Chapitre 1

Arithmétique, variables, instructions

Notions introduites

- les modes interactif et programme de Python
- expressions arithmétiques
- messages d'erreur
- manipuler des variables
- séquence d'instructions
- lire et écrire des chaînes de caractères

Le langage de programmation Python permet d'interagir avec la machine à l'aide d'un programme appelé *interprète Python*[1]. On peut l'utiliser de deux façons différentes. La première méthode consiste en un dialogue avec l'interprète. C'est le *mode interactif*. La seconde consiste à écrire un programme dans un fichier source puis à le faire exécuter par l'interprète. C'est le *mode programme*.

1.1 Mode interactif

En première approximation, le mode interactif de l'interprète Python se présente comme une calculatrice[2]. Les trois chevrons `>>>` constituent l'invite de commandes de Python, qui indique qu'il attend des instructions. Si par

1. Le site `https://www.nsi-premiere.fr` qui accompagne ce livre présente plusieurs environnements pour travailler avec Python.

2. On suppose ici que l'interprète Python vient d'être lancé, quelle que soit la solution retenue.

exemple on tape `1+2` puis la touche Entrée, l'interprète Python calcule et affiche le résultat.

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

Comme on le voit, les chevrons apparaissent de nouveau. L'interprète est prêt à recevoir de nouvelles instructions.

Arithmétique

En Python, on peut saisir des combinaisons arbitraires d'opérations arithmétiques, en utilisant notamment les quatre opérations les plus communes.

```
>>> 2 + 5 * (10 - 1 / 2)
49.5
>>>
```

L'addition est notée avec le symbole `+`, la soustraction avec le symbole `-`, la multiplication avec le symbole `*` et la division avec le symbole `/`. La priorité des opérations est usuelle et on peut utiliser des parenthèses. Dans l'exemple ci-dessus, on a utilisé des espaces pour améliorer la lisibilité. Ces espaces sont purement décoratifs et ne modifient pas la façon dont l'expression est calculée. En particulier, elles n'agissent pas sur la priorité des opérations.

```
>>> 1+2 * 3
7
```

Erreurs. L'interprète n'accepte que des expressions arithmétiques complètes et bien formées. Sinon, l'interprète indique la présence d'une erreur.

```
>>> 1 + * 2
  File "<stdin>", line 1
    1 + * 2
        ^
SyntaxError: invalid syntax
```

Ici, le message `SyntaxError: invalid syntax` indique une erreur de syntaxe, c'est-à-dire une instruction mal formée. Nous verrons plus loin d'autres catégories d'erreur. On peut ignorer pour l'instant la ligne `File "<stdin>", line 1`. Les deux lignes suivantes montrent à l'utilisateur l'endroit exact de l'erreur de syntaxe avec le symbole `^`, ici le caractère `*`.

Erreurs. Un autre type d'erreur se manifeste lorsque l'on donne à l'interprète une expression qui est correcte du point de vue de l'écriture mais dont le résultat n'a pas de sens. Ainsi toute tentative de division par zéro produit un message spécifique.

```
>>> 2 / (3 - 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Attention, les nombres « à virgule » s'écrivent à l'anglo-saxonne avec un point, et non avec une virgule qui a un autre sens en Python. En essayant d'appliquer des opérations arithmétiques à des nombres écrits avec des virgules on obtient toute une panoplie de comportements inattendus.

```
>>> 1,2 + 3,4
1, 5, 4
```

```
>>> 1,2 * 3
1, 6
```

En Python, la virgule sert à séparer deux valeurs. On en verra différentes utilisations aux chapitres 4, 5 et 14.

Les nombres de Python. Les nombres de Python sont soit des entiers relatifs, simplement appelés entiers, soit des nombres décimaux, appelés flottants.
Les entiers peuvent être de taille arbitraire (ce qui n'est pas le cas dans de nombreux langages de programmation). Ces entiers ne sont limités que par la mémoire disponible pour les stocker. L'exercice 6 page 22 propose une expérience pour observer que les entiers de Python peuvent être très grands. Le chapitre 19 reviendra sur la représentation des entiers dans un ordinateur.
Les nombres flottants, en revanche, ont une capacité limitée et ne peuvent représenter qu'une partie des nombres décimaux. Ainsi, des nombres décimaux trop grands ou trop petits ne sont pas représentables. Si on saisit un nombre décimal en tapant `1` suivi de cinq cents `0` et d'un point, on obtient la valeur `inf`, qui représente une valeur trop grande pour être représentée de cette manière. Des nombres comme π, $\sqrt{2}$, qui ne sont pas des nombres décimaux, ne peuvent être représentés que de manière approximative en Python. Ces approximations ne seront cependant pas problématiques dans les situations que nous rencontrerons. Le chapitre 20 reviendra sur la représentation des nombres réels dans un ordinateur.

À propos de la division. Si on effectue la division de deux entiers avec l'opération `/`, on obtient un nombre décimal. Ainsi, `7 / 2` donne le nombre `3.5`. (Le chapitre 20 expliquera comment de tels nombres décimaux sont représentés dans la machine.) Si on veut effectuer en revanche une division entière, alors il faut utiliser l'opération `//`. Ainsi, `7 // 2` donne le nombre `3`, qui est cette fois un entier, à savoir le quotient de 7 par 2 dans la division euclidienne. On peut également obtenir le reste d'une telle division euclidienne avec l'opération `%`. Ainsi, `7 % 2` donne l'entier `1`. (La division euclidienne est celle que l'on a apprise à l'école primaire.)
Attention : les opérations `//` et `%` de Python ne coïncident avec la division euclidienne que lorsque le diviseur est positif. Lorsqu'il est négatif, le reste est alors également négatif. Voici une illustration des quatre cas de figure :

	$a = 7$ $b = 3$	$a = -7$ $b = 3$	$a = 7$ $b = -3$	$a = -7$ $b = -3$
a `//` b	2	-3	-3	2
a `%` b	1	2	-2	-1

Dans tous les cas, on a l'égalité $a = (a \texttt{ // } b) \times b + a \texttt{ \% } b$ et l'inégalité $|a \texttt{ \% } b| < |b|$. Une autre façon de le voir, sans doute plus simple, consiste à définir $a \texttt{ // } b$ comme la partie entière (par défaut) du nombre réel a/b, c'est-à-dire le plus grand entier inférieur ou égal à a/b. Cela étant, il est rare que l'on divise par un nombre négatif. On peut même le décourager purement et simplement pour éviter toute différence accidentelle avec la division euclidienne.

Variables

Les résultats calculés peuvent être mémorisés par l'interprète, afin d'être réutilisés plus tard dans d'autres calculs.

```
>>> a = 1 + 1
>>>
```

La notation `a =` permet de donner un nom à la valeur à mémoriser. L'interprète calcule le résultat de `1+1` et le mémorise dans la *variable* `a`. Aucun résultat n'est affiché. On accède à la valeur mémorisée en utilisant le nom `a`[3].

```
>>> a
2
```

Plus généralement, la variable peut être réutilisée dans la suite des calculs.

3. On suppose que toutes les commandes tapées dans cette section sont tapées à la suite sans relancer à chaque fois l'interprète Python.

```
>>> a * (1 + a)
6
```

Le symbole `=` utilisé pour introduire la variable `a` désigne une opération d'*affectation*. Il attend à sa gauche un nom de variable et à sa droite une expression. On peut donner une nouvelle valeur à la variable `a` avec une nouvelle affectation. Cette valeur remplace la précédente.

```
>>> a = 3
>>> a * (1 + a)
12
```

Le calcul de la nouvelle valeur de `a` peut utiliser la valeur courante de `a`.

```
>>> a = a + 1
>>> a
4
```

Un nom de variable peut être formé de plusieurs caractères (lettres, chiffres et souligné). Il est recommandé de ne pas utiliser de caractères accentués et l'usage veut que l'on se limite aux caractères minuscules.

```
>>> cube = a * a * a
>>> ma_variable = 42
>>> ma_variable2 = 2019
```

Erreurs. Un nom de variable ne doit pas commencer par un chiffre et certains noms sont interdits (car ils sont des mots réservés du langage).

```
>>> 4x = 2
  File "<stdin>", line 1
    4x = 2
     ^
SyntaxError: invalid syntax
>>> def = 3
  File "<stdin>", line 1
    def = 3
        ^
SyntaxError: invalid syntax
```

Il n'est pas possible d'utiliser une variable qui n'a pas encore été définie.

```
>>> b + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Erreurs. Seul un nom de variable peut se trouver à la gauche d'une affectation.

```
>>> 1 = 2
  File "<stdin>", line 1
SyntaxError: can't assign to literal
```

Une variable peut être imaginée comme une petite boîte portant un nom (ou une étiquette) et contenant une valeur. Ainsi, on peut se représenter une variable déclarée avec `x = 1` par une boîte appelée `x` contenant la valeur 1.

Lorsque l'on modifie la valeur de la variable `x`, par exemple avec l'affectation `x = x + 1`, la valeur 1 est remplacée par la valeur 2.

État

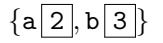
À chaque étape de l'interaction, chacune des variables introduites contient une valeur. L'ensemble des associations entre des noms de variables et des valeurs constitue l'*état* de l'interprète Python. Cet état évolue en fonction des instructions exécutées, et notamment en raison des affectations de nouvelles valeurs. On dit de ces instructions qui modifient l'état qu'elles ont un *effet de bord*.

On peut représenter l'état de l'interprète par un ensemble d'associations entre des noms de variables et des valeurs. Par exemple, l'ensemble {`a` 1, `b` 2, `c` 3} représente l'état dans lequel les variables `a`, `b` et `c` valent respectivement `1`, `2` et `3` et où aucune autre variable n'est définie.

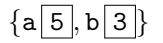
Pour simuler à la main une étape d'interaction, on indiquera l'état avant et après l'exécution de l'instruction. Considérons par exemple l'instruction suivante.

```
>>> a = a + b
```

Cette instruction modifie la valeur de `a` en fonction des valeurs de `a` et `b`. Par exemple, en partant de l'état



on obtient après exécution de l'instruction un état où l'association `a` 5 a remplacé `a` 2.



Différences entre mathématiques et informatique. En informatique, le terme de *variable* est utilisé pour indiquer le fait que la valeur associée peut *varier* au fur et à mesure de l'exécution du programme. Ce n'est donc pas la même notion que celle de variable en mathématiques, désignant elle une unique valeur (qui est, ne serait-ce que provisoirement, indéterminée).
Par ailleurs, on a parfois des manières différentes d'écrire les choses dans ces deux domaines, et le même symbole peut ne pas avoir la même signification en informatique et en mathématiques. C'est le cas en particulier du symbole d'égalité, dont l'utilisation en informatique peut paraître étrange si on la confond avec sa signification mathématique. La suite de symboles `a = a + 1` par exemple n'a pas de sens si on la voit comme une égalité. Il faut la voir comme une instruction, qui donne comme nouvelle valeur à la variable `a` le résultat de l'expression `a + 1` calculé avec la valeur courante de `a`. En l'occurrence, on vient donc d'augmenter la valeur de `a` d'une unité.
En conséquence de cette nature de l'instruction d'affectation, même la désignation de la variable par `a` peut avoir deux sens différents en informatique. La plupart du temps `a` désigne la valeur de la variable, c'est-à-dire le contenu de la boîte associée, mais à gauche du symbole d'affectation `a` est une *référence* à la boîte elle-même, dont on va modifier le contenu.
Comme il est courant de modifier la valeur d'une variable en lui ajoutant une certaine quantité, il existe en outre une instruction spécifique pour cela, notée `+=`. Ainsi, on peut, en informatique, écrire `a += 1` au lieu de `a = a + 1`.

Si d'autres variables sont définies dans l'état de départ et ne sont pas affectées par l'instruction, leur valeur reste inchangée. Ainsi, partant de

$$\{\texttt{a}\boxed{\texttt{0}}, \texttt{b}\boxed{\texttt{1}}, \texttt{d}\boxed{\texttt{-12}}, \texttt{x}\boxed{\texttt{3}}\}$$

on obtient l'état suivant.

$$\{\texttt{a}\boxed{\texttt{1}}, \texttt{b}\boxed{\texttt{1}}, \texttt{d}\boxed{\texttt{-12}}, \texttt{x}\boxed{\texttt{3}}\}$$

Lorsqu'une variable est affectée pour la première fois, l'association correspondante apparaît dans l'état. Ainsi, pour une instruction `a = b + c` et en partant de l'état

$$\{\texttt{b}\boxed{\texttt{-2}}, \texttt{c}\boxed{\texttt{5}}\}$$

on obtient l'état suivant.

$$\{\texttt{a}\boxed{\texttt{3}}, \texttt{b}\boxed{\texttt{-2}}, \texttt{c}\boxed{\texttt{5}}\}$$

Dans le cas particulier où on affecte une nouvelle variable avec la valeur d'une variable qui existe déjà, par exemple avec `d = a`, il est important de comprendre que `d` et `a` ne sont pas deux noms pour la même boîte, mais deux boîtes différentes, la boîte `d` recevant la valeur contenue dans la boîte `a`.

On peut observer que les deux variables sont effectivement indépendantes : modifier la variable `a`, par exemple avec `a = 7`, n'a pas d'effet sur la variable `d`.

1.2 Mode programme

Le mode programme de Python consiste à écrire une suite d'instructions dans un fichier et à les faire exécuter par l'interprète Python. Cette suite d'instructions s'appelle un *programme*, ou encore un *code source*. On évite ainsi de ressaisir à chaque fois les instructions dans le mode interactif. Par ailleurs, cela permet de distinguer le rôle de programmeur du rôle d'utilisateur d'un programme.

Affichage

En mode programme, les résultats des expressions calculées ne sont plus affichés à l'écran. Il faut utiliser pour ceci une instruction explicite d'affichage. En Python, elle s'appelle `print`. Par exemple, dans un fichier portant le nom `test.py`, on peut écrire l'instruction suivante.

```
print(3)
```

On peut alors faire exécuter ce programme par l'interprète Python [4], ce qui affiche `3` à l'écran. L'instruction `print` accepte une expression arbitraire. Elle commence par calculer le résultat de cette expression puis l'affiche à l'écran. Ainsi, l'instruction

```
print(1+3)
```

calcule la valeur de l'expression `1+3` puis affiche `4` à l'écran.

Affichage de textes

L'instruction `print` n'est pas limitée à l'affichage de nombres. On peut lui donner un message à afficher, écrit entre guillemets. Par exemple, l'instruction

4. Voir le site `https://www.nsi-premiere.fr` pour les différentes manières d'appeler l'interprète Python sur un fichier.