Chapitre 1

Premiers pas

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

1.1 Un exemple de programme

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

Voici le code de cette fonction :

```
from typing import List

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

    i_est_premier : bool = False  # indicateur de primalité

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

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

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

```
        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
```

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

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

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

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

— **Définitions de fonction**

La construction :

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

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

— **Variables**

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

— **Expressions**

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

— **Instructions**

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

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

La construction :

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

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

La construction :

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

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

1.2 Notion d'expression

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

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

Les expressions sont principalement de deux types :

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

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

1.2.1 Expressions atomiques

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

Considérons l'interaction suivante avec l'interprète Python :

```
>>> 42
42
```

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

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

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

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

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

Pour résumer, nous avons fait la distinction entre :

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

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

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

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

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

1.2.1.1 Les constantes logiques (ou booléens)

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

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

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

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

1.2.1.2 Les entiers

Les entiers sont écrits en notation mathématique usuelle.

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

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

```
>>> 232392873298372983827392847398473948374394873984792837293823922832323928732983729838273928473984739483743948739847928372938239228323928732983729838273928473984739483743948739847928372938239228
```

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

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

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

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

1.2.1.3 Les constantes à virgule flottante.

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

Les flottants sont des approximations informatiques des *nombres réels* de $\mathbb{R}$, qui eux n'existent qu'en mathématiques. Dans ce livre d'introduction, nous ne nous occuperons pas trop des problèmes liés aux approximations – du fait de leur complexité – mais nous discutons un peu de cette problématique au chapitre 3.

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

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

Par exemple :

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

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

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

Remarque : lors de certains calculs, on ne veut pas distinguer entre entiers et flottants (par exemple le calcul de la valeur absolue). Dans ce cas on utilisera le type `float`. En effet, dans la spécification des types pour Python [2], les entiers du type `int` doivent être acceptés en remplacement de `float`. C'est un peu comme l'ensemble des entiers que l'on peut considérer en mathématiques comme sous-ensemble des nombres réels.

1.2.1.4 Les chaînes de caractères

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

Une chaîne de caractères est un texte encadré :

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

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

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

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

2. Cf. PEP 484 https://www.python.org/dev/peps/pep-0484/.

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

1.2.2 Expressions composées

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

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

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

- les expressions atomiques d'entiers et de flottants vues précédemment
- des opérateurs arithmétiques `+`, `-`, `*`, etc.
- des applications de fonctions prédéfinies en langage Python ou définies par le programmeur.

1.2.2.1 Opérateurs arithmétiques

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

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

La notation utilisée suit l'usage courant des mathématiques.

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

```
>>> 2 + 1
3

>>> 2 + 3 * 9
29

>>> (2 + 3) * 9
45

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

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

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

Voici quelques exemples illustratifs.

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

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

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

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

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

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

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

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

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

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

Priorité des opérateurs

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

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

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