

Initiation à la programmation en Python

Damien Vergnaud

École Normale Supérieure

18 février 2015

Table des matières

1 Les listes (Première approche)

2 Les fonctions

3 Les modules

4 Les fichiers

Les listes

- Une **liste** est une structure de données qui contient une série de valeurs.
- Ces valeurs ne sont pas forcément du même type
- Une liste est déclarée par une série de valeurs séparées par des virgules, et le tout encadré par des crochets:

```
>>> animaux = ['girafe', 'hippopotame', 'singé', 'dahu']
>>> taille = [5.0, 1.0, 0.7, 2.0]
>>> mix = ['girafe', 5.0, 'dahu', 2]
>>> animaux
['girafe', 'hippopotame', 'singé', 'dahu']
>>> print animaux
['girafe', 'hippopotame', 'singé', 'dahu']
>>> taille
[5.0, 1.0, 0.7, 2.0]
>>> mix
['girafe', 5.0, 'dahu', 2]
```

Les listes

- On peut rappeler ses éléments par leur numéro de position (**indice**)
- les indices d'une liste de n éléments commencent à 0 et se terminent à $n - 1$

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu']
>>> animaux[0]
'girafe'
>>> animaux[1]
'hippopotame'
>>> animaux[3]
'dahu'
```

Les listes

- Tout comme les chaînes de caractères les listes supportent l'opérateur '+' de concaténation, ainsi que '*' pour la duplication:

```
>>> animaux = ['aigle', 'ecureuil']
>>> animaux + animaux
['aigle', 'ecureuil', 'aigle', 'ecureuil']
>>> animaux * 3
['aigle', 'ecureuil', 'aigle', 'ecureuil', 'aigle', 'ecureuil']
```

- On peut également considérer une chaîne de caractères comme une liste :

```
>>> animal = "hippopotame"
>>> animal
'hippopotame'
>>> animal[0]
'h'
>>> animal[4]
'o'
>>> animal[10]
'e'
```

Indiçage négatif

- La liste peut également être indexée avec des nombres négatifs selon le modèle suivant :

```
liste          : ['girafe', 'hippopotame', 'singe', 'dahu']
index positif  :      0           1           2           3
index négatif  :     -4         -3          -2          -1
```

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu']
>>> animaux[-4]
'girafe'
>>> animaux[-2]
'singe'
>>> animaux[-1]
'dahu'
```

Tranches

- Un autre avantage des listes est que l'on peut en sélectionner une **partie** en utilisant un indicé construit sur le modèle $[m:n+1]$

```
>>> animaux = ['girafe', 'hippopotame', 'singé', 'dahu', 'ornithorynque']
>>> animaux[0:2]
['girafe', 'hippopotame']
>>> animaux[0:3]
['girafe', 'hippopotame', 'singé']
>>> animaux[0:]
['girafe', 'hippopotame', 'singé', 'dahu', 'ornithorynque']
>>> animaux[:]
['girafe', 'hippopotame', 'singé', 'dahu', 'ornithorynque']
>>> animaux[1:]
['hippopotame', 'singé', 'dahu', 'ornithorynque']
>>> animaux[1:-1]
['hippopotame', 'singé', 'dahu']
```

Les instructions `range()` et `len()`

- L'instruction `range` vous permet de créer des listes d'entiers de manière simple et rapide

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0,5)
[0, 1, 2, 3, 4]
>>> range(15,21)
[15, 16, 17, 18, 19, 20]
>>> range(0,1000,100)
[0, 100, 200, 300, 400, 500, 600, 700, 800, 900]
>>> range(0,-10,-1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

- L'instruction `len` vous permet de connaître la longueur d'une liste

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu' , 'ornithorynque']
>>> len(animaux)
5
>>> len(range(10))
10
```

L'instruction for

Syntaxe

```
for variable in list:  
    instruction      # ou bloc d'instructions
```

- Parcourir une séquence :

Exemples

```
for lettre in "ciao":  
    print lettre, # c i a o
```

```
for x in [2, 'a', 3.14]:  
    print x, # 2 a 3.14
```

```
for i in range(5):  
    print i, # 0 1 2 3 4
```

L'instruction break

L'instruction `break` provoque le passage à l'instruction qui suit immédiatement le corps de la boucle `while` ou `for`.

Exemples

```
for x in range(1, 11):
    if x == 5:
        break
    print x,

print "\nBoucle interrompue pour x =", x

# affiche :
# 1 2 3 4
# Boucle interrompue pour x = 5
```

L'instruction continue

L'instruction continue fait passer à l'itération suivante les instructions `while` ou `for`.

Exemples

```
for x in range(1, 11):
    if x == 5:
        continue
    print x,

print "\nLa boucle a sauté la valeur 5"

# affiche :
# 1 2 3 4 6 7 8 9 10
# La boucle a sauté la valeur 5
```

Table des matières

1 Les listes (Première approche)

2 Les fonctions

3 Les modules

4 Les fichiers

Définition d'une fonction

- Les **fonctions** permettent de décomposer les programmes en sous-programmes et de réutiliser des morceaux de programmes.
- Une fonction est un programme Python défini à partir de paramètres d'entrées qui retourne éventuellement une valeur de sortie.
- La syntaxe d'une fonction Python est la suivante :

Syntaxe

```
def <nom de la fonction> ( <liste de paramètres> ) :  
    <bloc d'instructions>
```

- Une instruction `return <expression>` dans le bloc d'instructions définissant une fonction provoque la fin d'exécution de la fonction avec le retour de la valeur de l'expression qui suit.

Définition d'une fonction

```
>>> def compter_lettre(lettre, texte) :  
    n=0  
    for c in texte :  
        if c == lettre :  
            n += 1  
    return "nombre d'occurences de la lettre " \  
+ lettre + " : " + 'n'  
  
>>> print compter_lettre('e', 'je reviens')  
nombre d'occurrences de la lettre e : 3  
>>>
```

Appel d'une fonction

- Une fois qu'une fonction f a été définie, elle peut être utilisée dans une expression particulière qu'on nomme un appel de fonction et qui a la forme $f(v_1, v_2, \dots, v_n)$, où v_1, v_2, \dots, v_n sont des expressions dont la valeur est transmise au paramètres.
- On parle d'un appel de fonction **par valeur** par opposition à un appel **par référence**.
- Python offre un mécanisme d'instanciation des paramètres par défaut. On peut écrire la liste des paramètres en entête d'une définition de fonction comme suit :

Syntaxe

```
def <nom de la fonction> (p1, ..., pk, pk+1=expr1, ..., pk+n=exprn) :  
    <bloc d'instructions>
```

- Les k premiers paramètres doivent obligatoirement être précisés à l'appel de fonction mais pas les n derniers. L'appel de fonction se fait donc avec k arguments au minimum et $k + n$ arguments au maximum. Si un paramètre $pk+i$ n'est pas instancié explicitement, il prend la valeur par défaut de $expr_i$.

Appel d'une fonction

```
>>>def pluriel(mot, famille = 'standard'):  
    if famille == 'standard' :  
        return mot + 's'  
    if famille == 's':  
        return mot  
    if famille == 'oux' :  
        return mot + 'x'  
    if famille == 'al' :  
        return mot[:-1] + 'ux'  
  
>>> print pluriel('maison')  
'maisons'  
>>> print pluriel('souris', 's')  
'souris'  
>>> print pluriel('chou', 'oux')  
'choux'  
>>> print pluriel('cheval', 'al')  
'chevaux'
```

Variables locales et variables globales

- Les variables qui sont introduites dans la définition d'une fonction peuvent être utilisées dans la suite de la définition mais **pas à l'extérieur de la fonction**.
- Ces variables sont dites **locales** par opposition aux variables **globales** qui sont introduites à l'extérieur de la définition d'une fonction et qui peuvent être utilisées à l'intérieur comme à l'extérieur de cette définition.
- Lorsque le même nom est utilisé pour introduire une variable locale et une variable globale, Python distingue bien deux variables différentes mais à l'intérieur de la définition de la fonction, c'est à la variable locale auquel le nom réfère.

Variables locales et variables globales

```
>>> def f(x):  
    y=2  
    return x + y
```

```
>>> print f(3)
```

```
5
```

```
>>> print y
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#5>", line 1, in <module>
```

```
    print y
```

```
NameError: name 'y' is not defined
```

```
>>> u = 7
```

```
>>> def g(v):
```

```
    return u * v
```

```
>>> print g(2)
```

```
14
```

```
>>> def h(u):  
    return u
```

```
>>> print h(3)
```

```
3
```

```
>>> print u
```

```
7
```

```
>>> def k(w) :
```

```
    u = 5
```

```
    return w+u
```

```
>>> print k(3)
```

```
8
```

```
>>> print u
```

```
7
```

```
>>>
```

Table des matières

1 Les listes (Première approche)

2 Les fonctions

3 Les modules

4 Les fichiers

Modules

- On peut ranger les définitions de fonctions se rapportant à une même application au sein d'un script commun baptisé **module**.
- Un module est sauvegardé sous forme d'un fichier dont le nom a la forme `<nom du module>.py`.
- Pour utiliser un module, il faut se servir de l'instruction `import <nom du module>`.
- L'exécution de cette instruction consiste à exécuter le script définissant le module (ce script peut contenir des instructions autres que des définitions de fonctions).
- Pour importer un module, Python a besoin de connaître le chemin qui permet d'accéder au fichier correspondant. Ce chemin doit apparaître dans la liste des chemins possibles stockés dans la variable `path` du module `sys`.

Modules - Première méthode d'importation

```
>>> import random
>>> random.randint(0,10)
9
```

Regardons de plus près cet exemple :

- L'instruction `import` vous permet d'importer toutes les fonctions du module `random`
- Ensuite, nous utilisons la fonction (ou méthode) `randint(a,b)` du module `random`; attention cette fonction renvoie un nombre entier aléatoirement entre `a` inclus et `b` inclus.

Modules - Deuxième méthode d'importation

- Pour disposer d'une fonction du module:

Syntaxe

```
from [module] import [fonction]
```

- Pour disposer de toutes les fonctions d'un module:

Syntaxe

```
from [module] import *
```

```
from math import *  
racine = sqrt(49)  
angle = pi/6  
print sin(angle)
```

Modules courants

- `math` : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- `sys` : passage d'arguments, gestion de l'entrée/sortie standard etc...
- `os` : dialogue avec le système d'exploitation.
- `random` : génération de nombres aléatoires.
- `time` : permet d'accéder aux fonctions gérant le temps.
- `calendar` : fonctions de calendrier.
- `profile` : permet d'évaluer le temps d'exécution de chaque fonction dans un programme (profiling en anglais).
- `urllib` : permet de récupérer des données sur internet depuis python.
- `Tkinter` : interface python avec Tk (permet de créer des objets graphiques; nécessite d'installer Tk).
- `string` : opérations sur les chaînes de caractères ; à noter que la plupart des fonctions du module `string` sont maintenant obsolètes; il est maintenant plus correct d'utiliser les méthodes directement associées aux objets de type `string`.
- `re` : gestion des expressions régulières.
- `pickle` : écriture et lecture de structures python (comme les dictionnaires par exemple).
- ...

Obtenir de l'aide sur les modules importés

- Pour obtenir de l'aide sur un module rien de plus simple, il suffit d'invoquer la commande `help` :

```
>>> import random
>>> help(random)
```

- Il est aussi possible d'invoquer de l'aide sur une fonction particulière d'un module en la précisant de la manière suivante :

```
>>> help(random.randint)
```

Table des matières

1 Les listes (Première approche)

2 Les fonctions

3 Les modules

4 Les fichiers

Utilisation de fichiers

- Il est important de dissocier les données des programmes qui les utilisent en rangeant ces données dans des fichiers séparés.
- Le module `os` contient des fonctions qui permettent de localiser les fichiers :
 - `getcwd()`: Retourne le chemin du répertoire courant

`chdir(<ch>)`: Change le répertoire courant qui prend la valeur donnée par la chaîne de caractères `<ch>`

`path.isfile(<ch>)`: Retourne un booléen qui indique s'il existe un fichier dont le chemin est la chaîne de caractères `<ch>`

`path.isdir(<ch>)`: Retourne un booléen qui indique s'il existe un répertoire dont le chemin est la chaîne de caractères `<ch>`

```
>>> from os import chdir
>>> chdir("/home/exercices")
```

Les deux formes d'importation

```
>>>>> import os
>>> rep_cour = os.getcwd()
>>> print rep_cour
```

```
>>> from os import getcwd
>>> rep_cour = getcwd()
>>> print rep_cour
```

Utilisation de fichiers

- Pour utiliser un fichier identifié par le chemin `ch` dans un programme Python, il faut commencer par l'ouvrir par l'appel de fonction `open(<ch>, [<mode>])` qui retourne un objet de type `file`.
- Le paramètre facultatif `<mode>` indique le mode d'ouverture du fichier :
 - `r` : mode lecture (le fichier doit exister préalablement)
 - `w` : mode écriture (si le fichier existe, les données sont écrasées, sinon le fichier est créé)
 - `a` : mode ajout (si le fichier existe, les données écrites vont l'être après celles existantes, sinon le fichier est créé)
- Si le mode est omis, le mode par défaut est `r`.

Utilisation de fichiers

- un objet de type `file` est associé à des attributs et des méthodes. En voici quelques-unes :
 - `read([<n>])` : Retourne la chaîne des `<n>` caractères restants.
 - `write(<ch>)` : Écrit la chaîne de caractères `<ch>`.
 - `close()` : Ferme le fichier quand il est fini d'être utilisé.
 - `seek(<n>)` : Choisit le caractère `<n>` comme position courante du fichier.
 - `tell()` : Retourne le caractère en position courante.

Exemple

- Créez un fichier dans un éditeur de texte que vous sauvez dans votre répertoire avec le nom 'exemple.txt', par exemple :

```
Ceci est la premiere ligne  
Ceci est la deuxieme ligne  
Ceci est la troisieme ligne  
Ceci est la quatrieme et derniere ligne
```

```
>>> filin = open('exemple.txt','r')  
>>> filin  
<open file 'exemple.txt', mode 'r' at 0x40155b20>  
>>> filin.readlines()  
['Ceci est la premiere ligne\n', 'Ceci est la deuxieme ligne\n',  
'Ceci est la troisieme ligne\n',  
'Ceci est la quatrieme et derniere ligne\n']  
>>> filin.close()  
>>> filin  
<closed file 'exemple.txt', mode 'r' at 0x40155b20>
```

Exemple

```
Ceci est la premiere ligne  
Ceci est la deuxieme ligne  
Ceci est la troisieme ligne  
Ceci est la quatrieme et derniere ligne
```

```
>>> filin = open('exemple.txt','r')  
>>> lignes = filin.readlines()  
>>> for i in lignes:  
...     print i  
...  
Ceci est la premiere ligne  
  
Ceci est la deuxieme ligne  
  
Ceci est la troisieme ligne  
  
Ceci est la quatrieme et derniere ligne  
>>> filin.close()
```

Exemple

```
Ceci est la premiere ligne  
Ceci est la deuxieme ligne  
Ceci est la troisieme ligne  
Ceci est la quatrieme et derniere ligne
```

```
>>> filin = open("exemple.txt","r")  
>>> filin.read()  
'Ceci est la premiere ligne\nCeci est la deuxieme ligne\nCeci est la troisi  
>>> filin.close()
```

Exemple

```
Ceci est la premiere ligne  
Ceci est la deuxieme ligne  
Ceci est la troisieme ligne  
Ceci est la quatrieme et derniere ligne
```

```
>>> filin = open("exemple.txt","r")  
>>> filin.tell()  
0L  
>>> filin.readline()  
'Ceci est la premiere ligne\n'  
>>> filin.tell()  
27L  
>>> filin.seek(0)  
>>> filin.tell()  
0L  
>>> filin.readline()  
'Ceci est la premiere ligne\n'  
>>> filin.close()
```

Exemple

```
>>> animaux = ['girafe', 'hippopotame', 'singe', 'dahu', 'ornithorynque']
>>> filout = open('exemple2.txt','w')
>>> for i in animaux:
...     filout.write(i)
...
>>> filout.close()
>>>
[fuchs@opera ~]$ more exemple2.txt
girafehippopotamesingedahuornithorynque
[fuchs@opera ~]$
```