Introduction

Dans ce chapitre, on présente les constructions qui constituent le noyau du langage du programmation Python :

- les littéraux, les types de base (`int`, `float`, `bool`);
- la notion de variable, l'instruction d'affectation et la distinction entre expression et instruction;
- les instructions constituant les briques de base d'un langage de programmation :
  - la boucle bornée `for`;
  - les instructions de branchement `if ... elif ... else`;
  - la boucle non bornée `while`.
- les fonctions qui permettent de réutiliser et rendre plus lisible le code.

1 Bases d'un langage de programmation : instructions, littéraux, expressions

1.1 Langage de programmation et instruction

Définition 1

Un **langage de programmation** permet d'écrire des programmes qui sont exécutés par un ordinateur.
Python est un **langage de programmation interprété** créé par Guido Van Rossum dans les années 1980.
On peut évaluer du code Python en **mode interactif** dans une console : un prompt précédé d'une invite comme »> ou `In[1]` attend la saisie d'une expression Python bien formatée, son évaluation est affichée directement en dessous avec éventuellement affichage d'un message d'erreur.
Le **mode interactif** est pratique pour tester de petits bouts de code mais on écrit un programme structuré dans le **mode programme** : le texte du programme est saisi dans un éditeur de texte puis exécuté avec la commande `python` ou le bouton `Exécuter` de l'environnement de programmation utilisé. Contrairement au **mode interactif** on peut enregistrer un programme dans une mémoire pérenne sous la forme d'un fichier texte. Par convention on donne l'extension `.py` aux programmes Python désignés souvent comme des **scripts**.

Mode interactif

```
>>> 1 + 2
3
>>> "1" + "2"
'12'
>>> "1" + 2
Traceback (most recent call last
    ):
  File "<stdin>", line 1, in <
      module>
TypeError: can only concatenate
    str (not "int") to str
```

Exécution d'un script Python

```
user@pc~$ python hello_world.py
Hello world
```

Programme 1

Voici un exemple de programme Python qui récupère des données sur son **entrée standard** (ici une saisie de l'utilisateur avec la fonction `input` mais ce pourrait être un fichier externe) , les traite puis renvoie des valeurs sur sa **sortie standard** (ici la console utilisateur avec la fonction `print` mais ce pourrait être un fichier externe) :

```
#Définition de fonctions
def f(x):
    return x ** 2 - 3

## Programme principal

#entrées
a = float(input('Borne inférieure ? '))
b = float(input('Borne supérieure ? '))
s = float(input("Seuil de l'encadrement ?"))

#traitement
while b - a > s: #boucle non bornée
    m = (a+b)/2 #affectation
    if f(m) < 0: #conditionnelle, branchement
        a = m
    else:
        b = m

#sorties
print(a, "<= racine(3) <= ", b)
```

Définition 2

Un **programme Python** est un texte structuré comme une séquence d'**instructions**.
Un **interpréteur Python** exécute le programme sur un ordinateur en mobilisant des ressources de calcul (processeur) et de mémoire.
L'exécution d'une instruction peut modifier **l'état courant du programme**, par **effet de bord**.
Après l'exécution d'une instruction, l'interpréteur évalue par défaut l'instruction sur la ligne suivante (les lignes vides ne sont pas prises en compte) mais certaines instructions se traduisent par des sauts en avant (branchement) ou en arrière (boucle) dans le texte du programme.
Les séquences de caractères précédées d'un dièse # ne sont pas interprétées, ce sont des **commentaires**.

1.2 Environnements pour programmer en Python

Méthode

Pour programmer en Python, on peut :

- ☞ Installer une distribution Python comprenant un interpréteur et un environnement de programmation :
  - la plus simple est Idle disponible sur le site officiel Python ;
  - une distribution complète avec une interface simple et un débogueur très visuel : `https://thonny.org/` ;
  - une distribution plus lourde mais plus complète avec tous les modules scientifiques est Anaconda.
- ☞ Utiliser un interpréteur intégré au navigateur Web :
  - `http://pythontutor.com/visualize.html#mode=edit` est idéal pour visualiser l'exécution du code mais propose peu de modules/bibliothèques externes ;
  - `https://console.basthon.fr/` est plus riche en modules/bibliothèques externes.
  - les activités **Capytale** partagées par les professeurs dans l'ENT (Ressources numériques) sont un autre moyen d'exécuter du code Python dans le navigateur.

1.3 Littéraux et types de base

Définition 3

Un **littéral** est un texte qui est interprété par Python pour créer un **objet** en mémoire avec une valeur bien spécifiée.
Un **objet** Python est caractérisé par son **type** et sa **valeur**.
On obtient le type d'un objet en lui appliquant la fonction `type`.

```
>>> type(842)
```

```
<class 'int'>
>>> type("842")
<class 'str'>
```

Méthode

Les objets de même type peuvent être combinés à l'aide d'opérateurs pour créer d'autres objets de même type.
Une opération entre des objets de types différents provoque en général une erreur sauf pour des cas particuliers comme les types numériques pour lesquels il existe des règles de conversion implicite.
Les quatre types de base sont :

Type	Domaine de valeurs	Opérateurs	Exemple de littéraux
`int`	entiers signés	`+ - * // % **`	`0 -4 842`
`float`	sous-ensemble des décimaux	`+ - * / **`	`0.0 -1.0 3.14`
`bool`	valeurs logiques	`not and or`	`True False`
`str`	chaînes de caractères	`+`	`'tb' " '2.4' 'Bonjour'`

On peut ajouter un type spécifique `None` sur lequel on ne définit pas d'opérateur car tous les objets de valeur `None` sont identiques.

Dans le cas d'une combinaison de plusieurs opérateurs, des **règles de précédence** (ou priorité) déterminent l'ordre dans lequel les opérations sont effectuées. Les priorités usuelles des opérations algébriques sont bien connues (attention l'exponentiation a la plus haut priorité). On peut changer l'ordre de priorité en utilisant des **parenthèses**. Pour les opérations booléennes, les opérateurs classés par ordre décroissant de priorité sont `not`, `and` puis `or`.
Il est fortement recommandé d'utiliser des parenthèses qui sont l'opérateur de plus haute priorité, lorsqu'on n'est pas sûr des règle de précédence ou pour s'en affranchir.

Exemple 1 *Opérations sur les types de base*

```
>>> 11 + 3  #addition
14
>>> 11 * 3 #multiplication
33
>>> 11 ** 3 #exponentiation
1331
>>> 11 // 3 #quotient de la division euclidienne
3
>>> 11 % 3  #reste de la division euclidienne
2
>>> 11 / 3  #division décimale
3.6666666666666665
>>> not True #négation booléenne
False
>>> True or False #disjonction booléenne
True
>>> True and False #conjonction booléenne
```

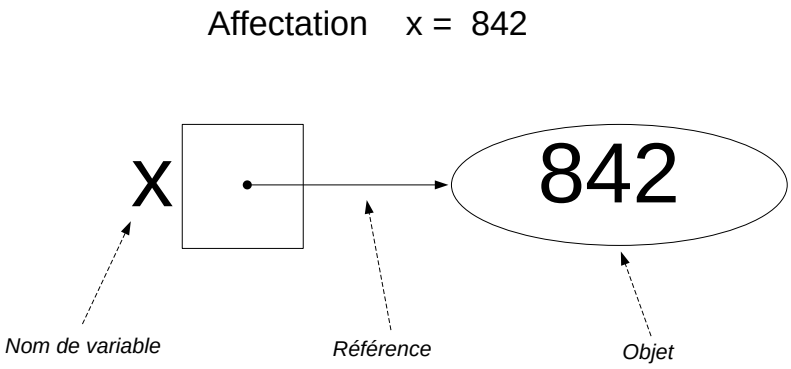
```
False
>>> '2' + 1   #impossible d'ajouter un 'str' et un 'int'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> '2' + '1'  #concaténation de 'str'
'21'
>>> not True and False  #not prioritaire sur and
False
>>> not(True and False) #dans le doute on met des parenthèses
True
```

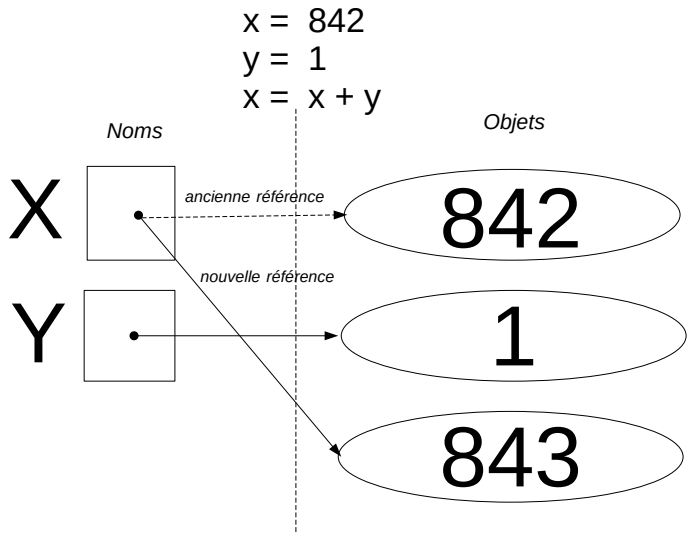
1.4 Variable et expression

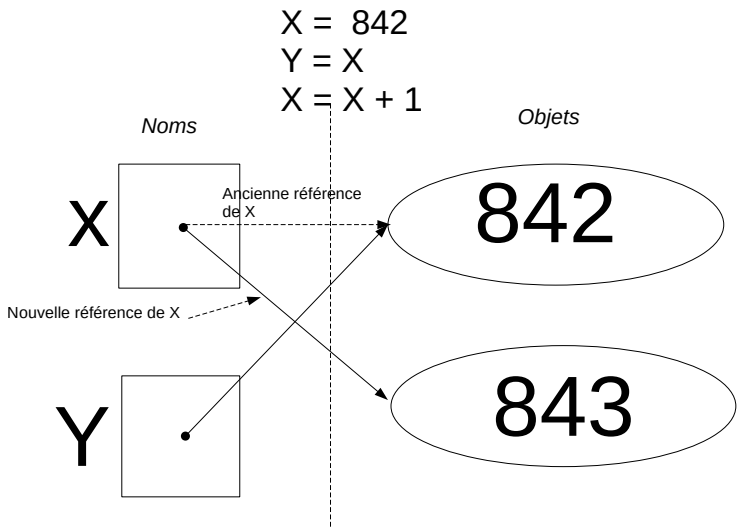
Définition 4

Dans un programme, pour manipuler des objets Python, on a besoin de les référencer par des noms.
Une **variable** est l'association entre un **nom** et un objet Python qu'on désigne souvent comme **valeur** de la variable.
L'opérateur = réalise cette association. L'interpréteur Python évalue d'abord le membre de droite pour créer l'objet puis l'associe au membre de gauche contenant le **nom**.
Cette **instruction** s'appelle une **affectation de variable**. Comme elle modifie **l'état courant** du programme on parle d'**effet de bord**.

Voici les représentations de quelques séquences d'affectations :







Définition 5

Une **expression** est une combinaison de littéraux et de variables dont la valeur peut être évaluée par l'interpréteur en remplaçant les noms de variables par leur valeur.
Par exemple si dans l'état courant du programme la valeur de `x` est 842 alors l'expression `x + 1` a pour valeur 843.

Remarque 1

- Les noms de variables en Python doivent obéir à certaines règles syntaxiques (ne pas commencer par un chiffre, ne peut pas contenir de tiret haut). Il est recommandé de les choisir en minuscules et d'utiliser le tiret bas comme séparateur si un nom est constitué de plusieurs mots comme `compteur_vie`.
- L'instruction `x = x + 1` permet d'incrémenter la valeur de la variable `x` (à condition qu'elle soit définie sinon cela provoque une erreur).

  Dans une affectation, l'opérateur `=` n'est pas un opérateur d'égalité et `x = x + 1` ne doit

pas être interprété comme une égalité. Le `x` à droite de l'opérateur représente la valeur associée précédemment au nom `x` tandis que le `x` à gauche est un nom auquel sera associée la valeur de l'expression `x + 1`.

Méthode

Pour bien comprendre un programme on peut compléter un **tableau d'état** : on exécute le programme comme le ferait l'interpréteur, en notant les valeurs de toutes les variables pour chaque instruction qui une modifie l'état de la mémoire par effet de bord.
Cet exemple classique permet de comprendre qu'il faut une variable de stockage pour sauvegarder la valeur de `a` si on veut échanger les valeurs des variables `a` et `b`

```
1 a = 842
2 b = 843
3 a = b
4 b = a
```

Instruction (numéro de ligne)	a	b
ligne 1	842	
ligne 2	842	843
ligne 3	843	843
ligne 4	843	843

Pour échanger les valeurs des variables `a` et `b` on peut utiliser l'un des deux programme ci-dessous, celui de droite plus pythonique utilise le déballage de `tuple`.

```
1 a = 842
2 b = 843
3 c = a
4 a = b
5 b = c
```

```
1 a = 842
2 b = 843
3 a, b = b, a
```

2 Instructions conditionnelles (ou de branchement)

Exemple 2

Considérons l'exemple classique d'un programme de résolution dans l'ensemble des réels d'une équation du second degré à coefficients réels.

```
a = input('a ?')
b = input('b ?')
c = input('c ?')
delta = b**2 - 4*a*c
if delta >= 0:
    x1 = (-b - sqrt(delta)) / (2 * a)
    x2 = (-b + sqrt(delta)) / (2 * a)
    print("Deux racines distinctes de valeurs x1 et x2")
elif delta == 0:
    x0 = -b/(2*a)
    print(Une racine double de valeur x0)
else:
```

```
    print("Pas de racine réelle")
```

Définition 6

Dans l'exemple précédent, des **instructions de test** permettent de choisir entre plusieurs branches pour continuer l'exécution du programme selon le signe du discriminant. La syntaxe générale des instructions de branchement est :

```
if condition1:
    instruction     #
      ...           # bloc d'instructions 1
    instruction     #
elif condition2:
    instruction     #
      ...           # bloc d'instructions 2
    instruction     #
    instruction
elif ...
    .....
else :
    instruction     #
      ...           # dernier bloc d'instructions
    instruction     #
```

Seul le premier mot clé `if` est obligatoire, les autres `elif` et `else` sont optionnels. Noter que `else` n'est pas suivi d'une condition.
Les conditions sont des variables ou des expressions (souvent des comparaisons) de type booléen.
Si `condition1` a pour valeur `True` alors le bloc d'instructions 1 est exécuté.
Si ce n'est pas le cas, `condition2` est évaluée. Si sa valeur est `True` alors le bloc d'instructions 2 est exécuté.
Sinon on passe au `elif` suivant et ainsi de suite.
Si aucune des conditions présentes derrière un des mot-clé `elif` n'est évaluée à `True`alors le dernier bloc est exécuté.
Remarque : Un seul des blocs d'instructions peut être exécuté : le premier possible ceci même si l'état courant rend plusieurs des conditions valides (`True`).
Les conditions sont souvent construites à l'aide des opérateurs de comparaison et des opérateurs logiques pour créer des expressions booléennes.

Principaux opérateurs de comparaison de variables

`x == y`	`x est égal à y`
`x != y`	`x est différent de y`
`x > y`	`x est strictement supérieur à y`
`x < y`	`x est strictement inférieur à y`
`x >= y`	`x est supérieur ou égal à y`
`x <= y`	`x est inférieur ou égal à y`

Attention à ne pas confondre l'opérateur d'égalité == avec l'opérateur d'affectation = .

Principaux opérateurs sur des expressions booleenes

`E and F`	`Vraie si E est Vraie ET F est Vraie`
`E or F`	`Vraie si E est Vraie OU F est Vraie`
`not E`	`Vraie si E est Fausse`

On peut combiner des opérateurs arithmétiques, de comparaison et logiques pour créer des expressions booléennes complexes. Il faut prêter attention aux ***règles de précédence****. Les parenthèses sont prioritaires sur tous les autres opérateurs donc on peut les utiliser quand on n'est pas certain des* ***règles de précédence*** *ou pour s'en affranchir.*

```
>>> 3<4 and 5 == 2*2+1 #on fait confiance aux règles de précédence
True
>>> (3<4) and (5 == 2*2+1) #avec des parenthèses, + sur et + lisible
True
```

3 Boucle bornée

Définition 7 ***Boucle*** `for`

Lorsque l'on veut répéter un certain nombre de fois un ensemble d'instructions on utilise la **boucle bornée** ou boucle `for` :

Programme 2

```
>>> for k in range(4):
        print("bonjour")

bonjour
bonjour
bonjour
bonjour
```

Programme 3

```
>>> for k in range(4):
        print(k)

0
1
2
3
```

Ici la fonction range renvoie un *itérateur* qui produit consécutivement les valeurs entières de 0 à 3. La boucle `for` ne fait que parcourir les valeurs de cet itérateur.
Plus généralement, la boucle `for` permet de parcourir tout objet *itérable* :

Programme 4

```
>>> for c in 'AB':
        print(c)

A
B
```

Programme 5

```
>>> for x in [2, 6]:
        print(x**2)

4
36
```

La syntaxe générale d'une boucle inconditionnelle `for` est :

```
for element in iterable:
    instruction   #
    instruction   # bloc d'instructions
    instruction   #
```

La fonction `range` possède trois arguments dont deux sont optionnels :

- `range(n)` renvoie un itérateur parcourant les entiers consécutifs entre 0 et `n` exclu.
- `range(m,n)` renvoie un itérateur parcourant les entiers consécutifs entre `m` compris et `n` exclu.
- `range(m,n,s)` renvoie un itérateur parcourant les entiers consécutifs entre `m` compris et `n` exclu avec un pas de `s`.

Remarque 2

En Python, le **bloc** d'une boucle inconditionnelle est délimité par :

- Un **marqueur de début de bloc**, le caractère : à la fin de l'instruction définissant la boucle ;
- Une **indentation** (décalage en espaces par rapport à la marge de gauche) commune à toutes les lignes d'instruction appartenant au bloc de la boucle.

Les instructions exécutées après la boucle ont un niveau d'indentation inférieur à celui du **bloc** de boucle.

*En Python, **l'indentation** n'est pas un simple élément de présentation mais bien un élément de syntaxe,* `IndentationError` *est un message d'erreur classique :*

```
(base) fjunier@fjunier:~$ cat test.py
s  = 0
for k in range(10):
s = s + k
(base) fjunier@fjunier:~$ python3 test.py
  File "test.py", line 3
    s = s + k
    ^
IndentationError: expected an indented block
```

Méthode

Le bloc d'une boucle peut contenir une autre boucle, on parle alors de **boucles imbriquées**.

- Pour énumérer tous les couples (i, j) avec $1 \leqslant i \leqslant 3$ et $1 \leqslant j \leqslant 3$, on peut écrire le programme `couple_avec_repetition.py` :

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i, j)
```

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

- Pour énumérer tous les couples (i, j) avec $1 \leqslant i < j \leqslant 3$, on peut écrire le programme `couple_ordre_croissant.py` :

```
for i in range(1,4):
    for j in range(i+1,4):
        print(i, j)
```

```
fjunier@fjunier:~$ python3
    couple_ordre_croissant.py
1 2
1 3
2 3
```

4 Boucle non bornée

Définition 8

Dans une boucle `for`, le nombre d'itérations est connue à l'avance (au plus tard lors de la première exécution de l'instruction). Lorsque l'on ne sait pas à l'avance combien de fois la boucle devra être exécutée, on utilise une **boucle non bornée** `while`.
La syntaxe est la suivante :

```
while condition:
    instruction   #
    instruction   #   bloc d'instructions
    instruction   #
```

La `condition` est une expression qui doit pouvoir être évaluée sous forme de booléen. Tant que sa valeur est True, le bloc d'instructions est exécuté.

Lors de l'utilisation de `while`, il faudra s'assurer que la condition finisse par prendre la valeur `False` sans quoi la boucle ne se terminera pas!!
Remarque : pour faire une opération jusqu'à la vérification d'une certaine condition il suffit d'ajouter l'opérateur logique `not()`.

Exemple 3

- Un exemple classique est celui du calcul du PGCD de deux entiers `a` et `b` non tous nuls par la méthode d'Euclide.

```
a = int(input('Entier a ?'))
b = int(input('Entier b ?'))
while b != 0:
    tmp = a
    a = b
    b = tmp % b   #reste de la division euclidienne
print("PGCD : ", a)
```

- Les algorithmes de seuil, sont un classique de l'enseignement de l'algorithmique dans le secondaire. Par exemple si on considère qu'une population augmente de 2 % par an on peut déterminer le nombre d'années au bout duquel sa valeur initiale sera doublée avec le programme suivant :

```
population = int(input("Population initiale ? "))
double = 2 * population
n = 0
taux = 2 / 100
while population < double:
    population = population * (1 + taux)
    n = n + 1
print("Doublement en ", n, "années")
```

5 Fonctions

5.1 Définir une fonction

Définition 9

Lorsqu'on a besoin de réutiliser tout un bloc d'instructions, on peut l'encapsuler dans une **fonction**. On étend ainsi le langage avec une nouvelle instruction. Une fonction sert à factoriser et clarifier le code, elle facilite la maintenance et le partage. C'est un outil de **modularité**.
Pour déclarer une fonction, on définit son **en-tête** (ou **signature**) avec son **nom** et des **paramètres formels** d'entrée. Vient ensuite le bloc d'instructions, décalé d'une indentation et qui constitue le **corps** de la fonction.

Fonction avec `return`

```
def mafonction(parametre1, parametre2): #signature
    bloc d'instructions (optionnel)
    return valeur
```

Fonction sans `return`

```
def mafonction(parametre1, parametre2): #signature
```

```
    bloc d'instructions (non vide)
```

Si le corps de la fonction contient au moins une instruction préfixée par le mot clef `return` alors l'exécution d'un `return` termine l'exécution du corps de la fonction et renvoie une valeur au programme principal. Si le `return` est dans une structure de contrôle (boucle, test), il est possible que le corps de la fonction ne soit pas entièrement exécuté, on parle de **sortie prématurée**.
Une fonction sans `return` s'appelle une **procédure**, elle modifie l'état du programme principal par **effet de bord**. En Python, une procédure renvoie quand même la valeur spéciale `None` au programme principal.

On exécute une fonction en substituant aux **paramètres formels** des valeurs particulières appelées **paramètres effectifs**. On parle d'**appel de fonction**, on peut l'utiliser comme une **expression** si une valeur est renvoyée ou comme une **instruction** s'il s'agit d'une procédure.

Exemple 4

Par exemple une fonction `carre` qui prend en paramètre un nombre `x` et qui renvoie son carré, s'écrira :

Programme 6

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

Une fonction peut prendre plusieurs paramètres. Par exemple une fonction `carre_distance_origine(x,y)` qui prend en paramètres deux nombres `x` et `y` et qui renvoie le carré de la distance d'un point de coordonnées `(x, y)` à l'origine d'un repère orthonormal, s'écrira :

Programme 7

```
def carre_distance_origine(x, y):
    return x ** 2 + y ** 2
```

Une fonction peut retourner un tuple de valeurs. Par exemple une fonction `coord_vecteur` qui prend en paramètres quatre nombres `xA, yA, xB, yB` et qui retourne les coordonnées du vecteur lié dont les extrémités ont pour coordonnées `(xA, yA)` et `(xB, yB)`, s'écrira :

Programme 8

```
def coord_vecteur(xA, yA, xB, yB):
    return (xB - xA, yB - yA)
```

Voici un exemple de fonction sans paramètres d'entrée, ni valeur de retour (il s'agit donc d'une procédure).

Programme 9

```
def message_defaite():
    print("Vous avez perdu, merci d'avoir participé")
```

Attention, `return valeur` renvoie `valeur` qu'on peut capturer dans une variable alors que `print(valeur)` affiche `valeur` sur la sortie standard (la fenêtre où s'exécute l'interpréteur) mais `valeur` ne peut alors être capturée dans une variable. On donne ci-dessous un extrait de console Python, où on a défini maladroitement une fonction cube avec un `print` à la place d'un `return`. On ne récupère pas la valeur de retour souhaitée mais `None` lorsqu'on appelle la fonction.

```
In [10]: def cube(x):
    ...:     print(x ** 3)
    ...:

In [11]: cube(4)
64

In [12]: b = cube(5)
125

In [13]: b

In [14]: print(type(b))
<class 'NoneType'>

In [15]: print(b + 1)

TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

5.2 Utiliser des bibliothèques de fonctions

Méthode

On a parfois besoin d'utiliser des fonctions de Python qui ne sont pas chargées pas défaut. Ces fonctions sont stockées dans des programmes Python appelées **modules** ou **bibliothèques**. Par exemple le module `math` contient les fonctions mathématiques usuelles et le module `random` contient plusieurs types de générateurs de nombres pseudo-aléatoires.
Pour importer une fonction d'un module on peut procéder de deux façons :

```
#import du module de mathématique (création d'un point d'accès)
import math
```

```
#pour utiliser la fonction sqrt, on la préfixe du nom du module et d'un
    point
racine = math.sqrt(2)
```

Première façon

```
#import de la fonction sqrt du module math
from math import sqrt

racine = sqrt(2)

#Pour importer toutes les fonctions de math, ecrire
#from math import *
```

Deuxième façon

Pour obtenir de l'aide sur le module math dans la console Python, il faut d'abord l'importer avec `import math` puis taper `help(math)`, mais le mieux est encore de consulter la documentation en ligne https://docs.python.org/3/.

Méthode *Bibliothèques utiles pour les mathématiques*

- Le module `math` rassemble les fonctions mathématiques usuelles :

Fonction	Spécification
`math.sqrt(x)`	renvoie la racine carré du réel $x \geqslant 0$
`math.cos(x)`	renvoie le cosinus du réel x
`math.sin(x)`	renvoie le sinus du réel x
`math.exp(x)`	renvoie l'exponentielle du réel x
`math.log(x)`	renvoie le logarithme népérien du réel $x > 0$

- Le module `random` rassemble diverses fonctions simulant le hasard :

Fonction	Spécification
`random.randrange(a,b)`	renvoie un entier aléatoire dans [a;b[
`random.randint(a,b)`	renvoie un entier aléatoire dans [a;b]
`random.random()`	renvoie un décimal aléatoire dans [0;1[
`random.uniform(a,b)`	renvoie un décimal aléatoire dans [a;b]

6 Erreurs

Définition 10

On distingue plusieurs types d'**erreurs** ou **bugs** lors de l'exécution d'un programme Python. Le **débogage** d'un programme s'appuie d'abord sur la lecture attentive des messages d'erreurs de l'interpréteur et sur un traçage des valeurs par exemple en insérant des `print` ou en utilisant les outils de l'environnement de programmation (points d'arrêt, débogueur).
Python caractérise chaque erreur par un **type**, voici les plus courants :

- une **erreur de syntaxe** se produit lorsque la syntaxe du langage Python n'est pas respectée :

```
>>> 4 = a
  File "<stdin>", line 1
SyntaxError: cannot assign to literal
```

- une **erreur de définition** se produit lorsqu'on évalue une expression avec des valeurs de variables non définies :

```
>>> 3 * a +1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

- une **erreur de type** se produit lorsqu'on évalue une expression avec des valeurs de variables non définies :

```
>>> 2 + '2'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- une **erreur d'exécution** se produit lorsque l'interpréteur ne peut pas évaluer une expression ou exécuter une instruction :

```
>>> def division(a, b):
...     return a / b
...
>>> x = 8
>>> y = 0
>>> z = division(x, y)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in division
ZeroDivisionError: division by zero
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

- une **erreur de logique** se produit lorsque le programme ne produit pas le résultat attendu ou ne se termine pas (boucle infinie) :

```
>>> x = 1
>>> while x > 0:
...     x = x +1     #boucle infinie, pour en sortir CTRL + C
^CTraceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```