Cours sur Python

Session 1

Introduction

Python est né au début des années 1990, son père est Guido van Rossum. Les deux versions actuelles sont la 2.7.2 et la 3.2.2.

Python est présent partout, vous l'utilisez tous les jours avec Ubuntu, Red Hat en use et abuse, de même que Google (van Rossum est employé par Google). Vous avez aussi très certainement vu Python en action avec Launchpad.

Ce cours est une introduction au langage, en abordant certaines notions de base de la programmation. Le but est d'être clair pour les gens qui n'ont jamais programmé, donc certains « pythonismes » seront volontairement mis de côté. On approfondira dans d'autres sessions.

L'interpréteur de commandes

Python est un langage « interprété », c'est-à-dire qu'un script ne nécessite pas de compilation pour être exécuté. Il existe un interpréteur qui permet d'interagir avec l'utilisateur. Pour le lancer, démarrez un terminal et lancez :

```
python
```

Commençons par une utilisation très simple, en faisant faire du calcul à python :

```
3*6
4-9
```

Notion de variable

Les variables correspondent à des zones de stockage de valeurs. Elles sont désignées par des mots contenant les lettres minuscules, majuscules, les chiffres et le tiret bas « _ »[1)]. Une variable ne peut avoir un nom commençant par un chiffre.

A une variable on assigne une valeur grâce à l'instruction « = », par exemple (dans l'interpréteur) :

```
a = 4
b = 5
a + b
```

Les valeurs contenues par les variables a et b ont été utilisées pour réaliser l'opération.

Une variable doit avoir été initialisée pour être utilisée :

```
c
c = 1
c
```

La première et la troisième instruction sont identiques, mais le résultat diffère suivant l'initialisation.

Une variable peut « dépendre » d'elle même :

```
a = 4
a = a + 1
a
```

Attention à la casse ! « A » est différent de « a ».

Notion de type

On n'a utilisé que des entiers pour l'instant. Python l'a remarqué :

```
a = 5
a / 2
b = -5
b / 2
```

On obtient 2, et pas 2.5 pour a, et -3 et pas -2.5 pour b. Pourquoi ? python arrondi tout simplement à l'entier inférieur car on lui a suggéré que a est un entier.

En Python 3, le résultat est bien 2.5 pour a et -2.5 pour b

Comment faire alors ? Préciser à python que a est un « flottant » :

```
a = 5.0
a / 2
```

Un autre type très courant est la « chaine de caractère » (*string* en anglais). Elle est définie en plaçant du texte entre « “ » ou entre « ' » :

```
une_chaine = "Je suis une chaine."
une_chaine
```

On peut « ajouter » des chaines, on appelle ça la concaténation :

```
a = "Bonjour "
b = "Monde !"
a + b
```

Pour mettre le doigt sur une possible confusion, essayons ceci :

```
a = 1
a
b = "1"
b
```

Notez la différence entre les 2 résultats. La première fois python affiche un entier, la seconde une chaine de caractères (entre « ' »).

```
 a+b
```

Une erreur résulte de ce calcul, puisque *a* et *b* sont de type différents.

Premier script

Un des intérêts d'un programme est de pouvoir interagir avec l'utilisateur. On va voir pour ça deux instructions et comment les utiliser :

- `raw_input()` permet de demander à l'utilisateur de saisir un texte (texte au sens large, on commencera avec des nombres) ;
- `print("texte")` permet d'afficher du texte.

Ce sont deux fonctions, qui utilisent un ou plusieurs arguments (des données qu'elles traiteront). Nous aborderons les fonctions un peu plus loin.

`raw_input()` s'utilise de cette manière :

```
resultat = raw_input("Veuillez saisir quelque chose : ")
```

La chaine de caractère sera affichée à l'écran, et le curseur attendra que l'utilisateur saisisse du texte puis « Entrée ». Ce qui a été saisi sera alors assigné à la variable `résultat` (en tant que chaine de caractère).

Pour ensuite afficher ce texte on peut utiliser :

```
print "Du texte qui ne change pas et autre chose :", resultat
```

Les éléments à afficher sont séparés par une virgule, qui apparaitre comme un simple espace lors de l'exécution.

"`Du texte qui ne change pas et autre chose :`" et `resultat` sont deux arguments de la fonction `print`.

Regardons maintenant cet exemple :

```
#!/usr/bin/env python
# -*- coding: UTF8 -*-

# On demande le nom
nom = raw_input("Quel est votre nom ? ")
```

```
# On demande le prénom
prenom = raw_input("Quel est votre prénom ? ")

# On affiche le tout
print ("bonjour", prenom, nom)
```

Quelques petites remarques :

- les lignes qui commencent par « # » ne sont pas lues par Python, ce sont des lignes de commentaires. Elle sont utiles pour détailler certains éléments de votre script ;
- la première ligne est un « shebang », qui permet à votre shell que savoir avec quel programme lancer votre script lorsqu'il est exécutable (**./exemple1.py**) ;
- la seconde définit l'encodage, c'est essentiel pour nous francophones qui utilisont des accents (Python n'aime pas vraiment les accents sans encodage précisé).

Copiez ce script dans un fichier « exemple.py » et exécutez :

```
python exemple.py
```

En guise d'exercice, écrivez un script qui demande l'âge de 2 personnes, et qui affiche la différence d'âge. Rappelez vous que *raw_input()* récupère une chaine de caractère, qu'il faudrait transformer en entier pour faire le calcul. La fonction *int* qui prend en argument une chaine de caractère, et retourne sa conversion en entier.

Conditions

Pour que le script réagisse suivant ce que l'utilisateur à saisi, on utilise des instructions de contrôle, `if` et `else`. Elle s'utilisent comme ceci :

```
  if (une_condition_est_vraie):
        # on execute
        # une série
        # d'instructions
  else:
        # sinon
        # on fait autre chose
```

La syntaxe est (je pense) assez claire, mais elle amène quelques nouvelles notions.

Les booléens

Une variable booléennes est une variable qui n'accepte que 2 valeurs : vrai ou faux (`True` ou `False` en Python). On peut étendre ceci aux nombres et dire : si c'est 0 c'est faux, sinon (dans tous les autres cas), c'est vrai. “une_condition_est_vraie” va donc être une expression qui sera soit vraie (ou non nulle), soit fausse (nulle). Si elle est vraie, on exécute la première partie de la condition (juste après `if`, sinon la deuxième (ce qui suit `else`).

L'indentation

Pour connaître toutes les instructions à exécuter si la condition est vraie, on définit un “bloc”. Ce bloc est défini par une indentation (<tab> en général) :

```
      # on execute
      # une série
      # d'instructions
```

Ce bloc sera exécuté si la condition est vérifiée.

Prenons un exemple de comparaison d'entiers :

```
a = 1
b = 2
if (a > b):
      print ("a est supérieur à b")
else:
      print ("b est supérieur ou égal à a")
```

Quel sera le résultat du script ?

L'expression qui définit la condition est souvent une comparaison, qui utilise les symboles suivants :

- > et < pour 'strictement supérieur/inférieur à'
- >= et ⇐ pour 'supérieur/inférieur ou égal à'
- == et != pour 'égal à ou différent de'

Les conditions peuvent être multiples et inclure des `or` et `and` :

- `a or b` sera vraie si soit a, soit b est vraie
- `a and b` sera vraie si a et b sont vraie

Vous pouvez maintenant reprendre l'exercice de tout à l'heure et systématiquement afficher un résultat positif.

Listes

Définir une liste

Utiliser une variable par élément à saisir, ça devient très vite ingérable (imaginez ce que ça donnerait pour gérer une liste de 10000 clients). Python possède un autre type de données, les listes. Il s'agit en fait d'un “tas” de variables groupées en une seule. Par exemple (pour notre liste de choses à faire) on pourrait avoir :

```
a = "coup de fil à maman"
b = "acheter du café"
c = "upgrader vers gutsy"
```

A chaque nouvel élément il faudrait ajouter une variable, c'est ingérable. On peut alors utiliser :

```
todo = ["coup de fil à maman", "acheter du café", "upgrader vers gutsy"]
```

Notez que si l'on a déjà initialisé a, b et c on peut aussi utiliser :

```
todo = [a, b, c]
```

On n'a plus qu'une seule variable, qui contient un ensemble cohérent d'éléments, et qui peut bien entendu être modifiée.

Une liste est ce qu'on appelle un 'objet' (comme n'importe quel élément en python, mais peu importe pour le moment). Et à un objet correspondent des 'méthodes'. Ces méthodes sont des actions que l'on peut appliquer à l'objet. Par exemple, on peut ajouter un élément à notre liste :

```
todo.append("préparer le cours python sur l'orienté objet")
```

Dans l'interpréteur, affichez maintenant todo:

```
todo
```

Le nouvel élément a bien été ajouté à notre liste.

`todo` est l'objet auquel on s'intéresse, append() la méthode. ATTENTION, todo.method != todo.method(). todo.method correspond à la suite d'instruction qui définissent les actions de la méthode, alors que todo.method() correspond au fait d'appliquer ces instructions. Vous noterez que 'todo' a été modifié sans que l'on ait besoin de lui réassigner une nouvelle valeur ; on n'a pas eu besoin de faire :

```
todo = todo.methode()
```

Accéder aux éléments

On accède aux éléments d'une liste par leur indice (de 0 à (nombre_d_elements - 1)). Le premier élément est donc accessible par :

```
todo[0]
```

Le nombre d'éléments contenus dans une liste est donné par len(liste) :

```
nb = len(todo)
```

Donc le dernier élément de la liste est :

```
todo[nb - 1]
```

Si l'on veut agir sur tous les éléments d'une liste, on utilise l'instruction `for` :

```
for item in todo:
```

```
      # on commence un bloc d'instructions
      # avec une nouvelle indentation
      # 'item' est une variable à laquelle on assigne
      # la valeur de l'élément courant du tableau
```

Pour afficher un élément du tableau par ligne on peut donc utiliser :

```
for untodo in todo:
      print " - %s" % untodo
```

Notez au passage l'utilisation particulière de `print`. `%s` sera remplacé par la valeur de `untodo` lors de l'affichage. Notez aussi que la variable `untodo` aurait pu prendre n'importe quel nom (elle s'appelait `item` un peu plus haut).

Fonctions - introduction

On a parlé tout à l'heure des méthodes pour un objet. Le terme 'méthode' est lié à la programmation orientée objet, un terme plus générique étant 'fonction'.

L'intérêt d'une fonction est d'éviter des répétitions du même code. Par exemple, votre programme va appliquer la même mise en page à du texte à plusieurs reprises, il est alors intéressant d'utiliser une fonction. Cette fonction pourrait être définie comme suit :

```
def list_print(texte):
    n_texte = " - %s - " % texte
    return n_texte
```

- `def` précise à python que l'on débute la description d'une fonction ;
- `texte` est un paramètre ;
- `return` permet de mettre fin à la fonction, et de renvoyer le contenu d'une variable.

Notez l'indentation pour la définition de bloc.

Une fonction ne doit pas forcément retourner quelque chose, elle peut par exemple juste ecrire du texte.

Dans un script, cette fonction pourra être appelée comme ceci :

```
txt1 = "hello"
txt2 = "bye"
ntxt = (list_print(txt1), list_print(txt2))
print """%s
%s
""" % (ntxt[0], ntxt[1])
```

A noter :

- l'utilisation des triples ” pour une chaîne de caractères qui s'affichera sur plusieurs lignes ;
- l'utilisation de “%s %s” % (a, b) : si l'on a plus de 2 chaînes à remplacer, on les met entre parenthèses.

1)

En anglais : *underscore*.