r`
3e	`'c'`	`'cba'`
4e	`'d'`	`'dcba'`
sortie	`-`	*`'dcba'`*

5.3.3.2 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, s2):
    """str * 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 :

```python
def entrelacement(s1, s2):
    """str * str -> str

    renvoie la chaîne constituée par l'entrelacement
            des caractères des chaînes s1 et s2."""

    # i : int
    i = 0  # indice pour parcourir les deux chaînes

    # r : str
    r = ''  # chaîne résultat

    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
```

```python
# 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 `entrelacement('ace', 'bdfghi')` donc pour `s1='ace'` et `s2='bdfghi'` :

Pour la boucle, la simulation est la suivante :

tour de boucle	variable `r`	variable `i`
entrée	`''`	0
1er	`'ab'`	1
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'
```

5.4 Exercices corrigés

Exercice (5.1) : Intervalles (corrigé)

Cet exercice permet de se familiariser avec les intervalles (type `range`) et les boucles d'itérations avec `for ... in ...`.

Les réponses aux questions doivent donc impérativement exploiter ces constructions.

Question 1

Donner une définition de la fonction `somme_carres` qui, étant donné un entier naturel `n`, retourne la somme des carrés des nombres entiers inférieurs ou égaux à `n`.

Réponse

```
def somme_carres(n):
    """int -> int
    Hypothèse : n >= 0

    retourne la somme des carrés des entiers inférieurs ou
             égaux à n."""

    # s : int
    s = 0  # la somme à calculer

    # i : int (entier courant)
    for i in range(1, n + 1):
        s = s + i * i

    return s
```

```
# Jeu de tests
assert somme_carres(0) == 0
assert somme_carres(1) == 1
assert somme_carres(2) == 5
assert somme_carres(3) == 14
assert somme_carres(4) == 30
assert somme_carres(5) == 55
```

Remarque : contrairement aux variables locales, les variables d'itérations sont déclarées directement au dessus du `for` (cf. le cours). En effet, dans "notre" Python on considère que le `for` introduit une portée pour la variable d'itération, ce qui n'est malheureusement pas implémenté par Python. La variable `i` ci-dessus “existe” donc encore après la boucle `for` mais on interdit à nos étudiants de la référencer. Cela correspond bien sûr à une bonne pratique de programmation en Python.

Question 2

Soit la fonction mystere `f` suivante :

```
def f(m, n):
    """ ??? -> ???
        ???
    """

    # p: int
    p = 1

    # k : int
    k = m
    while k < n:
        p = p * k * k * k
        k = k+1

    return p
```

Expliquer ce que fait cette fonction et, en complément, donner une définition mathématique de ce calcul et compléter sa spécification.

En déduire une définition complète (avec spécification et jeu de tests) de cette fonction mystère (renommée pour l’occasion) en utilisant une boucle `for ... in ...` en remplacement de `while`.

Effectuer une simulation de votre fonction pour `m=4` et `n=8`.

Réponse

Cette fonction calcule le produit des cubes des entiers k dans l’intervalle $[m, n[$.

$$f(m, n) = \prod_{k=m}^{n-1} k^3$$

```
def produit_cubes(m, n):
    """int * int -> int
    Hypothèse : (0 <= m) and (m <= n)

    retourne le produit des cubes des entiers dans l'intervalle [m,n[."""

    # p : int
    p = 1   # le produit à calculer

    # k : int  (entier courant)
    for k in range(m, n):
        p = p * k * k * k
```

```python
    return p

produit_cubes(4, 8)
```

```python
# Jeu de tests
assert produit_cubes(1, 4) == 1 * 8 * 27
assert produit_cubes(13, 21) == f(13, 21)
assert produit_cubes(4, 8) == 592704000
```

Simulation de `produit_cubes(4, 8)` :

Tour de boucle	variable `k`	variable `p`
entrée	-	**1**
1er	4	64
2e	5	8000
3e	6	1728000
4e	7	592704000
sortie	-	**592704000**

Remarques :

- contrairement aux simulation de `while` la sortie est sur une ligne séparée.
- la variable d'itération contient la valeur courante de l'itération, tout se passe comme si l'affectation de `k` se faisait "avant" chaque tour de boucle.
- il est **interdit** de référence la variable d'itération après la boucle `for` donc on met un tiret en entrée et en sortie pour cette variable.

Exercice (5.2) : Fonction mystère (corrigé)

Le but de cet exercice est de réussir à déterminer ce que calcule une fonction mystère en simulant une boucle for.

Considérer la fonction «mystère» `f` suivante:

```python
def f(a):

    b = 0
    for c in a:
        if c >= '0' and c <= '9':
            b = b + 1

    return b
```

Question 1

Compléter la définition ci-dessus avec la signature de la fonction ainsi que les déclarations de variables.

Réponse

```python
def f(a):
    """
    str -> int
    """

    # b : int
    b = 0

    # c : str
    for c in a:
        if c >= '0' and c <= '9':
            b = b + 1

    return b
```

Question 2

Effectuer une simulation de boucle correspondant à l'évaluation de l'application `f('10 août')`

Calculer «à la main» les valeurs de `f` pour 'bonjour', 'un : 1', '606060'.

Réponse

Tour de boucle	variable `c`	variable `b`
entrée	-	0
1e	'1'	1
2e	'0'	2
3e	' '	2
4e	'a'	2
5e	'o'	2
6e	'û'	2
7e	't'	2
sortie	-	2

```python
>>> f('bonjour')
0
```

```python
>>> f('un : 1')
1
```

```python
>>> f('606060')
6
```

Question 3

Conjecturer ce que calcule `f`.

En déduire une définition complétée de la fonction, en lui trouvant un nom plus explicite.

Réponse

D'après la simulation précédente, la fonction `f` ajoute 1 à `b` à chaque fois que le caractère `c` est un chiffre. On conjecture donc que la fonction compte le nombre de chiffres dans la chaîne donnée en entrée.

```
def nb_chiffres(s):
    """ str -> int

    Renvoie le nombre de chiffres de s"""

    # nb : int
    nb = 0

    # c : str
    for c in s:
        if c >= '0' and c <= '9':
            nb = nb + 1

    return nb
```

```
# Jeu de tests
assert nb_chiffres('bonjour') == 0
assert nb_chiffres('12345') == 5
assert nb_chiffres('0') == 1
assert nb_chiffres('') == 0
assert nb_chiffres('a1b2c3ed4') == 4
```

Exercice (5.4) : Palindromes (corrigé)

Question 1

Donnez la spécification et une définition de la fonction `est_palindrome` telle que `est_palindrome(s)` retourne `True` si `s` est un palindrome, c'est-à-dire une chaîne qui est la même si on la lit de la gauche vers la droite ou de la droite vers la gauche.

Par exemple :

```
>>> est_palindrome('')
True
```

```
>>> est_palindrome('je ne suis pas un palindrome')
False
```

```
>>> est_palindrome('aba')
True
```

```
>>> est_palindrome('amanaplanacanalpanama')
True
```

Réponse

```
def est_palindrome(s):
    """str -> bool
```

```python
    retourne True si et seulement si s est un palindrome."""

    # i : int (indice courant)
    for i in range(0, len(s)):
        if s[i] != s[len(s)-i-1]:
            return False  # différence trouvée

    # La boucle s'est terminée sans qu'on trouve de différence
    # entre la lecture dans un sens et dans l'autre
    return True
```

```python
# Jeu de tests
assert est_palindrome('') == True
assert est_palindrome('je ne suis pas un palindrome') == False
assert est_palindrome('aaaa') == True
assert est_palindrome('aba') == True
assert est_palindrome('amanaplanacanalpanama') == True
```

Variante plus efficace (contrôle que la moitié de gauche correspond à la moité de droite)

```python
def est_palindrome_bis(s):
    """str -> bool

    retourne True si et seulement si s est un palindrome."""

    # i : int
    for i in range(0, len(s)//2):
        if s[i] != s[len(s)-i-1]:
            return False

    return True
```

```python
# Jeu de tests
assert est_palindrome_bis('')
assert not est_palindrome_bis('je ne suis pas un palindrome')
assert est_palindrome_bis('aaaa')
assert est_palindrome_bis('aba')
assert est_palindrome_bis('amanaplanacanalpanama')
```

Une variante de cette dernière fonction qui exploite l'utilisation des entiers négatifs pour l'accès aux éléments d'une chaîne de caractères (vu en cours) :

```python
def est_palindrome_ter(s):
    """str -> bool

    retourne True si et seulement si s est un palindrome."""

    # i : int
    for i in range(0,len(s)//2):
        if s[i] != s[-i-1]:
            return False
```

```
    return True
```

```
# Jeu de tests
assert est_palindrome_ter('')
assert not est_palindrome_ter('je ne suis pas un palindrome')
assert est_palindrome_ter('aaaa')
assert est_palindrome_ter('aba')
assert est_palindrome_ter('amanaplanacanalpanama')
```

Question 2

On se propose maintenant de définir une fonction de création automatique de palindromes.

Cette fonction `miroir` prend une chaîne de caractères en paramètre et retourne un palindrome à partir de cette chaîne, correspondant simplement au miroir de la chaîne.

Par exemple :

```
>>> miroir('abc')
'abccba'
```

```
>>> miroir('amanaplanacanal')
'amanaplanacanallanacanalpanama'
```

```
>>> miroir('do-re-mi-fa-sol')
'do-re-mi-fa-sollos-af-im-er-od'
```

Remarque : dans le jeu de tests on exploitera le fait que la chaîne miroir est un palindrome.

Réponse

```
def miroir(s):
    """str -> str

    retourne le palindrome miroir de la chaîne s."""

    # r : str
    r = ''   # la chaîne inversée

    # ch : str  (caractère courant)
    for ch in s:
        r = ch + r

    return s + r
```

```
# Jeu de tests
assert miroir('abc') == 'abccba'
assert est_palindrome(miroir('abc'))
assert est_palindrome(miroir('amanaplanacanal'))
assert est_palindrome(miroir('do-re-mi-fa-sol'))
```

Variante exploitant les indices négatifs dans les chaînes de caractères :

```python
def miroir_bis(s):
    """str -> str

    retourne le palindrome miroir de la chaîne s."""

    # r : str
    r = ''   # la chaîne inversée

    # i : int  (position du caractère courant)
    for i in range(1,len(s)+1):
        r = r + s[-i]

    return s + r
```

```python
# Jeu de tests
assert miroir_bis('abc') == 'abccba'
assert est_palindrome(miroir_bis('abc'))
assert est_palindrome(miroir_bis('amanaplanacanal'))
assert est_palindrome(miroir_bis('do-re-mi-fa-sol'))
```

Exercice (5.5) : Suppressions (corrigé)

Question 1

Donner une définition de la fonction `suppression` telle que `suppression(c, s)` supprime toutes les occurrences du caractère `c` dans la chaîne `s`.

Par exemple :

```python
>>> suppression('a','')
''
```

```python
>>> suppression('a','aaaaa')
''
```

```python
>>> suppression('p', 'le papa noel')
'le aa noel'
```

```python
>>> suppression('a', 'bbbbb')
'bbbbb'
```

Réponse

```python
def suppression(c, s):
    """str * str -> str
    Hypothèse : len(c) == 1

    retourne la chaîne s privée de toutes les occurrences du caractère c."""

    # res : str
```

```python
    res = '' # on accumulera le résultat dans res

    # d : str
    for d in s:
        if d != c:
            res = res + d
    return res
```

```python
# Jeu de tests
assert suppression('a','') == ''
assert suppression('a','aaaaa') == ''
assert suppression('p', 'le papa noel') == 'le aa noel'
assert suppression('a', 'bbbbb') == 'bbbbb'
```

Question 2

Donner une définition de la fonction `suppression_debut` telle que `suppression_debut(c,s)` supprime la *première* occurrence du caractère `c` dans la chaîne `s`.

Par exemple :

```python
>>> suppression_debut('a', '')
''
```

```python
>>> suppression_debut('a', 'aaaaa')
'aaaa'
```

```python
>>> suppression_debut('p', 'le papa noel')
'le apa noel'
```

```python
>>> suppression_debut('a', 'bbbbb')
'bbbbb'
```

Réponse

```python
def suppression_debut(c, s):
    """str * str -> str
    Hypothèse : len(c) == 1

    Retourne la chaîne s privée de la première occurrence du caractère c."""

    # premiere_trouvee : bool
    premiere_trouvee = False # indique si la première occurrence de c a été vue ou non

    # res : str
    res = '' # Contient le résultat

    # d : str
    for d in s:
        if d != c:
            res = res + d
        elif  not premiere_trouvee:
            premiere_trouvee = True
```

```python
        else:
            res = res + d

    return res
```

```python
# Jeu de tests
assert suppression_debut('a', '') == ''
assert suppression_debut('a', 'aaaaa') == 'aaaa'
assert suppression_debut('p', 'le papa noel') == 'le apa noel'
assert suppression_debut('a', 'bbbbb') == 'bbbbb'
```

Question 3

Donner la spécification et une définition de la fonction `suppression_derniere` telle que `suppression_derniere(c, s)` supprime la *dernière* occurrence du caractère `c` dans la chaîne `s`.

Par exemple :

```python
>>> suppression_derniere('a','')
''
```

```python
>>> suppression_derniere('a', 'aaaaa')
'aaaa'
```

```python
>>> suppression_derniere('p', 'le papa noel')
'le paa noel'
```

```python
>>> suppression_derniere('a', 'bbbbb')
'bbbbb'
```

Réponse

```python
def suppression_derniere(c, s):
    """str * str -> str
    Hypothèse : len(c) == 1

    retourne la chaîne s privée de la dernière occurrence du caractère c."""

    # derniere_trouvee : bool
    derniere_trouvee = False # indique si la dernière occurrence de c a été vue ou non

    # res : str
    res = '' # le résultat à retournere

    # i : int
    i = len(s) - 1 # indice du caractère courant, en partant de la fin
    while i >= 0:
        if s[i] != c:
            res = s[i] + res
        elif  not derniere_trouvee:
            derniere_trouvee = True
        else:
```

```
            res = s[i] + res

        i = i - 1

    return res
```

```
# Jeu de tests
assert suppression_derniere('a','') == ''
assert suppression_derniere('a', 'aaaaa') == 'aaaa'
assert suppression_derniere('p', 'le papa noel') == 'le paa noel'
assert suppression_derniere('a', 'bbbbb') == 'bbbbb'
```

6 Listes

6.1 Les listes

Nous avons introduit dans le cours précédent deux types de *séquences* :

— 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 cours, 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.

6.1.1 Définition et opérations de base

Définition : Une **liste** de type `list[`α`]` est une séquence dont tous les éléments sont du même type α.

Remarque : en remplacement des lettres grecques α, β, γ ... on pourra écrire en toutes lettres : `alpha`, `beta`, `gamma`, ...

Dans ce cours, nous utiliserons les lettres grecques dans le fil du texte et l'écriture en toutes lettres dans les programmes : spécifications de fonctions, déclarations de variables, etc.

6.1.1.1 Construction explicite

Une **expression atomique de liste** de type `list[`α`]` est soit :

— la *liste vide* notée `[]`
— une liste spécifique d'éléments notée `[`e_0 `,` e_1 `, ... ,` e_{n-1} `]` où chacun des e_i est une expression quelconque de type α.

Par exemple :

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

qui est une liste de type `list[int]`, tout comme :

```
>>> [1, 2, 3, 2*2, 4+1]
[1, 2, 3, 4, 5]
```

```
>>> ['chat', 'ours', 'pomme']
['chat', 'ours', 'pomme']
```

qui est une liste de type `list[str]`, ou encore :

```
>>> [True, False]
[True, False]
```

qui est une liste de type `list[bool]`

Pour chaque liste ainsi créée, il faut donc remplacer l'identifiant α générique par un type spécifique : `int`, `float`, `bool`, `str` (ou bien des combinaisons plus complexes, nous le verrons lors du prochain cours).

Le cas de la liste vide est un petit peu particulier puisque cette dernière est compatible avec tous les types d'élément : `[]` 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[`α`]` pour n'importe quel α.

Remarque : dans ce cours, nous manipulerons uniquement des *listes homogènes* qui ne contiennent que des éléments d'un même type α donné. Le langage Python ne fait pas la différence entre listes homogènes et listes hétérogènes, c'est-à-dire contenant des éléments de types indéterminés.

Par exemple :

```
>>> type([1, 2, 3, 4, 5])
list
```

```
>>> type(['chat', 'ours', 'pomme'])
list
```

```
>>> type([True, False])
list
```

```
>>> type([])
list
```

Dans chaque cas, Python répond «le type `list`» sans information sur le type des éléments contenus dans les listes. Cependant, nous verrons que la connaissance du type des éléments d'une liste est une information primordiale pour pouvoir résoudre les problèmes posés. En pratique, les listes ne sont jamais véritablement hétérogènes et le programmeur doit toujours comprendre le type des éléments qu'elles contiennent.

A retenir : c'est au programmeur (donc à *vous*) qu'incombe la responsabilité de garantir que les éléments contenus dans une liste de type `list[`α`]` soient bien tous du *même* type α.

6.1.1.2 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 `, ... ,` 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[alpha] -> int`

Par exemple :

`[1, 2, 3, 4, 5]` est de longueur 5

```
>>> len([1, 2, 3, 4, 5])
5
```

`["chat", "ours", "pomme"]` est de longueur 3

```
>>> len(["chat", "ours", "pomme"])
3
```

`[True,False]` est de longueur 2

```
>>> len([True, False])
2
```

Et bien sûr la liste vide est de longueur 0.

```
>>> len([])
0
```

6.1.1.3 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[`α`]`,

Remarque : par convention, nous utiliserons des identifiants commençant par une majuscule pour les listes, le plus souvent `L`, `L1`, `L2`, etc.

`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.

6.1.1.4 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[alpha] * list[alpha] -> list[alpha]`

Attention : on ne peut concaténer que des listes dont les éléments sont du même type `alpha`.

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 autorisait cette dernière concaténation, il ne serait pas possible de définir le type de la liste concaténée : il s'agirait d'une liste hétérogène **interdite** dans notre cours !

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 dans les deux premiers exemples ci-dessus la liste vide est considérée de type `list[int]` alors que dans le dernier cas on la considère plutôt de type `list[bool]`. Cette flexibilité concernant le type de la liste vide est nécessaire entre autre pour pouvoir la considérer comme élément neutre de la concaténation.

De façon générale, pour toute liste `L` on a les égalités suivantes :

```
L + [] == [] + L == L
```

6.1.1.5 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[`α`]` et `<élément>` une expression de type α.

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.

Remarque : cette classification en structures de données mutables ou immutables est de nature assez complexe. Dans ce cours d'introduction, en guise de simplification nous nous bornerons à utiliser la méthode `append` pour la problématique de construction des listes et nous n'aborderons pas les autres opérations de mutation de liste.

A titre d'illustration, nous allons définir une fonction `liste_pairs` permettant de construire la liste des entiers pairs dans l'intervalle $[1, n]$. La spécification de cette fonction est la suivante :

```python
def liste_pairs(n):
    """int -> list[int]
    Hypothèse: 1 <= n

    retourne 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 :

```python
def liste_pairs(n):
    """int -> list[int]
    Hypothèse: 1 <= n

    retourne la liste des entiers pairs dans l'intervalle [1,n]."""

    # L : list[int]
    L = []  # la liste résultat, initialement vide
```

```python
    # 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
```

```python
# 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]
```

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]`

6.1.1.6 Accès aux éléments

Comme pour les chaînes de caractères, les indices des éléments d'une liste `L` de type `list[`α`]` 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 α 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[`α`]` (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 α donc d'un type *différent* de la liste `L`

Ceci est une des raisons pour lesquelles les chaînes ne sont pas équivalentes à des listes de type `list[str]` (car chaque élément est de type `str` et non `list[str]`). L'autre raison (plus importante) de cette distinction est par souci d'efficacité. Ainsi les chaînes de caractères sont presque toujours fournies de façon spécifique dans les langages de programmation (il y a des contre-exemples comme le langage C et le langage *Haskell* notamment).

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]`

6.1.1.7 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[`α`]` permet de construire une nouvelle liste également de type `list[`α`]` et composée des éléments de `L` situés entre deux indices.

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 `Comptine` :

```python
# Comptine : list[str]
Comptine = ['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

```python
>>> Comptine[1:3]
['stram', 'gram']
```

```
>>> Comptine[3:4]
['pic']
```

Les expressions de découpage reconstruisent les listes. La liste de départ référencée par la variable `Comptine` est donc restée inchangée.

```
>>> Comptine
['am', 'stram', 'gram', 'pic', 'pic', 'col', 'gram']
```

Remarque : si $i \geq j$ alors `L[i:j]` renvoie la liste vide.

```
>>> Comptine[3:3]
[]
```

```
>>> Comptine[5:3]
[]
```

Cas particulier : reconstruction de la liste de départ.

On peut bien sûr reconstruire complètement la liste de départ.

```
>>> Comptine[0:len(Comptine)]    # autrement dit:  Comptine[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 :

```
>>> Comptine[:]
['am', 'stram', 'gram', 'pic', 'pic', 'col', 'gram']
```

6.1.1.7.1 Autres cas particuliers : liste des premiers éléments et liste des derniers éléments.

- `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]`.
- `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)]`.

```
>>> Comptine[:4]
['am', 'stram', 'gram', 'pic']
```

```
>>> Comptine[0:4]
['am', 'stram', 'gram', 'pic']
```

```
>>> Comptine[3:]
['pic', 'pic', 'col', 'gram']
```

```
>>> Comptine[3:len(Comptine)]
['pic', 'pic', 'col', 'gram']
```

6.1.1.7.2 Elément ou liste d'un seul élément ?

Attention : pour une liste `L` de type `list[`α`]` ne pas confondre

- `L[i]` qui renvoie un élément de type α
- `L[i:i+1]` qui renvoie une liste de type `list[`α`]` de longueur 1.

Par exemple :

```
>>> Comptine[3]
'pic'
```

```
>>> Comptine[3:4]
['pic']
```

6.1.1.7.3 Complément : 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 `Comptine` 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 :

- `Comptine[-4:-1]` renvoie la même liste que `Comptine[3:-1]` et donc que `Comptine[3:6]`.

```
>>> Comptine[-4:-1]
['pic', 'pic', 'col']
```

```
>>> Comptine[-4:-1] == Comptine[3:-1] == Comptine[3:6]
True
```

- `Comptine[-6:-2]` renvoie la même liste que `Comptine[1:-2]` et donc que `Comptine[1:5]`.

```
>>> Comptine[-6:-2]
['stram', 'gram', 'pic', 'pic']
```

```
>>> Comptine[-6:-2] == Comptine[1:-2] == Comptine[1:5]
True
```

Exercice : selon les mêmes principes, donner le résultat de l'évaluation des expressions suivantes :

— `Comptine[-5:-3]`
— `Comptine[-4:-4]`

6.1.1.7.4 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.

```
>>> Comptine[1:5:2]
['stram', 'pic']
```

Cas particulier : `L[i:j:1]` renvoie la même liste que `L[i:j]`

```
>>> Comptine[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.

6.1.1.7.5 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.

```
>>> Comptine[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 ;

```
>>> Comptine[6:2:-1]
['gram', 'col', 'pic', 'pic']
```

```
>>> Comptine[5:0:-1]
['col', 'pic', 'pic', 'gram', 'stram']
```

```
>>> Comptine[5::-1]
['col', 'pic', 'pic', 'gram', 'stram', 'am']
```

```
>>> Comptine[::-1]
['gram', 'col', 'pic', 'pic', 'gram', 'stram', 'am']
```

Exercice : Donner des découpages de `Comptine` 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.

6.2 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.

6.2.1 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.

6.2.1.1 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[Number] -> Number

    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[Number] -> Number

    Retourne la somme des éléments de la liste L."""

    # s : Number
    s = 0  # la somme cumulée des éléments
```

```python
    # n : Number  (élément courant)
    for n in L:
        s = s + n

    return s
```

```python
# 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
sortie	-	15

6.2.1.2 Exemple 2 : longueur de liste

Considérons un second exemple de réduction avec la fonction `longueur` qui calcule le nombre d'éléments d'une liste.

```python
def longueur(L):
    """list[alpha] -> int

    Retourne la longueur de la liste L."""

    # lng : int
    lng = 0  # comptage de la longueur initialement à zéro

    # e : alpha   (élément courant)
    for e in L:
        lng = lng + 1  # longueur incrémentée pour chaque élément

    return lng
```

Il s'agit ici d'une *réduction* vers le type `int`, ce qui correspond à la signature suivante :

`list[alpha] -> int`

Le `alpha` 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[alpha]` pour n'importe quel `alpha`. 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.

Attention : le type générique `alpha` doit être remplacé 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 l'identifiant α de la définition de la fonction "devient" `int` le temps de l'appel ci-dessus. Bien sûr, on peut l'appliquer sur une liste d'un type différent, par exemple `list[str]`

```
>>> longueur(['bla', 'bla', 'bla'])
3
```

Ici, l'identifiant α de la définition de `longueur` «devient» `str`, mais juste pendant la durée de cet appel spécifique.

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.

6.2.2 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[`α`]` de longueur n :

`[` e_0 `,` e_1 `,` $\ldots$ `,` e_{n-1} `]`

ainsi qu'une fonction `f` de signature α `->` β.

L'objectif est de construire la liste :

`[` $f(e_0)$ `,` $f(e_1)$ `,` $\ldots$ `,` $f(e_{n-1})$ `]` de type `list[`β`]` et de longueur n également.

Une transformation possède dans le cas général une signature de la forme :

`list[`α`] -> list[`β`]`

que l'on peut lire comme :

> Une transformation d'une liste de α vers une liste de β.

6.2.2.1 Exemple 1 : liste des carrés Pour illustrer les transformations de listes, considérons la spécification suivante :

```
def liste_carres(L):
    """list[Number] -> list[Number]

    retourne la liste des éléments de L élevés au carré."""
```

Ici 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 : $\alpha = \beta =$ `Number`

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):
    """Number -> Number

    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[Number] -> list[Number]

    Retourne la liste des éléments de L élevés au carré."""

    # LR : list[Number]
    LR = []  # liste résultat, initialement vide

    # x : Number (élément courant)
```

```python
    for x in L:
        LR.append(carre(x)) # ajoute le carré en fin de résultat


    return LR
```

```python
# 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 fonction dédiée.

6.2.2.2 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 :

```python
def liste_longueurs_chaines(L):
    """list[str] -> list[int]

    retourne la liste des longueurs des chaînes éléments de L."""
```

Par rapport au schéma général de transformation de `list[`α`]` vers `list[`β`]` on a ici un exemple où α et β sont distincts avec $\alpha =$ `str` et $\beta =$ `int`.

Par exemple :

```
>>> liste_longueurs_chaines(['e', 'ee', 'eee', 'eeee'])
[1, 2, 3, 4]
```

```
>>> liste_longueurs_chaines(['un', 'deux', 'trois', 'quatre'])
[2, 4, 5, 6]
```

```
>>> liste_longueurs_chaines([])
[]
```

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_chaines(L):
    """list[str] -> list[int]

    retourne la liste des longueurs des chaînes éléments de L."""

    # LR : list[int]
    LR = []  # liste résultat, initialement vide

    # s : str (élément courant)
    for s in L:
        LR.append(len(s)) # ajoute la longueur en fin de résultat


    return LR
```

```
# Jeu de tests
assert liste_longueurs_chaines(['e', 'ee', 'eee', 'eeee']) == [1, 2, 3, 4]
assert liste_longueurs_chaines(['un', 'deux', 'trois', 'quatre']) \
       == [2, 4, 5, 6]
assert liste_longueurs_chaines([]) == []
```

6.2.3 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[`α`]` de longueur n :

`[` a_0 `,` a_1 `, ... ,` a_{n-1} `]`

ainsi qu'un prédicat `p` de signature α `-> bool`.

L'objectif est de construire la liste également de type `list[`α`]` :

`[` b_0 `,` b_1 `, ... ,` b_{m-1} `]` de longueur $m \leq n$ avec :

— pour chaque b_k il existe un unique a_i tels que $b_k = a_i$ et $p(a_i)$ vaut `True` avec $k \leq i$
— et pour tout $l > k$, on vérifie $b_l = a_j$ implique $i < j$.

... 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[`α`] -> list[`α`]`

que l'on peut lire comme :

> un filtrage d'une liste de α.

6.2.3.1 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."""

    # LR : list[int]
    LR = []  # la liste filtrée, initialement vide
```

```python
    # n : int (élément courant)
    for n in L:
        if est_pair(n):
            LR.append(n)
        # sinon on ne fait rien

    return LR
```

```python
# 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`.

6.2.3.2 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 :

```python
def liste_superieurs(L, x):
    """list[Number] * Number -> list[Number]

    renvoie la liste des nombres éléments de L supérieurs
    au nombre x."""

    # LR : list[Number]
    LR = []  # la liste résultat initialement vide

    # y : Number (élément 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 `Number`.

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.

Cependant, la signature n'est pas `list[Number] -> list[Number]` car la fonction prend en entrée un second argument. C'est donc un filtrage un peu plus complexe mais qui reste bien dans la catégorie *filtrage de listes*.

6.2.3.2.1 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, x):
    """list[Number] * Number -> list[Number]

    renvoie la liste des nombres éléments de L supérieurs
    au nombre x."""

    # Voici la définition interne :
    def superieur_a_x(z):
        """Number -> bool

        Renvoie True si z est supérieur à x, False sinon."""
        return z > x

    # et maintenant le reste du corps de la fonction principale

    # LR : list[Number]
```

```python
    LR = []  # la liste résultat initialement vide

    # y : Number (élément nombre courant)
    for y in L:
        if superieur_a_x(y):
            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. Nous reviendrons sur ce point lors des prochains cours (toujours en guise de complément).

6.2.4 Autres problèmes

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.

6.2.4.1 Exemple 1 : liste des sommes cumulées

Certains problèmes assez simples sortent tout de même de notre classification. Le calcul de la liste des sommes cumulées est un exemple intéressant.

La spécification du problème est la suivante :

```python
def liste_sommes_cumulees(L):
    """list[Number] -> list[Number]

    retourne la liste des sommes cumulées de L."""
```

Quelques exemples permettent de comprendre le problème posé ici :

```python
>>> liste_sommes_cumulees([])
[]
```

```python
>>> liste_sommes_cumulees([42])
[42]
```

```python
>>> liste_sommes_cumulees([1, 2, 3, 4, 5])
[1, 3, 6, 10, 15]
```

```python
>>> 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[Number] -> list[Number]
```

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 :

```python
def liste_sommes_cumulees(L):
    """list[Number] -> list[Number]

    retourne la liste des sommes cumulées de L."""

    # LS : list[Number]
    LS = []  # liste des sommes cumulées en résultat

    # s : Number
    s = 0   #  somme cumulée

    # n : Number  (élément courant)
    for n in L:
        s = s + n
        LS.append(s)

    return LS
```

```python
# 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 ?

6.2.4.2 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 :

```python
def interclassement(L1, L2):
    """list[alpha] * list[alpha] -> list[alpha]
```

```
    Hypothèse : les éléments des listes L1 et L2 sont triés par ordre croissant.

    renvoie la liste triée résultant de l'interclassement de L1 et L2."""
```

Un point important dans la spécification est que l'on considère en hypothèse que les deux listes `L1` et `L2` sont triées (pour l'ordre du comparateur $<$ sur les éléments) et que leurs éléments sont du même type α. 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 dans le cours correspondant).

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.

Mais il est clair que comprendre comment imaginer et exprimer un tel algorithme demande de la pratique, d'où la nécessité de s'entraîner avec les nombreux exercices proposés en complément du cours.

Une fois que l'algorithme a été élucidé, nous pouvons en produire une version plus formelle dans le langage Python.

```python
def interclassement(L1, L2):
    """list[alpha] * list[alpha] -> list[alpha]
    Hypothèse : les éléments des listes L1 et L2 sont triés par ordre croissant.

    renvoie la liste triée résultant de l'interclassement de L1 et L2."""

    # i1 : int
    i1 = 0  #  indice de parcours de la liste L1
    # i2 : int
    i2 = 0  # indice de parcours de la liste L2

    # LR : list[alpha]
    LR = [] # liste résultat de l'interclassement

    while (i1 < len(L1)) and (i2< len(L2)):
        if L1[i1] <= L2[i2]:
```

```python
            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
```

Pour cette fonction, il est clair que notre jeu de tests devient primordial.

```python
# 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']
```

Et finalement, une simulation s'impose. 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]`.

Pour la boucle on obtient :

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.

Ceci clôt l'analyse de notre première fonction non-triviale (tout du moins en cours).

6.3 Exercices corrigés

Exercice (6.1) : Listes de répétitions (corrigé)

Dans cet exercice introductif, on s'intéresse à la construction de listes par répétition d'un élément ou d'une liste d'éléments.

Question 1

Donner une définition de la fonction `repetition` qui, étant donné un élément x et un entier naturel k, renvoie la liste contenant k occurrences de x.

Par exemple :

```
>>> repetition("thon", 4)
['thon', 'thon', 'thon', 'thon']
```

```
>>> repetition(3, 8)
[3, 3, 3, 3, 3, 3, 3, 3]
```

```
>>> repetition(5, 0)
[]
```

```
>>> repetition([1, 2, 3], 5)
[[1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3]]
```

Réponse

```
def repetition(x, k):
    """alpha * int -> list[alpha]
    Hypothèse: k >= 0

    retourne la liste composée de k occurrences de x."""

    # L : list[alpha]
```

```
    L = [] # la liste calculée

    # i : int (entier courant)
    for i in range(0, k):
        L.append(x)

    return L
```

```
# Jeu de tests
assert repetition(0, 4) == [0, 0, 0, 0]
assert repetition(4, 0) == []
assert repetition('pom', 5) == ['pom', 'pom', 'pom', 'pom', 'pom']
```

Question 2

Donner une définition de la fonction `repetition_bloc` qui, étant donné une liste L et un entier naturel k, renvoie la liste obtenue en concaténant k fois la liste L.

Par exemple :

```
>>> repetition_bloc(["chat", "thon", "loup"], 3)
['chat', 'thon', 'loup', 'chat', 'thon', 'loup', 'chat', 'thon', 'loup']
```

```
>>> repetition_bloc([1, 2, 3], 5)
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> repetition_bloc([1, 2, 3, 4, 5], 0)
[]
```

Réponse

```
def repetition_bloc(L, k):
    """list[alpha] * int -> list[alpha]
    Hypothèse: 0 <= k

    retourne la liste composée de k répétitions de la liste L."""

    # res : list[alpha]
    res = [] # la liste calculée

    # i : int (entier courant)
    for i in range(k):
        res = res + L

    return res
```

```
# Jeu de tests
assert repetition_bloc([1, 2], 4) == [1, 2, 1, 2, 1, 2, 1, 2]
assert repetition_bloc([4, 5, 2, 3], 0) == []
assert repetition_bloc(['pim', 'pam', 'poum'], 3)  ==  ['pim', 'pam', 'poum', \
                                                         'pim', 'pam', 'poum', \
                                                         'pim', 'pam', 'poum']
```

Remarque : en Python l'opérateur * permet de construire des listes de répétitions. Par exemple :

```
>>> ["thon"] * 4

['thon', 'thon', 'thon', 'thon']
```

```
>>> 8 * [3]

[3, 3, 3, 3, 3, 3, 3, 3]
```

```
>>> ["chat", "thon", "loup"] * 3

['chat', 'thon', 'loup', 'chat', 'thon', 'loup', 'chat', 'thon', 'loup']
```

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

En TD et TME on pourra utiliser cet opérateur pour éviter de définir 36 fois `repetition` ou `repetition_bloc` mais il ne figure pas dans la carte de référence et on ne pourra donc pas le supposer connu dans les contrôles.

Exercice (6.2) : Maximum d'une liste (corrigé)

Cet exercice propose quelques problèmes de réduction associés à la notion de maximum.

Question 1

Donner une définition de la fonction `max_liste` qui, étant donné une liste non vide de nombres, renvoie le plus grand élément de cette liste.

Par exemple :

```
>>> max_liste([3, 7, 9, 5.4, 8.9, 9, 8.999, 5])
9
```

Réponse

```
def max_liste(L):
    """list[Number] -> Number
    Hypothèse :  len(L) > 0

    retourne le plus grand élément de la liste L."""

    # res : Number
    res = L[0] # la valeur calculée

    # e : Number (élément courant)
    for e in L[1:]:
        if e > res:
            res = e
```

```python
    return res
```

```python
# Jeu de tests
assert max_liste([3, 7, 9, 5.4, 8.9, 9, 8.999, 5]) == 9
assert max_liste([-2, -1, -5, -3, -1, -4, -1]) == -1
```

Question 2

Donner une définition de la fonction `nb_occurrences` qui, étant donné une liste L et un élément x, renvoie le nombre d'occurrences de x dans L.

Par exemple :

```python
>>> nb_occurrences([3, 7, 9, 5.4, 8.9, 9, 8.999, 5], 9)
2
```

```python
>>> nb_occurrences(["chat", "ours", "chat", "chat", "loup"], "chat")
3
```

```python
>>> nb_occurrences(["chat", "ours", "chat", "chat", "loup"], "ou")
0
```

Réponse

```python
def nb_occurrences(L, x):
    """list[alpha] * alpha -> int

    retourne le nombre d'occurrences de x dans L."""

    # res : int
    res = 0 # (la valeur calculée)

    # e : alpha (élément courant)
    for e in L:
        if e == x:
            res = res + 1

    return res
```

```python
# Jeu de tests
assert nb_occurrences([3, 7, 9, 5.4, 8.9, 9, 8.999, 5], 9) == 2
assert nb_occurrences(["chat", "ours", "chat", "chat", "loup"], "chat") == 3
assert nb_occurrences(["chat", "ours", "chat", "chat", "loup"], "ou") == 0
```

Question 3

Donner une définition de la fonction `nb_max` qui, étant donné une liste non vide de nombres L, renvoie le nombre d'occurrences du maximum de L dans L.

Par exemple :

```python
>>> nb_max([3, 7, 9, 5.4, 8.9, 9, 8.999, 5])
2
```

```
>>> nb_max([-2, -1, -5, -3, -1, -4, -1])
3
```

Réponse

```python
# la solution "triviale" est d'utiliser les 2 fonctions précédentes:
def nb_max(L):
    """list[Number] -> int
    Hypothèse : len(L) > 0

    retourne le nombre d'occurrences du maximum de L dans L."""

    return nb_occurrences(L, max_liste(L))

# mais cette solution est coûteuse ! elle parcours 2 fois la liste...

# L'exercice 2 du thème 7 permet de revenir sur ce point et de
# proposer une version optimale de cette fonction.
```

```python
# Jeu de tests
assert nb_max([3, 7, 9, 5.4, 8.9, 9, 8.999]) == 2
assert nb_max([-2, -1, -5, -3, -1, -4, -1]) == 3
```

Exercice (6.4) : Liste de diviseurs (corrigé)

On construit dans cet exercice des listes d'entiers vérifiant certains prédicats (divisibilité, parité, imparité).

Question 1

Donner une définition de la fonction `liste_diviseurs` qui, étant donné un entier naturel non nul a, retourne la liste des entiers naturels qui sont diviseurs de a.

Par exemple :

```
>>> liste_diviseurs(18)
[1, 2, 3, 6, 9, 18]
```

Réponse

```python
def liste_diviseurs(a):
    """int -> list[int]
    Hypothèse : a > 0

    retourne la liste des diviseurs de a."""

    # L : list[int]
    L = [] # la liste calculée

    # i : int (entier courant)
```

```python
    for i in range(1, a + 1):
        if a % i == 0:
            L.append(i)
    return L
```

```python
# Jeu de tests
assert liste_diviseurs(2) == [1, 2]
assert liste_diviseurs(12) == [1, 2, 3, 4, 6, 12]
assert liste_diviseurs(25) == [1, 5, 25]
```

Question 2

Donner une définition de la fonction `liste_diviseurs_impairs` qui, étant donné un entier naturel non nul a, retourne la liste des entiers naturels impairs qui sont diviseurs de a.

Par exemple :

```python
>>> liste_diviseurs_impairs(24)
[1, 3]
```

```python
>>> liste_diviseurs_impairs(8)
[1]
```

```python
>>> liste_diviseurs_impairs(15)
[1, 3, 5, 15]
```

Réponse

```python
def liste_diviseurs_impairs(a):
    """int -> list[int]

    retourne la liste des diviseurs impairs de a
    hypothèse : a > 0."""

    # L : list[int]
    L = [] # la liste calculée

    # i : int
    i = 1 # candidat diviseur impair
          # 1 étant le plus petit possible.

    while i < a + 1:
        if a % i == 0:
            L.append(i)

        i = i + 2

    return L
```

```python
# Jeu de tests
assert liste_diviseurs_impairs(2) == [1]
assert liste_diviseurs_impairs(12) == [1, 3]
assert liste_diviseurs_impairs(30) == [1, 3, 5, 15]
```

```python
assert liste_diviseurs_impairs(15) == [1, 3, 5, 15]
```

Question 3

Donner une définition de la fonction `liste_diviseurs_pairs` qui, étant donné un entier naturel non nul a, retourne la liste des entiers naturels pairs qui sont diviseurs de a.

Par exemple :

```python
>>> liste_diviseurs_pairs(24)
[2, 4, 6, 8, 12, 24]
```

```python
>>> liste_diviseurs_pairs(7)
[]
```

```python
>>> liste_diviseurs_pairs(8)
[2, 4, 8]
```

Réponse

```python
def liste_diviseurs_pairs(a):
    """int -> list[int]

    retourne la liste des diviseurs pairs de a
    hypothèse : a > 0."""

    # L : list[int]
    L = [] # la liste calculée

    # i : int
    i = 2 # candidat diviseur pair
          # 2 étant le plus petit possible.

    if a % 2 == 1:
        return L
    else:
        while i < a + 1:
            if a % i == 0:
                L.append(i)

            i = i + 2

        return L
```

```python
# Jeu de tests
assert liste_diviseurs_pairs(2) == [2]
assert liste_diviseurs_pairs(12) == [2, 4, 6, 12]
assert liste_diviseurs_pairs(30) == [2, 6, 10, 30]
assert liste_diviseurs_pairs(15) == []
```

Exercice (6.5) : Fonction mystère (corrigé)

Déterminer ce que fait une fonction, sans en connaître ni le nom, ni la spécification mais simplement en étudiant son implémentation. Parcours de listes par indices, avec sortie anticipée ou pas.

Question 1

Soit la fonction «mystère» `f` ci-dessous :

```
def f(L):
    """
    ??? mystère !"""

    if (len(L) == 0) or (len(L) == 1):
        return True
    else:
        # i : ?
        for i in range(len(L) - 1):
            if L[i] >= L[i + 1]:
                return False
    return True
```

Compléter cette définition en donnant la signature de la fonction ainsi que le type à déclarer pour la variable `i`.

Réponse

```
def f(L):
    """list[Number] -> bool
    ??? mystère !"""

    if (len(L) == 0) or (len(L) == 1):
        return True
    else:
        # i : int (entier courant)
        for i in range(len(L) - 1):
            if L[i] >= L[i + 1]:
                return False
    return True
```

Question 2

Selon les principes vus en cours, effectuer une *simulation de boucle* correspondant à l'évaluation de l'application :

```
f([3, 5, 7, 10])
```

Dans cette simulation on montrera aussi les valeurs prises par `L[i]` et `L[i + 1]`.

Quelle est la valeur retournée par cette application ?

Mêmes questions pour :

```
f([3, 15, 7, 10])
```

Réponse

Simulation de boucle pour `f([3, 5, 7, 10])`:

tour de boucle	variable `i`	`L[i]`	`L[i+1]`
entrée	**0**	**3**	**5**
1er tour	1	5	7
2e	2	7	10
sortie	-	-	-

Sortie « normale » de boucle. Instruction suivante : `return True`. La valeur renvoyée est `True`.

Simulation de boucle pour `f([3, 15, 7, 10])`:

tour de boucle	variable `i`	`L[i]`	`L[i+1]`
entrée	**0**	**3**	**15**
1er tour	1	15	7
2e	2	7	10
sortie anticipée	-	-	-

Sortie « anticipée » de boucle. La valeur renvoyée est `False`.

Question 3

En déduire une définition complète et plus lisible de cette fonction, en particulier :

- proposer un nom plus pertinent pour la fonction
- compléter la description de la fonction
- proposer un jeu de tests pour valider la fonction.

Réponse

```
def est_croissante(L):
    """list[Number] -> bool

    retourne True si L est rangée en ordre strictement croissant et False sinon ;
    retourne True si L est vide ou si L n'a qu'un seul élément."""

    if (len(L) == 0) or (len(L) == 1):
        return True
    else:
        # i : int (entier courant)
        for i in range(len(L) - 1):
            if L[i] >= L[i + 1]:
                return False
    return True
```

```python
# Jeu de tests
assert est_croissante([1, 3, 4, 7, 8, 11, 13]) == True
assert est_croissante([1, 3, 4, 7, 8, 11, 9]) == False
assert est_croissante([1, 3, 4, 2, 5, 6]) == False
assert est_croissante([1, 3, 3, 4, 5, 6]) == False
assert est_croissante([]) == True
assert est_croissante([5]) == True
```

Question 4

Écrire une autre définition de cette fonction, en utilisant une instruction `while` avec une sortie anticipée de boucle et non une sortie anticipée de fonction, sans *sans* utiliser `return` dans le corps de la boucle.

Réponse

```python
def est_croissante_bis(L):
    """list[Number] -> bool

    retourne True si L est rangée en ordre strictement croissant et False sinon ;
    retourne True si L est vide ou si L n'a qu'un seul élément."""

    # i : int (entier courant)
    i = 0
    # b : bool
    b = True # la valeur calculée
    while (i < len(L) - 1) and b:
        b = L[i] < L[i + 1]
        i = i + 1
    return b
```

```python
# Jeu de tests
assert est_croissante_bis([1, 3, 4, 7, 8, 11, 13]) == True
assert est_croissante_bis([1, 3, 4, 7, 8, 11, 9]) == False
assert est_croissante_bis([1, 3, 4, 2, 5, 6]) == False
assert est_croissante_bis([1, 3, 3, 4, 5, 6]) == False
assert est_croissante_bis([]) == True
assert est_croissante_bis([5]) == True
```

7 N-uplets et décomposition de problèmes

7.1 Les n-uplets

Les séquences ne sont pas les seules données structurées disponibles dans le langage Python.

Une autre structure de données courante dans les langages de programmation et disponible en python est le **n-uplet** (ou *tuple* en anglais).

Définition : Pour un n fixé avec $n \geq 2$, un **n-uplet** est une donnée $(e_1, e_2, \ldots, e_n)$ composée d'exactement n éléments pouvant être chacun d'un type différent : e_1 de type T_1, e_2 de type T_2, $\ldots$, e_n de type T_n.

Le type d'un n-uplet est le *produit cartésien* $T_1 \times T_2 \times \ldots \times T_n$ que l'on notera :

`tuple[`T_1`,`T_2`,...,`T_n`]`

Remarques :

- un n-uplet contenant 2 éléments est appelé **couple**
- un n-uplet contenant 3 éléments est appelé **triplet**
- un n-uplet contenant 4 éléments est appelé **quadruplet**
- un n-uplet contenant 5 éléments est appelé **quintuplet**
- un n-uplet contenant 6 éléments est appelé **sextuplet**
- un n-uplet contenant 7 éléments est appelé 7-**uplet**
- un n-uplet contenant 8 éléments est appelé 8-**uplet**
- etc.

7.1.1 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')
```

On a construit ci-dessus 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]]`

Remarque : Le langage Python ne retient que le type `tuple` pour les n-uplets, sans information supplémentaire sur le type des éléments contenus.

Par exemple :

```
>>> type((1, True, 'blion'))
tuple
```

```
>>> type((1, 'deux', 3.0, 'quatre', 5))
tuple
```

```
>>> type((0, True, 2.0, 'trois', [4, 5], (6, 7.3)))
tuple
```

Important : c'est donc au programmeur de faire attention à ce que tout n-uplet t de type :

`tuple[`T_1`,`T_2`,`$\ldots$`,`T_n`]`

contienne bien *exactement* n éléments avec :

— un premier élément de type T_1
— un second élément de type T_2
— $\ldots$
— un n-ième élément de type T_n

7.1.2 Déconstruction

La *déconstruction* d'un n-uplet consiste à récupérer les différents éléments qu'il contient en les stockant dans des variables dites **variables de déconstruction**.

Une **instruction de déconstruction d'un n-uplet** t de type `tuple[`T_1`,`T_2`,`$\ldots$`,`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
— $\ldots$
— 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 obligatoire 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]
t = (1, True, 'blion')

n, b, s = t # déconstruction
```

D'après le type du n-uplet : `tuple[int, bool, str]` on sait que :

- la variable `n` est de type `int`

```
>>> n
1
```

- la variable `b` est de type `bool`

```
>>> b
True
```

- la variable `s` est de type `str`

```
>>> s
'blion'
```

Comme indiqué précédemment, il est parfois utile de préciser le type et le rôle des variables de déconstruction. On peut reprendre notre exemple de la façon suivante :

```
# t : tuple[int, bool, str]
t = (1, True, 'blion')

# n : int  (explication optionnelle)
# b : bool
# s : str
n, b, s = t # déconstruction
```

7.2 Utilisation des n-uplets

Les n-uplets servent à regrouper un nombre fixé d'éléments qui, une fois regroupés, forment une *entité*.

Dans ce cours, nous prendrons essentiellement deux exemples :

- une entité *point du plan* qui regroupe 2 éléments tous deux de type `Number`. Un point du plan est donc de type `tuple[Number, Number]`.
- une entité *enregistrement de personne* (dans une base de données) qui regroupe 4 éléments : un nom et un prénom de type `str`, un âge de type `int` et un statut marital (marié ou non) de type `bool`. Une personne est donc de type `tuple[str, str, int, bool]`.

7.2.1 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.

7.2.1.1 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 :

```
import math # pour math.sqrt
```

```
def distance(p1, p2):
    """tuple[Number, Number] * tuple[Number, Number] -> float

    retourne la distance entre les points p1 et p2."""

    # x1 : Number (abscisse de P1)
    # y1 : Number (ordonnée de P1)
    x1, y1 = p1

    # x2 : Number (abscisse de P2)
    # y2 : Number (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** des noms au type de n-uplets sur lesquels on travaille.

```
# type Point = tuple[Number, Number]
```

Ici on a déclaré un alias nommé `Point` pour le type `tuple[Number, Number]`. Donc pour tout exercice sur les points du plan, on peut utiliser `Point` comme identifiant de type plutôt que `tuple[Number, Number]`. Mais on se rappelle que c'est simplement un raccourci d'écriture, le «vrai» type des points du plan est bien `tuple[Number, Number]`.

Après cette déclaration du type Point (en dehors des fonctions du fichier courant) on peut proposer une spécification plus concise pour la fonction `distance` (l'implémentation et les jeux de test ne changent pas).

```python
def distance(p1, p2):
    """Point * Point -> float

    retourne la distance entre les points p1 et p2."""
```

7.2.1.2 Exemple 2 : enregistrement de personnes

L'entité personne correspond à une information de type : `tuple[str, str, int, bool]`.

Pour simplifier les déclarations de type, nous allons commencer par déclarer un *alias de type* :

```python
# type Personne = tuple[str, str, int, bool]
```

On peut maintenant dans les spécifications utiliser l'alias de type, c'est-à-dire écrire `Personne` plutôt que `tuple[str, str, int, bool]`. On y gagne ici clairement tant en terme de place qu'en facilité de compréhension.

Considérons par exemple la fonction suivante :

```python
def est_majeure(p):
    """Personne -> bool

    renvoie True si la personne est majeure, ou False sinon."""

    nom, pre, age, mar = p

    return age >= 18
```

```python
# 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`

7.2.2 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.

7.2.2.1 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`.

```python
def somme_et_moyenne(L):
    """list[Number] -> tuple[Number, float]
    Hypothèse : len(L) > 0

    retourne un couple formé de la somme et de la moyenne
    des éléments de la liste L."""

    # s : Number
    s = 0  # somme

    # x : Number (élément courant)
    for x in L:
        s = s + x

    return (s, (s / len(L)))
```

```python
# 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)
```

7.2.2.2 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.

```python
def mariage(p):
    """Personne -> Personne

    Retourne la personne avec un statut marital validé."""

    nom, pre, age, mar = p

    return (nom, pre, age, True)
```

```python
# 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.

7.2.3 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 cours et les listes vues lors du cours précédent.

Nous considérons respectivement des exemples de réduction, de transformation et de filtrage de listes de n-uplets.

7.2.3.1 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.

7.2.3.1.1 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 :

```python
def age_moyen(L):
    """list[Personne] -> float
    Hypothèse : len(L) > 0

    renvoie l'age moyen des personne enregistrées dans la liste L."""

    # age_total : int
    age_total = 0  # age cummulé

    # pers : Personne   (ou  tuple[str, str, int, bool])
    for pers in L:
        # age :int (age de la personne)
        nom, pre, age, mar = pers

        age_total = age_total + age

    return age_total / len(L)
```

```python
# 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
```

7.2.3.2 Variation : 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`.

```python
def age_moyen(L):
    """list[Personne] -> float
    Hypothèse : len(L) > 0

    renvoie l'age moyen des personne enregistrées dans la liste L."""

    # age_total : int
    age_total = 0  # age cummulé

    # age : int (age de la personne)
    for (nom, pre, age, mar) in L:
        age_total = age_total + age

    return age_total / len(L)
```

Remarque : on a uniquement déclaré le type de la variable de déconstruction `age` puisque c'est la seule dont on a besoin pour calculer l'âge moyen.

7.2.3.3 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.

```python
def extraction_noms(L):
    """list[Personne] -> list[str]

    Retourne la liste des noms des personnes de L."""

    # LR : list[str]
    LR = []

    for (nom, pre, age, mar) in L:
        LR.append(nom)

    return LR
```

```python
# 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([]) == []
```

7.2.3.4 Filtrages

Pour illustrer le filtrage des listes de n-uplets, on considère le filtrage de la base de données pour ne conserver que les personnes majeures.

```python
def personnes_majeures(L):
    """list[Personne] -> list[Personne]

    retourne la liste des personnes majeures dans la base L."""

    # LR : list[Personne]
    LR = []

    # p : Personne
    for p in L:
        if est_majeure(p):
            LR.append(p)

    return LR
```

```python
# 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)]
```

```python
assert personnes_majeures([('Un', 'Una', 1, True)]) == []
assert personnes_majeures([]) == []
```

7.3 Décomposition de problème

Dans cette dernière partie du cours, nous nous intéressons à un aspect méthodologique de la programmation : la **décomposition de problème**.

7.3.1 Maîtrise de la complexité

La plupart des problèmes que l'on cherche à résoudre en informatique sont de nature complexe. Or, dans ce cours, on tente d'identifier problèmes et fonctions. Notre *manifeste* est le suivant :

> un problème posé = une fonction Python pour le résoudre

La solution d'un problème complexe est forcément complexe. Cependant, notre objectif est de maîtriser cette complexité en précisant un peu notre *manifeste* :

> un problème complexe posé = une fonction Python simple pour le résoudre.

La question est alors la suivante :

> Comment peut-on proposer une fonction simple pour un problème complexe ?

Notre principal élément de réponse est méthodologique :

— il faut décomposer le problème complexe en un certain nombre de sous-problèmes plus simples
— chaque sous-problème simple peut être résolu par une fonction Python simple
— la solution du problème complexe est une fonction qui enchaîne simplement les solutions aux sous-problèmes.

Cette démarche se nomme la **décomposition de problèmes** et comme toute activité intellectuelle, elle fait appel à des facultés liées à notre intelligence et notamment : l'*exprérience* ainsi qu'une certaine dose de *créativité*.

Il n'est donc pas facile d'appliquer cette méthodologie, et la première chose à faire est simplement d'acquérir de l'expérience.

7.3.2 Exemple : le triangle de Pascal

Dans la plupart des exercices, la démarche de décomposition a été appliquée pour diviser l'énoncé en questions. Dans certains de ces exercices, chaque question correspond à un sous-problème et l'enchaînement des questions permet de résoudre un problème plus complexe.

7.3.2.1 Enoncé du problème

Pour ce cours, nous allons illustrer la démarche de décomposition sur un problème dont l'énoncé est le suivant.

Problème : on souhaite construire les n premières lignes du *triangle de Pascal* qui est une présentation des coefficients binomiaux $\binom{n}{k}$ (notés aussi C_n^k) sous la forme d'un triangle d'entiers naturels.

Les premières lignes du triangle de Pascal sont les suivantes :

```
           1
       1      1
     1    2     1
   1   3      3   1
 1   4    6     4   1
1   5   10    10   5   1
   ...      ...      ...
```

Au niveau de ce cours, on peut dire que ce problème, ainsi posé, est complexe puisqu'il n'est pas facile de le comprendre sans précision supplémentaire.

7.3.2.2 Représentation des données

Tout d'abord se pose le problème de la représentation des données du problème. Dans notre cas précis se pose le problème de la représentation d'un triangle de nombre.

Une façon assez naturelle de représenter le triangle est d'utiliser *une liste de listes d'entiers*, donc le type `list[list[int]]`, ainsi :

- chaque élément de la liste représentant le triangle est une *ligne* du triangle
- chaque *ligne* du triangle est une liste d'entiers.

Ce type complexe témoigne également de la nature complexe du problème posé.

Essayons d'encoder les premières lignes du triangle de Pascal avec le type `list[list[int]]`.

```
[[1],
 [1, 1],
 [1, 2, 1],
 [1, 3, 3, 1],
 [1, 4, 6, 4, 1],
 [1, 5, 10, 10, 5, 1]]
```

Avec des sauts de lignes bien choisis on peut se convaincre qu'il s'agit bien de l'encodage d'un triangle d'entiers.

7.3.2.3 Décomposition en sous-problèmes

Maintenant que nous savons comment représenter le triangle de Pascal, nous pouvons commencer à décomposer le problème complexe en sous-problèmes plus simples.

Important : il n'existe pas *une unique* manière de décomposer un problème complexe en sous-problèmes. Nous illustrons ici une des nombreuses approches possibles.

Pour cela, considérons en tant que sous-problème le passage d'une ligne à l'autre dans le triangle.

Question : pourquoi s'intéresse-t-on à ce phénomène ?

La réponse est assez subjective : on «sent», par expérience et sûrement avec un peu de créativité, qu'il s'agit d'une bonne piste pour la résolution du problème.

Observons notamment le passage de la ligne :

```
...         ...
  1    4     6     4    1
```

à la ligne :

```
1    5    10    10    5    1
   ...       ...       ...
```

Sous forme de listes, cela donne :

```
[1, 4, 6, 4, 1]
```

vers :

```
[1, 5, 10, 10, 5, 1]
```

Reproduisons la ligne de départ en opérant un petit décalage :

```
 1    4    6    4    1

      1    4    6    4    1
```

Pour répondre à la question :

> Pourquoi effectue-t-on cette juxtaposition ?

Cela reste toujours très subjectif : disons que cela semble un moyen judicieux d'obtenir des 5 et des 10 nécessaires pour la ligne suivante. En effet :

```
    1    4    6    4    1    0
 +
    0    1    4    6    4    1
    =====================
    1    5   10   10    5    1
```

Il s'agit du résultat attendu pour la ligne suivante !

Remarque : cette propriété assez géométrique peut être justifiée mathématiquement à partir de la récurrence : $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$. Mais nous nous limiterons à résoudre le problème sans justifier la correction de la solution.

On peut maintenant décomposer le processus permettant de passer d'une ligne à la ligne suivante, cette fois-ci en considérant des listes Python :

A partir de `[1, 4, 6, 4, 1]`

1. on décale la ligne vers la gauche en injectant un 0 à droite. On obtient :
   [1, 4, 6, 4, 1, 0]
2. on décale la ligne de départ cette fois-ci vers la droite. On obtient :
   [0, 1, 4, 6, 4, 1]
3. pour obtenir la ligne suivante, on réalise maintenant simplement une addition des deux lignes décalées à gauche et à droite :
   [1, 5, 10, 10, 5, 1]

Cette dernière liste est bien la ligne suivante recherchée.

On a donc décomposé le sous-problème `ligne_suivante` du passage d'un ligne du triangle de Pascal à la suivante en trois sous-problèmes :

1. Le sous-problème `decalage_gauche` qui décale une ligne vers la gauche en injectant un 0 à droite.
2. Le sous-problème `decalage_droite` qui décale une ligne vers la droite en injectant un 0 à gauche.
3. Le sous-problème `somme_listes` consistant à calculer la somme terme à terme de deux listes de même longueur.

Regardons ce que cela donne sur la prochaine étape :

A partir de : `[1, 5, 10, 10, 5, 1]`

1. par `decalage_gauche` on obtient :
   [1, 5, 10, 10, 5, 1, 0]
2. par `decalage_droite` on obtient :
   [0, 1, 5, 10, 10, 5, 1]
3. par `somme_listes` on obtient :
   [1, 6, 15, 20, 15, 6, 1]

On vérifiera (par exemple sur Wikipedia) que cette ligne suivante est correcte.

Finalement, on peut itérer la solution du problème `ligne_suivante` pour résoudre la construction du triangle de Pascal.

Commençons à résoudre nos sous-problèmes un par un.

7.3.2.4 Sous-problèmes : `decalage_gauche` et `decalage_droite` Les deux sous-problèmes les plus simples sont faciles à traduire en Python :

```python
def decalage_gauche(L):
    """ list[int] -> list[int]

    retourne le décalage à gauche de la liste L
    en injectant un 0 à droite."""

    return L + [0]
```

```python
# Jeu de tests
assert decalage_gauche([1, 4, 6, 4, 1]) == [1, 4, 6, 4, 1, 0]
assert decalage_gauche([1]) == [1, 0]
```

Le décalage à droite n'est pas beaucoup plus compliqué.

```python
def decalage_droite(L):
    """ list[int] -> list[int]

    retourne le décalage à droite de la liste L
    en injectant un 0 à gauche."""

    return [0] + L
```

```python
# Jeu de tests
assert decalage_droite([1, 4, 6, 4, 1]) == [0, 1, 4, 6, 4, 1]
assert decalage_droite([1]) == [0, 1]
```

7.3.2.5 Sous-problème : `somme_listes`

La solution du problème `somme_listes` n'est pas non plus très complexe.

```python
def somme_listes(L1, L2):
    """ list[int] * list[int] -> list[int]
    Hypothèse : les listes L1 et L2 sont de même longueur

    retourne la somme terme à terme des listes L1 et L2."""

    # LR : list[int]
    LR = []

    # i : int (indice courant)
    for i in range(0, len(L1)):
        LR.append(L1[i] + L2[i])

    return LR
```

```python
# Jeu de tests
assert somme_listes([0, 1, 4, 6, 4, 1], [1, 4, 6, 4, 1, 0]) \
       == [1, 5, 10, 10, 5, 1]
assert somme_listes([], []) == []
assert somme_listes([0, 0, 0, 0, 0], [1, 2, 3, 4, 5]) == [1, 2, 3, 4, 5]
assert somme_listes([1, 2, 3, 4, 5], [0, 0, 0, 0, 0]) == [1, 2, 3, 4, 5]
```

7.3.2.6 Sous-problème : `ligne_suivante`

La construction de la ligne suivante dans le triangle de Pascal consiste maintenant simplement à composer les fonctions définies précédemment.

```python
def ligne_suivante(L):
    """list[int] -> list[int]
    Hypothèse : L est une ligne du triangle de Pascal.
```

```python
    retourne la ligne suivant L dans le triangle de Pascal."""

    # LG : list[int]
    LG = decalage_gauche(L)

    # LD : list[int]
    LD = decalage_droite(L)

    return somme_listes(LG, LD)
```

```python
# Jeu de tests
assert ligne_suivante([1]) == [1, 1]
assert ligne_suivante([1, 1]) == [1, 2, 1]
assert ligne_suivante([1, 2, 1]) == [1, 3, 3, 1]
assert ligne_suivante([1, 3, 3, 1]) == [1, 4, 6, 4, 1]
assert ligne_suivante([1, 4, 6, 4, 1]) == [1, 5, 10, 10, 5, 1]
```

7.3.2.7 Problème : `triangle_pascal`

Nous n'avons plus qu'à itérer `ligne_suivante` pour construire les n premières lignes du triangle de Pascal.

```python
def triangle_pascal(n):
    """ int -> list[list[int]]
    Hypothèse : n >= 1

    retourne les n premières lignes du triangle
    de Pascal."""

    # Ligne : list[int]
    Ligne = [1]  # la ligne courante

    # Triangle : list[list[int]]
    Triangle = [Ligne]

    # k : int (ligne courante)
    for k in range(1, n):
        Ligne = ligne_suivante(Ligne)
        Triangle.append(Ligne)

    return Triangle
```

Exceptionnellement, plutôt qu'un jeu de test de validation, nous allons simplement essayer de construire les 10 premières lignes du triangle de Pascal.

```python
>>> triangle_pascal(10)
[[1],
 [1, 1],
 [1, 2, 1],
 [1, 3, 3, 1],
 [1, 4, 6, 4, 1],
```

```
 [1, 5, 10, 10, 5, 1],
 [1, 6, 15, 20, 15, 6, 1],
 [1, 7, 21, 35, 35, 21, 7, 1],
 [1, 8, 28, 56, 70, 56, 28, 8, 1],
 [1, 9, 36, 84, 126, 126, 84, 36, 9, 1]]
```

Il faudrait encore travailler un peu pour montrer la correction de notre solution, mais on peut dire que cela ressemble bien au triangle de Pascal.

On retiendra de cette section la *méthodologie de décomposition de problèmes*, qui a permis de définir des fonctions simples permettant de résoudre des sous-problèmes, qui une fois recomposés permettent de résoudre notre problème complexe de départ.

7.4 Exercices corrigés

Exercice (7.1) : Nombres complexes (corrigé)

Le but de cet exercice est de définir quelques opérations simples sur les nombres complexes ce qui amène à définir des fonctions simples manipulant des couples de flottants.

Question 1

Les nombres complexes sont décrits la plupart du temps par un couple de réels qui définit leur partie réelle et leur partie imaginaire. Définissons donc un type `Complexe = tuple[float float]` pour une telle représentation. Ainsi le nombre complexe $2 + 3i$ sera représenté par le couple $(2.0, 3.0)$, le nombre i par $(0.0, 1.0)$ et un réel r par $(r, 0.0)$.

Donner la spécification et une définition en Python des fonctions `partie_relle` et `partie_imaginaire` telles que `partie_reelle(c)` (resp. `partie_imaginaire(c)`) renvoie la partie réelle (resp. imaginaire) d'un nombre complexe. Par exemple :

```
>>> partie_reelle((2.0,3.0))
2.0
```

```
>>> partie_imaginaire((2.0,3.0))
3.0
```

```
>>> partie_reelle((0.0,1.0))
0.0
```

```
>>> partie_imaginaire((0.0,1.0))
1.0
```

```
>>> partie_reelle((4.0,0.0))
4.0
```

```
>>> partie_imaginaire((4.0,0.0))
0.0
```

Réponse

Remarques :

— cette question introduit sans le dire la notion d'accesseur, qu'on ne désire pas aborder dans le cours. On passera systématiquement par la décomposition des n-uplets par la suite, sans utiliser de tels accesseurs (cf les réponses aux questions suivantes). Cet exercice est donc là simplement pour proposer une première question très simple permettant de manipuler un couple.
— Il ne fait pas oublier de définir le type `Complexe` sous forme de commentaire "spécial" compréhensible par MrPython.
— le typage des variables de déconstruction est optionnel, mais recommandé lorsqu'il s'agit de décomposer un argument de la fonction.

```
# type Complexe = tup[float, float]

def partie_reelle(c):
    """Complexe -> float

    renvoie la partie réelle du nombre complexe c.
    """

    # re : float
    # im : float
    re, im = c

    return re
```

```
# Jeu de tests

assert partie_reelle((2.0,3.0)) == 2.0
assert partie_reelle((0.0,1.0)) == 0.0
assert partie_reelle((4.0,0.0)) == 4.0
```

```
def partie_imaginaire(c):
    """Complexe -> float

    renvoie la partie imaginaire du nombre complexe c.
    """

    # re : float
    # im : float
    re, im = c

    return im
```

```
# Jeu de tests

assert partie_imaginaire((2.0,3.0)) == 3.0
assert partie_imaginaire((0.0,1.0)) == 1.0
assert partie_imaginaire((4.0,0.0)) == 0.0
```

Question 2

Donner la spécification et une définition en Python de la fonction `addition_complexe` telle que `addition_complexe(c1, c2)` renvoie l'addition des nombres complexes `c1` et `c2`

Par exemple :

```
>>> addition_complexe((1.0, 0.0), (0.0, 1.0))
(1.0, 1.0)
```

```
>>> addition_complexe((2.0, 3.0), (0.0, 1.0))
(2.0, 4.0)
```

Réponse

```python
def addition_complexe(c1, c2):
    """Complexe * Complexe -> Complexe

    renvoie le nombre complexe correspondant à c1 + c2
    """

    # re1 : float
    # im1 : float
    re1, im1 = c1

    # re2 : float
    # im2 : float
    re2, im2 = c2

    return (re1 + re2 , im1 + im2)
```

```python
# Jeu de tests

assert addition_complexe((1.0, 0.0), (0.0, 1.0)) == (1.0, 1.0)
assert addition_complexe((2.0, 3.0), (0.0, 1.0)) == (2.0, 4.0)
```

Question 3

On rappelle que le produit de deux nombres complexes $(a + bi)$ et $(c + di)$ est donné par $(a + bi) * (c + di) = (ac - bd) + (ad + bc)i$. Donner la spécification et une définition en Python de la fonction `produit_complexe` telle que `produit_complexe(c1, c2)` renvoie le produit des nombres complexes `c1` et `c2`

Par exemple :

```
>>> produit_complexe((0.0, 0.0), (1.0, 1.0))
(0.0, 0.0)
```

```
>>> produit_complexe((0.0, 1.0), (0.0, 1.0))
(-1.0, 0.0)
```

```
>>> produit_complexe((2.0, 3.0), (0.0, 1.0))
(-3.0, 2.0)
```

Remarque : les nombres complexes existent en fait en `Python`. Ils ont pour type `complex`.

Réponse

```python
def produit_complexe(c1, c2):
    """Complexe * Complexe -> Complexe

    renvoie le nombre complexe correspondant à c1 * c2
    """

    # re1 : float
    # im1 : float
    re1, im1 = c1

    # re2 : float
    # im2 : float
    re2, im2 = c2

    return (re1*re2 - im1 * im2, re1 * im2 + im1 * re2)
```

```python
# Jeu de tests
assert produit_complexe((0.0, 0.0), (1.0, 1.0)) == (0.0, 0.0)
assert produit_complexe((0.0, 1.0), (0.0, 1.0)) == (-1.0, 0.0)
```

Exercice (7.2) : Nombre d'occurrences du maximum dans une liste (corrigé)

Cet exercice montre l'utilisation de n-uplet pour améliorer l'efficacité de la résolution d'un problème.

Question 1

Dans l'exercice 2 du Thème 6, nous avons écrit une fonction (`nb_max`) permettant de calculer le nombre de fois que le maximum d'une liste apparaît dans cette liste. La solution proposée consistait à parcourir une première fois la liste pour déterminer le maximum, puis une seconde fois pour compter le nombre d'occurrences de ce maximum.

Afin d'améliorer l'efficacité de cette fonction, on aimerait ne parcourir qu'une seule fois la liste afin de déterminer *à la fois* le maximum et le nombre d'occurrences de ce dernier.

Donner la spécification et une définition en Python de la fonction `nb_de_max` qui, étant donné une liste non vide de nombres, renvoie un couple contenant le maximum et le nombre de fois où ce maximum apparaît dans la liste. Par exemple

```python
>>> nb_de_max([10])
(10, 1)
```

```python
>>> nb_de_max([3, 7, 9, 5.4, 8.9, 9, 8.999, 5])
(9, 2)
```

```python
>>> nb_de_max([-2, -1, -5, -3, -1, -4, -1])
(-1, 3)
```

Réponse

```python
def nb_de_max(L):
    """list[Number] -> tuple[Number, int]
    hypothèse : len(L) > 0

    renvoie le couple (m,n) dans lequel m est le maxumum de L
            et n le nombre d'occurrences de m dans L.
    """

    # m : Number
    m=L[0] # candidat au maximum

    # n : int
    n=1 # nombre de fois où m est apparu dans L

    # x : Number
    for x in L[1:]:
        if x > m : # on a trouvé un nombre supérieur au maximum courant
            m = x
            n = 1
        elif x == m : # on a trouvé une nouvelle occurrence du maximum courant
            n = n + 1

    return (m,n)
```

```python
# Jeu de tests
assert nb_de_max([3, 7, 9, 5.4, 8.9, 9, 8.999, 5]) == (9, 2)
assert nb_de_max([-2, -1, -5, -3, -1, -4, -1]) == (-1, 3)
```

Exercice (7.4) : Tester l'alignement de points (corrigé)

Le but de cet exercice est de vérifier qu'une liste de points ne contient que des points alignés.

Le type Point sera défini dans tout l'exercice par un couple d'entiers. On choisit délibérément de travailler sur des entiers, afin d'éviter des problèmes d'arrondis. On définit donc formellement le type `Point = tuple[int, int]` que l'on pourra utiliser dans les signatures de fonctions.

Question 1

Donner la spécification et une définition en Python de la fonction `vecteur` telle que `vecteur(p1, p2)` renvoie le couple de coordonnées du vecteur formé par les points `p1` et `p2`.

On rappelle que les coordonnées du vecteur correspond au couple (différence des abscisses, différence des ordonnées).

Réponse

```python
# type Point = tuple[int, int]

def vecteur(p1, p2):
```

```python
    """Point * Point -> tuple[int, int]
    attention : un vecteur n'est pas un Point

    renvoie les coordonnées du vecteur p1p2.
    """

    x1, y1 = p1
    x2, y2 = p2

    return (x2-x1, y2-y1)
```

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

Question 2

Donner la spécification et une définition en Python de la fonction `alignes` telle que `alignes(p1, p2, p3)` renvoie le booléen `True` si les 3 points sont alignés et `False` sinon.

On rappelle que 3 points $p1, p2, p3$ sont alignés si les deux vecteurs $p1 \cdot p2$ et $p2 \cdot p3$ sont colinéaires, c'est-à-dire proportionnels.

Par exemple :

```python
>>> alignes((0,0), (1,1), (5,5))
True
```

```python
>>> alignes((0,0), (1,1), (1,2))
False
```

Réponse

```python
def alignes(p1, p2, p3):
    """Point * Point * Point -> bool

    renvoie un booléen indiquant si les 3 points sont alignés.
    """
    # v1 : tuple[int, int] (vecteur correspondant à p1p2)
    v1 = vecteur(p1, p2)

    # v2 : tuple[int, int] (vecteur correspondant à p2p3)
    v2 = vecteur(p2, p3)

    x1, y1 = v1
    x2, y2 = v2

    return x1 * y2 == x2 * y1
```

```python
# Jeu de tests

assert alignes((0,0), (1,1), (5,5)) == True
assert alignes((0,0), (1,1), (1,2)) == False
```

Question 3

Donner la spécification et une définition en Python de la fonction `alignement` qui, étant donné une liste `L` de points contenant au moins 3 éléments, renvoie le booléen `True` si tous les points de la liste `L` sont alignés et `False` sinon.

Par exemple :

```
>>> alignement([(0,0), (1,1), (5,5)])
True
```

```
>>> alignement([(0,0), (1,1), (5,5), (1,0)])
False
```

```python
def alignement(L):
    """list[Point] -> bool
    Hypothèse : L contient au moins 3 éléments.

    renvoie le booléen True si et seulement si tous les points de L sont alignés.
    """

    # rep : bool (contient la réponse courante)
    rep = True

    # i : int (indice courant)
    i = 0

    while rep and (i+2 < len(L)):
        rep = alignes(L[i], L[i+1], L[i+2])
        i = i + 1

    return rep
```

```python
# Jeu de tests
assert alignement([(0, 0), (1, 1), (5, 5)]) == True
assert alignement([(0, 0), (1, 1), (5, 5), (2, 2), (10, 10)]) == True
assert alignement([(0, 0), (1, 1), (5, 5), (1, 0)]) == False
```

8 Compréhensions de listes

8.1 Schémas de manipulation des listes

Les listes de Python sont plutôt versatiles, on les utilise pour résoudre différents types de problème dont le point commun est de nécessiter le stockage d'éléments en séquence, c'est-à-dire dans un ordre donné.

Dans ce cours, nous souhaitons prendre un peu de recul sur les manipulations de listes. Cela nous permettra d'aborder des constructions très expressives du langage Python pour manipuler les listes : les **compréhensions**. En guise de complément, et pour comprendre comment les compréhensions sont élaborées, nous enrichissant nos connaissances sur les **fonctionnelles** (ou fonctions d'ordre supérieur) pour systématiser les schémas de traitements sur les listes.

Nous avons vu que les schémas de manipulation de listes que l'on rencontre le plus fréquemment sont les suivants :

— le schéma de construction : lorsque l'on construit une liste de façon algorithmique à partir d'une séquence autre qu'une liste,
— le schéma de transformation: lorsque l'on transforme une liste en appliquant à chaque élément une fonction unaire donnée,
— le schéma de filtrage : lorsque l'on filtre les éléments d'un liste pour un prédicat donné,
— le schéma de réduction : lorsque l'on réduit une liste en une information synthétisée à partir de ses éléments.

Les **expressions de compréhension** de Python permettent de généraliser les trois premiers schémas. Le schéma de réduction peut également être généralisé à l'aide des **fonctionnelles**.

8.2 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.

8.2.1 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]`.

La définition ci-dessous exploite la boucle `while`.

```python
def naturels_while(n):
    """int -> list[int]
    Hypothèse: n > 0

    Retourne la liste des n premiers entiers naturels non-nuls."""
```

```python
    # k :int
    k = 1 # élément courant

    # LR : list[int]
    LR = [] # liste résultat

    while k <= n:
        LR.append(k)
        k = k + 1

    return LR
```

```python
# Jeu de tests
assert naturels_while(5) == [1, 2, 3, 4, 5]
assert naturels_while(1) == [1]
assert naturels_while(0) == []
```

Ce problème peut être vu de façon alternative comme la construction d'une liste à partir d'un intervalle d'entiers.

La définition ci-dessous adopte ce point de vue.

```python
def naturels_for(n):
    """int -> list[int]
    ..."""

    # LR : list[int]
    LR = [] # liste résultat

    # k : int
    for k in range(1, n + 1):
        LR.append(k)

    return LR
```

```python
# Jeu de tests
assert naturels_for(5) == [1, 2, 3, 4, 5]
assert naturels_for(1) == [1]
assert naturels_for(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 :

```python
>>> [k for k in range(1, 6)]
[1, 2, 3, 4, 5]
```

L'expression ci-dessous signifie presque littéralement :

construit 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]
```

8.2.2 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, string ou list.

Principe d'interprétation :

L'expression de compréhension ci-dessus signifie :

> Construit 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 : pour ne pas compromettre la concision des compréhensions, le type de la variable de compréhension <var> n'est pas déclaré. Ce type peut cependant être déduit 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).

Nous pouvons maintenant proposer une troisième définition pour la fonction naturels, cette fois-ci de façon particulièrement concise.

```
def naturels(n):
    """int -> list[int]
    ..."""
    return [k for k in range(1, n + 1)]
```

```python
# Jeu de tests
assert naturels_for(5) == [1, 2, 3, 4, 5]
assert naturels_for(1) == [1]
assert naturels_for(0) == []
```

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**.

8.2.3 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.

```python
def multiples(k, n):
    """int * int -> list[int]
    Hypothèse : 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)]
```

```python
# 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 : ré-écrire des version avec `while` et `for` de la fonction `multiples`.

8.2.4 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'`

Exercice : définir avec une compréhension la fonction `liste_caracteres` qui à partir d'une chaîne de caractères `s` retourne la liste des caractères de `s`.

8.3 Schéma de transformation

Le schéma de transformation, ou schéma `map`, consiste comme nous l'avons vu à appliquer une fonction unaire à chacun des éléments d'une liste pour reconstruire une liste transformée.

8.3.1 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.

```
def carre(n):
    """int -> int
    retourne le carré de l'entier n."""
    return n * n

def liste_carres_for(L):
    """ list[int] -> list[int]
    retourne la liste des carrés des éléments de la liste L."""

    # LR : list[int]
    LR = [] # liste résultat

    # e : int
    for e in L: # élément courant
        LR.append(carre(e))

    return LR
```

```
# Jeu de tests
assert liste_carres_for([1, 2, 3, 4, 5]) == [1, 4, 9, 16, 25]
assert liste_carres_for([10, 100, 1000]) == [100, 10000, 1000000]
assert liste_carres_for([]) == []
```

Remarque : la fonction `carre` n'est pas très intéressante, mais sa définition permet de mieux expliciter le schéma de transformation.

Utilisons une expression de compréhension correspondant au premier cas de test.

```
>>> [carre(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]`

En lisant la description ci-dessus, on devrait bien s'attendre au résultat suivant :

```
[1, 4, 9, 16, 25]
```

On peut donc redéfinir la fonction `liste_carres` avec une compréhension.

```
def liste_carres(L):
    """list[int] -> list[int]
    ..."""

    return [carre(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 cours sur les listes : la construction de la liste des longueurs de chaînes.

```
def liste_longueurs_for(L):
    """ list[str] -> list[int]

    retourne la liste des longueurs des chaînes de L."""

    # LR : list[int]
    LR = []  # liste résultat

    # s : str
    for s in L:
        LR.append(len(s))

    return LR
```

```
# Jeu de tests
assert liste_longueurs_for(['un', 'deux', 'trois', 'quatre', 'cinq']) \
       == [2, 4, 5, 6, 4]
assert liste_longueurs_for(['', '.', '..', '...']) == [0, 1, 2, 3]
assert liste_longueurs_for([]) == []
```

Ici on a une transformation d'une liste de chaînes de caractères en une liste d'entiers. Cette transformation peut être réécrite avec une compréhension.

```
def liste_longueurs(L):
    """ list[str] -> list[int]
    ..."""
    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`.

8.3.2 Complément : la fonctionnelle map

Il est possible d'écrire une fonction générique `map` qui offre une alternative aux compréhensions pour exprimer de façon concise des transformations de listes.

```
def map(f, L):
    """(alpha -> beta) * list[alpha] -> list[beta]

    Retourne la transformation de la liste L par la fonction f."""

    # LR : list[alpha]
    LR = []

    # e : alpha
    for e in L:
        LR.append( f(e) )

    return LR
```

```
# Jeu de tests
assert map(carre, [1, 2, 3, 4, 5]) == [1, 4, 9, 16, 25]
assert map(len, ['un', 'deux', 'trois', 'quatre', 'cinq']) == [2, 4, 5, 6, 4]
```

Dans la définition de la fonctionnelle `map`, la principale difficulté concerne la signature de la fonction, que l'on rappelle ci-dessous :

```
(alpha -> beta) * list[alpha] -> list[beta]
```

Cette signature s'interprète de la façon suivante.

Dans `map(f, L)` :

- le premier paramètre `f` est de type `alpha -> beta`, c'est-à-dire une fonction unaire qui transforme une donnée de type `alpha` en une donnée de type `beta`.
- le second paramètre `L` est une liste de type `list[alpha]`

— la valeur de retour de `map` est de type `list[beta]`.

Par exemple, dans l'expression :

```
map(carre, [1, 2, 3, 4, 5])
```

— le premier argument `carre` pour `f` est de type `int -> int` (donc `alpha`=`int` et `beta`=`int`)
— le second argument `[1, 2, 3, 4, 5]` pour `L` est de type `list[int]` (donc `list[alpha]` pour `alpha`=`int`)
— la valeur de retour est `[1, 4, 9, 16, 25]` de type `list[int]` (donc `list[beta]` pour `beta`=`int`)

De même, dans l'expression :

```
map(len, ['un', 'deux', 'trois', 'quatre', 'cinq'])
```

— le premier argument `len` pour `f` est de type `str -> int` (donc `alpha`=`str` et `beta`=`int`)
— le second argument `['un', 'deux', 'trois', 'quatre', 'cinq']` pour `L` est de type `list[str]` (donc `list[alpha]` pour `alpha`=`str`)
— la valeur de retour est `[2, 4, 5, 6, 4]` de type `list[int]` (donc `list[beta]` pour `beta`=`int`)

Exercice : Proposer une variante de la fonction `map` donnée ci-dessus en utilisant une expression de compréhension.

8.4 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.

8.4.1 Exemples : liste des entiers positifs et liste des entiers pairs

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.
```

```python
    """
    return (n % 2) == 0
```

```python
# Jeu de tests
assert est_pair(1) == False
assert est_pair(2) == True
assert est_pair(3) == False
```

Le filtrage pour le premier prédicat permet de ne conserver que les éléments positifs d'une liste d'entiers.

```python
def liste_positifs_for(L):
    """list[int] -> list[int]
    Retourne la sous-liste des entiers positifs de L."""

    # LR : list[int]
    LR = [] # liste résultat

    # n : int
    for n in L:
        if est_positif(n):
            LR.append(n)

    return LR
```

```python
# Jeu de tests
assert liste_positifs_for([1, -1, 2, -2, 3, -3, -4]) == [1, 2, 3]
assert liste_positifs_for([1, 2, 3]) == [1, 2, 3]
assert liste_positifs_for([-1, -2, -3]) == []
```

Pour le second prédicat on obtient la sous-liste des entiers pairs.

```python
def liste_pairs_for(L):
    """list[int] -> list[int]
    Retourne la sous-liste des entiers pairs de L."""

    # LR :list[int]
    LR = [] # list[int]

    # n : int
    for n in L:
        if est_pair(n):
            LR.append(n)

    return LR
```

```python
# Jeu de tests
assert liste_pairs_for([1, 2, 3, 4, 5, 6, 7]) == [2, 4, 6]
assert liste_pairs_for([2, 4, 6]) == [2, 4, 6]
assert liste_pairs_for([1, 3, 5, 7]) == []
```

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 positifs pour n dans la liste [1, -1, 2, -2, 3, -3, -4]

Un second exemple avec les pairs :

```
>>> [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 pairs pour n dans la liste [1, 2, 3, 4, 5, 6, 7]

8.4.2 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.

Nous pouvons maintenant redéfinir nos deux fonctions liste_positifs et liste_pairs de façon particulièrement concise.

```python
def liste_positifs(L):
    """list[int] -> list[int]
    Retourne la sous-liste des entiers positifs de `L`."""

    return [n for n in L if est_positif(n)]
```

```python
# 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]) == []
```

```python
def liste_pairs(L):
    """list[int] -> list[int]
    Retourne la sous-liste filtrée des entiers pairs de `L`."""

    return [n for n in L if est_pair(n)]
```

```python
# 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]) == []
```

8.4.3 Complément : la fonctionnelle filter

Tout comme pour `map` il est possible d'écrire une **fonctionnelle de filtrage**.

```python
def filter(predicat, L):
    """(alpha -> bool) * list[alpha] -> list[alpha]

    Retourne la sous-liste des éléments de L filtrés par le prédicat."""

    # LR : list[alpha]
    LR = [] # liste résultat

    # e: alpha
    for e in L:
        if predicat(e):
            LR.append(e)

    return LR
```

```python
# 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, que l'on rappelle ci-dessous :

```
(alpha -> bool) * list[alpha] -> list[alpha]
```

Cette signature s'interprète de la façon suivante.

Dans `filter(predicat, L)` :

— le premier paramètre `predicat` est de type `alpha -> bool`, c'est-à-dire une fonction unaire à valeur booléenne,
— le second paramètre `L` est une liste de type `list[alpha]`
— la valeur de retour de `filter` est de type `list[alpha]`.

Par exemple, dans l'expression :

```python
filter(est_positif, [1, -1, 2, -2, 3, -3, -4])
```

- le premier argument `est_positif` pour `predicat` est de type `int -> bool` (donc `alpha=int`)
- le second argument `[1, -1, 2, -2, 3, -3, -4]` pour `L` est de type `list[int]` (donc `list[alpha]` pour `alpha=int`)
- la valeur de retour est `[1, 2, 3]` de type `list[int]` (donc `list[alpha]` pour `alpha=int`)

Exercice : Proposer une variante de la fonction `filter` donnée ci-dessus en utilisant une expression de compréhension conditionnée.

8.5 Plus loin avec les compréhensions

Dans cette section, nous abordons certains aspects plus avancés des expressions de compréhensions.

8.5.1 Les schémas de construction-transformation-filtrage

La syntaxe des compréhensions autorise naturellement la combinaison de constructions, de transformations et du filtrages pour produire des listes complexes à partir d'autres séquences (intervalles, chaînes ou listes).

Considérons par exemple le problème de la construction de la liste des premiers entiers pairs élévés au carré.

Essayons directement avec une expression de compréhension pour les 10 premiers entiers naturels non-nuls.

```python
>>> [k*k for k in range(1, 11) if est_pair(k)]
[4, 16, 36, 64, 100]
```

On en déduit la fonction suivante :

```python
def liste_carres_pairs(n):
    """ int -> list[int]
    Hypothèse : n >= 1

    Retourne la liste des carrés des entiers naturels
    pairs dans l'intervalle [1;n]."""

    return [k*k for k in range(1, n + 1) if est_pair(k)]
```

```python
# Jeu de tests
assert liste_carres_pairs(10) == [4, 16, 36, 64, 100]
assert liste_carres_pairs(1) == []
assert liste_carres_pairs(2) == [4]
```

Exercice : proposer une définition alternative de la fonction `liste_carres_pairs` en utilisant une boucle `for` d'itération.

8.5.2 Les 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 :

```
def liste_prix(L):
    """ list[tuple[str, float]] -> list[float]
    Retourne la liste des prix des produits de L."""

    # LR : list[float]
    LR = []

    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.

Bien sûr, les conditions sont également disponibles.

Retournons par exemple la liste des produits dont le prix dépasse 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']
```

8.5.3 Les 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.

```python
def liste_couples_for(n):
    """ int -> list[tuple[int, int]]
    Hypothèse: n >= 0

    retourne la liste des couples (i,j) sur l'intervalle [1;n]."""

    # LR : list[tuple[int, int]]
    LR = []  # liste résultat

    # i : int
    for i in range(1, n + 1):
        # j : int
        for j in range(i, n + 1):
            LR.append( (i, j) )

    return LR
```

```python
# Jeu de tests
assert liste_couples_for(0) == []
assert liste_couples_for(1) == [(1, 1)]
assert liste_couples_for(2) == [(1, 1), (1, 2), (2, 2)]
assert liste_couples_for(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` :

```python
>>> [(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` :

```python
>>> [(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)]
```

8.5.3.1 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.

```python
def liste_couples(n):
    """ int -> list[tuple[int, int]]
    Hypothèse: n >= 0

    retourne la liste des couples (i,j) sur l'intervalle [1;n]."""

    return [(i, j) for i in range(1, n + 1) for j in range(i, n + 1)]
```

```python
# Jeu de tests
assert liste_couples(0) == []
assert liste_couples(1) == [(1, 1)]
```

```python
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 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>`.

8.5.3.2 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 :

```python
>>> [(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)]
```

Question : pouvez-vous donner une interprétation littérale de ce résultat ?

8.5.3.2.1 Exemple : liste des couples divisibles Dans le cours sur les boucles, nous avons compté l'ensemble des couples (i, j) dans un intervalle $[1; n]$ tels que j divise i.

Voici une définition de la fonction `liste_couples_divisibles` qui retourne la liste de ces couples.

```python
def liste_couples_divisibles(n):
    """ int -> list[tuple[int, int]]
    Hypothèse : 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]
```

```python
# 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.

8.6 Complément : le schéma de réduction

Le dernier schéma classique de manipulation des listes consiste à synthétiser une information «simple» à partir 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.

Reprenons un exemple classique de réduction : le calcul de la longueur d'une liste (sans utiliser la fonction `len`)

```
def longueur(L):
    """list[alpha] -> int

    Retourne la longueur de la liste `L`."""

    # n : int
    n = 0

    # e : alpha
    for e in L:
        n = n + 1

    return n
```

```
# Jeu de tests
assert longueur([1, 2, 3, 4, 5]) == 5
assert longueur([]) == 0
```

La somme des éléments d'un liste de nombres est un autre exemple classique de réduction.

```
def somme_liste(L):
    """list[Number] -> Number
    Retourne la somme des éléments de la liste L."""

    # s: int
    s = 0

    # n : Number
    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
```

Il est possible de généraliser ce schéma avec la **fonctionnelle de réduction** que l'on peut définir de la façon suivante.

```
def reduce(op, en, L):
    """(alpha*beta->beta) * beta * list[alpha] -> beta
    Retourne la réduction de la liste `L` selon pour l'opérateur binaire
    `op` associatif avec élément neutre `en`."""

    # reduc : beta
    reduc = en
```

```python
    # 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.

```python
def plus(a, b):
    """Number * Number -> Number
    Retourne l'addition de a et b."""

    return a + b
```

```python
>>> reduc(plus, 0, [1, 2, 3, 4, 5])
15
```

```python
>>> reduce(plus, 0, [])
0
```

Dans le même genre d'idée on peut calculer le produit des éléments d'une liste.

```python
def fois(a, b):
    """Number * Number -> Number
    Retourne le produit de a et b."""

    return a * b
```

```python
>>> reduce(fois, 1, [1, 2, 3, 4, 5])
120
```

C'est une façon un peu inefficace de calculer la factorielle… Pour la longueur, ce n'est pas beaucoup plus complexe.

```python
def incr_un(n,m):
    """int *int -> int
    Retourne toujours `m+1` quelque soit l'argument `n`.
    """
    return m + 1
```

```python
>>> reduce(incr_un, 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, que l'on rappelle ci-dessous :

```
(alpha*beta->beta) * beta * list[alpha] -> beta
```

Cette signature s'interprète de la façon suivante.

Dans `reduce(op, en, L)` :

— le premier paramètre `op` est de type `alpha * beta -> beta` qui correspond à une fonction binaire combinant une donnée de type `alpha` et une donnée de type `beta` pour produire

une donnée de type `beta`.
— le second paramètre `en` est de type `beta` qui correspond à un élément neutre pour `op`
— le troisième paramètre `L` est une liste de type `list[alpha]`
— la valeur de retour de `reduce` est de type `beta`.

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 `alpha=int` et `beta=int`)
— le deuxième argument `0` pour `en` est de type `int` (donc `beta` pour `beta=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[alpha]` pour `alpha=int`)
— la valeur de retour est `15` de type `list[int]` (donc `beta` pour `beta=int`)

Question : pouvez-vous trouver un exemple de réduction avec `alpha` et `beta` différents ?

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.

8.7 Exercices corrigés

Exercice (8.1) : Revisiter les listes (corrigé)

L'objectif de cet exercice consiste à revisiter certains exercices sur les listes du thème 6 et de proposer de nouvelles solutions basées sur les compréhensions.

Question 1

Proposer une solution basée sur les compréhensions pour :

— la question 1 de l'exercice 1 *Listes de Répétitions* du thème 6

Réponse

```
def repetition(x, k):
    """alpha * int -> list[alpha]
    Hypothèse: k >= 0

    retourne la liste composée de k occurrences de x."""

    return [x for i in range(1, k + 1)]
```

```
# Jeu de tests
assert repetition(0, 4) == [0, 0, 0, 0]
assert repetition(4, 0) == []
assert repetition('pom', 5) == ['pom', 'pom', 'pom', 'pom', 'pom']
```

Question 2

Proposer une solution basée sur les compréhensions pour :

— les questions 1 à 3 de l'exercice 4 *Liste de diviseurs* du thème 6

Réponse

```python
def liste_diviseurs(a):
    """int -> list[int]
    Hypothèse : a > 0

    retourne la liste des diviseurs de a."""

    return [i for i in range(1, a + 1) if a % i == 0]
```

```python
# Jeu de tests
assert liste_diviseurs(2) == [1, 2]
assert liste_diviseurs(12) == [1, 2, 3, 4, 6, 12]
assert liste_diviseurs(25) == [1, 5, 25]
```

```python
def liste_diviseurs_impairs(a):
    """int -> list[int]

    retourne la liste des diviseurs impairs de a
    hypothèse : a > 0."""

    return [i for i in range(1, a + 1)
              if (a % i == 0) and (i % 2 == 1)]
```

```python
# Jeu de tests
assert liste_diviseurs_impairs(2) == [1]
assert liste_diviseurs_impairs(12) == [1, 3]
assert liste_diviseurs_impairs(30) == [1, 3, 5, 15]
assert liste_diviseurs_impairs(15) == [1, 3, 5, 15]
```

```python
def liste_diviseurs_pairs(a):
    """int -> list[int]

    retourne la liste des diviseurs pairs de a
    hypothèse : a > 0."""

    return [i for i in range(1, a + 1)
              if (a % i == 0) and (i % 2 == 0)]
```

```python
# Jeu de tests
assert liste_diviseurs_pairs(2) == [2]
assert liste_diviseurs_pairs(12) == [2, 4, 6, 12]
assert liste_diviseurs_pairs(30) == [2, 6, 10, 30]
assert liste_diviseurs_pairs(15) == []
```

Question 3

Proposer une solution basée sur les compréhensions pour :

— les questions 1 et 2 de l'exercice 6 *Listes obtenues par multiplication ou division* du thème 6

Réponse

```python
def list_mult(L, k):
    """list[int] * int -> list[int]
    retourne la liste obtenue en multipliant par k
    tous les éléments de L."""

    return [k * e for e in L]
```

```python
# Jeu de tests
assert list_mult([3, 5, 2, 4], 5) == [15, 25, 10, 20]
assert list_mult([3, 5, 2, 4], -1) == [-3, -5, -2, -4]
assert list_mult([3, 5, 2, 4], 0) == [0, 0, 0, 0]
assert list_mult([], 10) == []
```

```python
def list_div(L, k):
    """list[int] * int -> list[int]
    hypothèse : k non nul
    retourne la liste obtenue en divisant par k les éléments de L
    qui sont multiples de k et en supprimant les autres."""

    return [e // k for e in L if e % k == 0]
```

```python
# Jeu de tests
assert list_div([3, 45, 10, 8, 50, 1], 5) == [9, 2, 10]
assert list_div([-3, 45, 10, 8, 50, 1], -3) == [1, -15]
assert list_div([3, 5, 7, 11], 2) == []
assert list_div([], 10) == []
```

Exercice (8.4) : Crible d'Eratosthène (corrigé)

Cet exercice consiste à implémenter le crible d'Eratosthène qui décrit un moyen de calculer la listes des nombres premiers inférieurs à un entier fixé. On rappelle qu'un nombre est premier s'il n'est divisible que par 1 et lui-même. On rappelle également que, par convention, 1 n'est pas un nombre premier.

La méthode du crible d'Eratosthène consiste à créer dans un premier temps la liste des entiers compris entre 2 et n, puis itérer le processus suivant :

1. Récupérer le premier élément de la liste (ce nombre est un nombre premier)
2. Retirer de la liste restante tous les multiples de ce nombre.
3. Recommencer à l'étape 1. tant qu'il reste des éléments dans la liste

Illustrons la méthode avec l'entier $n = 10$. On commence par créer la liste

`[2, 3, 4, 5, 6, 7, 8, 9, 10]`

On récupère alors le prochain nombre premier, `2`, et on calcule la liste dans laquelle les multiples de 2 ont été retirés, soit `[3, 5, 7, 9]`.

On récupère alors le prochain nombre premier, `3`, et on calcule la liste dans laquelle les multiples de 3 ont été retirés, soit `[5, 7]`.

On récupère alors le prochain nombre premier, `5`, et on calcule la liste dans laquelle les multiples de 5 ont été retirés, soit `[7]`.

On récupère alors le prochain nombre premier, `7`, et on calcule la liste dans laquelle les multiples de 7 ont été retirés, soit `[]`.

La liste est vide, donc le calcul est terminé. La liste des nombres premiers inférieurs ou égaux à 10 est donc `[2, 3, 5, 7]`.

Question 1

À l'aide d'une compréhension, donner la définition et la spécification de la fonction `liste_non_multiple` qui, étant donné un entier `n` non nul et une liste d'entiers `L`, renvoie la liste des éléments de `L` qui ne sont pas multiples de `n`. Par exemple :

```
>>> liste_non_multiple(2,[2,3,4,5,6,7,8,9,10])
[3, 5, 7, 9]
```

```
>>> liste_non_multiple(3,[2,3,4,5])
[2, 4, 5]
```

```
>>> liste_non_multiple(2,[2,4,6])
[]
```

```
>>> liste_non_multiple(2,[])
[]
```

```
>>> liste_non_multiple(7,[2,3,4,5])
[2, 3, 4, 5]
```

Réponse

```python
def liste_non_multiple(n,L):
    """int * list[int] -> list[int]
    Hypothèse : n != 0

    renvoie la liste des éléments de L qui ne sont pas multiples de n."""

    return [e for e in L if e % n != 0]
```

```python
# Jeu de tests
assert liste_non_multiple(2,[2,3,4,5]) == [3,5]
assert liste_non_multiple(2,[2,4,6]) == []
assert liste_non_multiple(3,[2,3,4,5]) == [2,4,5]
assert liste_non_multiple(2,[]) == []
assert liste_non_multiple(7,[2,3,4,5]) == [2,3,4,5]
```

Question 2

Donner la définition et la spécification de la fonction **eratosthene** qui calcule la liste des nombres premiers inférieurs ou égaux à un entier **n** (supérieur ou égal à 2) en utilisant le crible d'Eratosthène décrit ci-dessus. Par exemple :

```
>>> eratosthene(10)
[2, 3, 5, 7]
```

```
>>> eratosthene(2)
[2]
```

```
>>> eratosthene(50)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Réponse

```python
def eratosthene(n):
    """ int -> list[int]
    Hypothèse : n > 1
    renvoie la liste des nombres premiers inférieurs ou égaux à n."""

    #L : list[int]
    L = [ k for k in range(2,n+1)] # liste de départ

    #R : list[int]
    R = [] # liste contenant les nombres premiers

    #p : int
    p=0 # prochain nombre premier

    while len(L) > 0:
        p = L[0] # on récupère le prochain nombre premier
        R.append(p) # on le rajoute à la liste courante
        L = liste_non_multiple(p, L) # on calcule les entiers non multiples du nombre premier

    return R
```

```python
# Jeu de tests
assert eratosthene(10) == [2,3,5,7]
assert eratosthene(2) == [2]
```

Question 3

Un facteur premier d'un entier n est un nombre premier qui divise n. À l'aide de la fonction précédente, et en utilisant une compréhension, donner la définition et la spécification de la fonction `liste_facteurs_premiers` qui, étant donné un entier **n** supérieur ou égal à 2, calcule la liste des facteurs premiers de **n**, c'est à dire la liste des nombres premiers inférieurs ou égaux à **n** et qui divisent **n**. Par exemple :

```
>>> liste_facteurs_premiers(2)
[2]
```

```
>>> liste_facteurs_premiers(10)
[2, 5]
```

```
>>> liste_facteurs_premiers(2*3*7)
[2, 3, 7]
```

```
>>> liste_facteurs_premiers(2*3*4*7*9)
[2, 3, 7]
```

Réponse

```
def liste_facteurs_premiers(n):
    """int-> list[int]
    Hypothèse : n > 1

    renvoie la liste des facteurs premiers de n."""

    return [e for e in eratosthene(n) if n%e == 0]
```

```
# Jeu de tests
assert liste_facteurs_premiers(2) == [2]
assert liste_facteurs_premiers(10) == [2, 5]
assert liste_facteurs_premiers(2*3*7) == [2, 3, 7]
assert liste_facteurs_premiers(2*3*4*7*9) == [2, 3, 7]
```

9 Ensembles et dictionnaires

Dans ce cours, nous introduisons deux nouvelles structures de données :

— les **ensembles** qui sont l'équivalent informatique des ensembles mathématiques
— les **dictionnaires** qui associent des clés de recherche à des valeurs.

9.1 Les Ensembles

La notion d'*ensemble* est fondamentale en mathématique. De fait, l'essentiel des constructions mathématiques sont basées sur la *théorie des ensembles*.

9.1.1 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).

Pour notre cours, nous adopterons la définition suivante.

Définition : un **ensemble** de type `set[`α`]` est une collection d'**éléments** de type α et tous *distincts* (au sens de l'égalité de Python `==`)

Pour des raisons techniques, le type α des éléments d'un ensemble doit être *immutable*. A l'exception des listes et des ensembles eux-mêmes, tous les types que nous avons vus répondent à cette contrainte. On retiendra donc :

> Aucun type liste ou ensemble ne peut apparaître dans le α de `set[`α`]`.

9.1.1.1 Construction explicite d'un ensemble

Pour construire un ensemble, Python propose une syntaxe inspirée de la notation mathématique usuelle.

Syntaxe de la construction explicite d'un ensemble `E` de type `set[`α`]` :

`{`e_1`,`e_2`,`$\ldots$`, `e_n`}`

où tous les e_1, $\ldots$, e_n sont des expressions de type α.

Par exemple, pour construire l'ensemble $\{2, 5, 9\}$, c'est-à-dire l'ensemble contenant les éléments entiers 2, 5 et 9, on utilise l'expression suivante :

```
>>> {2, 5, 9}
{9, 2, 5}
```

Ici, l'ensemble construit est de type `set[int]`.

Attention : la construction de l'**ensemble vide** qui se note $\emptyset$ en mathématiques se note de façon spécifique en Python :

```
>>> set()
set()
```

Pour des raisons historiques , la notation `{}` est en effet réservée aux dictionnaires vides (mais pour éviter toute ambiguïté nous utiliserons plutôt `dict()`, cf. deuxième partie du cours).

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 fondamental : les éléments d'un ensemble ne sont pas ordonnés. Plus précisément, les éléments sont ordonnés (car en informatique tout est bien ordonné) de façon complètement 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.

Finalement, la dernière propriété importante est que les éléments d'un ensemble de type `set[`α`]` doivent *tous* être du même type α. Les ensemble que l'on considère doivent êtres *homogènes*, comme pour les listes. On s'éloigne ici un peu de la théorie dite *naïve* des ensembles qui permet de mélanger «les chaussettes et les culottes» (voir plus loin pour cet exemple naïf). D'ailleurs, et de façon similaire aux listes, Python ne fait pas de distinction sur le type des éléments d'un ensemble :

```
>>> type({2, 5, 9})
set
```

```
>>> type({'e', 'a', 'u'})
set
```

Puisqu'un programmeur (même confirmé) *doit* maîtriser le type des informations qu'il manipule, nous instituons comme pour les listes la règle du typage explicite et précis des éléments d'un ensemble.

A retenir : c'est au programmeur (donc à *vous*) qu'incombe la responsabilité de garantir que les éléments contenus dans un ensemble de type `set[`α`]` soient bien tous du *même* type α.

9.1.1.2 Test d'appartenance

Toujours d'après Wikipedia :

> la théorie des ensembles est une théorie de l'appartenance (un élément d'un ensemble est dit « appartenir » à cet ensemble).

(cf. http://fr.wikipedia.org/wiki/Appartenance_%28math%C3%A9matiques%29)

L'opération fondamentale des ensembles est donc le **test d'appartenance**.

En mathématique, si un élément e appartient à un ensemble E on note : $e \in E$.

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 α
- `<ensemble>` est une expression de type `set[`α`]`

Le **principe d'évaluation** associé est simplissime. 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 que dans la plupart des cas un seul test suffit pour répondre à la même question dans le cas d'un ensemble de n éléments.

9.1.1.3 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[`α`]` et `<elem>` une expression de type α.

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. Aucune valeur n'est ainsi retournée par la méthode `add`.

Pour observer ces phénomènes, considérons la définition de variable suivante :

```
# Ens : set[str]
Ens = {'a', 'e', 'u'}
```

Demandons à Python la valeur de la variable `Ens` :

```
>>> Ens
{'a', 'e', 'u'}
```

Ajoutons maintenant un élément :

```
>>> Ens.add('o')
```

Ici on observe bien que la méthode `add` ne retourne rien. En revanche, l'ensemble référencé par la variable `Ens` a été modifié directement en mémoire, comme l'atteste l'exemple ci-dessous :

```
>>> Ens
{'a', 'e', 'o', 'u'}
```

De façon similaire à la méthode `append` des listes, on utiliser la méthode `add` uniquement pour reconstruire des ensembles dans des fonctions dont le type de retour est de la forme `set[`α`]`.

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`.

```
def presences(L):
    """ list[int] -> set[int]
```

```python
    retourne l'ensemble des entiers qui apparaissent
    au moins une fois dans L."""

    # E : set[int]
    E = set()  # l'ensemble résultat, initialement vide

    # n : int
    for n in L:
        E.add(n)

    return E
```

```python
# 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.

9.1.1.4 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[`α`]` et `<elem>` une expression de type α.

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, sans reconstruction d'un ensemble. Aucune valeur n'est ainsi retournée par la méthode `remove`.

Pour observer ces phénomènes, reprenons la variable `Ens` définie précédemment :

```
>>> Ens
{'a', 'e', 'o', 'u'}
```

Extrayons maintenant la lettre `'e'` de l'ensemble.

```
>>> Ens.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 `Ens` :

```
>>> Ens
{'a', 'o', 'u'}
```

L'élément a bien été retiré de l'ensemble.

9.1.2 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 :

```
for <var> in <ensemble>:
    <corps>
```

où :

— `<ensemble>` est une expression d'ensemble de type `set[`α`]`
— `<var>` est la *variable d'itération d'élément* de type α
— `<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 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(E):
    """set[int] -> int

    Renvoie la somme des éléments de E."""

    # s : int
    s = 0  # somme

    # n : int (élément courant)
    for n in E:
        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 ...

9.1.3 Egalité ensembliste et notion de sous-ensemble

La notion de **sous-ensemble** est fondamentale en théorie des ensembles. Mathématiquement, on dirait qu'un ensemble E_1 est sous-ensemble d'un ensemble E_2, ce que l'on note :

$$E_1 \subseteq E_2 \text{ si et seulement si } \forall e \in E_1, e \in E_2$$

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 :

```python
def est_sous_ensemble(E1, E2):
    """set[alpha] * set[alpha] -> bool

    Renvoie True si E1 est sous-ensemble de E2, False Sinon."""

    # e : alpha (élément courant)
    for e in E1:
        if e not in E2:
            return False

    return True
```

```python
# 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(E1, E2)
```

On peut utiliser de façon équivalente (et préférée) l'expression suivante :

```
E1 <= E2
```

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 :

— `E1 < E2` correspond au comparateur *sous-ensemble strict* $E_1 \subset E_2$
— `E1 >= E2` correspond au comparateur *sur-ensemble* $E_1 \supseteq E_2$
— `E1 > E2` correspond au comparateur *sur-ensemble strict* $E_1 \supset E_2$

En mathématiques, deux ensembles E_1 et E_2 sont égaux si et seulement ils contiennent exactement les mêmes éléments, c'est-à-dire :

— $\forall e_1 \in E_1, e_1 \in E_2$
— $\forall e_2 \in E_2, e_2 \in E_1$

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 $E_1 \subseteq E_2$ *et* $E_2 \subseteq E_1$.

On peut donc directement traduire cette dernière définition de la **relation d'égalité sur les ensembles** en Python :

```
def ensembles_egaux(E1, E2):
    """set[alpha] * set[alpha] -> bool

    Renvoie True si E1 et E2 sont égaux, False Sinon."""

    return est_sous_ensemble(E1, E2) and est_sous_ensemble(E2, E1)
```

```
# 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(E1, E2)
```

s'écrit plus simplement (et de façon préférée) :

```
E1 == E2
```

De même, le test d'inégalité s'écrit tout simplement :

```
E1 != E2
```

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
```

9.1.4 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.

9.1.4.1 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 notation mathématique, l'**union** entre deux ensembles E_1 et E_2 se note $E_1 \cup E_2$.

La propriété fondamentale de l'union est que si un élément $e \in E_1 \cup E_2$ alors :

— e appartient à E_1 (ex.: *«e est une chaussette»*)
— **ou** e appartient à E_2 (ex.: *«e est une culotte»*)

Remarque : il est possible que l'élément e appartienne simultanément à E_1 et E_2 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 $E_1 \cup E_2$ consiste à composer un nouvel ensemble regroupant tous les éléments de E_1 et tous les éléments de E_2. Ceci se traduit très simplement en Python.

```python
def union(E1, E2):
    """set[alpha] * set[alpha] -> set[alpha]

    Retourne l'union des ensembles E1 et E2."""

    # U : set[alpha]
    U = set()   # ensemble résultat

    # e1 : alpha  (élément de E1)
    for e1 in E1:
        U.add(e1)

    # e2 : alpha (élément de E2)
    for e2 in E2:
        U.add(e2)

    return U
```

```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(E1, E2)`

s'écrit plus simplement (et de façon préférée) :

`E1 | E2`

Voici quelques exemples d'utilisation de cet opérateur d'union ensembliste :

```python
>>> {'a','e','u'} | {'o', 'i'}
{'a', 'e', 'i', 'o', 'u'}
```

```python
>>> {2, 5, 9} | {1, 3, 4, 8}
{1, 2, 3, 4, 5, 8, 9}
```

```python
>>> {1, 2, 5, 9} | {1, 3, 4, 8}
{1, 2, 3, 4, 5, 8, 9}
```

9.1.4.2 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»

En notation mathématique, l'**intersection** entre deux ensembles E_1 et E_2 se note $E_1 \cap E_2$.

La propriété fondamentale de l'intersection est que si un élément $e \in E_1 \cap E_2$ alors :

— e appartient à E_1 (ex.: *«e est une chaussette rouge»*)
— **et** e appartient à E_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 $E_1 \cap E_2$ consiste à composer un nouvel ensemble en prenant tous les éléments de E_1 qui sont également présent dans E_2.

Remarque : on pourrait de façon symétrique prendre tous les éléments de E_2 qui sont également dans E_1.

En Python, cet algorithme (ou son symétrique) s'exprime très simplement.

```python
def intersection(E1, E2):
    """set[alpha] * set[alpha] -> set[alpha]

    Retourne l'intersection des ensembles E1 et E2."""

    # I : set[alpha]
    I = set()    # ensemble intersection

    # e1 : alpha (élément de E1)
    for e1 in E1:
        if e1 in E2:
            I.add(e1)

    return I
```

```python
# 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(E1, E2)`

s'écrit plus simplement (et de façon préférée) :

`E1 & E2`

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()
```

9.1.4.3 Différence La différence ensembliste est la dernière opération que nous allons reconstruire dans ce cours.

> *«Il faudra trier les chaussettes sans trous, et jeter les autres»*

La **différence** entre deux ensembles E_1 et E_2 se note traditionnellement $E_1 \setminus E_2$ et consiste simplement à prendre les éléments de E_1 (ex.:*«toutes les chaussettes»*) et à en «retirer» les éléments qui sont également dans E_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 :

```python
def difference(E1, E2):
    """set[alpha] * set[alpha] -> set[alpha]

    Retourne la difference des ensembles E1 et E2."""

    # D : set[alpha]
    D = set()   # ensemble différence

    # e1 : alpha (élément de E1)
    for e1 in E1:
        if e1 not in E2:
            D.add(e1)

    return D
```

```python
# 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()
```

```python
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(E1, E2)`

s'écrit plus simplement (et à nouveau de façon préférée) :

`E1 - E2`

Par exemple :

```python
>>> {'a', 'i', 'o', 'y'} - {'u', 'i', 'o'}
{'a', 'y'}
```

```python
>>> { 2, 5, 9 } - {2, 3, 4, 8}
{5, 9}
```

```python
>>> { 2, 5, 9 } - {2, 3, 5, 8}
{9}
```

```python
>>> { 2, 5, 9 } - {2, 3, 5, 8, 9}
set()
```

9.2 Les dictionnaires

Les dictionnaires sont sans doute, avec les listes, les structures de données les plus couramment utilisées en Python.

9.2.1 Définition et opérations de base

Un dictionnaire Python exprime une relation entre un ensemble de *clés* - dites *de recherche* - et un ensemble de *valeurs*. Cette relation est mathématiquement une *application* (en anglais un *mapping*) qui impose qu'à chaque clé corresponde exactement une valeur. On dit que le couple formé d'une clé et de sa valeur associée forme une *association*.

Le type des dictionnaires Python est défini ci-dessous.

Définition : un **dictionnaire** de type `dict[`α:β`]` est un ensemble d'assocations entre :

— une **clé** (ou *clé de recherche*) de type α
— une **valeur** de type β 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 la première partie du cours, nous y reviendrons.

Important : puisque les clés d'un dictionnaire forment un ensemble, le type α de `dict[`α:β`]` doit être un type valide pour les ensembles. 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.

9.2.1.1 Construction explicite d'un dictionnaire

La construction explicite d'un dictionnaire consiste à énumérer l'ensemble des associations clé/valeur.

Syntaxe de la construction explicite d'un dictionnaire `D` de type `dict[`α`:`β`]` :

`{`k_1`:`v_1`,`k_2`:`v_2`,`$\ldots$`,` k_n`:`v_n`}`

où

— tous les k_1, …, k_n sont des expressions pour les clés de type α.
— tous les v_1, …, v_n sont des expressions pour les valeurs associées de type β.

Pour illustrer cette syntaxe, construisons un dictionnaire (très partiel) des constantes mathématiques :

```
>>> {'pi' : 3.14159265, 'sqrt2' : 1.41421356237, 'phi' : 1.6180339887 }
{'sqrt2': 1.41421356237, 'phi': 1.6180339887, 'pi': 3.14159265}
```

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, dans ce cours 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).

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.

```
# Constantes : dict[str:float]
Constantes =  {'pi' : 3.14159265,
               'sqrt2' : 1.41421356237,
               'phi' : 1.6180339887 }
```

Remarque : il serait possible d'«imiter» un dictionnaire Python de type `dict[`α`:`β`]` avec une liste de type `list[tuple[`α`,`β`]]`. Mais en pratique on ne le fait pas car cette imitation de dictionnaire serait très inefficace.

9.2.1.2 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[α:β]
- <clé> est une expression de type α

Le principe d'évaluation est le suivant :

- si la <clé> est présente dans le <dictionnaire>, alors la valeur associée de type β est retournée
- sinon, si la <clé> n'est pas présente, alors une erreur est signalée par l'interprète Python.

Remarque : il faudra donc 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).

9.2.1.3 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 cours) 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[`α`:`β`]`
- `<clé>` est une expression de type α

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.

9.2.1.4 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[`α`:`β`]`
- `<clé>` est une expression de type α
- `<valeur>` est une expression de type β

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
{'sqrt2': 1.41421356237, 'pi': 3.14159265, '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
{'sqrt2': 1.41421356237,
 'e': 2.71828182,
 'pi': 3.14159265,
 'phi': 1.6180339887}
```

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
{'sqrt2': 1.41421356237,
 'e': 2.71828182,
 'pi': 3.141592653589793,
 'phi': 1.6180339887}
```

La valeur associée à la clé `'pi'` a bien été remplacée.

9.2.1.4.1 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', 'e', 'b', 'c', 'j', 'd', 'b', 'j', 'a', 'b']
```

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', 'e', 'b', 'c', 'j', 'd', 'b', 'j', 'a', 'b'])
{'b': 4, 'a': 1, 'c': 2, 'e': 1, 'j': 2, 'd': 1}
```

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 entre les éléments (ici des assocations) est arbitraire.

La définition proposée pour la fonction de comptage d'occurrences est la suivante :

```
def compte_occurrences(L):
    """ list[str] -> dict[str:int]

    retourne le compte des occurrences des éléments
    de la liste L."""

    # D : dict[str:int]
    D = dict()  # le dictionnaire résultat

    # e : str (élément courant)
    for e in L:
        if e not in D: # première occurrence
            D[e] = 1
        else:  # k-ième occurrence, k > 1
            D[e] = D[e] + 1

    return D
```

```
# 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()
```

Question : la signature de la fonction `compte_occurrences` est-elle la plus générale possible ?

9.2.1.5 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[`α`:`β`]`

— <clé> est une expression de type α

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
{'sqrt2': 1.41421356237,
 'e': 2.71828182,
 'pi': 3.141592653589793,
 'phi': 1.6180339887}
```

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
{'e': 2.71828182, 'pi': 3.141592653589793, 'phi': 1.6180339887}
```

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.

9.2.2 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.

9.2.2.1 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 :

```
for <var> in <dictionnaire>:
    <corps>
```

où :

- `<dictionnaire>` est une expression de dictionnaire de type `dict[`α`:`β`]`
- `<var>` est la *variable d'itération de clé* de type α
- `<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[alpha, beta]` et retourne un ensemble de type `set[alpha]`.

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 :

```
def ensemble_des_cles(D):
    """dict[alpha:beta] -> set[alpha]

    Renvoie l'ensemble des clés du dictionnaire D."""

    # K : set[alpha]
    K = set()

    # k : alpha (clé courante)
    for k in D:
        K.add(k)

    return K
```

```
# Jeu de tests
assert ensemble_des_cles({'a':1, 'z':26, 't':20, 'q':17}) \
      == {'a', 'q', 't', 'z'}
```

```python
assert ensemble_des_cles({'e': 2.71828182,
                          'pi': 3.141592653589793,
                          'phi': 1.6180339887}) == {'e', 'pi', 'phi'}

assert ensemble_des_cles(dict()) == set()
```

9.2.2.2 Itération sur les associations Dans des cas un peu moins fréquents, on peut s'intéresser au parcours des associations clé/valeur dans un dictionnaire. Pour cela, on peut utiliser la syntaxe suivante :

```
for (<cvar>, <vvar>)  in <dictonnaire>.items():
     <corps>
```

où :

— `<dictionnaire>` est une expression de dictionnaire de type `dict[`α`:`β`]`
— `<cvar>` est la *variable d'itération de clé* de type α
— `<vvar>` est la *variable d'itération de valeur* de type β
— `<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_valeurs` prend en paramètre un dictionnaire de type `dict[alpha, beta]` et retourne une liste de type `list[beta]`.

Par exemple :

```python
>>> liste_des_valeurs({'a':1, 'b':2, 'c':2, 'd':3})
[3, 2, 1, 2]
```

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 :

```python
def liste_des_valeurs(D):
    """dict[alpha:beta] -> list[beta]

    Renvoie la liste des valeurs du dictionnaire D."""
```

```python
    # L : list[beta]
    L = []

    # k : alpha
    # v : beta
    for (k, v) in D.items():
        L.append(v)

    return L
```

```python
# Jeu de tests
assert liste_des_valeurs({'a':1, 'b':2, 'c':2, 'd':3}) == [3, 2, 1, 2]
assert liste_des_valeurs(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 a de fortes chances de changer. Ceci illustre que la récupération des valeurs d'un dictionnaire n'est pas une opération triviale.

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_valeurs` :

```python
def liste_des_valeurs(D):
    """dict[alpha:beta] -> list[beta]

    Renvoie la liste des valeurs du dictionnaire D."""

    # L : list[beta]
    L = []

    # k : alpha   (clé courante)
    for k in D:
        L.append(D[k])

    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).

9.3 Exercices corrigés

Exercice (9.1) : Différence symétrique (corrigé)

Dans le cours, nous avons introduit :

- l'union de deux ensembles : l'opérateur `|` de Python
- l'intersection de deux ensembles : l'opérateur `&`

— la différence entre deux ensembles : l'opérateur `-`

Dans cet exercice, nous construisons une nouvelle opération ensembliste : la **différence symétrique**.

Question 1

La différence ensembliste est une opération classique de la théorie des ensembles, que l'on peut expliquer de différentes façons. Dans cette première question, nous utilisons la définition suivante :

La **différence symétrique** entre deux ensembles E_1 et E_2, qui est notée $E_1 \triangle E_2$, représente l'ensemble des éléments e tels que :

— soit e appartient à E_1
— soit e appartient à E_2
— mais e ne peut appartenir *simultanément* à E_1 et E_2.

Sans utiliser les opérations ensemblistes prédéfinies, proposer une définition de la fonction `diff_sym` qui construit la différence symétrique entre deux ensembles `E1` et `E2`.

Par exemple :

```
>>> diff_sym({2, 5, 9}, {3, 5, 8})
{2, 3, 8, 9}
```

```
>>> diff_sym({2, 5, 9}, {2, 5, 8, 9})
{8}
```

```
>>> diff_sym({'a', 'b', 'c'}, {'d', 'e', 'f'})
{'a', 'b', 'c', 'd', 'e', 'f'}
```

```
>>> diff_sym({'a', 'b', 'c'}, set())
{'a', 'b', 'c'}
```

```
>>> diff_sym(set(), {'d', 'e', 'f'})
{'d', 'e', 'f'}
```

```
>>> diff_sym({'a', 'b', 'c'}, {'a', 'b', 'c'})
set()
```

Réponse

```
def diff_sym(E1, E2):
    """ set[alpha] * set[alpha] -> set[alpha]
    retourne la différence symétrique entre E1 et E2."""

    # E : set[alpha]
    E = set()  # ensemble résultat

    # e1 : alpha
    for e1 in E1:
        if e1 not in E2:
            E.add(e1)
```

```python
    # e2 : alpha
    for e2 in E2:
        if e2 not in E1:
            E.add(e2)

    return E
```

```python
# Jeu de tests
assert diff_sym({2, 5, 9}, {3, 5, 8}) == {2, 3, 8, 9}
assert diff_sym({2, 5, 9}, {2, 5, 8, 9}) == {8}
assert diff_sym({'a', 'b', 'c'}, {'d', 'e', 'f'}) \
       == {'a', 'b', 'c', 'd', 'e', 'f'}
assert diff_sym({'a', 'b', 'c'}, set()) == {'a', 'b', 'c'}
assert diff_sym(set(), {'d', 'e', 'f'}) == {'d', 'e', 'f'}
assert diff_sym({'a', 'b', 'c'}, {'a', 'b', 'c'}) == set()
```

Question 2

Proposer une seconde définition de la fonction `diff_sym` en exploitant directement la *propriété de différence symétrique* :

$$E_1 \triangle E_2 = (E_1 \setminus E_2) \cup (E_2 \setminus E_1)$$

Remarque : on utilisera bien sûr les opérateurs ensemblistes prédéfinis par Python.

Question : quelle définition de la fonction `diff_sym` est-elle selon-vous la plus efficace ?

Réponse

```python
def diff_sym(E1, E2):
    """ set[alpha] * set[alpha] -> set[alpha]
    retourne la différence symétrique entre E1 et E2."""

    return (E1 - E2) | (E2 - E1)
```

```python
# Jeu de tests
assert diff_sym({2, 5, 9}, {3, 5, 8}) == {2, 3, 8, 9}
assert diff_sym({2, 5, 9}, {2, 5, 8, 9}) == {8}
assert diff_sym({'a', 'b', 'c'}, {'d', 'e', 'f'}) \
       == {'a', 'b', 'c', 'd', 'e', 'f'}
assert diff_sym({'a', 'b', 'c'}, set()) == {'a', 'b', 'c'}
assert diff_sym(set(), {'d', 'e', 'f'}) == {'d', 'e', 'f'}
assert diff_sym({'a', 'b', 'c'}, {'a', 'b', 'c'}) == set()
```

Concernant l'efficacité, ici on a besoin de parcourir deux fois chaque ensemble, alors qu'un seul parcours suffit dans la première définition.

Exercice (9.3) : Répétitions dans les listes (corrigé)

L'analyse des répétitions dans des séquences comme les listes représente un cas d'utilisation typique des ensembles.

Question 1

Donner une définition de la fonction `repetes` qui, étant donné une liste `L`, retourne l'ensemble des éléments répétés au moins une fois dans cette liste.

Par exemple :

```
>>> repetes([1, 2, 23, 9, 2, 23, 6, 2, 9])
{2, 9, 23}
```

```
>>> repetes([1, 2, 3, 4])
set()
```

```
>>> repetes(['bonjour', 'ça', 'ça', 'va', '?'])
{'ça'}
```

Remarque : on supposera que le type des éléments de la liste `L` est compatible avec les ensembles (mais on n'écrira pas d'hypothèse correspondante).

Réponse

```
def repetes(L):
    """ list[alpha] -> set[alpha]
    Retourne l'ensemble des éléments répétés dans L."""

    # Vu : set[alpha]
    Vu = set()  # éléments déjà vus

    # R : set[alpha]
    R = set()  # ensemble résultat

    # e : alpha
    for e in L:
        if e in Vu:  # déjà vu ?
            R.add(e) # répétition
        else:
            Vu.add(e)

    return R
```

```
# Jeu de tests
assert repetes([1, 2, 23, 9, 2, 23, 6, 2, 9]) == {2, 9, 23}
assert repetes([1, 2, 3, 4]) == set()
assert repetes(['bonjour', 'ça', 'ça', 'va', '?']) == {'ça'}
```

Question 2

Donner une définition de la fonction `sans_repetes` qui étant donné une liste `L` retourne cette même liste `L` sans les répétitions éventuelles d'éléments. Votre fonction doit cependant concerver l'ordre dans lequel la première occurrence de chaque élément apparaît dans la liste `L`.

Par exemple :

```
>>> sans_repetes([1, 2, 23, 9, 2, 23, 6, 2, 9])
[1, 2, 23, 9, 6]
```

```
>>> sans_repetes([1, 2, 3, 4])
[1, 2, 3, 4]
```

```
>>> sans_repetes([2, 1, 2, 1, 2, 1, 2])
[2, 1]
```

```
>>> sans_repetes(['bonjour', 'ça', 'ça', 'va' , '?'])
['bonjour', 'ça', 'va', '?']
```

Réponse

```
def sans_repetes(L):
    """ list[alpha] -> list[alpha]
    Retourne la liste des éléments de L sans
    leurs répétitions."""

    # Vu : set[alpha]
    Vu = set()  # éléments déjà vus

    # LR : list[alpha]
    LR = []  # liste résultat

    # e : alpha
    for e in L:
        if e not in Vu:
            LR.append(e)
            Vu.add(e)

    return LR
```

```
# Jeu de tests
assert sans_repetes([1, 2, 23, 9, 2, 23, 6, 2, 9]) == [1, 2, 23, 9, 6]
assert sans_repetes([1, 2, 3, 4]) == [1, 2, 3, 4]
assert sans_repetes([2, 1, 2, 1, 2, 1, 2]) == [2, 1]
assert sans_repetes(['bonjour', 'ça', 'ça', 'va' , '?']) \
      == ['bonjour', 'ça', 'va', '?']
```

Question 3

Donner une définition de la fonction **`uniques`** qui, étant donné une liste `L`, retourne l'ensemble des éléments apparaissant exactement une fois dans cette liste.

Par exemple :

```
>>> uniques([1, 2, 23, 9, 2, 23, 6, 2, 1])
{6, 9}
```

```
>>> uniques([1, 2, 1, 1])
{2}
```

```
>>> uniques([1, 2, 1, 2, 1])
set()
```

Attention : votre fonction doit uniquement utiliser des ensembles et non des dictionnaires pour enregistrer les répétitions.

Réponse

```
def uniques(L):
    """ list[alpha] -> set[alpha]
    retourne l'ensemble des éléments apparaissant
    une seule fois dans L."""

    # Unefois : set[alpha]
    Unefois = set()   # éléments vus au moins une fois

    # Trop : set[alpha]
    Trop = set() # éléments vus plus d'une fois

    # e : alpha
    for e in L:
        if e in Unefois:
            # vu au moins 2 fois
            Trop.add(e)
        else:
            # vu pour la première fois
            Unefois.add(e)

    return Unefois - Trop
```

```
# Jeu de tests
assert uniques([1, 2, 23, 9, 2, 23, 6, 2, 1]) == {6, 9}
assert uniques([1, 2, 1, 1]) == {2}
assert uniques([1, 2, 1, 2, 1]) == set()
```

Question 4

Donner une définition de la fonction `frequences` telle que `frequences(L)` retourne le dictionnaire des fréquences des éléments de `L`, c'est à dire un dictionnaire dont les clés sont les éléments de `L` et qui leur associe comme valeur le nombre de fois où ils apparaîssent dans `L`.

Par exemple :

```
>>> frequences([])
{}
```

```
>>> frequences([2])
{2: 1}
```

```
>>> frequences([2, 2, 2])
{2: 3}
```

```
>>> frequences([1, 2, 23, 9, 2, 23, 6, 2, 9])
{1: 1, 2: 3, 9: 2, 6: 1, 23: 2}
```

Réponse

```
def frequences(L):
    """ list[alpha] -> dict[alpha:int]
    Retourne le dictionnaire des fréquences
    des éléments de L."""

    # Freqs : dict[alpha:int]
    Freqs = dict()  # dictionnaire des fréquences

    # e : alpha
    for e in L:
        if e in Freqs:
            Freqs[e] = Freqs[e] + 1
        else:
            Freqs[e] = 1

    return Freqs
```

```
# Jeu de tests
assert frequences([1, 2, 23, 9, 2, 23, 6, 2, 9]) \
       == {1:1, 2:3, 23:2, 9:2, 6:1}
assert frequences([]) == dict()
```

Question 5

À partir de la fonction `frequences`, donner une définition de la fonction `repetes_fois` telle que `repetes(k, L)` retourne l'ensemble des éléments répétés `k` fois dans `L` (pour `k > 0`).

Par exemple :

```
>>> repetes_fois(1, [1, 2, 23, 9, 2, 23, 6, 2, 9])
{1, 6}
```

```
>>> repetes_fois(2, [1, 2, 23, 9, 2, 23, 6, 2, 9])
{9, 23}
```

```
>>> repetes_fois(3, [1, 2, 23, 9, 2, 23, 6, 2, 9])
{2}
```

```
>>> repetes_fois(4, [1, 2, 23, 9, 2, 23, 6, 2, 9])
set()
```

Réponse

```
def repetes_fois(k, L):
    """ int * list[alpha] -> set[alpha]
```

```
    Hypothèse : k > 0
    Retourne l'ensemble des éléments répétés
    exactement k fois dans L."""

    # Freqs : dict[alpha:int]
    Freqs = frequences(L)  # dictionnaire des fréquences

    # E : set[alpha]
    E = set()  # ensemble résultat

    # e : alpha
    for e in Freqs:
        if Freqs[e] == k:
            E.add(e)

    return E
```

```
# Jeu de tests
assert repetes_fois(1, [1, 2, 23, 9, 2, 23, 6, 2, 9]) == {1, 6}
assert repetes_fois(2, [1, 2, 23, 9, 2, 23, 6, 2, 9]) == {23, 9}
assert repetes_fois(3, [1, 2, 23, 9, 2, 23, 6, 2, 9]) == {2}
assert repetes_fois(4, [1, 2, 23, 9, 2, 23, 6, 2, 9]) == set()
```

Exercice (9.4) : Recettes de cuisine (corrigé)

Dans cet exercice, on s'intéresse à la définition de fonctions permettant de manipuler un livre de recettes de cuisine. Comme tout bon livre de recettes qui se respecte, chaque recette décrit notamment l'ensemble des ingrédients qui la composent. À titre d'exemple, un livre de recettes de desserts pourrait contenir les informations suivantes :

Recette	Ingrédients
Gâteau au chocolat	chocolat, oeuf, farine, sucre, beurre
Gâteau au yaourt	yaourt, oeuf, farine, sucre
Crêpes	oeuf, farine, lait
Quatre-quarts	oeuf, farine, beurre, sucre
Kouign amann	farine, beurre, sucre

Un livre de recettes est donc représenté en Python par un dictionnaire de type `dict[str : set[str]]`

— les noms des recettes, de type `str`, comme clés
— l'ensemble des ingrédients, de type `set[str]`, comme valeurs associées.

Ainsi, l'exemple précédent donnerait le dictionnaire `Dessert` suivant :

```
Dessert = {
       'gateau chocolat' : {'chocolat', 'oeuf', 'farine', 'sucre', 'beurre'},
```

```python
       'gateau yaourt' : {'yaourt', 'oeuf', 'farine', 'sucre'},
       'crepes' : {'oeuf', 'farine', 'lait'},
       'quatre-quarts' : {'oeuf', 'farine', 'beurre', 'sucre'},
       'kouign amann' : {'farine', 'beurre', 'sucre'}
       }
```

Dans la suite de cet exercice, on pourra utiliser l'alias de type `Recette` pour `dict[str : set[str]]`

Question 1

Donner une définition de la fonction `nb_ingredients` qui, étant donnés un livre de recettes `D` et le nom d'une recette `r` **contenue dans** `D`, renvoie le nombre d'ingrédients nécessaires à la recette `r`.

Par exemple:

```python
>>> nb_ingredients(Dessert, 'crepes')
3
```

```python
>>> nb_ingredients(Dessert, 'gateau chocolat')
5
```

Réponse

```python
# type Recette = dict[str:set[str]]

def nb_ingredients(D, r):
    """ Recette * str -> int
        Hyp: r in D

        renvoie le nombre d'ingredients necessaires à la recette r"""

    return len(D[r])
```

```python
# Jeu de tests

# Livre de recettes de dessert
# Dessert : Recette
Dessert = {
       'gateau chocolat' : {'chocolat', 'oeuf', 'farine', 'sucre', 'beurre'},
       'gateau yaourt' : {'yaourt', 'oeuf', 'farine', 'sucre'},
       'crepes' : {'oeuf', 'farine', 'lait'},
       'quatre-quarts' : {'oeuf', 'farine', 'beurre', 'sucre'},
       'kouign amann' : {'farine', 'beurre', 'sucre'}}

assert nb_ingredients(Dessert, 'crepes') == 3
assert nb_ingredients(Dessert, 'gateau chocolat') == 5
```

Question 2

Donner une définition de la fonction `recette_avec` qui, étant donnés un livre de recettes `D` et le nom d'un ingrédient `i`, renvoie l'ensemble des recettes qui utilisent cet ingrédient.

Par exemple :

```
>>> recette_avec(Dessert, 'beurre')
{'gateau chocolat', 'kouign amann', 'quatre-quarts'}
```

```
>>> recette_avec(Dessert, 'lait')
{'crepes'}
```

```
>>> recette_avec(Dessert, 'fraise')
set()
```

Réponse

```
def recette_avec(D, i):
    """ Recette * str -> set[str]

        renvoie l'ensemble des recettes de D qui utilisent l'ingrédient i"""

    # SR : set[str]
    SR = set()

    # r : str
    for r in D:
        if i in D[r]:
            SR.add(r)

    return SR
```

```
# Jeu de tests
assert recette_avec(Dessert, 'beurre') == {'kouign amann',
                                           'quatre-quarts', 'gateau chocolat'}
assert recette_avec(Dessert, 'lait') == {'crepes'}
assert recette_avec(Dessert, 'fraise') == set()
```

Question 3

Donner une définition de la fonction `tous_ingredients` qui, étant donné un livre de recettes `D`, renvoie l'ensemble de tous les ingrédients apparaissant au moins une fois dans une recette de `D`.

Par exemple :

```
>>> tous_ingredients(Dessert)
{'beurre', 'chocolat', 'farine', 'lait', 'oeuf', 'sucre', 'yaourt'}
```

Réponse

```
def tous_ingredients(D):
    """ Recette -> set[str]

        renvoie l'ensemble des ingrédients apparaissant dans une recette de D"""

    # SR : set[str]
    SR = set()
```

```python
    # r : str
    for r in D:
        # i : str
        for i in D[r]:
            SR.add(i)

    return SR
```

```python
# Jeu de tests
assert tous_ingredients(Dessert) == {'farine', 'lait', 'yaourt', 'beurre',
                                     'chocolat', 'sucre', 'oeuf'}
assert tous_ingredients(dict()) == set()
```

Variante utilisant l'union :

```python
def tous_ingredients_union(D):
    """ Recette -> set[str]

        renvoie l'ensemble des ingrédients apparaissant dans une recette de D"""

    # SR : set[str]
    SR = set()

    # r : str
    for r in D:
        SR = SR | D[r]

    return SR
```

```python
# Jeu de tests
assert tous_ingredients_union(Dessert) == {'farine', 'lait', 'yaourt', 'beurre',
                                           'chocolat', 'sucre', 'oeuf'}
assert tous_ingredients_union(dict()) == set()
```

Question 4

Tout livre de recettes contient une *table des ingrédients* permettant d'associer à chaque ingrédient l'ensemble des recettes qui l'utilisent. Une telle table est représentée en Python par le type `dict[str : set[str]]` dans lequel une clé est un ingrédient dont la valeur associée est l'ensemble des recettes qui l'utilisent.

Donner une définition de la fonction `table_ingredients` qui, étant donné un livre de recettes `D`, renvoie la table des ingrédients associée.

Par exemple :

```python
>>> table_ingredients(Dessert)
{'lait': {'crepes'},
 'chocolat': {'gateau chocolat'},
 'beurre': {'gateau chocolat', 'kouign amann', 'quatre-quarts'},
 'yaourt': {'gateau yaourt'},
 'oeuf': {'crepes', 'gateau chocolat', 'gateau yaourt', 'quatre-quarts'},
 'farine': {'crepes',
```

```
  'gateau chocolat',
  'gateau yaourt',
  'kouign amann',
  'quatre-quarts'},
 'sucre': {'gateau chocolat',
  'gateau yaourt',
  'kouign amann',
  'quatre-quarts'}}
```

Réponse

```
def table_ingredients(D):
    """ Recette -> dict[str : set[str]]

        renvoie le dictionnaire associant à chaque ingredient l'ensembles recettes associees"""

    # DR : dict[str : set[str]}
    DR = dict()

    # ingredients : set[str]
    ingredients = tous_ingredients(D)

    # i : str
    for i in ingredients:
        DR[i] = recette_avec(D,i)

    return DR
```

```
# Jeu de tests
assert table_ingredients(Dessert) == {'lait': {'crepes'},
                                      'beurre': {'gateau chocolat', 'quatre-quarts',
                                                 'kouign amann'},
                                      'oeuf': {'gateau chocolat', 'quatre-quarts',
                                               'crepes', 'gateau yaourt'},
                                      'yaourt': {'gateau yaourt'},
                                      'sucre': {'kouign amann', 'gateau chocolat',
                                                'quatre-quarts', 'gateau yaourt'},
                                      'farine': {'kouign amann', 'gateau chocolat',
                                                 'quatre-quarts', 'crepes', 'gateau yaourt'},
                                      'chocolat': {'gateau chocolat'}}
assert table_ingredients(dict()) == dict()
```

Question 5

Donner une définition de la fonction `ingredient_principal` qui, étant donné un livre de recettes `D`, renvoie le nom de l'ingrédient utilisé par le plus grand nombre de recettes. On supposera ici que `D` contient au moins une recette.

Par exemple :

```python
>>> ingredient_principal(Dessert)
'farine'
```

Réponse

```python
def ingredient_principal(D):
    """ Recette -> str
        Hyp : len(D) > 0

        renvoie le nom de l'ingredient le plus utilise"""

    # table : dict[str : set[str]]
    table = table_ingredients(D)

    # ires : str
    ires = ''     # ingredient le plus utilise

    # n : int
    n = 0 # nombre de recettes dans lesquelles l'ingredient
          # le plus utilise apparait

    # i : str
    for i in table:
        if len(table[i]) > n:
            ires = i
            n = len(table[i])

    return ires
```

```python
# Jeu de tests
assert ingredient_principal(Dessert) == 'farine'
assert ingredient_principal({'A' : {'a', 'b'},
                             'B' : {'a', 'c'}}) == 'a'
assert ingredient_principal({'A' : {'a', 'b'},
                             'B' : {'a', 'b'}}) in {'a', 'b'}
```

Question 6

Certaines personnes sont allergiques à certains ingrédients. On aimerait donc pouvoir ne conserver d'un livre de recettes que celles qui n'utilisent pas un ingrédient donné.

Donner une définition de la fonction `recettes_sans` qui, étant donnés un livre de recettes `D` et un ingrédient `i`, renvoie un nouveau livre de recettes ne contenant que des recettes de `D` n'utilisant pas l'ingrédient `i`.

Par exemple :

```python
>>> recettes_sans(Dessert, 'farine')
{}
```

```python
>>> recettes_sans(Dessert, 'oeuf')
{'kouign amann': {'beurre', 'farine', 'sucre'}}
```

```python
>>> recettes_sans(Dessert, 'beurre')
{'gateau yaourt': {'farine', 'oeuf', 'sucre', 'yaourt'},
 'crepes': {'farine', 'lait', 'oeuf'}}
```

Réponse

```python
def recettes_sans(D, i):
    """ Recette * str -> Recette

        renvoie un livre de recettes n'utilisant pas l'ingrédent i"""

    # DR : dict[str : set[str]]
    DR = dict()

    # r : str
    for r in D:
        if i not in D[r]:
            DR[r] = D[r]

    return DR
```

```python
# Jeu de tests
assert recettes_sans(Dessert, 'beurre') == {'gateau yaourt' : {'yaourt', 'oeuf',
                                                               'farine', 'sucre'},
                                            'crepes' : {'oeuf', 'farine', 'lait'}}
assert recettes_sans(Dessert, 'oeuf') == {'kouign amann': {'farine', 'beurre', 'sucre'}}
assert recettes_sans(Dessert, 'farine') == {}
```

10 Compréhensions d'ensembles et de dictionnaires

Les constructions par compréhensions d'ensembles et de dictionnaires (sans oublier les listes) forment le fil conducteur de ce cours.

10.1 Notion d'itérable

Dans les compréhensions sur les listes, 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 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 lors du prochain cours.

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 de la même façon aux itérables.

Dans le cours 8, nous avons vu beaucoup de constructions de listes par compréhensions sur des séquences. Illustrons maintenant des constructions à partir d'autres itérables, par exemple à partir d'un ensemble :

```
>>> [k*k for k in {2, 4, 8, 16, 32, 64}]
[4096, 1024, 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.

On peut de façon plus intéressante 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_valeurs` du dernier cours, que l'on peut maintenant définir de façon très concise.

```
def liste_des_valeurs(D):
    """dict[alpha:beta] -> list[beta]

    Renvoie la liste des valeurs du dictionnaire D."""

    return [D[k] for k in D]    # ou [v for (k,v) in D.items()]
```

Par exemple :

```
>>> liste_des_valeurs({'a':1, 'b':2, 'c':2, 'd':3})
[2, 2, 3, 1]
```

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.

10.2 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.

10.2.1 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_for(L):
    """ list[alpha] -> set[alpha]
    Retourne l'ensemble des éléments apparaissant
```

```
    dans la liste L."""

    # E : set[alpha]
    E = set()  # ensemble résultat

    # e : alpha
    for e in L:
        E.add(e)

    return E
```

```
# Jeu de tests
assert elements_for([1, 3, 5, 5, 7, 9, 1, 11, 13, 13]) \
       == {1, 3, 5, 7, 9, 11, 13}
assert elements_for([1, 1, 1, 1, 1]) == {1}
assert elements_for([]) == 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, le type de la variable de compréhension <var> n'est pas déclaré. Ce type peut cependant être déduit du type de <iterable>.

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[alpha] -> set[alpha]
    Retourne l'ensemble des éléments apparaissant
    dans la liste L."""

    return {e for e in L}
```

```
# 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()
```

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.).

10.2.2 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 :

$$\{x \in E \mid P(x)\}$$

où :

— E est un ensemble
— P est une proposition

Cette construction décrit l'ensemble composé des seuls éléments x de E pour lesquels la proposition P est vraie 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 $[1; 10]$:

```python
>>> { 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.

10.2.2.1 Exemple : personnes célibataires

Dans le cours 7 nous avons manipulé des bases de données représentées par des listes de personnes, en utilisant l'alias de type:

```python
# type Personne = tuple[str, str, int, bool]
```

Prenons la base de données suivante :

```python
# 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, prenom, age, marie) in BD if not marie }
{'Diaprem', 'Ltar', 'Unfor'}
```

Ceci permet de proposer une définition très simple de la fonction :

```
def nom_celibataires(Personnes):
    """ list[Personne] -> set[str]
    Retourne l'ensemble des noms des Personnes
     célibataires."""

    return { nom for (nom, prenom, age, 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, age, 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, age, 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')}
```

10.2.3 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 cours précédent 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 `list` apparaît bien dans le 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 cours, 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`).

10.3 Compréhensions de dictionnaires

Les dictionnaires peuvent également être construits par compréhension, en indiquant ce que sont leurs clés et les valeurs associées.

10.3.1 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>`

10.3.1.1 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[`α`:`β`]` à partir d'une liste de type `list[tuple[`α`,`β`]]`.

Voici un exemple d'une telle construction :

```
>>> { nom:age for (nom, age) in [('dupont',41), ('dupond',27), ('dupons',31)]}
{'dupons': 31, 'dupond': 27, '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)]}
{'dupont': 13, 'dupons': 31, 'dupond': 27}
```

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, prenom, age, marie) 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.

10.3.1.2 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, prenom, age, marie) ) 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 cours 9.

Regardons la valeur de la variable `DBDIndex` :

```
>>> DBDIndex
{'Diaprem': 146,
 'Ltar': 22215,
 'Laveur': 4417,
 'Itik': 331,
 'Unfor': 282,
 'Potteuse': 14}
```

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)
```

10.3.1.3 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`.

Exercice : traduire la compréhension ci-dessus sous la forme d'une fonction utilisant des boucles `for` d'itération.

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.

10.3.2 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.

10.3.2.1 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', 8: 'huit', 2: 'deux', 4: 'quatre', 6: 'six'}
```

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', 9: 'neuf', 5: 'cinq', 7: 'sept'}
```

```
>>> { 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'}
```

10.3.2.2 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 (num, (nom, prenom, age, marie) ) in DBD.items() if marie }
{'Potteuse': 22, 'Itik': 17, 'Laveur': 38}
```

Si on veut plutôt les nom et numéros des personnes célibataires de plus de 20 ans, on pourra écrire :

```
>>> { nom:num for (num, (nom, prenom, 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 (num, (nom, prenom, age, marie)) in DBD.items()
               if age >= 30 }
{'Laveur', 'Unfor'}
```

10.4 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.

Finalement, il faut retenir 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.

A l'issue de ce cours, il faut :

— savoir effectuer des compréhensions d'ensembles : avec ou sans filtrage
— savoir effectuer des compréhensions de dictionnaires
  — à partir d'un itérable
  — à partir de plusieurs itérables en dissociant la construction des clés et des valeurs
  — avec filtrage sur les clés
  — avec filtrage sur les valeurs

10.5 Exercices corrigés

Exercice (10.1) : Compréhensions en vrac (corrigé)

Cet exercice, qui doit être fait sur papier et non sur machine, propose de manipuler les expressions de compréhensions de listes, d'ensembles et de dictionnaires.

Soient les variables suivantes :

```python
# Liste : list[int]
Liste = [ 2, 5, 12, 31, 2, 17, 31, 42, 2 ]
```

```python
# Dico : dict[str:str]
Dico = {'xx':'bli', 'yzy':'blo', 'cuicui':'toutou', 'miaou':'toutou' }
```

```python
# Ens : set[float]
Ens = { 2.7, 4.12, 3.31, 8.29, 1.13, 12.31 }
```

Question 1 : compréhensions de listes

Donner le résultat d'évaluation des expressions suivantes :

1) `[ (k,Dico[k]) for k in Dico ]`
2) `[ (k,v) for (k,v) in Dico.items() ]`
3) `[ Dico[k] for k in Dico ]`
4) `[ v for (k,v) in Dico.items() ]`
5) `import math`
   `[ math.ceil(v) for v in Ens if v < 5.0 ]`

Réponse

```
>>> [ (k,Dico[k]) for k in Dico ]
[('miaou', 'toutou'), ('xx', 'bli'), ('cuicui', 'toutou'), ('yzy', 'blo')]
```

```
>>> [ (k,v) for (k,v) in Dico.items() ]
[('miaou', 'toutou'), ('xx', 'bli'), ('cuicui', 'toutou'), ('yzy', 'blo')]
```

```
>>> [ Dico[k] for k in Dico ]
['toutou', 'bli', 'toutou', 'blo']
```

```
>>> [ v for (k,v) in Dico.items() ]
['toutou', 'bli', 'toutou', 'blo']
```

```
>>> import math
>>> [ math.ceil(v) for v in Ens if v < 5.0 ]
[2, 3, 4, 5]
```

Question 2 : compréhensions d'ensembles

Donner le résultat d'évaluation des expressions suivantes :

1) `{ c for c in 'a bac a cab' }`
2) `{ c for c in 'a bac a cab' if c != ' ' }`
3) `{ (k, Dico[k]) for k in Dico }`
4) `{ v for (k,v) in Dico.items() }`
5) `{ Dico[k] for k in Dico }`
6) `{ k + 's' for k in Dico if len(Dico[k]) > 3 }`
7) `{ n % 5 for n in Liste }`
8) `{ n % 10 for n in range(0, 20) }`

   cas 8) à comparer avec : `[ n % 10 for n in range(0, 20) ]`

Réponse

```
>>> { c for c in 'a bac a cab' }
{' ', 'a', 'b', 'c'}
```

```
>>> { c for c in 'a bac a cab' if c != ' ' }
{'a', 'b', 'c'}
```

```
>>> { (k, Dico[k]) for k in Dico }
{('cuicui', 'toutou'), ('miaou', 'toutou'), ('xx', 'bli'), ('yzy', 'blo')}
```

```
>>> { v for (k,v) in Dico.items() }
{'bli', 'blo', 'toutou'}
```

```
>>> { Dico[k] for k in Dico }
{'bli', 'blo', 'toutou'}
```

```
>>> { k + 's' for k in Dico if len(Dico[k]) > 3 }
{'cuicuis', 'miaous'}
```

```
>>> { n % 5 for n in Liste }
{0, 1, 2}
```

```
>>> { n % 10 for n in range(0, 20) }
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

```
>>> [ n % 10 for n in range(0, 20) ]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Question 3 : Compréhensions de dictionnaires

Donner le résultat d'évaluation des expressions suivantes :

1) `{ k:Dico[k] for k in Dico }`
2) `{ Dico[k]:k for k in Dico }`
3) `{ (v + v):k for (k, v) in Dico.items() }`
4) `{ (Dico[k] + Dico[k]):k for k in Dico }`
5) `{ k:v for (k, v) in zip(range(1, 8), Liste)}`
6) `{ k:v for (k, v) in zip(range(1, 10), Liste) if v > 15 }`
7) `{ k:v for k in range(1, 10) for v in Liste if v > 15 }`

Réponse

```
{ k:Dico[k] for k in Dico }
{'yzy': 'blo', 'cuicui': 'toutou', 'miaou': 'toutou', 'xx': 'bli'}
```

```
{ Dico[k]:k for k in Dico }
{'bli': 'xx', 'blo': 'yzy', 'toutou': 'miaou'}
```

```
{ (v + v):k for (k, v) in Dico.items() }
{'toutoutoutou': 'cuicui', 'blibli': 'xx', 'bloblo': 'yzy'}
```

```
{ (Dico[k] + Dico[k]):k for k in Dico }
{'toutoutoutou': 'cuicui', 'blibli': 'xx', 'bloblo': 'yzy'}
```

```
{ k:v for (k, v) in zip(range(1, 8), Liste)}
{1: 2, 2: 5, 3: 12, 4: 31, 5: 2, 6: 17, 7: 31}
```

```
{ k:v for (k, v) in zip(range(1, 10), Liste) if v > 15 }
{8: 42, 4: 31, 6: 17, 7: 31}
```

```
{ k:v for k in range(1, 10) for v in Liste if v > 15 }
{1: 42, 2: 42, 3: 42, 4: 42, 5: 42, 6: 42, 7: 42, 8: 42, 9: 42}
```

11 Ouverture sur la programmation orienté objet

Ce dernier cours propose de conclure notre apprentissage des concepts élémentaires de programmation par une ouverture sur la *programmation orientée objet*.

11.1 Paradigmes de programmation

Nous avons souvent illustré dans notre cours le fait suivant :

> Un problème (informatique) donné peut être résolu de différentes façons.

En pratique, nous pouvons souvent proposer plusieurs définitions d'une même fonction, c'est-à-dire répondant à la même spécification.

En matière de programmation, ce principe fondamental se retrouve dans la façon même dont on aborde la solution des problèmes. On parle alors de *paradigme de programmation*. Même si cette classification est de moins en moins d'actualité, on a coutume de distinguer deux familles de langages de programmation :

- les langages de programmation procédurale et impérative
- les langages de programmation fonctionnelle et récursive

Les représentants les plus connus de la première famille sont les langages Fortran, Pascal ainsi que le langage C, ce dernier étant encore très utilisé aujourd'hui. Dans la famille des langages de programmation fonctionnelle et récursive, on trouve notamment le langage Scheme (anciennement enseigné à la place de Python) et plus généralement la famille de Lisp, le langage Caml et le langage Haskell.

Mais il est possible de programmer de façon fonctionnelle et récursive en C, Fortran ou Pascal, comme il est possible de programmer de façon procédurale et impérative en Scheme, Caml et Haskell ! De plus, les langages modernes sont pour la plupart du temps dits *multi-paradigmes* donc permettant différents styles de programmation.

Sans nous lancer dans la rédaction d'un dictionnaire terminologique, expliquons certains de ces concepts et illustrons-les en Python.

11.1.1 Programmation impérative

En programmation impérative, les problèmes informatiques que l'on se pose sont résolus par une sorte de *machine* à l'image des ordinateurs. La mémoire de l'ordinateur est représentée par des *variables* dites *globales*. Le processeur de l'ordinateur est représenté par des *suites d'instructions* qui permettent notamment de lire et modifier les valeurs des variables – donc la mémoire de l'ordinateur. Pour permettre les traitements complexes, sont proposées des *structures de contrôles*, en particulier :

- les alternatives comme le `if-else` de Python
- les boucles comme le `while` de Python

Finalement, on dispose de moyens pour saisir de l'information en entrée (en général au clavier ou en lisant un fichier) ainsi que d'outils permettant la sortie (en général à l'écran ou en écrivant un fichier). Ce mode de programmation assez primitif est disponible en Python.

Prenons un exemple : le calcul de la factorielle.

```
print("Calcul de la factorielle n!")
print("===========================")

print("Saisir la valeur de n: ", end='')

entree = int(input())
sortie = 1

while entree > 0:
    sortie = sortie * entree
    entree = entree - 1

print("n! = " + str(sortie))
```

Ce programme commence par demander la saisie au clavier d'un entier `n` que l'on stocke dans la variable `entree`. Le calcul de la factoriel est ensuite effectué par une boucle `while` et le résultat est écrit dans la variable `sortie`. Ce résultat est finalement affiché par une instruction `print`. Voici un exemple d'affichage généré par ce programme :

```
Calcul de la factorielle n!
===========================
Saisir la valeur de n: 5
n! = 120
```

Ce type de programmation, si on le prend au sens strict, est très rudimentaire : il est très (trop) proche de la machine.

11.1.2 La programmation procédurale

Notre exemple de calcul de la factorielle programmé de façon impérative naïve contredit des principes fondamentaux de «bonne programmation»

— le **principe SOC** – *Separation Of Concerns* (séparation des préoccupations)

La description du programme entremêle des aspects différents qu'il faut clairement séparer, notamment :

— l'acquisition des données en entrée
— la réalisation du calcul (pour notre exemple, de la factorielle)
— la production des résultats en sortie

On a vu dans de nombreux exemples – par exemple dans la construction du triangle de *Pascal* – que la partie calcul elle-même devait souvent être décomposée en sous-problèmes plus simples à résoudre.

— le **principe DRY** – *Don't Repeat Yourself* (ne te répète pas)

Il est très fréquent qu'un même calcul soit utile à différents programme, ou même dans différentes parties d'un même programme. Par exemple, le calcul de la factorielle peut être utile dans de nombreux problèmes de dénombrements (combinaisons, arrangements, permutations, etc.). Il est évident que faire du copier/coller à chaque fois que l'on a besoin d'un calcul n'est pas une bonnée idée. Il est d'ailleurs connu en ingénierie logicielle que de nombreux *bugs* proviennent de mauvais copier/coller.

— le **principe KISS** – *Keep It Simple and Stupid* (Restons simples, pour être polis)

Le calcul de la factorielle n'est pas de nature très complexe. En lisant le programme de calcul ci-dessus, il n'est pas très difficile de voir qu'il s'agit effectivement d'un calcul de factorielle. Mais nous avons vu dans ce cours des calculs bien plus complexes – prenons encore une fois en exemple la construction du triangle de Pascal. On a résolu le problème comlexe par une combinaisons de petites fonctions Python dont la compréhension est quasiment immédiate. En abordant le problème de façon impérative naïve la solution obtenue serait très difficile à lire.

Les *programmation procédurale* permet de concilier la programmation impérative avec les principes de bonne programmation évoqués ci-dessus.

Une procédure est un sous-programme informatique composé :

— d'un *nom* permettant de le désigner de façon unique,
— de *paramètres* permettant de généraliser les traitements,
— de *variables locales* offrant une mémoire de calcul temporaire et indépendante,
— d'un *corps de procédure* : une suite d'instructions impérative décrivant les traitements à effectuer.

Puisque chaque procédure est autonome et nommée, cela offre un moyen de séparer les préoccupations. Pour le programme de calcul de la factorielle, on peut notamment définir :

— une procédure `acquerir_entier` permettant d'acquérir le nombre en entrée.
— une procédure `factorielle` effectuant le calcul de la factorielle proprement dit.
— une procédure `afficher_resultat` pour produire les affichages en résultat.
— une procédure `calculer_factorielle` représentant le programme principal.

Les définitions correspondantes sont données ci-dessous.

```python
def acquerir_entier():
    """ -> int
    Acquiert un nombre depuis le clavier."""

    return int(input())
```

```python
def factorielle(n):
    """ int -> int
    Hypothèse : n >= 0 """

    # resultat : int
    resultat = 1

    # k : int
    for k in range(2, n + 1):
```

```
        resultat = resultat * k

    return resultat
```

```
def afficher_resultat(resultat):
    """ int -> NoneType
    Affiche le résultat du calcul de n!."""

    print("n! = " + str(resultat))
```

```
def calculer_factorielle():
    """ -> NoneType
    Programme principal de calcul de la factorielle."""

    print("Calcul de la factorielle n!")
    print("===========================")

    print("Saisir la valeur de n: ", end='')

    entree = acquerir_entier()

    sortie = factorielle(entree)

    afficher_resultat(sortie)
```

Si on a peut-être un peu trop abusé des principes de la programmation procédurale, on peut tout de même affirmer que le programme dans cette nouvelle version est bien plus respectueux des principes de bonne programmation :

— le principe **SOC** est respecté puisque les parties de calcul «pur» et les entrées/sorties sont bien séparées

— le principe **DRY** est également respecté car si on a besoin du calcul de la factorielle (sans les entrées/sorties) à un autre endroit du programme ou dans un autre programme, il suffit d'appeler la procédure `factorielle` depuis cet autre endroit.

— le principe **KISS** est finalement respecté puisque chaque procédure définie est courte et simple à comprendre.

Et bien sûr notre programme fonction toujours, comme l'atteste la trace d'exécution ci-dessous :

```
>>> calculer_factorielle()
Calcul de la factorielle n!
===========================
Saisir la valeur de n: 5
n! = 120
```

11.1.2.1 Procédure vs. Fonction Python Dans le cours, nous avons plutôt parlé de *définition de fonction* et non de procédure. Il s'agit en fait de la terminologie usuelle employée dans de nombreux langages de programmation et notamment Python et le langage C. Il y a cependant une différence importante entre la notion de procédure informatique et la notion de fonction en mathématique :

— la procédure informatique décrit les traitements informatiques à effectuer
— la fonction mathématique correspond à un ensemble de couples (valeur en entrée $\mapsto$ résultat en sortie)

Pour la factorielle, cet amalgame ne pose pas vraiment de problème, car il y a un lien fort entre le résultat de l'expression mathématique $n!$ et le résultat retourné par l'appel `factorielle(n)` en Python. Par exemple, on a ci-dessous deux «verités» :

— une vérité mathématique : $5! = 120$
— une vérité informatique : `factorielle(5) == 120`

La fonction Python suivante est également proche de la notion mathématique de fonction :

```
def frequences(L):
    """ list[alpha] -> dict[alpha:int]
    Retourne le dictionnaire des fréquences
    des éléments de la liste L."""

    # freqs : dict[alpha:int]
    freqs = dict()

    # e : alpha
    for e in L:
        if e in freqs:
            freqs[e] = freqs[e] + 1
        else:
            freqs[e] = 1

    return freqs
```

Ici encore, pour chaque liste passée en argument correspond en résultat un unique dictionnaire que l'on construit pour l'occasion. Si on met de côté la description précise du calcul et que l'on retient uniquement l'ensemble des couples :

$$(\text{liste passée en paramètre} \mapsto \text{dictionnaire de fréquences en résultat})$$

alors on a bien une fonction au sens mathématique du terme.

Les fonctions `factorielle` et `frequences` ont un point commun : elle n'effectuent aucun *effet de bord* en dehors du corps de ces mêmes fonctions. La seule zone mémoire modifiée concerne les *variables locals* aux fonctions. Par exemple, la fonction `factorielle` ne modifie que la valeur de la variable `resultat` qui lui est locale. La case mémoire associée à cette variable est créée lorsque la fonction est appelée. Dans la boucle `while` la valeur contenue dans la case mémoire est modifiée. Mais après le retour de la fonction la case mémoire est détruite. C'est aussi le cas pour la fonction `frequences` avec la variable `freqs` qui est locale à la fonction. Dans les deux cas, aucune de ces modifications mémoires ne sont visible *après* le retour des fonctions.

Ce confinement des effets de bord est une contrainte forte que nous imposons dans ce cours. L'objectif est de rapprocher les fonctions Python que nous écrivons et les fonctions mathématiques.

Considérons une fonction Python qui ne satisfait pas la contrainte.

```python
def supprimer_repetitions(L):
    """ list[alpha] -> NoneType
    Supprime les éléments répétés de la liste L."""

    # freqs : dict[int:int]
    freqs = frequences(L)

    # e : alpha
    for e in L[:]:  # Remarque : il faut copier L
        if freqs[e] > 1:
            L.remove(e)
```

L'objectif de cette fonction Python est de supprimer les répétitions présentes dans la liste `L`. On remarque que le type de retour indiqué est `NoneType`, ce qui est souvent le signe qu'un *effet de bord non-confiné* se produit. De fait, la liste `L` passée en paramètre est directement modifiée en mémoire par la méthode `remove`.

Prenons un exemple de liste :

```python
# Ma_liste : list[int]
Ma_liste = [1, 2, 3, 1, 3, 23, 1, 23, 4, 53]
```

Si nous appliquons la fonction `supprimer_repetitions` sur cette liste, Python n'affiche rien.

```python
>>> supprimer_repetitions(Ma_liste)
```

Cependant, l'invocation de la méthode `remove` dans `L.remove(e)` a pour effet de bord de modifier la liste `L` directement en mémoire, en enlevant l'élément `e` si ce dernier est présent dans la liste. Constatons la modification directe de la liste `Ma_liste` en mémoire.

```python
>>> Ma_liste
[2, 4, 53]
```

Les répétitions ont bien été enlevée de la liste directement en mémoire.

Dans la fonction `supprimer_repetitions` nous avons effectué deux opérations "interdites" dans notre cours :

— le paramètre `L` est modifié, or ce n'est pas une variable locale
— on a utilisé la méthode `remove` qui permet d'enlever la première occurrence d'un élément d'une liste directement en mémoire

Il n'y a pas de problème de principe à autoriser les effets de bord non-confinés pour les fonctions – comme c'est le cas pour `supprimer_repetitions`. Il faut cependant se rendre compte que l'on s'éloigne alors de la notion fondamentale de fonction mathématique. En particulier, le principe de *composition de fonctions* n'es plus appliquable librement. Or c'est justement ce principe fondamental qui supporte la décomposition d'un problème complexe en sous-problèmes indépendants. De plus, l'analyse des programmes et en particulier la découverte des erreurs éventuelles (les fameux *bugs*) devient beaucoup plus complexe si on autorise les effets de bord arbitraires. Il s'agit d'un phénomène bien connu des ingénieurs informaticiens. Des questions essentielles peuvent alors devenir complexe, notamment : «comment tester la fonction ?». De plus, maîtriser les effets de bord nécessite une bonne connaissance du fonctionnement interne de l'ordinateur.

11.1.3 La programmation fonctionnelle

Pour simplifier le modèle de programmation, une première approche assez radicale consiste à tout simplement *bannir* les effets de bord. Plutôt que de s'inspirer de la machine (dites *machine de Von Neuman* du nom d'un des pionniers de l'architecture des ordinateurs) on peut s'inspirer de la discipline mathématique du *calcul effectif*. En particulier, la théorie des *fonctions effectivement calculables* envisage les programmes informatiques comme des fonctions au sens mathématique du terme. Plutôt qu'un long discours, voici une définition de la factorielle de ce point de vue :

$$n! = \left\{ \begin{array}{l} 1 \text{ si } n = 0 \\ n \times (n-1)! \text{ si } n > 0 \end{array} \right.$$

Ici, on ne se contente pas d'expliquer ce qu'est la factorielle, on décrit précisément *comment* la calculer tout en restant dans un univers parfaitement mathématique.

La traduction en Python de cette définition mathématique est d'ailleurs immédiate :

```python
def factorielle(n):
    """ int -> int
    Hypothèse : n >= 0
    Retourne la factorielle de n."""

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

Un avantage certain est que la factorielle est maintenant une fonction à la fois informatique *et* mathématique : on a «réconcilié» les deux disciplines. On y gagne notamment le fait de pouvoir étudier notre fonction d'un point de vue mathématique, en particulier faire des raisonnements par récurrence. Le principal inconvénient de la programmation fonctionnelle est qu'elle est parfois inefficace, notamment dans les langages qui ne l'encouragent pas : c'est le cas de Python ou du langage C. Dans les *langages fonctionnels*, elle est au contraire privilégiée (par exemple dans le langage Ocaml) ou même imposée (par exemple dans le langage Haskell).

La paradigme de programmation que nous suivons dans ce cours est une variante disons de *programmation quasi-fonctionnelle* qui autorise les effets de bord uniquement s'ils sont confinés à l'intérieur des fonctions. Certes les fonctions que nous écrivons ne sont pas mathématiques au sens strict, mais du point de vue extérieur elles sont cohérentes avec leur cousine mathématique : elles génèrent les mêmes couples (valeur en entrée $\mapsto$ résultat en sortie).

Pour illustrer ce point, reprenons la fonction `supprimer_repetitions` qui ne satisfait pas la contrainte de confinement, et essayons d'en déduire une variante «satisfaisante».

```python
def suppression_repetitions(L):
    """ list[alpha] -> list[alpha]
    Retourne une liste composée des éléments de L
    qui ne sont pas répétés."""

    # freqs : dict[int:int]
    freqs = frequences(L)
```

```python
    # LR : list[alpha]
    LR = []

    # e : alpha
    for e in L:  # remarque : pas besoin de copier L
        if freqs[e] == 1:
            LR.append(e)

    return LR
```

Si l'on reprend notre exemple d'interaction avec cette nouvelle fonction.

```python
# Ma_liste : list[int]
Ma_liste = [1, 2, 3, 1, 3, 23, 1, 23, 4, 53]
```

```python
>>> suppression_repetitions(Ma_liste)
[2, 4, 53]
```

Nous constatons une première différence importante. Dans la fonction `supprimer_repetitions` qui fonctionnait par effet de bord, aucun résultat n'était retourné. Ici, on récupère une liste correspondant à celle référencée par la variable `Ma_liste` mais dans laquelle les répétitions ont été supprimées. La deuxième différence importante est que la liste n'a pas été directement modifiée en mémoire. Il est facile de vérifier ce fait :

```python
>>> Ma_liste
[1, 2, 3, 1, 3, 23, 1, 23, 4, 53]
```

Ainsi, la fonction se comporte essentiellement comme une fonction purement mathématique. On obtient donc la plupart des avantages de la vision mathématique :

- simplicité conceptuelle
- possibilité de composer librement les fonctions (à condition de satisfaire les signatures et les hypothèses bien sûr)
- relative facilité pour notamment découvrir les erreurs éventuelles

Pour illustrer ce dernier point, on peut maintenant facilement créer un jeu de tests pour la fonction, ce qui n'était pas le cas de la version non-confinée.

```python
# Jeu de tests pour suppression_repetitions
assert suppression_repetitions([1, 2, 3, 1, 3, 23, 1, 23, 4, 53]) == [2, 4, 53]
assert suppression_repetitions([1, 1, 1, 1]) == []
assert suppression_repetitions([1, 2, 3, 4]) == [1, 2, 3, 4]
assert suppression_repetitions([]) == []
```

Il reste un inconvénient à la programmation fonctionnelle ou quasi-fonctionnelle : l'efficacité. Par exemple, modifier une seule association dans un dictionnaire avec confinement nécessite de le reconstruire entièrement. Par effet de bord sans confinement, on peut effectuer cette même opération en une seule étape. Il existe cependant des techniques pour obtenir une meilleure efficacité tout en confinant les effets de bord. Par exemple, on peut utiliser des *arbres binaires de recherche* pour construire des dictionnaires efficaces et *sans* effets de bord (mêmes confinés). Cela nous conduirait cependant au delà de ce cours d'introduction.

11.1.4 La programmation orientée objet (P.O.O.)

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 cours !

11.2 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.

```python
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 communes* 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.

```python
# 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.

11.3 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 cours, 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.

11.3.1 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 un cours précédent.

```
# 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.

```python
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
```

```python
    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 :

```python
# pers : Personne
pers = Personne('Yoda', 'Yoda', 700, False)
```

```python
>>> pers.nom()
'Yoda'
```

```python
>>> pers.age()
700
```

```python
>>> pers.anniversaire()
```

```python
>>> 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.

11.3.2 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.

```python
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
```

11.3.3 Types itérables

Dans le cours sur les compréhensions d'ensembles et de dictionnaires, 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()
```

```python
        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 :

```python
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
```

```python
# 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 :

```python
>>> [ c for c in CharRange('a', 'e')]
['a', 'b', 'c', 'd', 'e']
```

```python
>>> [ c for c in CharRange('a', 'a')]
['a']
```

```python
>>> [ c for c in CharRange('b', 'a')]
[]
```

11.4 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 cours, 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.

N'hésitez pas à glaner des information sur la page principale du langage :

https://www.python.org/

Après-propos

11.5 Conclusion

Le cours se termine ici, mais votre apprentissage de la programmation en général, et de Python en particulier, ne fait que commencer.

Et bien sûr, il y a de nombreux autres problèmes à résoudres, d'algorithmes à comprendre ou inventer, et de langages de programmation à découvrir !

11.6 Licence Creative Commons CC-BY-SA 4.0

cf. https://creativecommons.org/licenses/by-sa/4.0/deed.fr

Licence publique Creative Commons Attribution - Partage dans les mêmes conditions 4.0 International

Lorsque Vous exercez les Droits accordés par la licence (définis ci-dessous), Vous acceptez d'être lié par les termes et conditions de la présente Licence publique Creative Commons Attribution - Partage dans les mêmes conditions 4.0 International (la « Licence publique »). Dans la mesure où la présente Licence publique peut être interprétée comme un contrat, Vous bénéficiez des Droits accordés par la licence en contrepartie de Votre acceptation des présents termes et conditions, et le Donneur de licence Vous accorde ces droits en contrepartie des avantages que lui procure le fait de mettre à disposition l'Œuvre sous licence en vertu des présents termes et conditions.

Article 1 - Définitions.

Œuvre dérivée signifie œuvre protégée par les Droit d'auteur et droits connexes, dérivée ou adaptée de l'Œuvre sous licence et dans laquelle l'Œuvre sous licence est traduite, retouchée, arrangée, transformée, ou modifiée de telle façon que l'autorisation du Donneur de licence est nécessaire, conformément aux dispositions des Droit d'auteur et droits connexes. Dans le cas de la présente Licence publique, lorsque l'Œuvre sous licence est une œuvre musicale, une représentation publique ou un enregistrement sonore, la synchronisation de l'Œuvre sous licence avec une image animée sera considérée comme une Œuvre dérivée aux fins de la présente Licence publique. Licence d'Œuvre dérivée signifie licence par laquelle Vous accordez Vos Droit d'auteur et droits connexes portant sur Vos contributions à l'Œuvre dérivée, selon les termes et conditions de la présente Licence publique. Licence compatible BY-SA signifie licence figurant à l'adresse suivante creativecommons.org/compatiblelicenses, approuvée par Creative Commons comme étant essentiellement équivalente à la présente Licence publique. Droit d'auteur et droits connexes signifie droit d'auteur et/ou droits connexes incluant, notamment, la représentation, la radio et télédiffusion, l'enregistrement sonore et le Droit sui generis des producteurs de bases de données, quelle que soit la classification ou qualification juridique de ces droits. Dans le cadre de la présente Licence publique, les droits visés à l'Article 2(b)(1)-(2) ne relèvent ni du Droit d'auteur ni de droits connexes. Mesures techniques efficaces signifie mesures techniques qui, en l'absence d'autorisation expresse, ne peuvent être contournées dans le cadre de lois conformes aux dispositions de l'Article 11 du Traité de l'OMPI sur le droit d'auteur adopté le 20 Décembre 1996 et/ou d'accords internationaux de même objet. Exceptions et limitations signifie utilisation loyale et équitable (fair use et fair dealing) et/ou toute autre exception ou limitation applicable à Votre utilisation de l'Œuvre sous licence. Eléments de licence signifie les composantes de la licence figurant dans l'intitulé de la Licence publique Creative Commons. Les éléments de la présente Licence publique sont : Attribution et Partage dans les mêmes conditions. Œuvre sous licence signifie œuvre littéraire ou artistique, base de données ou toute autre œuvre pour laquelle le Donneur de licence a recours à la présente Licence publique. Droits accordés par la licence signifie droits qui Vous sont accordés selon les termes et conditions d'utilisation définis par la présente Licence publique, limités aux Droit d'auteur et droits connexes applicables à Votre utilisation de l'Œuvre sous licence et que le Donneur de licence a le droit d'accorder. Donneur de licence signifie un individu ou une entité octroyant la présente Licence publique et les droits accordés par elle. Partager signifie mettre une œuvre à la disposition du public par tout moyen ou procédé qui requiert l'autorisation découlant des Droits accordés par la licence, tels que les droits de reproduction, de représentation au public, de distribution, de diffusion, de communication ou d'importation, y compris de manière à ce que chacun puisse y avoir accès de l'endroit et au moment qu'il choisit individuellement. Droit sui generis des producteurs de bases de données signifie droits distincts du droit d'auteur résultant de la Directive 96/9/CE du Parlement européen et du Conseil du 11 mars 1996 sur la protection juridique des bases de données, ainsi que tout autre droit de nature équivalente dans le monde. Vous (preneur de licence) se rapporte à tout individu ou entité exerçant les Droits accordés par la licence. Votre et Vos renvoient également au preneur de licence.

Article 2 - Champ d'application de la présente Licence publique.

Octroi de la licence. Sous réserve du respect des termes et conditions d'utilisation de la présente Licence publique, le Donneur de licence Vous autorise à exercer pour le monde entier, à titre gratuit, non sous-licenciable, non exclusif, irrévocable, les Droits accordés par la licence afin de : reproduire et Partager l'Œuvre sous licence, en tout ou partie ; et produire, reproduire et Partager l'Œuvre dérivée. Exceptions et limitations. Afin de lever toute ambiguïté, lorsque les Exceptions et limitations s'appliquent à Votre utilisation, la présente Licence publique ne s'applique pas et Vous n'avez pas à Vous conformer à ses termes et conditions. Durée. La durée de la présente Licence publique est définie à l'Article 6(a). Supports et formats : modifications techniques autorisées. Le Donneur de licence Vous autorise à exercer les Droits accordés par la licence sur tous les supports et formats connus ou encore inconnus à ce jour, et à apporter toutes les modifications techniques que ceux-ci requièrent. Le Donneur de licence renonce et/ou accepte de ne pas exercer ses droits qui pourraient être susceptibles de Vous empêcher d'apporter les modifications techniques nécessaires pour exercer les Droits accordés par la licence, y compris celles nécessaires au contournement des Mesures techniques efficaces. Dans le cadre de la présente Licence publique, le fait de ne procéder qu'à de simples modifications techniques autorisées selon les termes du présent Article 2(a)(4) n'est jamais de nature à créer une Œuvre dérivée. Utilisateurs en aval. Offre du Donneur de licence – Œuvre sous licence. Chaque utilisateur de l'Œuvre sous licence reçoit automatiquement une offre de la part du Donneur de licence lui permettant d'exercer les Droits accordés par la licence selon les termes et conditions de la présente Licence publique. Offre additionnelle du Donneur de licence – Œuvre dérivée. Chaque utilisateur d'une Œuvre dérivée reçoit automatiquement une offre du Donneur de licence lui permettant d'exercer les Droits accordés par la licence sur l'Œuvre dérivée selon les termes et conditions de la Licence d'Œuvre dérivée que Vous appliquez. Pas de restrictions en aval pour les utilisateurs suivants. Vous ne pouvez proposer ou imposer des termes et conditions supplémentaires ou différents, ou appliquer quelque Mesure technique efficace que ce soit à l'Œuvre sous licence si ceux(celles)-ci sont de nature à restreindre l'exercice des Droits accordés par la licence aux utilisateurs de l'Œuvre sous licence. Non approbation. Aucun élément de la présente Licence publique ne peut être interprété comme laissant supposer que le preneur de licence ou que l'utilisation qu'il fait de l'Œuvre sous licence est lié à, parrainé, approuvé, ou doté d'un statut officiel par le Donneur de licence ou par toute autre personne à qui revient l'attribution de l'Œuvre sous licence, comme indiqué à l'Article 3(a)(1)(A)(i).

Autres droits. Les droits moraux, tel que le droit à l'intégrité de l'œuvre, ne sont pas accordés par la présente Licence publique, ni le droit à l'image, ni le droit au respect de la vie privée, ni aucun autre droit de la personnalité ou apparenté ; cependant, dans la mesure du possible, le Donneur de licence renonce et/ou accepte de ne pas faire valoir les droits qu'il détient de manière à Vous permettre d'exercer les Droits accordés par la licence. Le droit des brevets et le droit des marques ne sont pas concernés par la présente Licence publique. Dans la mesure du possible, le Donneur de licence renonce au droit de collecter des redevances auprès de Vous pour l'exercice des Droits accordés par la licence, directement ou indirectement dans le cadre d'un régime de gestion collective facultative ou obligatoire assorti de possibilités de renonciation quel que soit le type d'accord ou de licence. Dans tous les autres cas, le Donneur de licence se réserve expressément le droit de collecter de telles redevances.

Article 3 – Conditions d'utilisation de la présente Licence publique.

L'exercice des Droits accordés par la licence est expressément soumis aux conditions suivantes.

Attribution.

Si Vous partagez l'Œuvre sous licence (y compris sous une forme modifiée), Vous devez : conserver les informations suivantes lorsqu'elles sont fournies par le Donneur de licence avec l'Œuvre sous licence : identification du(des) auteur(s) de l'Œuvre sous licence et de toute personne à qui revient l'attribution de l'Œuvre sous licence, dans la mesure du possible, conformément à la demande du Donneur de licence (y compris sous la forme d'un pseudonyme s'il est indiqué) ; l'indication de l'existence d'un droit d'auteur ; une notice faisant référence à la présente Licence publique ; une notice faisant référence aux limitations de garantie et exclusions de responsabilité ; un URI ou un hyperlien vers l'Œuvre sous licence dans la mesure du possible ; Indiquer si Vous avez modifié l'Œuvre sous licence et conserver un suivi des modifications précédentes ; et Indiquer si l'Œuvre sous licence est mise à disposition en vertu de la présente Licence publique en incluant le texte, l'URI ou l'hyperlien correspondant à la présente Licence publique. Vous pouvez satisfaire aux conditions de l'Article 3(a)(1) dans toute la mesure du possible, en fonction des supports, moyens et contextes dans lesquels Vous Partagez l'Œuvre sous licence. Par exemple, Vous pouvez satisfaire aux conditions susmentionnées en fournissant l'URI ou l'hyperlien vers la ressource incluant les informations requises. Bien que requises aux termes de l'Article 3(a)(1)(A), certaines informations devront être retirées, dans la mesure du possible, si le Donneur de licence en fait la demande. Partage dans les mêmes conditions.

Outre les conditions indiquées à l'Article 3(a), si Vous Partagez une Œuvre dérivée que Vous avez réalisée, les conditions suivantes s'appliquent aussi. La Licence d'Œuvre dérivée que Vous appliquez doit être une licence Creative Commons avec les mêmes Eléments de licence, qu'il s'agisse de cette version ou d'une version ultérieure, ou une Licence compatible BY-SA. Vous devez inclure le texte, l'URI ou l'hyperlien correspondant à la Licence d'Œuvre dérivée que Vous appliquez. Ces conditions peuvent être satisfaites dans la mesure du raisonnable suivant les supports, moyens et contextes via lesquels Vous Partagez l'Œuvre dérivée. Vous ne pouvez pas proposer ou imposer des termes ou des conditions supplémentaires ou différents ou appliquer des Mesures techniques efficaces à l'Œuvre dérivée qui seraient de nature à restreindre l'exercice des Droits accordés par la Licence d'Œuvre dérivée que Vous appliquez.

Article 4 – Le Droit sui generis des producteurs de bases de données.

Lorsque les Droits accordés par la licence incluent le Droit sui generis des producteurs de bases de données applicable à Votre utilisation de l'Œuvre sous licence :

afin de lever toute ambiguïté, l'Article 2(a)(1) Vous accorde le droit d'extraire, réutiliser, reproduire et Partager la totalité ou une partie substantielle du contenu de la base de données ; si Vous incluez la totalité ou une partie substantielle du contenu de la base de données dans une base de données pour laquelle Vous détenez un Droit sui generis de producteur de bases de données, la base de données sur laquelle Vous détenez un tel droit (mais pas ses contenus individuels) sera alors considérée comme une Œuvre dérivée, y compris pour l'application de l'Article 3(b) ; et Vous devez respecter les conditions de l'Article 3(a) si Vous Partagez la totalité ou une partie substantielle du contenu des bases de données.

Afin de lever toute ambiguïté, le présent Article 4 complète mais ne remplace pas Vos obligations découlant des termes de la présente Licence publique lorsque les Droits accordés par la licence incluent d'autres Droit d'auteur et droits connexes.

Article 5 – Limitations de garantie et exclusions de responsabilité.

Sauf indication contraire et dans la mesure du possible, le Donneur de licence met à disposition l'Œuvre sous licence telle quelle, et n'offre aucune garantie de quelque sorte que ce soit, notamment expresse, implicite, statutaire ou autre la concernant. Cela inclut, notamment, les garanties liées au titre, à la valeur marchande, à la compatibilité de certaines utilisations particulières, à l'absence de violation, à l'absence de vices cachés ou autres défauts, à l'exactitude, à la présence ou à l'absence d'erreurs connues ou non ou susceptibles d'être découvertes dans l'Œuvre sous licence. Lorsqu'une limitation de garantie n'est pas autorisée en tout ou partie, cette clause peut ne pas Vous être applicable. Dans la mesure du possible, le Donneur de licence ne saurait voir sa responsabilité engagée vis-à-vis de Vous, quel qu'en soit le fondement juridique (y compris, notamment, la négligence), pour tout préjudice direct, spécial, indirect, incident, conséquentiel, punitif, exemplaire, ou pour toutes pertes, coûts, dépenses ou tout dommage découlant de l'utilisation de la présente Licence publique ou de l'utilisation de l'Œuvre sous licence, même si le Donneur de licence avait connaissance de l'éventualité de telles pertes, coûts, dépenses ou dommages. Lorsqu'une exclusion de responsabilité n'est pas autorisée en tout ou partie, cette clause peut ne pas Vous être applicable.

Les limitations de garantie et exclusions de responsabilité ci-dessus doivent être interprétées, dans la mesure du possible, comme des limitations et renonciations totales de toute responsabilité.

Article 6 – Durée et fin.

La présente Licence publique s'applique pendant toute la durée de validité des Droits accordés par la licence. Cependant, si Vous manquez à Vos obligations prévues par la présente Licence publique, Vos droits accordés par la présente Licence publique seront automatiquement révoqués.

Lorsque les Droits accordés par la licence ont été révoqués selon les termes de l'Article 6(a), ils seront rétablis : automatiquement, à compter du jour où la violation aura cessé, à condition que Vous y remédiiez dans les 30 jours suivant la date à laquelle Vous aurez eu connaissance de la violation ; ou à condition que le Donneur de licence l'autorise expressément. Afin de lever toute ambiguïté, le présent Article 6(b) n'affecte pas le droit du Donneur de licence de demander réparation dans les cas de violation de la présente Licence publique. Afin de lever toute ambiguïté, le Donneur de licence peut également proposer l'Œuvre sous licence selon d'autres termes et conditions et peut cesser la mise à disposition de l'Œuvre sous licence à tout moment ; une telle cessation n'entraîne pas la fin de la présente Licence publique. Les Articles 1, 5, 6, 7, et 8 continueront à s'appliquer même après la résiliation de la présente Licence publique.

Article 7 – Autres termes et conditions.

Sauf accord exprès, le Donneur de licence n'est lié par aucune modification des termes de Votre part. Tous arrangements, ententes ou accords relatifs à l'Œuvre sous licence non mentionnés dans la présente Licence publique sont séparés et indépendants des termes et conditions de la présente Licence publique.

Article 8 – Interprétation.

Afin de lever toute ambiguïté, la présente Licence publique ne doit en aucun cas être interprétée comme ayant pour effet de réduire, limiter, restreindre ou imposer des conditions plus contraignantes que celles qui sont prévues par les dispositions légales applicables. Dans la mesure du possible, si une clause de la présente Licence publique est déclarée inapplicable, elle sera automatiquement modifiée a minima afin de la rendre applicable. Dans le cas où la clause ne peut être modifiée, elle sera écartée de la présente Licence publique sans préjudice de l'applicabilité des termes et conditions restants. Aucun terme ni aucune condition de la présente Licence publique ne sera écarté(e) et aucune violation ne sera admise sans l'accord exprès du Donneur de licence. Aucun terme ni aucune condition de la présente Licence publique ne constitue ou ne peut être interprété(e) comme une limitation ou une renonciation à un quelconque privilège ou à une immunité s'appliquant au Donneur de licence ou à Vous, y compris lorsque celles-ci émanent d'une procédure légale, quel(le) qu'en soit le système juridique concerné ou l'autorité compétente.