Premiers pas en programmation

Chap. 2

Table des matières

On appelle **informatique** le *traitement automatisé des informations* par un ordinateur. Cette discipline s'appuie sur la **programmation**, activité qui consiste à apprendre à un ordinateur à effectuer des tâches qu'il n'est pas capable d'exécuter à sa conception. L'écriture d'un programme nécessite l'utilisation d'un **langage de programmation** ; dans ce cours nous utiliserons Python.

1 À quoi a-t-on accès lorsqu'on utilise un langage de programmation

Un langage de programmation doit :

- *fournir des objets (ou types) primitifs ;*

- *posséder une bibliothèque de fonctions prédéfinies ;*
- *permettre la manipulation des objets primitifs et des fonctions prédéfinies ;*
- *établir des règles qui permettent de construire de nouveaux objets (ou types) ou de nouvelles fonctions par combinaison des types primitifs et des fonctions prédéfinies.*

Dans ce chapitre nous allons aborder chacun de ces points.

2 Quelques objets primitifs en Python

Le langage Python possède un grand nombre d'objets primitifs. Parmi eux, on utilisera lors des premiers chapitres :

- Les **constantes entières** (« Integer », `int`)

```
>>> type(3)
   <type 'int'>
>>> type(-10)
   <type 'int'>
```

- Les **constantes « flottantes »** (type `float`). Il s'agit d'une *approximations des nombres non entiers*.

```
>>> type(0.1)
   <type 'float'>
>>> type(-2.9)
   <type 'float'>
>>> type(2e3)
   <type 'float'>
```

- Les **valeurs booléennes** (type `bool`) `True` et `False`.

```
>>> type(True)
   <type 'bool'>
>>> type(False)
   <type 'bool'>
```

- Les **chaînes de caractères** (« String », type `str`).

```
>>> type("abcd")
   <type 'str'>
>>> type('abcd')
   <type 'str'>
>>> type('a')
   <type 'str'>
```

3 Syntaxe et évaluation d'une expression en Python

Définition. *En mathématique et en informatique une expression est une formule exprimant la façon de calculer une valeur.*

3.1 Ambiguïté de l'écriture traditionnelle des expressions

En mathématique on écrit $3x^2+2x+4$ l'expression qui permet de calculer la valeur du polynôme du second degrés, pour toute valeur de la variable x. En programmation, peut-on se contenter d'utiliser la syntaxe des mathématiques ?
Si l'on se penche un peu plus attentivement sur l'écriture du polynôme, on remarque :

- qu'il faut savoir que $2x$ signifie la multiplication du nombre 2 par x ;
- que pour $3x^2$ il faut d'abord élever à la puissance 2 x avant de multiplier le résultat par 3
- que $2x+4$ n'est pas 2 multiplié par le résultat du calcul de $x+4$
- que cette écriture est une écriture à deux dimensions (élévation à la puissance 2 est indiquée en augmentant l'espace depuis la ligne de base).

De même, en mathématique, on écrit :

- $a+b$: l'opérateur + est entre les deux opérandes ;
- $\sin(x)$: la fonction est placée avant son argument ;
- f' pour indiquer la fonction dérivée de la fonction f (le symbole de la dérivée est placé après le nom de la fonction).

En conclusion, l'écriture mathématique traditionnelle nous parait claire car on l'utilise depuis les plus petites classes et parce qu'on a appris des règles telles que celles de la priorité des opérateurs. *En informatique, la syntaxe devra être plus rigoureuse, moins ambiguë car la machine effectuera une vérification pointilleuse de tout ce qui sera écrit.*

L'écriture la plus rigoureuse du polynôme est : $(3\times(x\hat{}2))+((2\times x)+4)$, heureusement le langage Python a intégré la règle de priorité des opérateurs et on pourra écrire : $3\times x\hat{}2+2\times x+4$.

Remarque. Les expressions en informatique ne se limitent bien évidemment pas au domaine des mathématiques.

3.2 Opérateurs arithmétiques en Python

Les opérateurs en Python se répartissent en trois catégories :

- les *opérateurs arithmétiques*, utilisés pour calculer des expressions ;
- les *opérateurs de comparaison et les opérateurs logiques*, utilisés dans l'alternative ;
- les *opérateurs d'affectation*, utilisés pour modifier la « valeur » d'une variable.

Dans ce chapitre nous allons nous contenter d'utiliser les opérateurs arithmétiques.

Opérateur	Expression	Description
`+`	`op1 + op2`	renvoie le résultat de l'addition de `op1` et `op2`
`-`	`op1 - op2`	renvoie le résultat de la soustraction de `op1` et `op2`
`-`	`-op`	renvoie la valeur opposée de `op`
`*`	`op1 * op2`	renvoie le résultat de la multiplication de `op1` et `op2`
`/`	`op1 / op2`	renvoie le résultat de la division de `op1` et `op2`
`//`	`op1 // op1`	renvoie le résultat de la division euclidienne de `op1` et `op2`
`%`	`op1 % op2`	renvoie le reste de la division euclidienne de `op1` et `op2`
`**`	`op**exp`	renvoie le résultat de `op` à la puissance `exp`.

Remarque.

- Le résultat de la division de deux nombres est toujours un `float`, *même s'il s'agit de la division de deux nombres entiers.*
- Le résultat de la *division euclidienne* de deux nombres est un `float` si l'un des nombres est un `float`, un `int` si les deux nombres sont des entiers.

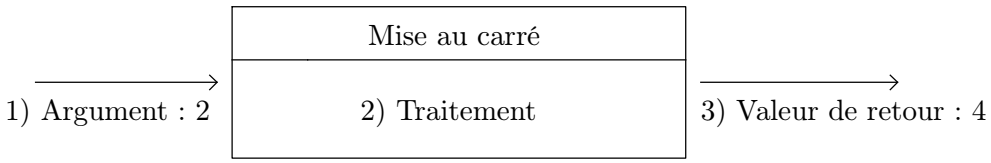
Exercice 1. Comment visualiser toutes les étapes de l'évaluation de l'expression $2 \times (3+4,3)^3 + 5 \times 6^2$ par l'interpréteur Python grâce au logiciel Thonny ?

4 Application d'une fonction

4.1 Schéma-bloc d'une fonction

Définition. *On appelle schéma-bloc la modélisation du fonctionnement d'une fonction. Cette modélisation est affichée sous forme graphique.*

Par exemple, le schéma-bloc de la fonction « mise au carré » est le suivant :



Ce schéma indique que :

1. La fonction accepte 0, 1 ou plusieurs arguments ;
2. La fonction effectue un traitement. *Ce traitement n'est pas indiqué dans le schéma-bloc car sa connaissance n'est pas nécessaire à un utilisateur.*
3. La fonction retourne la valeur calculée.

4.2 Fonctions intégrées du langage Python

Le langage Python possède de nombreuses fonctions prédéfinies. Les fonctions intégrées `dir` et `help` permettent de les découvrir.

```
>>> dir()
   ['__builtins__', '__doc__', 'ps_out']
```

```
>>> dir(__builtins__)
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BufferError',
BytesWarning', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EnvironmentError',
Exception', 'False', 'FloatingPointError', 'FutureWarning', 'GeneratorExit',
IOError', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError',
KeyError', 'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError', 'None',
NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError',
PendingDeprecationWarning', 'ReferenceError', 'RuntimeError', 'RuntimeWarning',
StandardError', 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
SystemExit', 'TabError', 'True', 'TypeError', 'UnboundLocalError',
UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
UnicodeWarning', 'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError',
'__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs', 'all', 'any',
apply', 'basestring', 'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable',
chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright', 'credits',
delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'execfile', 'exit', 'file',
filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash',
help', 'hex', 'id', 'input', 'int', 'intern', 'isinstance', 'issubclass', 'iter',
len', 'license', 'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next',
object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range',
raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round', 'set', 'setattr',
slice', 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'unichr',
'unicode', 'vars', 'xrange', 'zip']
```

```
>>> help(abs)
```

```
Help on built-in function abs in module __builtin__:

abs(...)
    abs(number) -> number

    Return the absolute value of the argument.
```

L'exemple ci-dessus pourrait laisser penser que le nombre de fonctions n'est pas aussi grand qu'annoncé. Elles sont en fait organisées au sein de modules : `math`, `random`, `time` qu'il faut importer.

```
>>> import math
>>> dir(math)
```

```
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh',
atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf',
erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi',
'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

```
>>> help(math.tan)
```

```
Help on built-in function tan in module math:

tan(...)
    tan(x)

    Return the tangent of x (measured in radians).
```

```
>>> import random
>>> dir(random)
   ['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
   'SystemRandom', 'TWOPI', 'WichmannHill', '_BuiltinMethodType', '_MethodType',
   '__all__', '__builtins__', '__doc__', '__file__', '__name__', '__package__', '_acos',
   '_ceil', '_cos', '_e', '_exp', '_hashlib', '_hexlify', '_inst', '_log', '_pi',
   '_random', '_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_warn',
   'betavariate', 'choice', 'division', 'expovariate', 'gammavariate', 'gauss',
   'getrandbits', 'getstate', 'jumpahead', 'lognormvariate', 'normalvariate',
   'paretovariate', 'randint', 'random', 'randrange', 'sample', 'seed', 'setstate',
   'shuffle', 'triangular', 'uniform', 'vonmisesvariate', 'weibullvariate']
>>> help(random.random)
   Help on built-in function random:

   random(...)
       random() -> x in the interval [0, 1).
```

Remarque. On constate que pour utiliser une fonction contenue dans le module `module`, on doit utiliser la syntaxe `module.nom_fonction`.

4.3 Syntaxe de l'application d'une fonction

L'appel (ou l'application) d'une fonction obéit à une syntaxe bien particulière, semblable à celle utilisée en math. Par exemple, l'appel de la fonction `abs` avec l'argument `-3` s'écrit `abs(-3)`.

4.4 Évaluation de l'argument d'une fonction

Il n'est pas nécessaire de fournir une constante comme argument à une fonction, on peut lui passer une expression. Par exemple l'appel suivant `abs(3 + 4 / 2 + 7 // 3)` est tout à fait valide. L'expression est évaluée avant que la fonction n'ait commencé son traitement.

5 Définition d'une fonction

Malgré leur grand nombre, les fonctions prédéfinies du langage peuvent ne pas être suffisantes. Il faut alors définir soi-même une nouvelle fonction.

5.1 Type et signature d'une fonction

En mathématique, on définie la fonction f qui calcule le carré d'un nombre réel de la sorte :

$$\begin{array}{rccc} f\colon & \mathbb{R} & \to & \mathbb{R} \\ & x & \longmapsto & x^2 \end{array}$$

En informatique, on s'appuie sur cette écriture, en la précisant un peu, on dit que la fonction possède un **nom**, un **type**, une **signature**.

$$\underbrace{\underbrace{f\colon}_{Nom}\ \underbrace{\mathbb{R}\to\mathbb{R}}_{Type}}_{Signature}$$

Définition.

- *Le type d'une fonction est constitué par les types de ses paramètres et par le type de la valeur qu'elle retourne.*
- *La signature d'une fonction est l'association du nom de cette fonction et de son type.*

La connaissance de la signature d'une fonction est fondamentale, ce n'est pas parce qu'une fonction se nomme `sum` qu'elle est capable de faire la somme de deux nombres (cf. exercice 2) ! Le type de la fonction indique quels arguments elle accepte et donne une première (mais insuffisante !) indication sur la façon de l'utiliser.

Remarque. En informatique, on aura très souvent à utiliser des fonctions plus complexes que celle présentée ici, en particulier, des fonctions à plusieurs paramètres. Sans explication supplémentaire, il vous faudra admettre qu'une fonction nommée `surface` qui calcule la surface d'un rectangle de côtés a et b a pour définition

$$\begin{aligned} \text{surface} : \quad & \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\ & (a, b) \longmapsto a \times b \end{aligned}$$

5.2 Définition d'une fonction en Python

La syntaxe utilisée pour définir une fonction en Python est assez proche de celle des mathématiques :

```
def f(x: float) -> float:
    return x**2
```

On reconnait tout de suite la signature de la fonction `def f(x: float) -> float` qui indique bien que la fonction possède un paramètre `x` de type `float` et retourne une valeur de type `float`. *Chaque paramètre doit donc être accompagné de son type !*

Les deux points `:` servent à indiquer que le *bloc d'instructions* qui suit la signature constitue *le corps de la fonction*, c'est à dire l'ensemble des étapes qui sont réalisées lorsqu'on appelle la fonction. *Un bloc d'instructions doit toujours être indenté par rapport à la marge !*

L'instruction `return` *indique à la fonction de retourner la valeur qu'elle vient de calculer*.

La définition de la fonction `surface` est la suivante :

```
def surface(a: float, b: float) -> float:
    return a * b
```

5.3 Spécification d'une fonction

L'utilisation d'une fonction nécessite la description précise :

1. *des conditions dans lesquelles elle peut être utilisée : c'est le rôle de la* ***signature****.*
2. *de la valeur qu'elle retourne : c'est le rôle de la* ***documentation****.*

Il est donc indispensable d'établir *un contrat entre le concepteur de la fonction et son utilisateur*.

Définition. *On appelle spécification l'association de la* ***signature*** *et de la* ***documentation*** *de la fonction.*

Spécification = Signature + Commentaires

Remarque. *Une spécification ne s'intéresse pas au « comment » !* L'utilisateur d'une fonction n'a pas besoin de connaître en détail toutes les étapes du traitement du problème par la fonction, il doit juste savoir *quelle valeur elle retourne et comment il faut l'utiliser*.

On peut écrire la spécification suivante pour la fonction qui élève au carré :

```
def f(x: float) -> float:
    """ Retourne le carré du nombre x passé en argument. """
```

En fait, la spécification est la seule contrainte qu'est tenu de respecter le concepteur d'une fonction. Il faut donc faire la distinction entre toutes les spécifications suivantes :

– Le programmeur ne s'occupe pas vraiment des conditions d'utilisation de la fonction. Il présuppose que tous les utilisateurs penseront à vérifier que l'argument est bien positif lors de l'appel de la fonction. *Il incombe à l'utilisateur de bien utiliser la fonction.*

```
def racine_carree(x: float) -> float:
    """ Retourne la racine carrée du nombre x. """
```

– Le programmeur avertit l'utilisateur qu'il existe une condition de bonne utilisation de la fonction (l'argument doit être positif ou nul) mais n'empêche pas l'utilisateur d'appeler la fonction avec un mauvais argument. *Ici aussi il incombe à l'utilisateur de bien utiliser la fonction.*

```
def racine_carree(x: float) -> float:
    """ Retourne la racine carrée du nombre x positif ou nul. """
```

– Le programmeur prend en charge la bonne utilisation de la fonction. *Il prend en charge le bon fonctionnement de la fonction.*

```
def racine_carree(x: float) -> float:
    """
    Retourne la racine carrée du nombre x si positif ou nul.
    Retourne une erreur si x négatif.
    """
```

6 Exercices du chapitre

Exercice 2. Découverte de fonctions

Les deux fonctions intégrées `dir` et `help` sont fondamentales. Les utiliser *dans la console* pour répondre aux questions ci-dessous.

1. La fonction intégrée `sum` intégrée au langage calcule-t-elle la somme de deux nombres ?
2. Que réalisent les fonctions intégrées `int`, `float`, `str`, `bool` ?
3. Quel traitement réalise la fonction `pow` du module `math` ? Donner un exemple d'utilisation.
4. Comment se nomme la fonction du module `math` qui calcule la racine carrée d'un nombre passé en argument ?

Exercice 3. Utilisation d'une fonction

1. Peut-on appeler la fonction `randint` du module `random` sans argument ?

   Rappel. La fonction intégrée `help` permet d'obtenir la documentation de n'importe quelle fonction.

2. Que retourne l'appel de la fonction `randint` du module `random` ?

3. Appeler cette fonction `randint` de façon à obtenir un nombre entier aléatoire compris entre 1 et 10.

4. Est-il possible que l'appel précédent retourne 1 ou 10 ?

Exercice 4. Définition d'une fonction

1. La fonction définie ci-dessous est-elle syntaxiquement correcte ?

   ```
   def retire_un(n):
       return 1 - n
   ```

2. La spécification est la suivante :

   ```
   def retire_un(n: float) -> float:
       """ Retourne la valeur de n - 1 """
   ```

   La fonction telle qu'elle est définie à la question 1 correspond-elle à cette spécification ?

3. Corriger le corps de la fonction `retire_un` de façon à ce que le calcul effectué corresponde à ce qu'annonce la spécification.

Exercice 5. Calcul de fonctions polynomiales

1. Définir et tester une fonction écrite en Python, nommée `polynomiale`, telle que `polynomiale(a, b, c, d, x)` retourne la valeur de la fonction qui à x associe ax^3+bx^2+cx+d.

   La documentation complète de cette fonction est la suivante :

   ```
   def polynomiale(a:int, b:int, c:int, d:int, x: float) -> float:
       """ Retourne la valeur ax^3 + bx^2 + cx + d
       >>> polynomiale(1, 1, 1, 1, 2) 15.0
       >>> polynomiale(1, 1, 1, 1, 3) 40.0
       """
   ```

2. Définir et tester une fonction écrite en Python, nommée `polynomiale_carre` qui retourne la valeur de la fonction ax^4+bx^2+c.

   La documentation complète de cette fonction est la suivante :

   ```
   def polynomiale_carre(a:int, b:int, c:int, x: float) -> float:
       """ Retourne la valeur ax^4 + bx^2 + c
       >>> polynomiale(1, 1, 1, 1, 2) 15.0
       >>> polynomiale(1, 1, 1, 1, 3) 40.0
   ```

```
    """
```

Exercice 6. Calcul de la moyenne de trois nombres

1. Définir et tester la fonction nommée `somme`, écrite en Python, dont la spécification est :

```
def somme(x:float, y: float, z: float) -> float:
    """ Retourne la somme des trois nombres passés en argument. """
```

2. Définir et tester une fonction écrite en Python, nommée `moyenne`, qui détermine la moyenne arithmétique de trois nombres.

   Remarque. La fonction `moyenne` doit utiliser la fonction `somme` définie à la questions précédente pour effectuer son traitement.

   La spécification de la fonction est la suivante :

```
def moyenne(a: float, b: float, c: float) -> float:
    """ Retourne la moyenne des trois nombres passés comme arguments. """
```

3. Définir et tester une fonction écrite en Python, nommée `moyenne_ponderee`, qui détermine la moyenne pondérée de trois nombres avec des coefficients variables. Les paramètres devront être écrits dans l'ordre suivant : *d'abord les trois nombres, puis les trois coefficients*.

   Remarque. La fonction `moyenne_ponderee` doit effectuer deux appels à la fonction `somme` pour effectuer son traitement.

   La documentation complète de la fonction est la suivante :

```
def moyenne_ponderee(x:float, y:float, z:float, a:int, b:int, c:int) -> float:
    """ Retourne la moyenne pondérée des nombres x, y et z par les coefficients a,
    b et c.

    >>> moyenne_ponderee(5, 10, 15, 1, 1, 1)
    10.0

    >>> moyenne_ponderee(5, 10, 15, 1, 1, 0)
    7.5
    """
```

Exercice 7. Calculs de surfaces et de volumes

1. Définir et tester une fonction écrite en Python, nommée `surface_rectangle`, qui détermine la surface d'un rectangle de longueur a et de largeur b.

   Préciser la spécification de la fonction.

2. Définir et tester une fonction écrite en Python, nommée `volume_parallelepipede`, qui détermine le volume d'un parallélépipède rectangle de longueur a, de largeur b et de hauteur h.

   Préciser la spécification de la fonction.

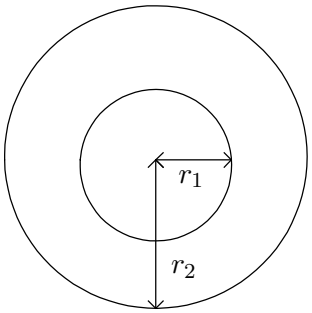
Cette fonction devra utiliser la fonction `surface_rectangle` *pour effectuer son traitement son traitement.*

3. Définir et tester une fonction écrite en Python, nommée `surface_disque`, qui détermine la surface d'un disque de rayon r.

   Rappel. Le module `math` possède une variable `pi`.

   Préciser la spécification de la fonction.

4. Définir et tester une fonction écrite en Python, nommée `surface_couronne`, qui détermine la surface d'une couronne de rayon intérieur r_1 et de rayon extérieur r_2.



   Préciser la spécification de la fonction.

   Cette fonction devra utiliser la fonction `surface_disque` *pour effectuer son traitement.*

5. Définir et tester une fonction écrite en Python, nommée `volume_tube`, qui détermine le volume de la partie pleine d'un tube de longueur l, dont la section est une couronne de rayon intérieur r_1 et de rayon extérieur r_2.

   Préciser la spécification de la fonction.

   Cette fonction devra utiliser la fonction `surface_couronne` *pour effectuer son traitement.*

Exercice 8. Calcul d'une distance

Définir et tester une fonction écrite en Python, nommée `distance` qui détermine la distance séparant deux points de coordonnées (x_1, y_1) et (x_2, y_2) d'un plan.

La documentation complète de la fonction est la suivante :

```
def distance(x1: float, y1: float, x2: float, y2: float) -> float:
    """ Retourne la distance dans le plan entre les deux points de coordonnées
    (x1, y1) et (x2, y2).

     >>> distance(0, 0, 1, 1)
     1.4142135623730951
     """
```

Exercice 9. Reprise de l'exercice 1

L'objectif de cet exercice est de retrouver une partie du comportement de la fonction `randint` du module `random` à partir de la fonction `random` de ce même module.

Définir et tester une fonction écrite en Python, nommée `tirage_entier`, telle que `tirage_entier(a)` retourne un nombre entier compris entre 1 et a (inclus), au hasard. *La fonction* `tirage_entier` *doit utiliser la fonction* `random` *du module* `random`.

Remarque. Lire attentivement la documentation de la fonction `random` afin de comprendre ce qu'elle retourne.

Rappel. La fonction intégrée `int` permet de convertir n'importe quel nombre en nombre entier.