Cours de Python

http://www.dsimb.inserm.fr/~fuchs/python/

Patrick Fuchs et **Pierre Poulain**
prénom [point] nom [arobase] univ-paris-diderot [point] fr

version du 26 août 2014

Université Paris Diderot-Paris 7, Paris, France
Institut Jacques Monod (CNRS UMR 7592) et DSIMB (INSERM UMR_S 665)

Bienvenue au cours de Python !

Ce cours a été conçu à l'origine pour les étudiants débutants en programmation Python des filières biologie et biochimie de l'Université Paris Diderot - Paris 7 ; et plus spécialement pour les étudiants du master Biologie Informatique.

Ce cours est basé sur les versions 2.x de Python et ne prend pas en compte, pour l'instant, les nouvelles versions 3.x. Néanmoins, nous essaierons de le mettre rapidement à jour de manière à traiter la compatibilité entre les deux générations. Si vous relevez des erreurs à la lecture de ce document, merci de nous les signaler.

Le cours est disponible en version HTML et PDF.

Remerciements

Un grand merci à Sander du CMBI de Nijmegen pour la première version de ce cours.

Merci également à tous les contributeurs, occasionels ou réguliers : Jennifer Becq, Virginie Martiny, Romain Laurent, Benoist Laurent, Benjamin Boyer, Hubert Santuz, Catherine Lesourd, Philippe Label, Rémi Cuchillo, Cédric Gageat, Philibert Malbranche, Mikaël Naveau, Amélie Bacle. Nous remercions aussi Denis Mestivier de qui nous nous sommes inspirés pour certains exercices.

Enfin, merci à vous tous, les curieux de Python, qui avez été nombreux à nous envoyer des retours sur ce cours, à nous suggérer des corrections et à nous signaler des coquilles.

De nombreuses personnes nous ont aussi demandé les corrections des exercices. Nous ne les mettons pas sur le site afin d'éviter la tentation de les regarder trop vite, mais vous pouvez nous écrire et nous vous les enverrons.

Table des matières

1 Introduction

1.1 Avant de commencer

Avant de débuter ce cours, voici quelques indications générales qui pourront vous servir pour la suite.

- Familiarisez-vous avec le site www.python.org. Il contient énormément d'informations et de liens sur Python et vous permet en outre de le télécharger pour différentes plateformes (Linux, Mac, Windows). La page d'index des modules est particulièrement utile.
- Pour aller plus loin avec Python, Gérard Swinnen a écrit un très bon livre intitulé *Apprendre à programmer avec Python* et téléchargeable gratuitement. Les éditions Eyrolles proposent également la version papier de cet ouvrage.
- Ce cours est basé sur une utilisation de Python sous Linux mais il est parfaitement transposable aux systèmes d'exploitation Windows et Mac.
- L'apprentissage d'un langage informatique comme Python va nécessiter d'écrire des lignes de codes à l'aide d'un éditeur de texte. Si vous êtes débutants, nous vous conseillons vivement d'utiliser *gedit* ou *nedit*, qui sont les plus proches des éditeurs que l'on peut trouver sous Windows (*notepad*). Des éditeurs comme *emacs* et *vi* sont très puissants mais nécessitent un apprentissage particulier.

1.2 Premier contact avec Python sous Linux

Python est un langage interprété, c'est-à-dire que chaque ligne de code est lue puis interprétée afin d'être exécutée par l'ordinateur. Pour vous en rendre compte, lancez la commande :

```
python
```

Celle-ci va démarrer l'**interpréteur Python**. Vous devriez obtenir quelque chose de ce style :

```
[fuchs@opera ~]$ python
Python 2.5.1 (r251:54863, Jul 10 2008, 17:25:56)
[GCC 4.1.2 20070925 (Red Hat 4.1.2-33)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Le bloc `[fuchs@opera ~]$` représente l'invite de commande de votre *shell* sous Linux. Le triple chevron >>> est l'invite de Python (*prompt* en anglais), ce qui signifie que Python attend une commande. Tapez par exemple l'instruction

```
print "Hello world !"
```

puis validez votre commande avec la touche `Entrée`.

Python a exécuté la commande directement et a affiché le texte `Hello world !`. Il attend ensuite votre prochaine instruction en affichant l'invite de l'interpréteur Python (>>>). En résumé, voici ce qui a du apparaître sur votre écran :

```
>>> print "Hello world !"
Hello world !
>>>
```

Vous pouvez refaire un nouvel essai en vous servant cette fois de l'interpréteur comme d'une machine à calculer.

```
>>> 1 + 1
2
>>>
```

À ce stade, vous pouvez entrer une autre commande ou bien quitter l'interpréteur à l'aide des touches Ctrl-D. Finalement l'interpréteur Python est un système interactif dans lequel vous pouvez entrer des commandes, que Python exécutera sous vos yeux (au moment où vous validerez la commande en tapant sur `Entrée`).

Il existe de nombreux autres langages interprétés tels que Perl ou R. Le gros avantage est que l'on peut directement tester une commande à l'aide de l'interpréteur, ce qui est très utile pour débugger (c'est-à-dire corriger les éventuelles erreurs d'un programme). Gardez bien en mémoire cette propriété de Python qui pourra parfois vous faire gagner un temps précieux !

1.3 Premier programme Python

Bien sûr, l'interpréteur présente vite des limites dès lors que l'on veut exécuter une suite d'instructions plus complexe. Comme tout langage informatique, on peut enregistrer ces instructions dans un fichier, que l'on appelle communément un script Python.

Pour reprendre l'exemple précédent, ouvrez un éditeur de texte (par exemple *gedit* ou *nedit*) et entrez le code suivant.

```
print 'Hello World !'
```

Ensuite enregistrez votre fichier sous le nom *test.py*, puis quittez l'éditeur de texte. L'extension standard des scripts Python est `.py`. Pour exécuter votre script, vous avez deux moyens.

1. Soit donner le nom de votre script comme argument à la commande Python :

   ```
   [fuchs@opera ~]$ python test.py
   Hello World !
   [fuchs@opera ~]$
   ```

2. Soit rendre votre fichier exécutable. Pour cela deux opérations sont nécessaires :
   - Précisez au *shell* la localisation de l'interpréteur Python en indiquant dans la première ligne du script :
     `#!/usr/bin/env python`
     Dans notre exemple, le script *test.py* contient alors le texte suivant :

     ```
     #!/usr/bin/env python

     print 'Hello World !'
     ```
   - Rendez votre script Python exécutable en tapant :
     `chmod +x test.py`

   Pour exécuter votre script, tapez son nom précédé des deux caractères `./` (afin de préciser au *shell* où se trouve votre script) :

   ```
   [fuchs@opera ~]$ ./test.py
   Hello World !
   [fuchs@opera ~]$
   ```

1.4 Commentaires

Dans un script, tout ce qui suit le caractère `#` est ignoré par Python jusqu'à la fin de la ligne et est considéré comme un commentaire. Une exception notable est la première ligne de votre script qui peut être `#!/usr/bin/env python` et qui a alors une signification particulière pour Python.

Les commentaires sont indispensables pour que vous puissiez annoter votre code. Il faut absolument les utiliser pour décrire les principales parties de votre code dans un langage humain.

```
#!/usr/bin/env python

# votre premier script Python
print 'Hello World !'

# d'autres commandes plus utiles pourraient suivre
```

1.5 Séparateur d'instructions

Python utilise l'espace comme séparateur d'instructions. Cela peut sembler évident, mais il est tout de même important de le préciser. Par exemple :

```
>>> print 1
1
>>> print1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'print1' is not defined
```

Omettre l'espace entre l'instruction `print` et le chiffre `1` renvoie une erreur.

1.6 Notion de bloc d'instructions et d'indentation

Pour terminer ce chapitre d'introduction, nous allons aborder dès maintenant les notions de **bloc d'instructions** et d'**indentation**.

En programmation, il est courant de répéter un certain nombre de choses (avec les boucles, voir chapitre 5) ou de faire des choix (avec les tests, voir chapitre 6).

Par exemple, imaginez que vous souhaitiez répéter 10 fois 3 instructions différentes, les unes à la suite des autres. Regardez l'exemple suivant en pseudo-code :

```
instruction_qui_indique_à_Python_de_répéter_10_fois_ce_qui_suit:
    sous-instruction1
    sous-instruction2
    sous-instruction3
instruction_suivante
```

La première ligne correspond à une instruction qui va indiquer à Python de répéter 10 fois d'autres instructions (il s'agit d'une boucle, on verra le nom de la commande exacte plus tard). Dans le pseudo-code ci-dessus, il y a 3 instructions à répéter, nommées `sous-instruction1` à `sous-instruction3`.

Notez bien les détails de la syntaxe. La première ligne indique que l'on veut répéter une ou plusieurs instructions, elle se termine par `:`. Ce symbole `:` indique à Python qu'il doit attendre un bloc d'instructions, c'est-à-dire un certains nombres de sous-instructions à répéter. Python

reconnaît un bloc d'instructions car il est indenté. L'indentation est le décalage d'un ou plusieurs espaces ou tabulations des instructions `sous-instruction1` à `sous-instruction3`, par rapport à
`instruction_qui_indique_à_Python_de_répéter_10_fois_ce_qui_suit:`.

Pratiquement, l'indentation en Python doit être homogène (soit des espaces, soit des tabulations, mais pas un mélange des deux). Une indentation avec 4 espaces est le style d'indentation le plus traditionnel et celui qui permet une bonne lisibilité du code.

Enfin, la ligne `instruction_suivante` sera exécutée une fois que le bloc d'instructions sera terminé.

Si tout cela vous semble un peu fastidieux, ne vous inquiétez pas. Vous allez comprendre tous ces détails, et surtout les acquérir, en continuant ce cours chapitre par chapitre.

2 Variables

Une **variable** est une zone de la mémoire dans laquelle une **valeur** est stockée. Aux yeux du programmeur, cette variable est définie par un **nom**, alors que pour l'ordinateur, il s'agit en fait d'une adresse (*i.e.* une zone particulière de la mémoire).

En Python, la **déclaration** d'une variable et son **initialisation** (c.-à-d. la première valeur que l'on va stocker dedans) se fait en même temps. Pour vous en convaincre, regardez puis testez les instructions suivantes après avoir lancé l'interpréteur :

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

Dans cet exemple, nous avons déclaré, puis initialisé la variable `x` avec la valeur 2. L'interpréteur nous a ensuite permis de regarder le contenu de la variable juste en tapant son nom. Retenez ceci car c'est une **spécificité de l'interpréteur Python**, très pratique pour chasser les erreurs (*debug*) dans un programme. Par contre, la ligne d'un script Python qui contient seulement le nom d'une variable (sans aucune autre fonction) n'affichera pas la valeur de la variable à l'écran.

2.1 Types

Le **type** d'une variable correspond à la nature de celle-ci. Les trois types principaux dont nous aurons besoin sont les entiers, les réels et les chaînes de caractères. Bien sûr, il existe de nombreux autres types (par exemple, les nombres complexes), c'est d'ailleurs un des gros avantages de Python (si vous n'êtes pas effrayés, vous pouvez vous en rendre compte ici).

Dans l'exemple précédent, nous avons stocké un nombre entier (*int*) dans la variable `x`, mais il est tout a fait possible de stocker des nombres réels (*float*) ou des chaînes de caractères (*string*) :

```
>>> y = 3.14
>>> y
3.1400000000000001
>>> a = "bonjour"
>>> a
'bonjour'
>>> b = 'salut'
>>> b
'salut'
>>> c = '''girafe'''
>>> c
'girafe'
```

Vous remarquez que Python reconnaît certains types de variable automatiquement (entier, réel). Par contre, pour une chaîne de caractères, il faut l'entourer de guillemets (simples, doubles voire triples) afin d'indiquer à Python le début et la fin de la chaîne.
Vous vous posez sans doute une question concernant l'exemple du réel. Si vous avez entré le chiffre `3.14`, pourquoi Python l'imprime-t-il finalement avec des zéros derrière (`3.1400000000000001`) ? Un autre exemple :

```
>>> 1.9
1.8999999999999999
>>>
```

Cette fois-ci, Python écrit `1.9` sous la forme `1.8` suivi de plusieurs `9` ! Sans rentrer dans les détails techniques, la manière de coder un réel dans l'ordinateur est rarement exacte. Jusqu'aux versions 2.6 Python ne nous le cache pas et le montre lorsqu'il affiche ce nombre.
À noter : à partir des versions 2.7 et supérieures, cet affichage redevient "caché" par Python :

```
>>> 3.14
3.14
>>>
```

2.2 Nommage

Le nom des variable en Python peut-être constitué de lettres minuscules (a à z), de lettres majuscules (A à Z), de nombres (0 à 9) ou du caractère souligné (_).

Néanmoins, un nom de variable ne doit pas débuter ni par un chiffre, ni par _ et ne peut pas contenir de caractère accentué. Il faut absolument éviter d'utiliser un mot « réservé » par Python comme nom de variable (par ex. : `print`, `range`, `for`, `from`, etc.).

Python est sensible à la casse, ce qui signifie que les variables `TesT`, `test` ou `TEST` sont différentes. Enfin, n'utilisez jamais d'espace dans un nom de variable puisque celui-ci est le sépérateur d'instructions.

2.3 Opérations

Les quatre opérations de base se font de manière simple sur les types numériques (nombres entiers et réels) :

```
>>> x = 45
>>> x + 2
47
>>> y = 2.5
>>> x + y
47.5
>>> (x * 10) / y
180.0
```

Remarquez toutefois que si vous mélangez les types entiers et réels, le résultat est renvoyé comme un réel (car ce type est plus général).

L'opérateur puissance utilise le symbole `**` et pour obtenir le reste d'une division entière, on utilise le symbole modulo `%` :

```
>>> 2**3
8
>>> 3 % 4
3
>>> 8 % 4
0
```

Opérations sur les chaînes de caractères

Pour les chaînes de caractères, deux opérations sont possibles, l'addition et la multiplication :

```
>>> chaine = "Salut"
>>> chaine
'Salut'
```

```
>>> chaine + " Python"
'Salut Python'
>>> chaine * 3
'SalutSalutSalut'
```

L'opérateur d'addition `+` permet de concaténer (assembler) deux chaînes de caractères et l'opérateur de multiplication `*` permet de dupliquer plusieurs fois une chaîne.

Opérations illicites

Attention à ne pas faire d'opération illicite car vous obtiendriez un message d'erreur :

```
>>> 'toto' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Notez que Python vous donne le maximum d'indices dans son message d'erreur. Dans l'exemple précédent, il vous indique que vous ne pouvez pas mélanger des objets de type `str` (*string*, donc des chaînes de caractères) avec des objets de type `int` (donc des entiers), ce qui est assez logique.

Fonction type()

Si vous ne vous souvenez plus du type d'une variable, utilisez la fonction `type` qui vous le rappelle.

```
>>> x = 2
>>> type(x)
<type 'int'>
>>> x = 2.0
>>> type(x)
<type 'float'>
>>> x = '2'
>>> type(x)
<type 'str'>
```

Faites bien attention, pour Python, la valeur `2` (nombre entier) est différente de `2.0` (nombre réel), de même que `2` (nombre entier) est différent de `'2'` (chaîne de caractères).

3 Écriture

Nous avons déjà vu au chapitre précédent la fonction `print` qui permet d'afficher une chaîne de caractères. Elle permet en plus d'afficher le contenu d'une ou plusieurs variables :

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

Python a donc écrit la phrase en remplaçant les variables `x` et `nom` par leur contenu. Vous pouvez noter également que pour écrire plusieurs blocs de texte sur une seule ligne, nous avons utilisé le séparateur `,` avec la commande `print`. En regardant de plus près, vous vous aperçevrez que Python a automatiquement ajouté un espace à chaque fois que l'on utilisait le séparateur `,`. Par conséquent, si vous voulez mettre un seul espace entre chaque bloc, vous pouvez retirer ceux de vos chaînes de caractères :

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

Pour imprimer deux chaînes de caractères l'une à côté de l'autre sans espace, vous devrez les concaténer :

```
>>> ani1 = 'chat'
>>> ani2 = 'souris'
>>> print ani1, ani2
chat souris
>>> print ani1 + ani2
chatsouris
```

3.1 Écriture formatée

Imaginez que vous vouliez calculer puis afficher la proportion de GC d'un génome. Notez que la proportion de GC s'obtient comme la somme des bases Guanine (G) et Cytosine (C) divisée par le nombre total de bases. Sachant que l'on a 4500 bases G, 2575 bases C pour un total de 14800 bases, vous pourriez faire comme suit (notez bien l'utilisation des parenthèses pour gérer les priorités des opérateurs) :

```
>>> propGC = (4500.0 + 2575)/14800
>>> print "La proportion de GC est", propGC
La proportion de GC est 0.478040540541
```

Remarquez que si vous aviez fait le calcul avec `(4500 + 2575)/14800`, vous auriez obtenu 0 car tous les nombres sont des entiers et le résultat aurait été, lui aussi, un entier. L'introduction de `4500.0` qui est un réel résoud le problème, puisque le calcul se fait alors sur des réels.

Néanmoins, le résultat obtenu présente trop de décimales (onze dans le cas présent). Pour pouvoir écrire le résultat plus lisiblement, vous pouvez utiliser l'opérateur de formatage `%`. Dans votre cas, vous voulez formater un réel pour l'afficher avec deux puis trois décimales :

```
>>> print "La proportion de GC est %.2f" % propGC
Le proportion de GC est 0.48
>>> print "La proportion de GC est %.3f" % propGC
La proportion de GC est 0.478
```

Le signe `%` est appelé une première fois (`%.2f`) pour

1. désigner l'endroit où sera placée la variable dans la chaîne de caractères ;
2. préciser le type de la variable à formater, ici `f` pour *float* donc pour un réel ;
3. préciser le formatage voulu, ici `.2` soit deux chiffres après la virgule.

Le signe `%` est rappelé une seconde fois (`% propGC`) pour indiquer les variables à formater. Notez que les réels ainsi formatés sont arrondis et non tronqués.

Vous pouvez aussi formatez des entiers avec `%i` (`i` comme *integer*)

```
>>> nbG = 4500
>>> print "Le génome de cet exemple contient %i guanines" % nbG
Le génome de cet exemple contient 4500 guanines
```

ou mettre plusieurs nombres dans une même chaîne de caractères.

```
>>> nbG = 4500
>>> nbC = 2575
>>> print "Ce génome contient %i G et %i C, soit une prop de GC de %.2f" \
... % (nbG,nbC,propGC)
Ce génome contient 4500 G et 2575 C, soit une prop de GC de 0.48
```

Remarque :

1. Dans l'exemple précédent, nous avons utilisé le symbole `%` pour formater des variables. Si vous avez besoin d'écrire le symbole pourcentage `%` sans qu'il soit confondu avec celui servant pour l'écriture formatée, il suffit de le doubler. Toutefois, si l'écriture formatée n'est pas utilisée dans la même chaîne de caractères où vous voulez utilisez le symbole pourcent, cette opération n'est pas nécessaire. Par exemple :

   ```
   >>> nbG = 4500
   >>> nbC = 2575
   >>> percGC = propGC * 100
   >>> print "Ce génome contient %i G et %i C, soit un %%GC de %.2f" \
   ... % (nbG,nbC,percGC) ,"%"
   Ce génome contient 4500 G et 2575 C, soit un %GC de 47.80 %
   ```

2. Le signe `\` en fin de ligne permet de poursuivre la commande sur la ligne suivante. Cette syntaxe est pratique lorsque vous voulez taper une commande longue.

Enfin, il est possible de préciser sur combien de caractères vous voulez qu'un résultat soit écrit. Dans la portion de code suivant, le caractère `;` sert de séparateur entre les instructions sur une même ligne :

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

Ceci est également possible sur les chaînes de caractères (avec `%s`, `s` comme *string*) :

```
>>> print "atom HN" ; print "atom HDE1"
atom HN
atom HDE1
>>> print "atom %4s" % "HN" ; print "atom %4s" % "HDE1"
atom   HN
atom HDE1
```

Vous voyez tout de suite l'énorme avantage de l'écriture formatée. Cela permet d'écrire en colonnes parfaitement alignées. Nous verrons que ceci est très pratique si l'on veut écrire les coordonnées des atomes d'une molécule au format PDB.

Alternative :

Il existe une autre façon d'écrire du texte formaté : la méthode `format()`.

```
>>> x = 32
>>> nom = 'John'
>>> print "{0} a {1} ans".format(nom,x)
John  a  32  ans

>>> nbG = 4500
>>> nbC = 2575
>>> propGC = (4500.0 + 2575)/14800
>>> print "On a {0} G et {1} C -> prop GC = {2:.2f}".format(nbG,nbC,propGC)
On a 4500 G et 2575 C -> prop GC = 0.48
```

La syntaxe est légèrement différente :

1. `.format(nom,x)` est utilisé pour indiquer les variables (`nom` et `x`) à afficher ;
2. `{0}` et `{1}` désigne l'endroit où seront placées les variables. Les chiffres indiquent la position de la variable dans `format` (0=nom et 1=x) ;
3. Pour le 2ème exemple, on précise la position de la variable (2) et le formatage voulu, ici `.2f` soit deux chiffres après la virgule pour un réel. Notez qu'un `:` est ajouté pour signifier l'ajout de règles de formatage.

Cette méthode vous est présentée à titre d'information car elle deviendra standard dans la version Python 3.

3.2 Exercices

Conseil : utilisez l'interpréteur Python pour les exercices 2 à 5.

1. Ouvrez l'interpréteur Python et tapez `1+1`. Que se passe-t-il ? Écrivez la même chose dans un script `test.py`. Exécutez ce script en tapant `python test.py` dans un *shell*. Que se passe-t-il ? Pourquoi ?
2. Calculez le pourcentage de GC avec le code

   ```
   percGC = ((4500 + 2575)/14800)*100
   ```

   puis affichez le résultat. Que se passe-t-il ? Comment expliquez-vous ce résultat ? Corrigez l'instruction précédente pour calculer correctement le pourcentage de GC. Affichez cette valeur avec deux décimales de précision.
3. Générez une chaîne de caractères représentant un oligonucléotide polyA (AAAA...) de 20 bases de longueurs, sans taper littéralement toutes les bases.
4. Suivant le modèle du dessus, générez en une ligne de code un polyA de 20 bases suivi d'un polyGC régulier (GCGCGC...) de 40 bases.
5. En utilisant l'écriture formatée, affichez en une seule ligne les variables `a`, `b` et `c` dont les valeurs sont respectivement `"salut"`, `102` et `10.318`.