Chapitre 1
Entiers, flottants, variables et fonctions

1 Type d'un objet

Lorsque l'on manipule un objet, mathématique ou informatique, il est essentiel de garder en tête la nature de l'objet, c'est-à-dire son **type**. Pour connaitre le type d'une expression informatique `expression`, on exécute l'instruction suivante.

```
type(expression)
```

En mathématiques, nous avons vu des objets de divers types : des nombres, des couples de nombres, des fonctions, des suites, des ensembles, des propositions logiques. En informatique, nous verrons cette année différents types de nombres, des fonctions, des booléens, des chaines de caractères, des listes et des dictionnaires.

```
type([-1, 3, 7, 4])

type(-35)
```

La partie suivante présente deux types de nombres existant en Python : les entiers et les flottants.

2 Entiers et flottants

En Python, on distingue deux types de nombres :

- ⋆ les **entiers** relatifs, dont le type est `int` (pour *integer*, en anglais) ;
- ⋆ les **flottants**, c'est-à-dire les nombres réels ou nombres à virgule, dont le type est `float`.

Les nombres flottants sont écrits comme des nombres décimaux. Comme Python parle anglais, le séparateur décimal n'est pas une virgule mais un point.

```
type(5.2)

type(-3)

type(-3.0)
```

Remarquons que les nombres -3 et -3.0 n'ont pas le même type (même s'ils ont la même valeur). Les représentations informatiques des entiers et des flottants sont très différentes ; on ne rentrera pas dans le détail de ces représentations, mais on en verra quelques conséquences pratiques à la partie 2.3.

2.1 Opérations sur les entiers et les flottants

Les opérations sur les entiers et sur les flottants sont les suivantes.

Opérations sur les flottants	
$+$	Addition
$-$	Soustraction
$*$	Multiplication
$**$	Puissance
$/$	Division

```
23.0 / 4.0

5.2e4 # La notation scientifique est représentée par la lettre e
      # (comme exposant), qui n'a rien à voir ici avec l'exponentielle.

# Derrière un dièse, on peut écrire un commentaire,
# et c'est bien pratique !
# Ce qui est placé après un dièse n'est pas lu par l'interpréteur.
```

Opérations sur les entiers	
$+$	Addition
$-$	Soustraction
$*$	Multiplication
$**$	Puissance
$//$	Division entière (quotient de la division euclidienne)
$\%$	Reste de la division euclidienne

```
23 // 4

23 % 4

24 % 4

2 ** 3
```

2.2 Conversion entiers-flottants

⋆ Pour convertir un entier en flottant, on lui applique la fonction **`float`**.

⋆ Réciproquement, pour convertir un flottant en entier (lorsque c'est pertinent, c'est-à-dire lorsque le flottant a une partie décimale nulle), on lui applique la fonction **`int`**.

```
float(12)

type(float(12))

int(12.0)

type(int(12.0))
```

En fait, Python convertit automatiquement un entier en flottant pour faire un calcul dans le monde des flottants. Par exemple, si l'un des opérandes du calcul est flottant et l'autre entier, Python convertira l'entier en flottant et le résultat sera flottant. Dans le cas d'une division flottante (de symbole /), les deux opérandes seront convertis si besoin en flottants et le résultat sera toujours flottant.

```
23. / 4

23 / 4

24 / 4

type(24 / 4)

3 * 5.2
```

2.3 Représentation machine des flottants et erreurs d'arrondi

En Python, les flottants sont représentés par leur développement dyadique, c'est-à-dire leur développement en somme de puissances de 2 (l'écriture décimale, elle, est la représentation en somme de puissances de 10). Cependant, la plupart des nombres ont un développement dyadique infini (c'est par exemple le cas de 0.1), qu'il est donc impossible de stocker entièrement dans la mémoire de l'ordinateur, ce qui contraint à arrondir les nombres flottants en ne gardant qu'une partie de leur développement dyadique.
Ainsi, les calculs effectués par Python sur les nombres flottants ne fournissent pas des résultats exacts, mais des approximations — très bonnes, du reste ! — du résultat exact.

```
0.1 + 0.2

1.1 + 2.7 - 3.8

0.7 ** 2

(1 / 730) * 730

(1 / 729) * 729
```

Par ailleurs, la représentation des flottants étant plus lourde que celle des entiers, la taille des flottants gérés par Python est limitée alors que celle des entiers ne l'est pas.

```
2 ** 10000

2.0 ** 10000
```

En pratique, dans les problèmes d'analyse numérique, si l'on veut comparer deux nombres flottants, on les comparera à une certaine précision près.

3 Variables

Dans les exemples précédents, les instructions tapées dans l'interprète de commandes fournissaient des résultats qui étaient immédiatement oubliés ensuite. Afin de conserver en mémoire un résultat et de pouvoir le réutiliser ensuite, on le stocke dans une **variable**.

3.1 Affectation

Une variable est l'association d'un identifiant (le nom de la variable) à une valeur.
L'instruction

```
variable = expression
```

évalue l'expression `expression`, puis affecte le résultat obtenu à l'identifiant `variable`.
Si la variable `variable` était déjà définie, l'instruction précédente modifie le contenu de la variable `variable`, qui prend à présent la valeur de l'expression `expression`. L'ancienne valeur de la variable est alors oubliée, écrasée.

```
a = 2 * 3 + 4

a
```

3.1.1 Syntaxe et bonnes pratiques

Voici quelques règles et recommandations sur la dénomination des variables.

- ⋆ Le nom d'une variable peut contenir des lettres (majuscules ou minuscules), des chiffres et des tirets bas _, mais pas d'espace, de parenthèse, de point, ni de tiret (qui serait un signe moins). Il est par ailleurs déconseillé d'utiliser des accents dans les noms de variables (les accents ne sont pas acceptés par toutes les machines).

- ⋆ Le nom d'une variable ne peut pas commencer par un chiffre.
- ⋆ Certains noms, déjà définis en Python, sont réservés et ne peuvent pas devenir le nom d'une variable : `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `False`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `None`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `True`, `try`, `while`, `with`, `yield`.
- ⋆ Il est toujours préférable de choisir des noms de variables explicites et pertinents : par exemple, on appellera plus volontiers le résultat d'une addition `somme` que `toto` ou que `b`. Donner des noms explicites aux variables vous permet de ne pas vous perdre dans vos variables et facilite également la lecture de vos programmes par votre dévouée correctrice.

De plus, dans une affectation, la variable à laquelle on affecte une valeur doit toujours se trouver à gauche du signe =.

```
4 = x # Là, l'ordinateur comprend que nous voulons redéfinir 4
      # et, fort heureusement, nous l'interdit !

x = 4 # C'est mieux !
```

3.1.2 Exemples et remarques

Une même variable peut, au cours de l'exécution d'un programme, prendre des valeurs différentes et même des valeurs de types différents.

```
a = 2

a

type(a)

a = 5.4

a

type(a)

a = [1, 2, 3]

type(a)
```

Une fois définie, la valeur d'une variable peut être utilisée dans un calcul ou dans la définition d'autres variables.

```
largeur = 2

longueur = 7

aire = longueur * largeur

aire
```

```
largeur = largeur + 4
# Ici, on évalue l'expression largeur + 4 : elle vaut 6 (car largeur vaut 2).
# On affecte ensuite 6 à la variable largeur
# (en écrasant la valeur précédente de largeur).

largeur

aire
```

 On remarque que la variable `aire` n'a pas été modifiée après la modification de la variable `largeur`. C'est normal : le calcul

```
aire = longueur * largeur
```

utilise les valeurs des variables `largeur` et `longueur` au moment où il est exécuté, mais n'établit en aucune manière de relation durable entre les variables `aire`, `largeur` et `longueur`.

3.1.3 Incrémentations

Il est très classique de modifier une variable pour lui ajouter 1 — on dit alors que la variable est **incrémentée** (de 1) — ou plus généralement, de lui ajouter un nombre quelconque (c'est notamment le cas lorsque la variable est un compteur). Pour ajouter le nombre `increment` à la variable `variable`, on peut écrire

```
variable = variable + increment
```

ou, plus efficacement,

```
variable += increment
```

Sur le même principe, pour soustraire ou pour multiplier un nombre à variable, on pourra utiliser les opérateurs `-=` et `*=`.

```
a = 2

a += 1 # Cette instruction a le même effet que a = a + 1.

a

a += 4

a

a -= 1 # Cette instruction a le même effet que a = a - 1.

a

a *= 3 # Cette instruction a le même effet que a = a * 3.

a
```

3.2 Affectation simultanée

Il est possible en Python d'effectuer plusieurs affectations d'un seul coup, simultanément, en affectant à un uplet d'identifiants un uplet de valeurs.

```
(abscisse, ordonnee) = (2, 3)

abscisse

ordonnee

abscisse, ordonnee = 5, 6.0
# En Python, les parenthèses dans le couple sont optionnelles.

abscisse

ordonnee
```

Bien que l'affectation simultanée soit bien pratique dans certains cas, il est en général plus lisible de définir une seule variable à la fois.

4 Fonctions

Dans cette partie, nous allons voir comment définir des fonctions, qui seront de type `function`, en Python.

4.1 Écriture d'une fonction

4.1.1 Syntaxe et premiers exemples

Une fonction est un objet qui à un certain nombre de variables d'entrée, appelées **arguments** de la fonction, associe un résultat, appelé **sortie**. On dira aussi que la fonction **renvoie** ou **retourne** cette sortie.

La syntaxe de définition d'une fonction est la suivante :

```
def nom_de_la_fonction (arguments):
    bloc indenté d'instructions
    dépendant éventuellement
    des arguments
    return sortie
```

Voici quelques remarques importantes sur l'écriture des fonctions en Python.

- ⋆ Le corps de la fonction est **indenté** d'un cran par rapport à la première ligne. Tout ce qui est ainsi indenté fait partie de la fonction. Dès qu'une ligne n'est plus indentée, elle ne fait plus partie de la fonction.
  Il est crucial de respecter l'indentation du code (c'est à peu près le seul point de syntaxe du langage Python !).

- ⋆ Les deux points à la fin de la première ligne sont un symbole discret mais indispensable !
- ⋆ Les arguments sont toujours donnés entre parenthèses. S'il y a en plusieurs, il seront séparés par des virgules.

  Contrairement aux fonctions que l'on a vues en mathématiques, une fonction Python peut aussi n'avoir aucun argument. Dans ce cas, les parenthèses seront toujours là, mais elles seront vides.

Dans Spyder, à chaque fois que l'on définira une fonction, on écrira sa définition dans l'éditeur de scripts dans un fichier, qu'il conviendra d'enregistrer. On exécutera alors le code, via le menu Exécuter par exemple. On testera alors la fonction ainsi définie sur des exemples bien choisis. On effectuera ces tests dans l'interprète de commandes.

Écrivons la définition suivante dans l'éditeur de scripts puis exécutons le fichier.

```
def double(x):
    return 2 * x
```

Cette fonction prend en argument un nombre `x` (entier ou flottant) et renvoie le double de `x`. C'est la fonction $x \longmapsto 2x$.

Testons-la dans l'interprète de commandes.

```
double(3)

double(-5.1)

y = 4

type(double(y))

type(double) # Comme en maths, on ne confondra pas
             # une fonction avec sa valeur en un point !
```

Une fois définie, on peut **appeler** la fonction, c'est-à-dire l'appliquer, dans le calcul d'une expression.

```
z = 3 * double(2.5) - 1 # double(2.5) est un appel de la fonction double.

z
```

On peut aussi, bien sûr, l'appeler à l'intérieur d'une autre fonction.

```
def beau_calcul(a, b):
    d = double(b) # La variable d contient le résultat
                  # de la fonction double appliquée à b.
    return double(a + d) + 1

beau_calcul(1, 3)
```

Une fonction pourra contenir plusieurs fois la commande `return`, notamment lorsque la fonction contiendra des tests (voir le chapitre 2).

⚠ Toutefois, au premier `return` exécuté, la fonction se termine.

```
def mauvaise_pratique(a, b):
    return a + b
    return a - b

mauvaise_pratique(2, 5)
# Le second return n'est même pas lu :
# on sort de la fonction après l'exécution du premier return.
```

Si l'on veut renvoyer plusieurs résultats, on renvoie plutôt un uplet de résultats.

```
def meilleure_pratique(a, b):
    return (a + b, a - b)

meilleure_pratique(2, 5)
```

4.1.2 Documentation d'une fonction

Comme tout bon code, l'écriture d'une fonction se doit d'être la plus lisible possible. Pour cela, la fonction et toutes les variables qui apparaissent dans le corps de la fonction doivent porter des noms pertinents. De plus, les étapes compliquées de la fonction, s'il y en a, doivent être expliquées en commentaire (après un dièse).

À toutes ces bonnes pratiques (qui ne sont pas spécifiques aux fonctions) s'ajoute la possibilité d'ajouter une **documentation** (*docstring*) à la fonction, c'est-à-dire une description de ce que fait la fonction. Elle comprend une liste des arguments de la fonction et de leur type, ainsi qu'une description de la sortie de la fonction.

Cette documentation, qui n'est pas un commentaire, se place au tout début de la fonction, juste en dessous de la première ligne, entre triples guillemets :

```
def nom_de_la_fonction (arguments):
    """
    Documentation de la fonction.
    """
    bloc indenté d'instructions
    dépendant éventuellement
    des arguments
    return sortie
```

On accède ensuite à la documentation d'une fonction à l'aide de la commande **`help`** :

```
help(nom_de_la_fonction)
```

Ajoutons à la fonction `double` précédente une documentation.

```
def double(x):
    """
    Argument / entrée : un nombre x, entier ou flottant.
    Sortie : le double de x.
    """
    return 2 * x
```

Renseignons-nous à présent sur la fonction `double`.

```
help(double)
```

4.1.3 Fonctions pré-implémentées et bibliothèques

De nombreuses fonctions sont déjà implémentées dans Python. Cependant, ces fonctions ne sont pas disponibles directement : elles sont stockées dans des **bibliothèques** ou **modules**, que l'on importe pour pouvoir utiliser des fonctions supplémentaires adaptées à un usage spécifique.

Par exemple, le langage Python possède évidemment toutes les fonctionnalités d'une calculatrice scientifique (et bien d'autres encore !) : la plupart des fonctions et constantes mathématiques usuelles sont implémentées dans la bibliothèque **`numpy`** ou dans la bibliothèque **`math`**.

Afin d'utiliser une bibliothèque, on commence par l'importer.

```
import numpy
```

Pour accéder ensuite à une fonction (ou à une constante) de la bibliothèque, on fait précéder le nom de la fonction (ou de la constante) du nom de la bibliothèque et d'un point.

```
numpy.cos(0)

pi # Cette variable n'est pas définie !

numpy.pi # C'est mieux !

numpy.exp(1)

numpy.log(numpy.e) # La fonction log est le logarithme népérien.

numpy.log10(1000) # La fonction log10 est le logarithme décimal.
```

Pour alléger les notations, on importe généralement la bibliothèque en lui donnant un autre nom (un *alias*), qui en général est une abréviation du nom d'origine. On se réfère alors à la bibliothèque par cet alias dans toute la suite.

```
import numpy as np

np.sqrt(25) # La fonction sqrt est la fonction racine carrée
            # (square root, en anglais).

np.abs(-12.4) # La fonction abs est la fonction valeur absolue.
```

4.2 Portée des variables : variables locales et globales

Toute variable dont l'affectation a lieu dans une fonction est dite **locale** à la fonction : c'est une variable qui n'a d'existence qu'à l'intérieur de la fonction et qui n'a rien à voir avec une variable qui porterait le même nom existant en dehors de la fonction. De même, les arguments d'une fonction sont des variables locales à cette fonction.

Lors de l'exécution de la fonction, Python crée et manipule des variables locales, puis les oublie une fois la fonction terminée. Il est donc impossible d'accéder à une variable locale en dehors de la fonction où elle est définie.

Par contre, la localité des variables permet une plus grande liberté : il est possible d'utiliser les mêmes noms de variables locales dans des fonctions différentes (et donc en particulier de réutiliser des noms pertinents).

À l'inverse, les variables définies en dehors de toute fonction sont dites **globales**. Python connait les valeurs de ces variables tout au long du script et il est donc possible d'y accéder à l'extérieur ou à l'intérieur d'une fonction.

```
i = 42 # Cette variable i est globale.

def mensonge():
    """
    Entrée : aucune.
    Sortie : 10.
    """
    i = 10 # Cette variable i est locale à la fonction.
    return i

i

mensonge()

i
```

La variable globale `i` n'a pas été modifiée par la fonction `mensonge`.

Étudions un autre exemple important.

```
import numpy as np

def distance(x, y):
    """
    Arguments : deux flottants x et y.
    Sortie : la distance entre x et y, c'est-à-dire
             la valeur absolue de leur différence.
    """
    difference = y - x
    dist = np.abs(difference)
    return dist
```

```
distance(2, 3)

difference

dist

x
# Les variables difference, dist, x et y
# sont locales à la fonction distance.
```

La fonction suivante est mal définie :

```
def mal_definie(x, y):
    distance(x, y)
    return dist ** 2
```

En effet, la variable `dist`, qui est locale à la fonction `distance`, n'est pas définie.

```
mal_definie(2, 5)
```

Pour corriger la fonction, on doit donc définir la variable `dist` à l'intérieur de notre nouvelle fonction.

```
def bien_definie(x, y):
    dist = distance(x, y)
    return dist ** 2
```

```
bien_definie(2, 5)
```