

Initiation à la programmation en Python

Damien Vergnaud

École Normale Supérieure

24 février 2010

Table des matières

- 1 Introduction
- 2 Variables
- 3 Entrées-Sorties
- 4 Les listes (Première approche)
- 5 Les structures de contrôle

Organisation du cours

Organisation des séances

2 heures

- \simeq 30 min de cours
- \simeq 1h30 de TP

(à adapter)

Support de cours

Disponible progressivement à l'adresse réticulaire

<http://www.math.unicaen.fr/~vergnaud/initPython.php>

- transparents
- feuilles de TPs
- corrections

Organisation du cours

Cours libre : pas d'examen, mais un projet (avec soutenance) délivrant 3 ECTS
Le cours peut s'adapter aux besoins

Contact

Damien Vergnaud

`damien.vergnaud@ens.fr`

Département d'informatique

Hauts du DI 4, Bâtiment principal

Escalier A, 3ème étage

Quelques références

- <http://www.python.org>
- Apprendre à programmer avec Python – Gérard SWINNEN
http://fr.wikibooks.org/wiki/Apprendre_à_programmer_avec_Python
- Cours de Python – Patrick FUCHS
<http://www.dsimb.inserm.fr/~fuchs/python/>
- Notes de cours Python – Robert CORDEAU
<http://www.iut-orsay.fr/dptmphy/Pedagogie/>
- Introduction to Programming using Python – Programming Course for Biologists at the Pasteur Institute
www.pasteur.fr/formation/infobio/python/
- Ingénierie Linguistique en Python (TP) <http://igm.univ-mlv.fr/ens/Master/M1/2009-2010/IngenierieLinguistique1/index.php>

Caractéristiques de Python

Python

- est **portable** (Unix, MacOS, Windows, ...)
- est **gratuit** (sans restriction dans des projets commerciaux).
- est **simple** (programmes très compacts et très lisibles).
- **gère ses ressources** (mémoire, descripteurs de fichiers...) sans intervention du programmeur
- est **orienté-objet**.
- est **dynamiquement typé**.
- est **extensible** (on peut l'interfacier avec des bibliothèques C existantes).
- donne accès à une **grande variété de services**

Écrire un programme Python : Méthode 1

Utiliser l'interpréteur

- 1 Lancez la console/terminal puis tapez :

```
python
```

- 2 Après quelques lignes indiquant la version de Python, vous voyez l'invite :

```
>>>
```

- 3 En tapant

```
>>> print 'Hello world!'
```

puis la touche entrée vous voyez :

```
>>> print 'Hello world!'
Hello world!
```

Pour quitter le terminal Python, et revenir au terminal classique, utiliser **Ctrl+D**.

Écrire un programme Python : Méthode 2

Utiliser un éditeur de texte

- L'interpréteur présente vite des limites si l'on veut exécuter une suite d'instructions plus complexe.
- On peut enregistrer ces instructions dans un fichier (que l'on appelle un **script Python**) en utilisant un **éditeur de texte** (Emacs, Scite, ...).
- Pour reprendre l'exemple précédent, ouvrez un fichier et tapez le code suivant.

```
print 'Hello world!'
```

- Ensuite enregistrez votre fichier sous le nom `test.py`
- Dans une console, donner le nom de votre script comme argument à la commande `python` :

```
python test.py
```


Écrire un programme Python : Méthode 3

Rendre un script exécutable

Il est possible de rendre un script python exécutable. Deux opérations :

- 1 Indiquer au shell la localisation de l'interpréteur Python en ajoutant au fichier test.py la première ligne comme suit :

```
#!/usr/bin/python  
print 'Hello world!'
```

- 2 Rendre le script exécutable en tapant dans la console :

```
chmod +x test.py
```

Pour lancer le script, taper son nom précédé des deux caractères `./`

```
./test.py
```

Les commentaires

Syntaxe

Les commentaires débutent par # et s'étendent jusqu'à la fin de la ligne courante.

Exemple

```
# Ceci est un commentaire
```

Recommandation : commenter selon les niveaux suivants :

- programme : pour indiquer le nom de l'auteur, la date de création, les dates et auteurs des différentes modifications, ainsi que la raison d'être du programme ;
- fonction : pour indiquer les paramètres et la raison d'être de la fonction ;
- groupe d'instructions : pour exprimer ce que réalise une fraction significative d'une procédure ;
- déclaration ou instruction : le plus bas niveau de commentaire.

Table des matières

1 Introduction

2 Variables

3 Entrées-Sorties

4 Les listes (Première approche)

5 Les structures de contrôle

Variables

- Une **variable** est une zone de la mémoire dans laquelle on stocke une valeur;
- cette variable est définie par un **nom**
(pour l'ordinateur, il s'agit en fait d'une adresse – une zone de la mémoire).
- Les noms de variables sont des noms que vous choisissez.
- Ce sont des suites de **lettres** (non accentuées) ou de **chiffres**.
- Le premier caractère est **obligatoirement** une lettre.
(Le caractère `_` est considéré comme une lettre)
- Python distingue les minuscules des majuscules.

Noms de variables et mots réservés

- Un nom de variable ne peut pas être un mot réservé du langage :

```
and      assert  break   class
continue def     del     elif
else     except  exec    finally
for      from    global  if
import   in      is      lambda
not      or      pass    print
raise    return  try     while
yield
```

Recommandations

- Utiliser des identificateurs significatifs.
- Réserver l'usage des variables commençant par une majuscule pour les **classes**.

Type de variable

- Le type d'une variable correspond à la nature de celle-ci.
- Les 3 types principaux dont nous aurons besoin sont :
 - les **entiers**,
 - les **flottants** et
 - les **chaînes de caractères**.
- Il existe de nombreux autres types (e.g. pour les nombres complexes), c'est d'ailleurs un des gros avantages de Python.

Déclaration et assignation

- En python, la **déclaration** d'une variable et son assignation (c.à.d. la première valeur que l'on va stocker dedans) se fait en même temps.

```
>>> x = 2
>>> x
2
>>>
```

- Dans cet exemple, nous avons stocké un entier dans la variable x, mais il est tout a fait possible de stocker des réels ou des chaînes de caractères :

```
>>> x = 3.14
>>> x
3.1400000000000001
>>> x = 'Bonjour !'
>>> x
'Bonjour !'
>>>
```

Le type entier

- Opérations arithmétiques

```
20 + 3 # 23
20 - 3 # 17
20 * 3 # 60
20 ** 3 # 8000
20 / 3 # 6 (division entière)
20 % 3 # 2 (modulo)
```

- Les entiers longs (seulement limités par la RAM)

```
2 ** 40 # 1099511627776L
3 * 72L # 216L
```


Le type flottant

- Les flottants sont notés avec un point décimal ou en notation exponentielle :

```
2.718
3e8
6.023e23
```

- Ils supportent les mêmes opérations que les entiers, sauf :

```
20.0 / 3 # 6.666666666666667
20.0 // 3 # 6 (division entière forcée)
```

- L'import du module `math` autorise toutes les opérations mathématiques usuelles :

```
from math import sin, pi
print sin(pi/4) # 0.70710678118654746
```

Les expressions booléennes

- Deux valeurs possibles : False, True.
- Opérateurs de comparaison : ==, !=, >, >=, <, <=

```
2 > 8 # False
2 <= 8 < 15 # True
```

- Opérateurs logiques : not, or, and

```
(3 == 3) or (9 > 24) # True (dès le premier membre)
(9 > 24) and (3 == 3) # False (dès le premier membre)
```

- Les opérateurs logiques et de comparaisons sont à valeurs dans False, True

Les chaînes de caractères

- Elles peuvent être incluses entre simples quotes (apostrophes) ou doubles quotes (guillemets):

```
>>> 'Une chaine'  
'Une chaine'  
>>> 'n\'est-ce pas'  
"n'est-ce pas"  
>>> "n'est-ce pas"  
"n'est-ce pas"  
>>> '"Oui," dit-il.'  
'"Oui," dit-il.'  
>>> "\"Oui,\" dit-il."  
'"Oui," dit-il.'  
>>> '"N\'est-ce pas," repondit-elle.'  
'"N\'est-ce pas," repondit-elle.'
```

Les chaînes de caractères

- Les textes dans les chaînes peuvent se poursuivre sur plusieurs lignes

```
print """
Usage: trucmuche [OPTIONS]
    -h                Affiche cette notice d'usage
    -H hôte           hôte auquel il faut se connecter
"""
```

- Les chaînes peuvent être concaténées avec l'opérateur +, et répétées avec *:

```
>>> word = 'Help' + 'A'
>>> word
'HelpA'
>>> '<' + word*5 + '>'
'<HelpAHelpAHelpAHelpAHelpA>'
```

Les méthodes associées aux chaînes de caractères

Quelques exemples :

```
>>> x = 'CECI EST UN TEXTE EN MASJUSCULE'  
>>> x.lower()  
'ceci est un texte en masjuscule'  
>>> x  
'CECI EST UN TEXTE EN MASJUSCULE'  
>>> 'ceci est un texte en minuscule'.upper()  
'CECI EST UN TEXTE EN MINUSCULE'  
>>>
```

```
>>> ligne = 'Ceci est une ligne'  
>>> ligne.split()  
['Ceci', 'est', 'une', 'ligne']  
>>> ligne = 'Cette:ligne:est:separee:par:des:deux-points'  
>>> ligne.split(':')  
['Cette', 'ligne', 'est', 'separee', 'par', 'des', 'deux-points']
```

Les méthodes associées aux chaînes de caractères

et encore beaucoup d'autres :

```
>>> dir(ligne)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__ge__',
 '__getattr__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__',
 '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
 '__str__', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
 'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper',
 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',
 'title', 'translate', 'upper', 'zfill']
```

```
>>> help(ligne.split)
```

Help on built-in function split:

```
split(...)
```

```
S.split([sep [,maxsplit]]) -> list of strings
```

Return a list of the words in the string S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator.

(END)

Conversion de types

On est souvent amené à convertir les types, i.e. passer d'un type numérique à une chaîne de caractères ou vice-versa.

```
>>> i = 3
>>> str(i)
'3'
>>> i = '456'
>>> int(i)
456
>>> float(i)
456.0
>>> i = '3.1416'
>>> float(i)
3.1415999999999999
>>>
```

Table des matières

1 Introduction

2 Variables

3 Entrées-Sorties

4 Les listes (Première approche)

5 Les structures de contrôle

Les entrées

- L'instruction `raw_input()` effectue toujours une saisie en mode texte que l'on peut ensuite transtyper :

```
f1 = raw_input("Entrez un flottant : ")
f1 = float(f1) # transtypage en flottant
# ou plus brièvement :
f2 = float(raw_input("Entrez un autre flottant : "))
```

- L'instruction `input()` permet de saisir une entrée au clavier. Comme c'est une évaluation, elle effectue un typage dynamique. Elle permet également d'afficher une invite :

```
n = input("Entrez un entier : ")
```

Les sorties

- L'instruction `print` permet d'afficher des sorties à l'écran :

```
a = 2
b = 5
print a # 2
print "Somme :", a + b # 7
print "Différence :", a - b, # -3
print "Produit :", a * b # 10
```

- Le séparateur virgule (,) permet d'empêcher le retour à la ligne.

```
>>> x = 32
>>> nom = 'John'
>>> print nom , ' a ' , x , ' ans'
John a 32 ans
```

Écriture formatée

- Comment convertir des valeurs en chaînes de caractères ?
↪ `repr()` ou écrire juste la valeur entre des guillemets renversés

```
>>> x = 10 * 3.14
>>> y = 200*200
>>> s = 'La valeur de x est ' + 'x' + ', et y est ' + 'y' + '...'
>>> print s
La valeur de x est 31.4, et y est 40000...
>>> p = [x, y]
>>> ps = repr(p)
>>> ps
'[31.4, 40000]'
```

Écriture formatée

- La deuxième manière est d'utiliser l'opérateur % avec une chaîne de caractères comme argument de gauche

```
>>> print 10 ; print 100 ; print 1000
10
100
1000
>>> print "%4i" % 10 ; print "%4i" % 100 ; print "%4i" % 1000
   10
  100
 1000
```

- entier : %i
- flottant : %f (2 chiffres significatifs : %.2f ...)
- chaîne : %s (5 caractères : %5s)

Table des matières

- 1 Introduction
- 2 Variables
- 3 Entrées-Sorties
- 4 Les listes (Première approche)**
- 5 Les structures de contrôle

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 commence à 0 et se termine à $n - 1$

```
>>> animaux = ['girafe', 'hippopotame', 'singé', '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
```

Table des matières

- 1 Introduction
- 2 Variables
- 3 Entrées-Sorties
- 4 Les listes (Première approche)
- 5 Les structures de contrôle

Les instructions composées

- Elles se composent :
 - d'une ligne d'en-tête terminée par deux-points ;
 - d'un **bloc d'instructions** indenté au même niveau.
- S'il y a plusieurs instructions indentées sous la ligne d'en-tête, elles doivent l'être **exactement** au même niveau (comptez un décalage de 4 caractères, par exemple).
- On peut imbriquer des instructions composées pour réaliser des structures de décision complexes.

L'instruction if

Syntaxe

```
if expression1:  
    instruction1  
elif expression2:  
    instruction2  
else:  
    instruction3
```

Description

- La valeur de `expression1` est évaluée et, si elle est `True`, `instruction1` est exécutée.
- si `expression1` est `False`, la valeur de `expression2` est évaluée et, si elle est `True`, `instruction2` est exécutée.
- si `expression1` et `expression2` sont `False` alors `instruction3` est exécutée.
- `instruction1`, `instruction2` et `instruction3` peuvent être des instructions simples ou des blocs.
- La clause `else` peut être omise.

L'instruction if

Exemple

```
if x < 0:
    print "x est négatif"
elif x % 2:
    print "x est positif et impair"
else:
    print "x n'est pas négatif et est pair"

# Test d'une valeur booléenne :
if x: # mieux que (if x is True:) ou que (if x == True:)
```

L'instruction if

Trouver, par exemple, le minimum de deux nombres :

```
x, y = 4, 3
```

```
# Ecriture classique :
```

```
if x < y:
```

```
    plusPetit = x
```

```
else:
```

```
    plusPetit = y
```

```
# Utilisation de l'opérateur ternaire :
```

```
plusPetit = x if x < y else y
```


L'instruction while

Syntaxe

```
while expression:  
    instruction      # ou bloc d'instructions
```

Description

- instruction est exécutée de façon répétitive aussi longtemps que le résultat de expression est True.
- expression est évaluée avant chaque exécution de instruction.

```
cpt = 0  
while x > 0:  
    x = x // 2 # division avec troncature  
    cpt += 1   # incrémentation  
print "L'approximation de log2 de x est", cpt
```

```
n = input('Entrez un entier [1 .. 10] : ')  
while (n < 1) or (n > 10):  
    n = input('Entrez un entier [1 .. 10], S.V.P. : ')
```

L'instruction for

- 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
```