TP N°1.0 - Python
Syntaxe de base

Thibault Napoléon
thibault.napoleon@isen-ouest.yncrea.fr

Notions de syntaxe abordées :

- ✓ Variables
- ✓ Instructions conditionnelles
- ✓ Boucles
- ✓ Fonctions
- ✓ Entrées/sorties
- ✓ Chaines de caractères
- ✓ Listes, dictionnaires et sets
- ✓ Entrées/sorties sur fichier
- ✓ Importations de modules

1 Installation de l'environnement de développement sous Linux

Installation de *python3*

Normalement toutes distributions récentes de Linux contiennent *python3*. Cependant si ce n'est pas le cas vous pouvez l'installer grâce à :

```
sudo apt-get install python3
```

Installation de l'éditeur *Atom*

Afin de programmer efficacement en *Python*, différents éditeurs s'offrent à vous. Si vous n'avez pas d'éditeur favoris et que vous souhaitez une installation fonctionnelle de *Python*, je ne saurais que vous conseiller l'éditeur *Atom*. Si vous ne l'avez pas vous pouvez l'installer grâce à :

```
sudo add-apt-repository ppa:webupd8team/atom
sudo apt-get update
sudo apt-get install atom
```

Configuration d'*Atom* pour *Python*

Afin de pouvoir utiliser efficacement *Python* dans *Atom* (complétion, détection des erreurs), il est possible d'installer un ensemble de plugins :

```
sudo apt-get install python3-pip
pip3 install flake8
pip3 install flake8-docstrings
apm install linter
apm install linter-flake8
```

2 Premier programme en *Python*

Création du fichier et ouverture

Python est un langage de programmation interprété. Cela implique que les fichiers sources, dont l'extension est *.py*, peuvent être exécutés directement par l'interpréteur *Python*. Pour créer et éditer un fichier *Python* dans le terminal, utilisez les commandes suivantes :

```
mkdir TP1
cd TP1
atom helloworld.py
```

Premier programme : *Hello World !*

Afin d'écrire votre premier programme, saisissez les instructions suivantes :

```
1 """Hello Wolrd."""
2
3
4 # Mon premier programme.
5 print('Hello World!')
```

- La première ligne permet de définir la *docstring* associée au fichier. Il s'agit d'une documentation à introduire dans chaque fichier pour le documenter. Elle est toujours suivie de deux lignes vides. Attention une *docstring* n'est pas un commentaire.
- La quatrième ligne permet de définir un commentaire.
- La cinquième ligne permet d'afficher sur la console la chaine *"Hello World !"*

Notez qu'aucune fonction particulière n'est nécessaire pour interpréter ce programme.

Première interprétation

Le langage *Python* est un langage interprété. Cela implique qu'aucune compilation n'est nécessaire mais qu'un outil, appelé interpréteur, doit interpréter votre programme. Il s'agit dans notre cas de *python3*. Pour lancer l'interprétation du code source précédent, utilisez la commande suivante :

```
python3 helloworld.py
```

3 Syntaxe de base

Les variables

En *Python* les variables ne sont pas typées. Cela implique que lors de la déclaration d'une variable on ne spécifie pas son type et que celui-ci est déduit à partir de l'affectation. Cela implique aussi que le type d'une variable peut changer. Exemple :

```
1 variable = 8
2 print(variable)
3 variable = 4.5
4 print(variable)
5 variable = 'h'
6 print(variable)
7 variable = 'bonjour'
8 print(variable)
```

Il est possible de convertir une variable d'un type vers un autre avec :

```
entier = int('23') # Conversion vers un entier.
chaine = str(45) # Conversion vers une chaine.
```

Il est possible de vérifier si une chaine contient uniquement des chiffres ou des caractères avec :

```
chaine = '512'
print(chaine.isdigit()) # Affiche True.
print(chaine.isalpha()) # Affiche False.
```

Les instructions conditionnelles

Comme dans tout langage de programmation il est possible d'exécuter une partie du code sous condition. Pour cela on utilise généralement l'instruction conditionnelle *if ... else*. En *Python* elle s'utilise de la manière suivante :

```
valeur = 23
if valeur >= 100:
  print('La valeur vaut plus de 99')
else:
  print('La valeur vaut moins de 100')
```

On peut utiliser les mots clé *not*, *or* et *and* pour paramétrer l'instruction de contrôle :

```
valeur = '23'
if not valeur.isdigit() or int(valeur) <= 100:
  print('Ce n\'est pas un entier supérieur à 100')
```

Attention, en *Python* les blocs sont définis grâce à l'indentation et non pas avec des accolades.

Les boucles

En *Python* seule l'instruction *while* existe (le *do ... while* n'existe pas). Elle s'utilise comme suit :

```
compteur = 5
while compteur > 0:
  print(compteur)
  compteur -= 1 # Affiche les entiers de 5 à 1.
```

En ce qui concerne l'instruction *for*, elle s'utilise comme suit :

```
for index in range(0, 5):
  print(index) # Affiche les entiers de 0 à 4.
```

Les fonctions

Afin de rendre vos programmes réutilisables mais aussi plus lisibles, il est nécessaire de définir des fonctions. Celles-ci peuvent prendre un ensemble de paramètres, séparés par des virgules en entrée. Elles peuvent aussi retourner un résultat en utilisant le mot-clé *return* (notez qu'il est possible de renvoyer plusieurs valeurs à travers le mot-clé *return*). Afin de documenter les

fonctions il est nécessaire, après la première ligne de définition de la fonction, d'inclure une *docstring*. En *Python* les fonctions se définissent comme suit :

```
def plusMoins(v1, v2):
  """Fonction pour additionner et soustraire."""
  plus = v1 + v2
  moins = v1 - v2
  return plus, moins


# Programme principal.
res1, res2 = plusMoins(8, 5)
```

Notez de nouveau l'indentation des blocs entre la fonction et le programme principal.

Les entrées/sorties

L'un des intérêts d'un langage de programmation est sa possibilité d'interagir avec l'utilisateur. Pour cela on utilise l'entrée standard, i.e. le clavier, ainsi que la sortie standard, i.e. l'écran. En *Python* les saisies utilisateur sont considérer comme des chaines de caractère. Il faut donc les convertir explicitement en entier avec *int()* si besoin. Exemple :

```
v1 = input('Entrez un premier entier > ')
v2 = input('Entrez un second entier > ')
print('La somme {0} + {1} = {2}'.format(v1, v2, int(v1) + int(v2)))
```

Ici dans la fonction *print()*, les éléments *{0}*, *{1}* et *{2}* sont remplacés respectivement par les trois expressions de la fonction *format()*.

Les chaines de caractères

Afin de définir une chaine de caractère en *Python*, on utilise indifféremment les simples quotes ou les doubles quotes. Afin de manipuler rapidement ces chaines, différentes fonctions peuvent y être appliquées : majuscules, minuscules, concaténation... Attention, en *Python* les chaines sont immutables. Cela implique qu'une chaine ne peut pas être modifiée, elle sera donc toujours recopiée. Les différentes fonctions de chaines s'utilisent comme suit :

```
chaine = 'Bonjour?'
print(chaine.lower()) # Minuscules.
print(chaine.upper()) # Majuscules.
print(chaine.replace('?', '!')) # Change le ? par un !.
print(len(chaine)) # Donne la taille de la chaine (8).

# Parcours de la chaine.
for letter in chaine:
  print('-> ' + letter)
```

En *Python* une chaine de caractères est un tableau qui contient dans chaque case un caractère. On peut accéder aux différentes *lettres* grâce aux crochets. Il est aussi possible d'extraire une sous chaine grâce à la notion de slice :

```
chaine = 'Bonjour?'
print(chaine[2]) # Affiche : n.
print(chaine[2:4]) # Affiche : nj.
```

```
print(chaine[:3]) # Affiche : Bon.
print(chaine[3:]) # Affiche : jour?.
print(chaine[::2]) # Affiche : Bnor (un sur deux).
```

Les listes, dictionnaires et sets

Une liste *Python* est une collection d'éléments qui peuvent avoir un type différents. On accède aux différents éléments de la même manière qu'aux lettres d'une chaine de caractères et il est possible de créer des *slices*. Différentes fonctions permettent de manipuler les listes pour en autre ajouter, supprimer, insérer un élément. Les listes sont très utilisées car elles sont mutables (contrairement aux chaines de caractères). Elles s'utilisent comme suit :

```
l1 = list()
l1.append('3')
l2 = ['1', 2, 'test']
print(l2[1]) # Affiche : 2.
l2.remove('test')
print(len(l1)) # Affiche : 1.

# Affiche les elements de la liste l2.
for element in l2:
  print(element)
```

Il existe aussi des dictionnaires qui permettent d'associer une valeur à une clé :

```
d = {}
d['1'] = 'bonjour'
d['test'] = 'salut'
print(d) # Affiche : {1: 'bonjour', 'test': 'salut'}.
print(d['1']) # Affiche : 'bonjour'.
del d['test']
print(d.keys())
```

Finalement le set est un conteneur qui contient des éléments de façon unique :

```
s1 = {'apple', 'orange', 'apple', 'pear'}
s2 = {'orange', 'pear', 'cherry'}
print(s1) # Affiche : {'apple', 'orange', 'pear'}.
s1 - s2 # Elements dans s1 mais pas dans s2.
s1 | s2 # Elements dans s1 ou s2.
s1 & s2 # Elements dans s1 et s2.
s1.add('banana')
s2.remove('pear')
```

Les entrées/sorties sur fichier

La lecture sur un fichier se fait simplement à l'aide des opérations suivantes :

```
fileRead = open('data.txt', 'r')
lines = fileRead.readlines()
fileRead.close()
```

```
fileRead = open('data2.txt')
line = fileRead.readline()
fileRead.close()
```

- *readline()* : permet de lire une ligne.
- *readlines()* : permet de lire toutes les lignes et de les placer dans une liste.
- *read()* : permet de lire toutes les lignes et de les placer dans une chaine de caractère.

Dans le cas où l'on veut lire un fichier sans avoir besoin de le fermer, on utilise l'instruction suivante :

```
with open('data.txt') as fileRead:
  lines = fileRead.readlines()
```

Pour faire une écriture on utilise :

```
fileWrite = open(data.txt', 'w')
fileWrite.write('v0')
l = ['v1', 'v2', 'v3']
fileWrite.writelines(l)
fileWrite.close()
```

- *write()* : permet d'écrire une ligne de type chaine de caractère.
- *writelines()* : permet d'écrire un ensemble de lignes de type chaines de caractères à partir d'une liste.

Pour faire des lectures et écritures sur fichier il est nécessaire de spécifier le mode d'ouverture du fichier. Le tableau suivant reprend les différents modes d'ouverture :

	r	**r+**	**w**	**w+**	**a**	**a+**
Lecture du fichier	✓	✓		✓		✓
Écriture dans le fichier		✓	✓	✓	✓	✓
Création du fichier			✓	✓	✓	✓
Remise à zéro du fichier			✓	✓		
Position initiale au début	✓	✓	✓	✓		
Position initiale à la fin					✓	✓

Les importations de modules

Pour finir il est parfois utile d'importer des modules déjà existants en *Python*, par exemple pour tirer un nombre aléatoire. Dans ce cas on utilise :

```
import random

print(random.randint(0, 10)) # Affiche un nombre aleatoire entre 0 et 10.
```

Ou :

```
from random import randint

print(randint(0, 10)) # Affiche un nombre aleatoire entre 0 et 10.
```

Pour importer toutes les fonctions d'un fichier :

```
from monfichier import * # Importe les fonctions de monfichier.py.
```