RES 302 : Réseaux IP

TP 1 : Introduction à la programmation en Python

Les exercices proposés dans ce TP sont à faire **individuellement**. Le travail sera à réaliser en binôme à partir du TP4.

Objectif du TP

L'objectif de ce TP est de prendre en main le langage de script Python, afin de se préparer au projet de programmation réseau qui débutera dans quelques semaines. Nous verrons notamment comment réaliser quelques fonctions spécifiques liées au projet à venir. Par ailleurs, en début de séance, il faudra configurer un espace de travail GIT qui servira durant tout le semestre.

1 Préparer un espace de travail sous GIT

GIT est un logiciel de gestion de version, principalement utilisé pour gérer le code source de projets. L'usage qu'on aura de GIT est le suivant : un serveur GIT hébergera votre code, et vous pourrez créer des copies de ce repertoire de travail localement sur votre mahcine (en salle de TP par exemple). Si vous faîtes des modifications sur votre copie locale, vous aurez alors le loisir de remonter ces changements sur le code présent sur le serveur. Cela permet notamment de facilier le travail à plusieurs sur un même code.

Pour l'ensemble des TP de RES 302, l'ensemble du code produit devra être disponible sur le projet GIT de chaque élève (ou binôme pour les séances de TP ultérieures).

Pour créer un projet GIT, il faut vous connecter sur le site `https://redmine-df.telecom-bretagne.eu` avec votre compte de l'école. Vous pourrez ensuite suivre la démarche de création (en cherchant "Nouveau projet"). Vous veillerez à activer le module "Repository", et nommerez votre projet **RES302-*votre_nom***

Pour l'utilisation de GIT, une aide pourra être trouvée en ligne (par exemple ici `https://git-scm.com/doc`). Voici quelques commandes qui pourront vous être utiles :

- *git clone <addr>* : Crée dans votre répertoire local de votre machine un nouveau répertoire avec le code source issu du serveur. Attention, cette commande écrase les fichiers que vous pourriez déjà avoir localement. Typiquement, cette commande n'est à faire que la première fois. Vous pouvez exécuter cette commande même si votre projet sur le serveur n'a pas de fichier, cela vous permet de créer le répertoire GIT localement sur votre machine.
- *git add <name>* : Ajoute un fichier ou un repértoire dans votre projet GIT (mais n'implique pas de changement sur le serveur).
- *git pull* : Rapatrie les changements présents sur le serveur, dans votre répertoire local. Ceci permet de récupérer la dernière version du code présent sur le serveur. Il faut être à la racine de votre répertoire de travail pour effectuer cette commande.
- *git push* : Déploie vos changements sur le serveur distant. Attention, il faut avoir fait la commande *git commit* avant.
- *git commit* : Enregistre les changements que vous avez fait localement. A faire avant de faire un *git push* si vous voulez déposer vos changements sur le serveur. Il faut être à la racine de votre répertoire de travail pour effectuer cette commande.

Typiquement, voici les commandes que vous pourrez faire pour démarrer votre projet (après avoir crée votre projet sur le redmine) :

```
mkdir RES302
cd RES302
git clone https://redmine-df.telecom-bretagne.eu/git/RES302-<votre_nom>
//à présent vous avez un repertoire nommé RES302-<votre_nom>
mkdir TP1
git add TP1
git commit
git push
```

2 Programmation Python

Python est un langage de script de haut niveau développé en 1993 par Guido Van Rossum. C'est un langage orienté objet, mais qui dispose d'outils permettant de se livrer à la programmation fonctionnelle ou impérative.

Ce TP est fortement inspiré de l'article wikibooks sur la programmation Python disponible à l'adresse : `http://en.wikibooks.org/wiki/Python_Programming`

3 Prise en main

3.1 Premier programme

Le premier programme que tout programmeur écrit s'intitule *Hello World !*. Ce programme affiche simplement la phrase *Hello World !* et quitte.

Ouvrez un éditeur de texte et créez un fichier nommé *hello.py*. Dans ce fichier, écrivez :

```
print ("Hello World")
```

Pour lancez le programme, tapez simplement dans un terminal :

```
python hello.py
```

Félicitations, vous avez écrit et lancé votre premier programme python !

3.2 Variables

Une variable peut être vue comme une boîte dans laquelle on peut placer des choses (e.g. des nombres). Voici un programme qui utilise une variable pour stocker un nombre :

```
lucky = 7
print (lucky)
```

Dans ce programme, nous avons créé une variable appelée *lucky* à laquelle nous avons assigné l'entier 7. Puis, nous souhaitons visualiser le contenu de la variable à l'aide de la fonction *print*.

En plus de stocker des choses dans les variables, nous pouvons également modifier la valeur de variable déjà initialisées :

```
changing = 3
print (changing)

changing = 9
print (changing)

red = 5
blue = 10
print (red, blue)

yellow = red
print (yellow, red, blue)

red = blue
print (yellow, red, blue)
```

3.3 Chaîne de caractères

Une chaîne de caractère (*string* en anglais) consiste simplement en une liste ordonnée de caractères. Par exemple, *"Hello"* est un string de 5 caractères.

Il y a trois manière de déclarer un string en Python : avec des apostrophes simples (*simple quote*) (’), des apostrophes doubles (") ou des apostrophes triples (”’). Dans chacun des cas, il faut débuter et terminer la déclaration en utilisant le même type de quote.

```
print ('I am a single quoted string')
print ("I am a double quoted string")
print ('''I am a triple quoted string''')
```

Il est également possible de concaténer deux strings (ou plus) ensembles à l’aide de l’opérateur **+**, de répéter un affichage à l’aide de l’opérateur ***** et de connaître la longueur (le nombre de caractère) d’un string à l’aide de la fonction **len()** :

```
print ("Hello " + "World !")
print ("Hello World !" * 5)
print (len ("Hello World !"))
```

Il est également possible de récupérer du texte depuis l’entrée standard à l’aide de la fonction **raw_input()** :

```
question = "What did you have for lunch ?"
print (question)

answer = raw_input()
print (answer)
```

Vous pouvez constater que la fonction **raw_input()** est bloquante, i.e. que tant que l’utilisateur n’appuiera pas sur la touche entrée, le programme est bloquée sur cette instruction.

Il peut être utile de convertir une chaîne de caractères en entier pour pouvoir effectuer des opérations arithmétiques. Pour ce faire, on peut utiliser la fonction **int()** :

```
print ("Please give me a number :")
answer = raw_input()

number = 10 + int(answer)

print ("If we add 10 to your number, we get " + str(number))
```

Notez comment se fait l'affichage du contenu de la variable *number*.

3.4 Exercices

1. Ecrire un programme qui demande à l'utilisateur d'entrer une chaîne de caractères, et afficher la longueur de cette chaîne.
2. Ecrire un programme qui demande à l'utilisateur d'entrer une chaîne de caractères et un nombre, et afficher *nombre* fois la chaîne en question.
3. Que se passe-t-il si un utilisateur entre un mot au lieu d'un nombre ?

4 Structures conditionnelles

Comme dans la majorité de langage de programmation, il est possible d'effectuer des actions différentes en fonction de conditions. La structure est assez classique *si alors sinon*. Il faut terminer une condition par deux points (" :") et marquer une indentation de quatre espace pour les instructions devant être exécutées si la condition est vérifiée. La syntaxe est la suivante :

```
name = raw_input("What is your name ? ")
password = raw_input("What is the password ? ")

if name == "Josh" and password == "Friday" :
    print ("Welcome Josh !")
elif name == "Fred" and password == "Rock" :
    print ("Welcome Fred !")
else :
    print ("I don't know you")
```

Voici les symboles pour effectuer les différents tests possibles ainsi que les différents mots clefs pour enchaîner plusieurs conditions :

<	inférieur		and	et
<=	inférieur ou égal		or	ou
>	supérieur			
>=	supérieur ou égal			
==	égal			
!=	différent			

TABLE 1 – Symboles pour les conditions

4.1 Exercices

1. Ecrire un programme qui génère un nombre aléatoire entre 0 et 10 et qui demande à l'utilisateur de deviner ce nombre. Suivant ce qu'entre l'utilisteur (le nombre correct ou non) le programme doit afficher un message ne conséquence. Pour générer un nombre aléatoire, vous pouvez utiliser la fonction *random.randint(a,b)* ou *a* et *b* sont les bornes de l'intervalle dans lequel le nombre aléatoire sera tiré. Pour utiliser cette fonction, vous devez importer le module *random* à l'aide de l'instruction : *import random* en début de fichier.
2. Ecrire un programme qui demande à l'utilisateur d'entrer un nombre et qui affiche si ce nombre est pair ou impair (vous pourrez vous servir de la fonction modulo pour vérifier la parité, qui s'utilise avec le caractère %)

5 Boucles et itérations

Pour effectuer une action répétitive, on utilise généralement une boucle. Python permet d'utiliser deux types de boucles.

5.1 Boucles while

Le premier type de boucle est de type *tant que* ***condition*** *faire* ***action***. La syntaxe est la suivante :

```
a = 0

while a < 10 :
    a += 1
    print (a)

print ("The final value of a is " + str(a))
```

Notez que la dernière instruction n'est pas incluse dans la boucle. Cela est dû à l'indentation (l'instruction est au même niveau que le *while*.

5.2 Boucles for

Le deuxième type de boucle est de type *pour* ***valeur*** *allant de de* ***telle borne à telle borne*** *faire* ***action***. La syntaxe est la suivante :

```
sum = 0

for item in range(0,10) :
    sum += 1
    print (sum)

print ("The final value of a is " + str(sum))
```

Il est également possible d'avoir un pas d'itération négatif. La syntaxe est la suivante :

```
sum = 0

for item in range(10, 0, -1) :
    sum += 1
    print (sum)

print ("The final value of a is " + str(sum))
```

5.3 Instruction *break* et *continue*

Il est possible d'arrêter une boucle prématurément (avant d'atteindre la condition d'arrêt) à l'aide de l'instruction *break*. La syntaxe est la suivante (fonctionne également pour les boucles for) :

```
a = 0

while a < 10 :
    a += 1
    print (a)
    break

print ("The final value of a is " + str(a))
```

Il est également possible d'aller directement à l'itération suivante sans effectuer la suite des instructions à l'aide de l'instruction *continue*. La syntaxe est la suivante :

```
sum = 0

for item in range(10, 0, -1) :
    sum += 1
    continue
    print (sum)

print ("The final value of a is " + str(sum))
```

5.4 Exercices

1. Modifiez votre programme de la question 1 de la section 2.1 pour tirer un nombre aléatoire entre 0 et 100. Ensuite, faites en sorte que, suivant le nombre entré par l'utilisateur, le programme affiche :
   — *vous avez gagne !* si le nombre est exact ;
   — *plus grand* si le nombre entré est plus petit que celui qu'il faut deviner ;
   — *plus petit* si le nombre entré est plus grand que celui qu'il faut deviner ;
   et ce jusqu'à ce que l'utilisateur trouve le bon nombre.
2. Ecrire un programme qui affiche la suite de *Fibonacci*. Le programme doit demander à l'utilisateur le rang jusqu'au quel il doit afficher la suite. Pour rappel, la suite de Fibonacci est définie par :
   $F_{n+2} = F_{n+1} + F_n$ avec $F_0 = 0$, $F_1 = F_2 = 1$
3. Ecrire un programme qui demande à l'utilisateur d'entrer une suite de nombre de manière itérative et qui affiche la moyenne à chaque nouveau nombre entré. L'utilisateur pourra par exemple taper q ou quit pour quitter le programme.

6 Séquence

Une séquence permet de stocker plusieurs valeurs de manière organisée. Python supporte principalement 4 types de séquences : strings, lists, tuples et dictionnaires. Les exemples suivants peuvent être exécutés en ouvrant un interpréteur python dans un terminal : tapez *python* dans un terminal, et ensuite vous pourrez entrer ces différentes commandes. Pour quitter l'interpréteur, vous pourrez taper *quit()*.

6.1 String

Nous avons déjà abordé les chaînes de caractères (string) dans la première partie de ce TP. Dans la plupart des langages, les éléments d'un tableau ou les caractères d'une chaîne peuvent être récupérés à l'aide des crochets. En python cela fonctionne de la même manière. Testez les instructions suivantes :

```
 "Hello, world !"[0]
"Hello, world !"[1]
"Hello, world !"[2]
"Hello, world !"[3]
"Hello, world !"[-2]
```

Les indices sont numérotés de 0 à n-1 où n est le nombre de caractères. Les indices négatifs sont comptés depuis la fin de la chaîne.

En python il est également possible de récupérer un sous ensemble de la chaîne à l'aide du symbole **:** Testez les instructions suivantes :

```
"Hello, world !"[3 :9]
string = "Hello, world !"
string[ :5]
string[-6 :-1]
string[-9 :]
string[ :]
```

6.2 List

Une liste est simplement une liste de valeur. Une liste est créée à l'aide des crochets. Par exemple, une liste vide est créée de la manière suivante :

```
spam = [ ]
```

Les valeurs d'une liste sont séparées par des virgules :

```
spam = ["bacon", "eggs", 42]
```

Comme vous pouvez le voir, une liste peut contenir des objets de types différents. Comme les caractères dans une chaîne, les objets d'une liste peuvent être récupérés par leurs indices et on peut extraire des sous-listes :

```
spam = ["bacon", "eggs", 42]
spam
spam[0]
spam[-2]
spam[1 :2]
```

Pour compter le nombre d'objet dans une liste, on utilise la fonction **len()** :

```
spam = ["bacon", "eggs", 42]
len(spam)
```

Les objets dans une liste peuvent également être remplacés (ce qui n'est pas possible avec les chaînes de caractères) :

```
spam = ["bacon", "eggs", 42]
spam
spam[1]
spam[1] = "ketchup"
spam
```

Pour ajouter des éléments dans une liste, on peut utiliser les fonctions **append()** (insertion en fin) ou **insert()** (insertion en précisant la position) :

```
spam = ["bacon", "eggs", 42]
spam.append(10)
spam
spam.insert(1, "and")
spam
```

Il est également possible de supprimer des objets d'une liste à l'aide de l'instruction **del** :

```
spam = ["bacon", "eggs", 42]
spam
del spam[1]
spam
```

Enfin, les listes ont un comportement non intuitif : supposons deux listes a et b. Si on affecte a à b et que l'on modifie ensuite a, b sera également modifié :

```
a = [1, 2, 3]
b = a
del a[2]
print a
print b
```

Ce comportement peut être contourné en utilisant le type d'affectation b = a[:]

6.3 Tuple

Les tuples sont très similaires aux listes, à la différence qu'ils ne sont pas modifiables. Lorsque vous avez créé un tuple, vous ne pouvez pas y ajouter, modifier ou supprimer un élément. Pour déclarer un tuple, il faut utiliser les virgules :

```
unchanging = "rocks", 0, "the universe"
```

Il est souvent nécessaire d'utiliser des parenthèses pour différencier différents tuples. Par exemple lors de l'affectation multiple sur une même ligne :

```
foo, bar = "rocks", (0, "the universe")
print foo
print bar
```

6.4 Dictionnaire

Les dictionnaires sont également très similaires aux listes et ils sont modifiables. Chaque élément dans un dictionnaire comporte deux parties : une clé et une valeur. Appeler une clé d'un dictionnaire renvoie la valeur liée à cette clé :

```
definitions = {"guava" : "a tropical fruit", "python" : "a programming language"}
print definitions
print definitions["python"]
print len(definitions)
```

Pour ajouter un élément dans un dictionnaire, il suffit de le déclarer comme une variable :

```
definitions = "guava" : "a tropical fruit", "python" : "a programming language"
definitions["new key"] = "new value"
print definitions
```

Ou alors on peut utiliser la fonction update :

```
definitions = "guava" : "a tropical fruit", "python" : "a programming language"
definitions.update("animal" : "loup")
print definitions
```

Noter que les valeurs du dictionnaire peuvent être différentes, y compris des tuples :

```
definitions = "guava" : "a tropical fruit", "python" : "a programming language"
definitions.update("animal" : ("loup", 2))
print definitions
print definitions["animal"][0]
print definitions["animal"][1]
```

7 Fonction

Une fonction est défini en Python avec le format suivant :

```
def functionname(arg1, arg2, ...) :
    statement 1
    statement 2
```

8 Exercices

1. Définissez une liste contenant 5 nombres aléatoires entre 0 et 100. Puis effectuez les actions suivantes :
   — triez et affichez la liste ;
   — ajoutez l'élément 12 à la liste et affichez la liste ;
   — ajoutez un nombre aléatoire à la liste de manière à ce qu'elle reste triée ;
   — affichez la sous-liste du 3e élément à la fin de la liste.
2. Implémentez une pile LIFO avec une liste. Pour ce faire, vous utiliserez trois fonctions :
   — pile : qui crée et retourne une pile à partir d'une liste de 4 variables d'éléments passés en paramètre (par exemple 4 nombre aléatoires ;
   — empile : empile un élément en haut de la pile ;
   — dépile : dépile un élément du haut de la pile (et renvoie cet élément)

3. De la même manière, implémentez une queue FIFO avec une liste.

9 Le passage d'arguments en ligne de commande

Python supporte complètement la création de programmes qui peuvent être lancés en ligne de commande, à l'aide d'arguments et de drapeaux longs ou cours pour spécifier diverses options. Commençons par un exemple simple, en utilisant *sys.argv* :

```
import sys

for arg in sys.argv :
    print arg
```

Chaque argument de ligne de commande passé au programme est ajouté à sys.argv, qui est un objet liste. Ici le script affiche chaque argument sur une ligne séparée. Le premier élément de sys.argv est le nom du script que vous appelez. Les arguments de la ligne de commande doivent être séparés par des espaces et chacun se présente comme un élément distinct dans la liste sys.argv. Les drapeaux de la ligne de commande, comme *–help*, se présentent également comme des éléments propres dans la liste sys.argv.

Comme vous pouvez le voir maintenant, vous disposez indiscutablement de toutes les informations passées à la ligne de commande. Pour des programme simples qui ne nécessite qu'un seul argument et pas de drapeau, vous pouvez simplement utiliser sys.argv[1] pour accéder à l'argument. Cependant il n'est pas aisé de fonctionner de cette manière quand vous voulez définir plusieurs options d'arguments, associés à des drapeaux. Pour cela, il vous faut recourir au module *getopt*.

La fonction *getopt* du module *getopt* prend trois paramètres : la liste des arguments (que vous obtenez à partir de sys.argv[1 :]), une chaîne contenant tous les drapeaux courts possibles acceptés par le programme et une liste des drapeaux plus longs qui correspondent aux versions courtes. La syntaxe utilisé est la suivante. On indique une lettre uniquement pour indiquer une option sans argument (par exemple *-h*), et une lettre avec deux points pour indiquer que l'option prend un paramètre (par exemple *f :*). La fonction renvoie un tuple, dont le premier élément contient une liste de tuples (drapeau, argument), et le deuxième élément contient un liste des arguments donnés sans drapeau. Voici un exemple :

```
import sys, getopt

try :
    opts, args = getopt.getopt(sys.argv[1 :], "hf :s", ["help", "first="])

except getopt.GetoptError as err :
    print "not properly used - print the options"
    sys.exit(2)

for opt, arg in opts :
    if opt in ("-h", "–help") :
        print arg
        sys.exit()
```

9.1 Exercice

Ecrire un programme qui accepte trois drapeaux :
— -h / –help : affiche l'aide au lancement du programme
— -s / –string : attend alors en paramètre une chaine de caractères
— -n / –number : attend alors en paramètre un nombre entier

L'utilisateur du programme devra utiliser obligatoirement les deux options -s et -n. Votre programme devra afficher la nième lettre de la chaîne de caractères entrée. Par exemple, si l'utilisateur appelle votre programme ainsi : *./programme -s coucou -n 2*, votre programme devra afficher o. Si l'utilisateur entre un chiffre plus grand que la taille de la chaîne de caractère, il faut afficher l'aide et sortir du programme. Si l'utilisateur n'entre pas les deux valeurs (-n et -s), il faudra également sortir du programme en affichant l'aide également.