Python 3 — Introduction

Yves Scherrer

Septembre 2015

1 Introduction

1.1 Objectifs

Ce document s'adresse à des étudiants de maîtrise qui ont des connaissances approfondies d'un ou de plusieurs autres langages de programmation. Son objectif est d'introduire le langage de programmation Python de façon succincte. Il introduit les structures de données et les structures de contrôles définies dans Python, et décrit le contenu de quelques modules importants.

1.2 Lectures complémentaires

Vous pouvez trouver des informations complémentaires dans le matériel suivant:

- Tutoriel officiel:
  https://docs.python.org/3/tutorial/index.html
- Documentation officielle complète:
  https://docs.python.org/3/library/index.html
- Cours d'introduction destiné à des programmeurs avancés, aussi disponible en PDF:
  http://www.diveintopython3.net/
- Manuel complet à propos de Python 3, avec beaucoup d'exemples:
  Mark Lutz (2010): Programming Python. O'Reilly.

Le matériel suivant est très utile pour apprendre Python 2, mais n'a pas été mis à jour pour Python 3:

- Un cours en ligne par Google:
  https://developers.google.com/edu/python/
- Référence compacte des commandes et modules les plus utilisés:
  http://rgruet.free.fr/PQR27/PQR2.7.html

2 Comment utiliser Python

Python peut être utilisé de deux manières distinctes, soit pour exécuter des commandes individuelles dans l'interpréteur interactif, soit pour exécuter des scripts. Ces deux modes sont disponibles dans l'éditeur IDLE qui est compris dans toute installation Python. Nous allons utiliser IDLE comme environnement de travail, du moins au début du cours.

2.1 Interpréteur interactif avec IDLE

Lorsque vous lancez IDLE, vous êtes directement présentés avec un interpréteur interactif, identifiable par le prompt >>>. Vous pouvez y écrire des commandes individuelles et les exécuter une par une. Ce mode est utile pour effectuer des tâches simples ou pour tester des commandes. En revanche, il n'est pas adapté pour exécuter des programmes plus complexes.

2.2 Exécution de scripts avec IDLE

Dans IDLE, vous pouvez créer des nouveaux fichiers et y écrire vos programmes. Un fichier ouvert est exécuté à l'aide du menu *Run* → *Run Module* ou de la touche F5. Les scripts sont sauvegardés avec l'extension *.py*.

2.3 Interpréteur interactif sans IDLE

Si vous ne disposez pas de IDLE, vous pouvez lancer l'interpréteur interactif en ouvrant un terminal du système et en tapant *python3* (Linux et Mac) ou *py -3* (Windows).

Vous sortez de Python avec *Ctrl+D* (UNIX) ou *Ctrl+Z* puis *Enter* (Windows).

2.4 Exécution de scripts sans IDLE

Ouvrez un terminal, positionnez-vous dans le répertoire où se trouve votre script, et lancez-le en tapant *python3 monscript.py* ou *py -3 monscript.py*).

Si vous travaillez sur un système UNIX, vous pouvez également lancer votre script de façon plus simple en tapant *./monscript.py*. Pour cela, deux conditions doivent être remplies:

- Le chemin d'accès de l'interpréteur doit être indiqué à la première ligne du script de la façon suivante:

  ```
  #! /usr/bin/python3
  ```

  ou:

  ```
  #! /usr/bin/env python3
  ```

- Le script doit avoir des permissions d'exécution. Ces permissions peuvent être accordées avec la commande suivante (dans un terminal UNIX, pas dans Python):

  ```
  chmod u+x monscript.py
  ```

2.5 Quelques éléments de syntaxe

- Les **commentaires** sont introduits par le symbole # et courent jusqu'à la fin de la ligne.

```
>>> c = 13.2          # ceci est un commentaire
```

- Le caractère de fin de ligne termine l'instruction.
  - Il n'est pas nécessaire de terminer les instructions par des points-virgules.
  - Le point-virgule est nécessaire seulement pour séparer plusieurs instructions mises sur la même ligne:

```
>>> r = 2; s = 7
```

- Les variables ne sont pas typées. Elles peuvent changer de type en cours d'exécution.
- Il n'y a pas besoin de déclarer les variables explicitement.

3 Types de base

3.1 Nombres

L'assignation de valeurs se fait avec =:

```
>>> a = 5.0
>>> b = 2.5
```

Python peut être utilisé comme une calculatrice simple. La syntaxe des expressions correspond à celle utilisée dans d'autres langages, avec les opérateurs +, -, * and /:

```
>>> a + b
7.5
>>> a - b
2.5
>>> a * b
12.5
>>> a / b
2.0
```

Incrémentations:

```
>>> a += 1        # a = a + 1  (a++ ne marche pas)
>>> a -= 3        # a = a - 3
```

Conversions:

```
>>> int(2.7)      # convertit un réel en entier
2
>>> float(2)      # convertit un entier en réel
2.0
```

Autres opérations:

```
>>> a ** b            # a puissance b
55.901699437494742
>>> abs(-3.0)         # valeur absolue
3.0
```

Division réelle et division entière:

```
>>> a = 5
>>> b = 2
>>> a / b         # division réelle
2.5
>>> a // b        # division entière
2
>>> a % b         # modulo
1
```

Python dispose également d'un type booléen, avec les constantes et opérations suivantes:

```
>>> x = True      # constantes booléennes avec majuscules
>>> y = False
>>> not y         # négation
True
>>> x and y       # conjonction
False
>>> x or y        # disjonction
True
```

3.2 Chaînes de caractères

On peut délimiter les chaînes soit par ", soit par ':

```
>>> 'hello'
>>> 'how\'s it going?'
>>> "how's it going?"
>>> 'This is "strange"'
```

En utilisant des guillemets triples (soit """ soit '''), on peut indiquer des chaînes de caractères multi-lignes qui retiennent les fins de lignes:

```
>>> """
... This is a string that when
... printed maintains its format
... and its end of lines.
... """
>>> print("""
... This is a string that when
... printed maintains its format
```

```
... and its end of lines.
... """)
```

On peut concaténer des chaînes avec l'opérateur + et les répéter avec l'opérateur *:

```
>>> 'Hello ' + ', ' + 'how are you'
>>> 'help' + '!'*5
```

On peut accéder des caractères individuels d'une chaîne de caractères en utilisant des indices. Le premier caractère a l'indice 0. Des sous-chaînes peuvent être spécifiées avec deux indices séparées par un deux-points:

```
>>> s = "bonjour tout le monde"
>>> s[0]         # extraction du premier caractère
'b'
>>> s[3:8]       # extraction d'une sous-chaîne
'jour'
>>> s[:3]        # extraction à partir du début
'bon'
>>> s[13:]       # extraction jusqu'à la fin
'le monde'
```

Les indices négatifs sélectionnent des caractères ou des sous-chaînes depuis la fin. La fonction *len()* retourne la longueur de la chaîne:

```
>>> len(s)       # longueur de la chaîne
21
>>> s[len(s)-1]  # extraction du dernier caractère
'e'
>>> s[-1]        # même signification
'e'
>>> s[4:-4]      # extraction d'une sous-chaîne
'our tout le m'
```

Opérations sur les chaînes de caractères:

```
>>> s = '  Bonjour tout le "MONDE"!  '
>>> s.strip()         # enlève les espaces au début et à la fin
'Bonjour tout le "MONDE"!'
>>> s.lower()                    # tout en minuscules
'  bonjour tout le "monde"!  '
>>> s.upper()                    # tout en majuscules
'  BONJOUR TOUT LE "MONDE"!  '
```

Recherche et remplacement:

```
>>> r = 'Bonjour tout le "MONDE"!'
>>> r.startswith('Bon')
True
>>> r.endswith("bla")
False
```

```
>>> r.find(”tout”)        # retourne l’indice du 1er caractère
                          # de la sous-chaîne
8
>>> r.find(”rien”)        # pas trouvé => -1
-1
>>> r.count(”ou”)         # dans ”Bonjour” et dans ”tout”
2
>>> r.replace(”Bonjour”, ”Bonsoir”)   # toutes les occurrences
’Bonsoir tout le ”MONDE”!’
```

Conversion de nombres:

```
>>> a = 5
>>> b = 7
>>> a + b
12
>>> c = str(a)      # transformer le int en str
>>> d = str(b)
>>> c + d
’57’
```

Davantage de méthodes sur:
https://docs.python.org/3/library/stdtypes.html#string-methods

3.3 Listes

Une liste est déclarée avec []. Il n’est pas nécessaire de spécifier la taille de la liste lors de la déclaration. Les listes peuvent être hétérogènes:

```
>>> maListe = [’pomme’, ’poire’, ’orange’, 34]
```

Comme avec les chaînes de caractères, on peut sélectionner les éléments à l’aide de leur indice:

```
>>> maListe[0]
’pomme’
>>> len(maListe)          # nombre d’éléments
4
>>> maListe[-1]
34
>>> maListe[1:3]          # extraction de sous-liste
[’poire’, ’orange’]
```

On peut vérifier l’appartenance d’un élément à une liste avec le mot-clé *in*:

```
>>> ’pomme’ in maListe
True
>>> 35 in maListe
False
```

Les listes sont mutables, c'est-à-dire qu'on peut ajouter, enlever et modifier des éléments après la déclaration de la liste:

- Modification d'un élément existant:

```
>>> maListe[2] = 'citron'
>>> maListe
['pomme', 'poire', 'citron', 34]
```

- Ajout en fin de liste:

```
>>> maListe.append('banane')
>>> maListe
['pomme', 'poire', 'citron', 34, 'banane']
```

- Position d'insertion spécifique:

```
>>> maListe.insert(1, 55)
>>> maListe
['pomme', 55, 'poire', 'citron', 34, 'banane']
```

- Concaténation de deux listes:

```
>>> maListe = maListe + liste2
>>> maListe
['pomme', 55, 'poire', 'citron', 34, 'banane', 4, 7, 9]
```

- Supprimer le dernier élément:

```
>>> maListe.pop()            # retourne l'élément supprimé
9
>>> maListe
['pomme', 55, 'poire', 'citron', 34, 'banane', 4, 7]
```

- Supprimer un élément spécifique:

```
>>> del maListe[0]            # si on connaît l'indice
>>> maListe
[55, 'poire', 'citron', 34, 'banane', 4, 7]
>>> maListe.remove('poire')   # si on connaît le contenu
>>> maListe
55, 'citron', 34, 'banane', 4, 7]
```

Python met à disposition des fonctions pour trier une liste. La commande *sorted()* crée une nouvelle liste triée, tandis que la méthode *sort()* trie une liste en place. *reverse()* inverse les éléments d'une liste en place:

```
>>> m = ['pomme', 'poire', 'citron', 'banane']
>>> sorted(m)                    # ordre alphabétique
['banane', 'citron', 'poire', 'pomme']
>>> m.sort()
>>> m
```

```
['banane', 'citron', 'poire', 'pomme']
>>> m.reverse()
>>> m
['pomme', 'poire', 'citron', 'banane']
```

Transformer une chaîne de caractères en liste de mots:

```
>>> c = "ceci est une phrase."
>>> c.split()        # séparateur par défaut: l'espace
['ceci', 'est', 'une', 'phrase.']
```

Opération inverse:

```
>>> m = ['ceci', 'est', 'une', 'phrase.']
>>> '-'.join(m)
'ceci-est-une-phrase.'
```

Transformer une chaîne de caractères en liste de caractères:

```
>>> list("deux mots")
['d', 'e', 'u', 'x', ' ', 'm', 'o', 't', 's']
```

3.4 Tuples

Un tuple est une liste non-modifiable. On ne peut ni insérer des éléments, ni en supprimer, ni en changer le contenu. Un tuple est déclaré avec des parenthèses normales:

```
>>> monTuple = (1, 2, 'three')
>>> monTuple[0]
500
>>> len(monTuple)
2
>>> 1 in monTuple
True
>>> tuple2 = (1, 2, (3, 5))    # tuple imbriqué
```

La modification en place d'un élément du tuple cause une erreur:

```
>>> monTuple[0] = 300
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

En général, on utilise des tuples pour **grouper** différentes valeurs d'une même entité. Ils fonctionnent comme des enregistrements simples.

3.5 Dictionnaires

Un **dictionnaire** (ou **tableau associatif**, ou **table de hachage**) est une généralisation de la structure de liste. Si une **liste** associe des **clés numériques consécutives** à des **valeurs**, un dictionnaire permet l'utilisation d'un choix plus large de clés: tout objet immutable (nombre, chaîne de caractère, tuple) peut être utilisé comme clé.

Un dictionnaire peut être considéré comme un ensemble non-ordonné de paires clé-valeur, avec la contrainte que les clés doivent être uniques. Les dictionnaires sont conçus pour pouvoir accéder aux valeurs à partir des clés de façon efficace.

```
>>> d = {'fruitJaune':'banane', 'fruitVert':'poire',
         'fruitOrange': 'orange'}
```

Lorsqu'on connaît la clé, on peut accéder directement à sa valeur:

```
>>> d['fruitVert']
'poire'
```

On peut ajouter et supprimer des entrées:

```
>>> d['fruitBleu'] = 'myrtille'
>>> del d['fruitJaune']
```

Les clés et les valeurs peuvent être de différents types:

```
>>> d['unNombre'] = 89.4           # valeur réelle
>>> d[43] = 'quarante-trois'       # clé entière
>>> d
{43: 'quarante-trois', 'unNombre': 89.4, 'fruitBleu':
'myrtille', 'fruitVert': 'poire', 'fruitOrange': 'orange'}
```

Les éléments d'un dictionnaire ne sont pas – et ne peuvent pas être – ordonnés.

Transformer un dictionnaire en une liste de tuples, extraire la liste des clés et la liste des valeurs:

```
>>> d.items()
[('fruitJaune', 'banane'), ('fruitVert', 'poire'), ('fruitOrange', '
   orange')]
>>> d.keys()
['fruitJaune', 'fruitVert', 'fruitOrange']
>>> d.values()
['banane', 'poire', 'orange']
```

L'opérateur *in* regarde les clés, pas les valeurs:

```
>>> "fruitOrange" in d
True
>>> "poire" in d      # poire n'est pas une clé
False
```

Pourquoi "dictionnaire"?

```
>>> {'pomme':'apple', 'poire':'pear', 'banane':'banana'}
```

4 Entrées et sorties basiques

Lorsqu'on utilise l'interpréteur interactif, les expressions sont immédiatement évaluées et leur résultat affiché:

```
>>> ”test”.upper()
TEST
```

Ceci n'est pas le cas dans un fichier script. Pour afficher un résultat sur la ligne de commande, il faut utiliser la commande *print()*:

```
t = ”test”.upper()
print(t)              # affichage: TEST
```

On peut afficher plusieurs éléments (même de types différents), séparés par des virgules:

```
a = 3; b = 5.6; c = ”test”
print(c, a, b)        # affichage: test 3 5.6
```

Par défaut, *print()* ajoute un retour à la ligne. La syntaxe suivante évite cela:

```
print(c, end=””)
print(a)              # affichage: test 3 (sur la même ligne)
```

La commande *input()* permet de solliciter une entrée à la ligne de commande. Cette entrée est toujours de type chaîne de caractères et doit être convertie si nécessaire.

```
x = input(”Entrez un nombre entre 1 et 10:”)
x2 = int(x)
```

4.1 Formatage de chaînes de caractères

La fonction *print()* permet l'affichage de plusieurs éléments de types différents, séparés par des espaces:

```
>>> m = 10
>>> print(”Le nombre magique est:”, m)
Le nombre magique est 10
```

Le même effet peut être obtenu par conversion et concaténation:

```
>>> m = 10
>>> s = ”Le nombre magique est:” + str(10)
>>> print(s)
Le nombre magique est 10
```

Cependant, ces deux façons de faire peuvent vite devenir encombrantes. Pour cela, on peut utiliser la méthode *format()*:

- Dans une chaîne de caractères, on définit des champs. Ces champs consistent en un entier entouré d'accolades.

- On appelle la méthode *format()* sur cette chaîne de caractères, avec autant de paramètres qu'on a défini de champs.
- La valeur du premier paramètre est substitué au champ 0, la valeur du deuxième paramètre au champ 1, et ainsi de suite. Ceci permet notamment d'inverser les paramètres dans la chaîne.

```
>>> s = "First argument: {0}, second one: {1}".format(47,11)
>>> print(s)
First argument: 47, second one: 11
>>> s = "Second argument: {1}, first one: {0}".format(47,11)
>>> print(s)
Second argument: 11, first one: 47
```

Cette méthode est particulièrement utile pour afficher des nombres réels de façon lisible. L'exemple suivant permet d'arrondir le nombre à deux décimales après la virgule:

```
>>> m = 5 / 3
>>> print("Le nombre magique est:", m)
Le nombre magique est: 1.6666666666666667
>>> s = "Le nombre magique est {0:.2f}".format(m)
>>> print(s)
Le nombre magique est 1.67
```

Le formatage de chaînes de caractères est bien plus puissant que ces quelques exemples le suggèrent. La documentation complète est disponible sur:
https://docs.python.org/3/library/string.html#format-string-syntax

5 Structures de contrôle

5.1 Distinction de cas avec *if*

```
if n > 0:
    print("positif")
    print("trois mots inutiles")
elif n == 0:
    print("zéro")
else:
    print("négatif")
print("toujours affiché")
```

- Les mots-clés utilisés sont *if, else, elif*.
- Les comparateurs sont ==, !=, <, >, <=, >=, *in*.
- Contrairement à d'autres langages, l'expression de comparaison ne doit pas être entourée de parenthèses. Elle est nécessairement terminée par le symbole :
- Les blocs d'instructions ne sont marqués ni avec des { } ni avec des BEGIN/END. **Seule l'indentation compte.**

5.2 Indentation

Python utilise uniquement l'indentation pour délimiter les blocs d'instructions:

- Une indentation plus grande qu'à la ligne précédente indique le début d'un bloc d'instructions.
- Une indentation plus petite qu'à la ligne précédente indique la fin d'un bloc d'instructions.

On peut indenter le code avec des tabulateurs…

```
if␣n␣>␣0:
␉print(”positif”)
```

…ou avec des espaces:

```
if␣n␣>␣0:
␣␣␣␣print(”positif”)
```

Il est recommandé d'utiliser toujours 4 espaces par niveau d'indentation (comme dans le deuxième exemple ci-dessus). C'est par ailleurs le comportement par défaut d'IDLE.

5.2.1 Blocs dans l'interpréteur interactif

Dès que vous écrivez du code contenant des blocs d'instructions, il est plus facile d'écrire le code dans un fichier au lieu d'utiliser l'interpréteur interactif. Cependant, il est possible d'écrire des blocs de code complexes avec ce dernier:

```
[1]  >>> n = 5
[2]  >>> while n < 10:
[3]  ...     print(n)
[4]  ...     n += 1
[5]  ...
[6]  5
[7]  6
[8]  7
[9]  8
[10] 9
[11] >>>
```

[2] L'interpréteur se rend compte que l'instruction while n'est pas complète. Il le signale avec les ... à la ligne suivante.

[3] L'intérieur de la boucle doit être indenté. Il faut donc taper n espaces.

[4] L'interpréteur conserve le même niveau d'indentation à la ligne suivante.

[5] Pour signaler la fin de la boucle, on efface les espaces, et on passe à la ligne suivante.

[6] Le bloc de code est complet, et il est exécuté.

5.3 Boucle *while*

```
x = 10
while (x > 0):
    print(x)
    x = x - 1
```

- L'instruction *break* sort immédiatement de la boucle.
- L'instruction *continue* passe immédiatement à la prochaine itération de la boucle.

5.4 Boucle *for*

L'instruction *for* itère sur les éléments d'une séquence (listes, chaînes de caractères, clés d'un dictionnaire, ...) en utilisant le mot-clé *in*. Ainsi, elle diffère considérablement de l'instruction *for* dans des langages de programmation plus traditionnels:

```
liste = [4, 7, 12, 52, 3]
for element in liste:
    print(element)    # affiche un nombre par ligne
```

Attention: La variable *element* désigne la valeur de l'élément, pas son indice (sa position dans la liste). Pour retrouver à la fois les indices et les valeurs, on peut utiliser la fonction *enumerate()*:

```
for i, v in enumerate(liste):
    print(i, v)       # affichage: 0 4 // 1 7 // 2 12 // ...
```

Parcours d'un dictionnaire:

```
d = {'jaune':7, 'vert':2, 'rouge': 4}
for k in d:
    print(k, '===>', d[k])   # affichage: jaune ===> 7 // ...
```

Parcours d'un dictionnaire par ordre alphabétique des clés:

```
for k in sorted(d):
    print(k, '===>', d[k])
```

- Techniquement, *sorted(d)* renvoie une liste triée des clés, sur laquelle la boucle *for* itère. Le dictionnaire en lui-même n'est pas trié.

Pour itérer sur des séquences de nombres entiers (comme le ferait une boucle *for* traditionnelle), il faut d'abord créer une telle séquence à l'aide de *range*:

```
for i in range(5):      # range(5) = [0, 1, 2, 3, 4]
    print(i)            # affichage: 0 // 1 // 2 // 3 // 4
```

5.5 Compréhensions de listes

Les compréhensions de listes permettent de définir des listes de manière concise à l'aide de règles. Typiquement, on utilise des compréhensions de listes lorsqu'on veut créer des nouvelles listes où chaque élément est le résultat d'une opération appliquée à chaque membre d'une liste existante, ou lorsqu'on veut filtrer une liste existante selon un critère prédéfini.

Cet exemple crée une liste des carrés de 0 à 10:

```
carres = []
for x in range(10):
    carres.append(x**2)       # carres = [0,1,4,9,16,25,36,49,64,81]
```

On peut obtenir le même effet avec une compréhension de liste, en déplaçant la boucle *for* à l'intérieur des crochets de la liste:

```
carres = [x**2 for x in range(10)]
```

Cet exemple crée une liste d'initiales à partir d'une liste de chaînes de caractères:

```
noms = ['Jean', 'Marie', 'Paul', 'Anne']
initiales = []
for x in noms:
    initiales.append(x[0])    # initiales = ['J', 'M', 'P', 'A']
```

Avec une compréhension de liste:

```
noms = ['Jean', 'Marie', 'Paul', 'Anne']
initiales = [x[0] for x in noms]
```

Cet exemple garde seulement les noms qui commencent par J:

```
noms = ['Jean', 'Marie', 'Paul', 'Jacques', 'Anne']
nomsJ = []
for x in noms:
    if x[0] == 'J':
        nomsJ.append(x)       # nomsJ = ['Jean', 'Jacques']
```

Avec une compréhension de liste, en déplaçant également la condition *if* à l'intérieur des crochets de la liste:

```
noms = ['Jean', 'Marie', 'Paul', 'Jacques', 'Anne']
nomsJ = [x for x in noms if x[0] == 'J']
```

6 Fonctions

En Python, on se sert de **fonctions** (ou **procédures** ou **sous-routines**) pour les mêmes raisons que dans les autres langages de programmation:

- Modularité du code

- Fonctions souvent utilisées
- Éviter la répétition du même code
- Récursivité

En Python, tout est appelé **fonction**: avec ou sans paramètres, avec ou sans valeur de retour.

On déclare une fonction avec le mot-clé *def*, le nom de la fonction, et la liste des paramètres entre parenthèses. La première ligne du corps de la fonction peut être une chaîne de caractères qui décrit la fonction; cette ligne est utilisée par des générateurs de documentation:

```
def moyenne(a, b, c):
    """Retourne la moyenne des trois nombres"""
    resultat = (a + b + c) / 3
    return resultat            # valeur de retour
```

Appel de la fonction:

```
print(moyenne(3, 5, 9))        # affiche 5.666666666666667
```

Pour des fonctions sans paramètres, on utilise une paire de parenthèses vide:

```
def procedureSansParametres():
    ...
```

Les paramètres ne sont pas typés. On ne peut pas enforcer un type aux paramètres:

```
print(moyenne(3.2, 6.4, 4.6))       # OK: réels
print(moyenne("bla", "test", "a"))  # erreur: chaînes
```

Les nombres, les chaînes de caractères et les tuples sont passés par **valeur**. Les listes et les dictionnaires sont passés par **référence**.

On peut définir des **valeurs par défaut** pour des paramètres:

```
def moyenne(a, b, c=1):  # valeur par défaut pour c
    ...
print(moyenne(3,5))      # affiche la moyenne de 3,5,1
print(moyenne(3,5,9))    # affiche la moyenne de 3,5,9
```

Techniquement, toutes les fonctions ont une valeur de retour. Si aucune instruction *return* est spécifiée, la valeur de retour est *None* (équivalent à *null* dans d'autres langages de programmation). A l'appel d'une fonction, on peut choisir de récupérer la valeur de retour ou non:

```
def procAvecRetour():
    ...
    return 89

def procSansRetour():
    ...

a = procAvecRetour()   # a = 89
b = procSansRetour()   # b = None
```

```
procAvecRetour()       # on perd la valeur de retour 89
procSansRetour()       # on perd la valeur 'None'
```

7 Lecture et écriture de fichiers

Ouvrir un fichier:

```
f = open("nom_du_fichier.txt", "r")
```

- *f* est un descripteur de fichier, à partir duquel on peut lire ou écrire des données
- Le deuxième paramètre de *open()* indique si on veut lire (*r*) ou écrire (*w*).

On peut spécifier l'encodage à utiliser:

```
f = open("nom_du_fichier.txt", "r", encoding="utf8")
```

Lire des données:

- *f.read()* renvoie une chaîne de caractères avec le contenu entier.
- *f.readlines()* renvoie une liste dont chaque élément représente une ligne du texte.
- *f.readline()* renvoie une chaîne de caractères avec une ligne du texte.
- *f* est un itérable qui peut être utilisé dans une boucle *for...in*:

  ```
  for ligne in f:
      print(ligne)
  ```

- Attention: Les lignes lues contiennent le(s) symbole(s) de fin de ligne. Utiliser *ligne.strip()* pour les enlever.

Écrire des données:

```
f.write(s)          # s est une chaîne de caractères
```

Fermer le fichier:

```
f.close()
```

8 Modules

Une bonne pratique de la programmation consiste à diviser des grands programmes en parties plus petites. En Python, on peut diviser des programmes en **modules**. Python propose également une large librairie de modules prédéfinis.

Pour pouvoir utiliser un module, il faut l'importer:

```
>>> import math
```

Pour utiliser une constante ou une fonction définie dans un module, il faut le préfixer avec le nom du module:

```
>>> math.pi
>>> math.log(2.35)
```

Si vous écrivez vos propres modules, il faut respecter les consignes suivantes:

- N'importe quel script Python peut être importé par un autre, à condition qu'il puisse être localisé par l'interpréteur.
- Le plus simple est de placer le script importé dans le même répertoire que le script importateur.
- Le script *exemple.py* est importé avec la commande *import exemple*.
- Si vous modifiez le contenu d'un module, vous devez le recharger avec *reload(modulename)*.

8.1 Modules importants

math Le module *math* met à disposition des constantes mathématiques et des fonctions de calcul: https://docs.python.org/3.0/library/math.html

re Le module *re* permet de définir et d'utiliser des expressions régulières. Plus de détails dans un document séparé.

sys Le module *sys* permet, entre autres, de récupérer les arguments fournis lors de l'appel du script sur la ligne de commande.

- Ceci marche uniquement lorsqu'on appelle le script directement depuis la ligne de commande, mais pas lorsqu'on exécute le script depuis IDLE.

Le tableau *argv* du module *sys* contient les arguments:

```
import sys
print("Argument 0:", sys.argv[0])   # nom du script
print("Argument 1:", sys.argv[1])   # premier paramètre
print("Argument 2:", sys.argv[2])   # deuxième paramètre
```

Voici un exemple d'invocation de ce script par la ligne de commande:

```
$ python monscript.py orange pomme
Argument 0: ./monscript.py
Argument 1: orange
Argument 2: pomme
```

9 Classes et orientation objet

Python permet également de faire de la programmation orientée-objet. Une **classe** est définie à l'aide du mot-clé *class*, suivi du nom de la classe. Le corps de la classe doit être indenté, et la première instruction peut être une chaîne de caractères de documentation.

Une **méthode** est une fonction qui "appartient" à une classe. Les méthodes ont toujours comme premier paramètre la variable *self*. Celle-ci contient la référence vers l'objet depuis lequel elle est appelée. (*self* est équivalent à la variable *this* en Java, avec la différence que *self* doit être spécifié explicitement dans l'entête des méthodes).

```
class MyClass:
    """A very simple class"""

    def greet(self, name):
        return "Hello, " + name
```

Une classe est instanciée en un objet, et les méthodes d'un objet sont appelés comme dans d'autres langages de programmation:

```
c = MyClass()
c.greet("Paolo")        # c correspond au paramètre self
```

Les classes peuvent avoir une méthode d'initialisation *__init__()* [deux soulignés à gauche et deux soulignés à droite du nom], similaire aux constructeurs dans d'autres langages. Cette méthode est appelée quand la classe est instanciée; elle peut avoir des paramètres:

```
class Greeter:
    """A simple class"""
    def __init__ (self, greeting):
        self.greeting = greeting
    def greet(self, name):
        return self.greeting + ", " + name

c2 = Greeter("Hi")
c2.greet("Tim")
```

Une classe peut être dérivée d'une autre classe comme suit:

```
class GreeterExt(Greeter):
    """A derived class"""
    def bye(self):
        return "Bye Bye"
```

```
c3 = GreeterEx(”hello”)
c3.greet(”mr smith”)
c3.bye()
```

Cette classe contient toutes les méthodes définies dans la superclasse *Greeter* ainsi que la nouvelle méthode *bye()*.

Par défaut, toutes les méthodes et tous les attributs sont publics. Il est possible de les rendre “pseudo-privés” en ajoutant deux soulignés au début de leur nom (p.ex. *def __bye(self)*).

10 Aide interactive

dir(module) affiche toutes les classes, variables et procédures définies par un module:

```
>>> import math
>>> dir(math)
[’__doc__’, ..., ’acos’, ’acosh’, ’asin’, ’asinh’, ’atan’, ’atan2’, ’
   atanh’, ’ceil’, ’copysign’, ’cos’, ’cosh’, ’degrees’, ’e’, ’erf’,
   ’erfc’, ’exp’, ’expm1’, ’fabs’, ’factorial’, ...]
```

help(module) affiche un texte d’aide pour le module:

```
>>> import math
>>> help(math)
Help on built-in module math:
NAME
    math
DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.
FUNCTIONS
    acos(...)
        acos(x)
        Return the arc cosine (measured in radians) of x.
    acosh(...)
        acosh(x)
        Return the hyperbolic arc cosine (measured in radians) of x.
...
```

- Taper *q* pour sortir de l’aide.

On peut aussi afficher directement la documentation d’une classe ou d’une fonction:

```
>>> help(math.log)
Help on built-in function log in module math:
log(...)
    log(x[, base])
    Return the logarithm of x to the given base.
```

```
    If the base not specified, returns the natural logarithm (base e)
        of x.
```

Sources

Une partie de ce document est une traduction des tutoriels disponibles sur
http://www.inf.ed.ac.uk/teaching/courses/inf2a/tutorials/.