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 14 septembre 2015

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 la version 2.7 de Python, version que l'on trouve par défaut dans les distributions Linux actuelles (comme Ubuntu 14.04 ou Fedora 22). Le cours sera mis à jour vers la version 3.x de Python dans les prochains mois (dès que python 3 sera le standard dans les distributions Linux). Nous avons ajouté un paragraphe dans l'introduction traitant des principales différences entre python 2 et python 3 .

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 (ici la même page pour python 3.*).
- 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.

1.7 Python 2 ou Python 3 ?

Important : pour les débutants, nous vous conseillons de sauter la lecture de ce paragraphe (retenez seulement que ce cours est en python 2.7, donc installez puis lancez python 2.7 pour les futurs exercices).

Vous êtes nombreux à nous demander la mise à jour du cours vers Python 3. Nous avons décidé d'effectuer cette mise à jour au moment où python 3 deviendra standard dans les distributions Linux. En effet pour l'instant sur mon Ubuntu 12.04, lorsque je lance l'interpréteur c'est bien python 2.7 qui est lancé (il en est de même pour la dernière Ubuntu 14.04) :

```
[fuchs@rome cours_python]$ python
Python 2.7.3 (default, Jun 22 2015, 19:33:41)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Toutefois, dans les futures versions d'Ubuntu (e.g. la 15.04), python 3 deviendra le standard. Alors quelles sont les différences majeures entre python 2 et python 3 ? Une documentation exhaustive est disponible ici, mais pour ce qui nous intéresse dans le cadre de ce cours, quatre différences méritent d'être citées :

1. la fonction `print` : en python 3, la fonction print utilise des parenthèses comme tout autre fonction python, par exemple : `print("Bonjour")`. Par conséquent, la syntaxe `print "Bonjour"` renverra une erreur.
2. le formattage de chaine de caractères : bien que cela existe depuis python 2, la méthode `format()` - plus élégante - est vivement conseillée pour formatter les chaines de caractères :

   ```
   >>> print("{0:7.3f}".format(2.0/3.0))
     0.667
   >>>
   ```

   L'ancienne syntaxe `"%7.3f" % (2.0/3.0)` reste toutefois encore fonctionnelle.
3. la division d'entier : en python 3, une division d'entier comme `3/5` renverra un `float` :

   ```
   >>> 3/5
   0.6
   >>>
   ```

   On pourra noter que cela était déjà possible en python 2, en utilisant le module `__future__` :

```
>>> 2/4
0
>>> from __future__ import division
>>> 2/4
0.5
>>>
```

4. la fonction `range` : en python 3, `range()` se comporte comme `xrange()` en python 2, c'est à dire qu'il s'agit d'un itérateur (plus efficace en mémoire) plutôt qu'un simple générateur de liste. Si on veut générer une liste, il suffit d'utiliser la fonction `list()` :

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

Vous avez maintenant quelques pointeurs vers les nouvelles fonctionnalités de python 3 avant que nous mettions à jour le cours, nous vous souhaitons un bon codage en Python !

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`.

4 Listes

4.1 Définition

Une **liste** est une structure de données qui contient une série de valeurs. Python autorise la construction de liste contenant des valeurs de type différent (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité. Une liste est déclarée par une série de valeurs (ne pas oublier les guillemets, simples ou doubles, s'il s'agit de chaînes de caractères) séparées par des **virgules**, et le tout encadré par des **crochets**. En voici quelques exemples :

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> tailles = [5, 2.5, 1.75, 0.15]
>>> mixte = ['girafe', 5, 'souris', 0.15]
>>> animaux
['girafe', 'tigre', 'singe', 'souris']
>>> tailles
[5, 2.5, 1.75, 0.15]
>>> mixte
['girafe', 5, 'souris', 0.15]
```

Lorsque l'on affiche une liste, Python la restitue telle qu'elle a été saisie.

4.2 Utilisation

Un des gros avantages d'une liste est que vous pouvez appeler ses éléments par leur position. Ce numéro est appelé **indice** (ou index) de la liste.

```
liste  : ['girafe', 'tigre', 'singe', 'souris']
indice :        0        1        2         3
```

Soyez très **attentifs** au fait que les indices d'une liste de *n* éléments commence à *0* et se termine à *n-1*. Voyez l'exemple suivant :

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> animaux[0]
'girafe'
>>> animaux[1]
'tigre'
>>> animaux[3]
'souris'
```

Par conséquent, si on appelle l'élément d'indice 4 de notre liste, Python renverra un message d'erreur :

```
>>> animaux[4]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

N'oubliez pas ceci ou vous risqueriez d'obtenir des bugs inattendus !

4.3 Opération sur les listes

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

```
>>> ani1 = ['girafe','tigre']
>>> ani2 = ['singe','souris']
>>> ani1 + ani2
['girafe', 'tigre', 'singe', 'souris']
>>> ani1 * 3
['girafe', 'tigre', 'girafe', 'tigre', 'girafe', 'tigre']
```

4.4 Indiçage négatif et tranches

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

```
liste          : ['girafe', 'tigre', 'singe', 'souris']
indice positif :        0        1        2         3
indice négatif :       -4       -3       -2        -1
```

ou encore :

```
liste          : ['A','C','D','E','F','G','H','I','K','L']
indice positif :   0   1   2   3   4   5   6   7   8   9
indice negatif : -10  -9  -8  -7  -6  -5  -4  -3 - 2  -1
```

Les indices négatifs reviennent à compter à partir de la fin. Leur principal avantage est que vous pouvez appeler le dernier élément d'une liste à l'aide de l'indice -1 sans pour autant connaître la longueur de la liste.

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> animaux[-4]
'girafe'
>>> animaux[-2]
'singe'
>>> animaux[-1]
'souris'
```

Un autre avantage des listes est la possibilité de sélectionner une partie en utilisant un indiçage construit sur le modèle `[m:n+1]` pour récupérer tous les éléments, du émième au énième (de l'élément `m` inclus à l'élément `n+1` exclus). On dit alors qu'on récupère une **tranche** de la liste, par exemple :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> animaux[0:2]
['girafe', 'tigre']
>>> animaux[0:3]
['girafe', 'tigre', 'singe']
>>> animaux[0:]
['girafe', 'tigre', 'singe', 'souris']
>>> animaux[:]
['girafe', 'tigre', 'singe', 'souris']
>>> animaux[1:]
['tigre', 'singe', 'souris']
>>> animaux[1:-1]
['tigre', 'singe']
```

Remarquez que lorsqu'aucun indice n'est indiqué à gauche ou à droite du symbole `:`, Python prend par défaut tous les éléments depuis le début ou tous les éléments jusqu'à la fin respectivement.

Dans les versions récentes de Python, on peut aussi préciser le pas en ajoutant un : supplémentaire et en indiquant le pas par un entier.

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> animaux[0:3:2]
['girafe', 'singe']
>>> x = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::1]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x[::2]
[0, 2, 4, 6, 8]
>>> x[::3]
[0, 3, 6, 9]
>>> x[1:6:3]
[1, 4]
```

Finalement, on voit que l'accès au contenu d'une liste avec des crochets fonctionne sur le modèle `liste[début:fin:pas]`.

4.5 Fonctions range et len

L'instruction `range()` vous permet de créer des **listes d'entiers** (et d'entiers uniquement) de manière simple et rapide. Voyez plutôt :

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

L'instruction `range()` fonctionne sur le modèle `range([début,] fin[, pas])`. Les arguments entre crochets sont optionnels.

L'instruction `len()` vous permet de connaître la longueur d'une liste, ce qui parfois est bien pratique lorsqu'on lit un fichier par exemple, et que l'on ne connaît pas *a priori* la longueur des lignes. Voici un exemple d'utilisation :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> len(animaux)
4
>>> len(range(10))
10
```

4.6 Listes de listes

Pour finir, sachez qu'il est tout-à-fait possible de construire des listes de listes. Cette fonctionnalité peut être parfois très pratique. Par exemple :

```
>>> enclos1 = ['girafe', 4]
>>> enclos2 = ['tigre', 2]
>>> enclos3 = ['singe', 5]
>>> zoo = [enclos1, enclos2, enclos3]
>>> zoo
[['girafe', 4], ['tigre', 2], ['singe', 5]]
```

Dans cet exemple, chaque sous-liste contient une catégorie d'animal et le nombre d'animaux pour chaque catégorie. Pour accéder à un élément de la sous-liste, on utilise un double indiçage.

```
>>> zoo[1]
['tigre', 2]
>>> zoo[1][0]
'tigre'
>>> zoo[1][1]
2
```

On verra un peu plus loin qu'il existe en Python les dictionnaires qui sont très pratiques pour faire ce genre de choses. On verra aussi qu'il existe un module nommé `numpy` permettant de gérer des listes ou tableaux de nombres (vecteurs et matrices), ainsi que de faire des opérations dessus.

4.7 Exercices

Conseil : utilisez l'interpréteur Python.

1. Constituez une liste `semaine` contenant les 7 jours de la semaine. À partir de cette liste, comment récupérez-vous seulement les 5 premiers jours de la semaine d'une part, et ceux du week-end d'autre part (*utilisez pour cela l'indiçage*) ? Cherchez un autre moyen pour arriver au même résultat (*en utilisant un autre indiçage*).
2. Trouvez deux manières pour accéder au dernier jour de la semaine.
3. Inversez les jours de la semaine en une commande.
4. Créez 4 listes `hiver`, `printemps`, `ete` et `automne` contenant les mois correspondant à ces saisons. Créez ensuite une liste `saisons` contenant les sous-listes `hiver`, `printemps`,`ete` et `automne`. Prévoyez ce que valent les variables suivantes, puis vérifiez-le dans l'interpréteur :
   - `saisons[2]`
   - `saisons[1][0]`
   - `saisons[1:2]`
   - `saisons[:][1]`

   Comment expliquez-vous ce dernier résultat ?
5. Affichez la table des 9 en une seule commande avec l'instruction `range()`.
6. Avec Python, répondez à la question suivante en une seule commande. Combien y a-t-il de nombres pairs dans l'intervalle [2 , 10000] inclus ?

5 Boucles et comparaisons

5.1 Boucles for

Imaginez que vous ayez une liste de quatre éléments dont vous voulez afficher les éléments les uns après les autres. Dans l'état actuel de vos connaissances, il faudrait taper quelque chose du style :

```
animaux = ['girafe','tigre','singe','souris']
print animaux[0]
print animaux[1]
print animaux[2]
print animaux[3]
```

Si votre liste ne contient que quatre éléments, ceci est encore faisable mais imaginez qu'elle en contienne 100 voire 1000 ! Pour remédier à cela, il faut utiliser les boucles. Regardez l'exemple suivant :

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> for animal in animaux:
...     print animal
...
girafe
tigre
singe
souris
```

En fait, la variable `animal` va prendre successivement les différentes valeurs de la liste `animaux` à chaque itération de la boucle. Notez bien les types de variable : `animaux` est une liste sur laquelle on itère, et `animal` est une chaîne de caractère qui à chaque itération de la boucle prendra les valeurs successives de la liste `animaux` (car les éléments de la liste sont des chaînes de caractère, mais on verra plus loin que cela est valable pour n'importe quel type de variable).

D'ores et déjà, remarquez avec attention l'**indentation**. Vous voyez que l'instruction `print animal` est décalée par rapport à l'instruction `for animal in animaux:`. Outre une meilleure lisibilité, ceci est formellement requis en Python. Toutes les instructions que l'on veut répéter constituent le **corps de la boucle** (ou un bloc d'instructions) et **doivent être indentées** d'un(e) ou plusieurs espace(s) ou tabulation(s). Dans le cas contraire, Python vous renvoie un message d'erreur :

```
>>> for animal in animaux:
... print animal
  File "<stdin>", line 2
    print animal
        ^
IndentationError: expected an indented block
```

Notez également que si le corps de votre boucle contient plusieurs instructions, celles-ci doivent être indentées de la même manière (e.g. si vous avez indenté la première instruction avec deux espaces, vous devez faire de même avec la deuxième instruction, etc).

Remarquez également un autre aspect de la syntaxe, une instruction `for` doit **absolument** se terminer par le signe deux-points **:** .

Il est aussi possible d'utiliser une tranche d'une liste :

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> for animal in animaux[1:3]:
...     print animal
...
tigre
singe
```

On a vu que les boucles `for` pouvaient utiliser une liste contenant des chaînes de caractères, mais elles peuvent tout aussi bien utiliser des listes numériques. En utilisant l'instruction `range` on peut facilement accéder à une liste d'entiers.

```
>>> for i in range(4):
...     print i
...
0
1
2
3
```

Notez ici que nous avons choisi le nom `i` pour la variable d'itération. Ceci est un standard en informatique et indique en général qu'il s'agit d'un entier (le nom `i` vient sans doute du mot indice). Nous vous conseillons de toujours suivre cette convention afin d'éviter les confusions, si vous itérez sur les indices vous pouvez appeler la variable `i` (par exemple dans `for i in range(4):`), si vous itérez sur une liste comportant des chaînes de caractères, mettez un nom explicite pour la variable d'itération (par exemple `for prenom in ['Joe','Bill']:`).

Revenons à notre liste `animaux`. Nous allons maintenant parcourir cette liste, mais cette fois par une itération sur ses indices :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> for i in range(4):
...     print animaux[i]
...
girafe
tigre
singe
souris
```

La variable `i` prendra les valeurs successives 0, 1, 2 et 3 et on accèdera à chaque élément de la liste `animaux` par son indice (i.e. `animaux[i]`)

Sur ces différentes méthodes de parcours de liste, quelle est la la plus efficace ? **La méthode à utiliser** pour parcourir avec une boucle `for` les éléments d'une liste est celle qui réalise **les itérations directement sur les élements**.

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> for animal in animaux:
...     print animal
...
girafe
tigre
singe
souris
```

Si vous avez besoin de parcourir votre liste par ses indices, préférez la fonction `xrange()` qui s'utilise de la même manière que `range()` mais qui ne construit pas de liste et qui est donc

beaucoup plus rapide. La fonction `range()` ne sera donc employée que pour créer des listes d'entiers.

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> for i in xrange(4):
...     print animaux[i]
...
girafe
tigre
singe
souris
```

Enfin, si vous avez besoin de l'indice d'un élément d'une liste **et** de l'élément lui-même, la fonction `enumerate()` est pratique.

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> for i, ani in enumerate(animaux):
...     print i, ani
...
0 girafe
1 tigre
2 singe
3 souris
```

5.2 Comparaisons

Avant de passer à une autre sorte de boucles (les boucles `while`), nous abordons tout de suite les **comparaisons**. Celles-ci seront reprises dans le chapitre des tests. Python est capable d'effectuer toute une série de comparaisons entre le contenu de différentes variables, telles que :

syntaxe Python	signification
==	égal à
!=	différent de
>	supérieur à
>=	supérieur ou égal à
<	inférieur à
<=	inférieur ou égal à

Observez l'exemple suivant sur des nombres entiers.

```
>>> x = 5
>>> x == 5
True
>>> x > 10
False
>>> x < 10
True
```

Python renvoie la valeur `True` si la comparaison est vraie et `False` si elle est fausse. `True` et `False` sont des booléens.

Faites bien attention à ne pas confondre l'**opérateur d'affectation** = qui donne une valeur à une variable et l'**opérateur de comparaison** == qui compare les valeurs de deux variables.

Vous pouvez également effectuer des comparaisons sur des chaînes de caractères.

```
>>> animal = "tigre"
>>> animal == "tig"
False
>>> animal != "tig"
True
>>> animal == 'tigre'
True
```

Dans le cas des chaînes de caractères, *a priori* seuls les tests == et != ont un sens. En fait, on peut aussi utiliser les opérateurs <, >, <= et >=. Dans ce cas l'ordre alphabétique est pris en compte, par exemple :

```
>>> "a" < "b"
True
```

`"a"` est *inférieur à* `"b"` car il est situé avant dans l'ordre alphabétique. En fait, c'est l'ordre ASCII des caractères qui est pris en compte (*i.e.* chaque caractère est affecté à un code numérique), on peut donc comparer aussi des caractères spéciaux (comme # ou ~) entre eux. On peut aussi comparer des chaînes à plus d'un caractère.

```
>>> "ali" < "alo"
True
>>> "abb" < "ada"
True
```

Dans ce cas, Python compare caractère par caractère de la gauche vers la droite (le premier avec le premier, le deuxième avec le deuxième, etc). Dès qu'un caractère est différent entre l'une et l'autre des chaînes, il considère que la chaîne « la plus petite » est celle qui présente le caractère ayant le plus petit code ASCII (les caractères suivants de la chaîne sont ignorés dans la comparaison), comme dans notre test `"abb" < "ada"` ci-dessus.

5.3 Boucles while

Une autre alternative à l'instruction `for` couramment utilisée en informatique est la boucle `while`. Le principe est simple. Une série d'instructions est exécutée tant qu'une condition est vraie. Par exemple :

```
>>> i = 1
>>> while i <= 4:
...     print i
...     i = i + 1
...
1
2
3
4
```

Remarquez qu'il est encore une fois nécessaire d'indenter le bloc d'instructions. Une boucle `while` nécessite **trois éléments** pour fonctionner correctement :

1. l'initialisation de la variable de test avant la boucle ;
2. le test de la variable associé à l'instruction `while` ;
3. l'incrémentation de la variable de test dans le corps de la boucle.

Faites bien attention aux tests et à l'incrémentation que vous utilisez car une erreur mène souvent à des boucles infinies qui ne s'arrêtent pas. Vous pouvez néanmoins toujours stopper l'exécution d'un script Python à l'aide de la combinaison de touches Ctrl-C.

5.4 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

1. Soit la liste `['vache','souris','levure','bacterie']`. Affichez l'ensemble des éléments de cette liste (un élément par ligne) de trois manières différentes (deux avec `for` et une avec `while`).
2. Constituez une liste `semaine` contenant les 7 jours de la semaine. Écrivez une série d'instructions affichant les jours de la semaine (en utiliser une boucle `for`), ainsi qu'une autre série d'instructions affichant les jours du week-end (en utilisant une boucle `while`).
3. Avec une boucle, affichez les nombres de 1 à 10 sur une seule ligne.
4. Soit `impairs` la liste de nombres `[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]`. Écrivez un programme qui, à partir de la liste `impairs`, construit une liste `pairs` dans laquelle tous les éléments de `impairs` sont incrémentés de 1.
5. Voici les notes d'un étudiant `[14, 9, 6, 8, 12]`. Calculez la moyenne de ces notes. Utilisez l'écriture formatée pour afficher la valeur de la moyenne avec deux décimales.
6. Soit la liste `X` contenant les nombres entiers de 0 à 10. Calculez le produit des nombres consécutifs deux à deux de `X` en utilisant une boucle. Exemple pour les premières itérations :

```
0
2
6
12
```

7. **Triangle.** Écrivez un script qui dessine un triangle comme celui-ci :

```
*
**
***
****
*****
******
*******
********
*********
**********
```

8. **Triangle inversé.** Écrivez un script qui dessine un triangle comme celui-ci :

```
**********
*********
********
*******
******
*****
****
***
**
*
```

9. **Triangle gauche.** Écrivez un script qui dessine un triangle comme celui-ci :

```
          *
         **
        ***
       ****
      *****
     ******
    *******
   ********
  *********
 **********
```

10. **Triangle isocèle.** Écrivez un script qui dessine un triangle comme celui-ci :

```
          *
         ***
        *****
       *******
      *********
     ***********
    *************
   ***************
  *****************
 *******************
```

11. Exercice +++. **Suite de Fibonacci**. La suite de Fibonacci est une suite mathématique qui porte le nom de Leonardo Fibonacci, un mathématicien italien du XIII^{e} siècle. Initialement, cette suite a été utilisée pour décrire la croissance d'une population de lapins mais elle est peut également être utilisée pour décrire certains motifs géométriques retrouvés dans la nature (coquillages, fleurs de tournesol...).

    Par définition, les deux premiers termes de la suite de Fibonacci sont 0 et 1. Ensuite, le terme au rang *n* est la somme des nombres aux rangs *n - 1* et *n - 2*. Par exemple, les 10 premiers termes de la suite de Fibonacci sont 0, 1, 1, 2, 3, 5, 8, 13, 21, 34.

    Écrivez un script qui construit la liste des 20 premiers termes de la suite de Fibonacci puis l'affiche.

6 Tests

6.1 Définition

Les **tests** sont un élément essentiel à tout langage informatique si on veut lui donner un peu de complexité car ils permettent à l'ordinateur de prendre des décisions si telle ou telle condition est vraie ou fausse. Pour cela, Python utilise l'instruction `if` ainsi qu'une comparaison que nous avons abordée au chapitre précédent. En voici un exemple :

```
>>> x = 2
>>> if x == 2:
...     print "Le test est vrai !"
...
Le test est vrai !
>>> x = "souris"
>>> if x == "tigre":
...     print "Le test est vrai !"
...
```

Plusieurs remarques concernant ces deux exemples :
- Dans le premier exemple, le test étant vrai, l'instruction `print "Le test est vrai !"` est exécutée. Dans le second exemple, le test est faux et rien n'est affiché.
- Les blocs d'instruction dans les tests doivent forcément être indentés comme les boucles `for` et `while`. L'indentation indique la portée des instructions à exécuter si le test est vrai.
- L'instruction `if` se termine comme les instructions `for` et `while` par le caractère **`:`**.

6.2 Tests à plusieurs cas

Parfois, il est pratique de tester si la condition est vraie ou si elle est fausse dans une même instruction `if`. Plutôt que d'utiliser deux instructions `if`, on peut se servir de `if` et de `else` :

```
>>> x = 2
>>> if x == 2:
...     print "Le test est vrai !"
... else:
...     print "Le test est faux !"
...
Le test est vrai !
>>> x = 3
>>> if x == 2:
...     print "Le test est vrai !"
... else:
...     print "Le test est faux !"
...
Le test est faux !
```

On peut utiliser une série de tests dans la même instruction `if`, notamment pour tester plusieurs valeurs d'une même variable. Par exemple, on se propose de tirer au sort une base d'ADN puis d'afficher le nom de cette dernière. Dans le code suivant, nous utilisons l'intruction `random.choice(liste)` qui renvoie un élément choisi au hasard dans une `liste`. L'instruction `import random` sera vue plus tard, admettez pour le moment qu'elle est nécessaire.

```
>>> import random
```

```
>>> base = random.choice(["a", "t", "c", "g"])
>>> if base == "a":
...     print "choix d'une adénine"
... elif base == "t":
...     print "choix d'une thymine"
... elif base == "c":
...     print "choix d'une cytosine"
... elif base == "g":
...     print "choix d'une guanine"
...
choix d'une cytosine
```

Dans cet exemple, Python teste la première condition, puis, si et seulement si elle est fausse, teste la deuxième et ainsi de suite... Le code correspondant à la première condition vérifiée est exécuté puis Python sort du `if`.

Remarque De nouveau, faites bien attention à l'indentation dans ces deux derniers exemples ! Vous devez être très rigoureux à ce niveau là. Pour vous en convaincre, exécutez ces deux scripts dans l'interpréteur Python :

Script 1 :

```
nombres = [4, 5, 6]
for nb in nombres:
    if nb == 5:
        print "Le test est vrai"
        print "car la variable nb vaut", nb
```

Script 2 :

```
nombres = [4, 5, 6]
for nb in nombres:
    if nb == 5:
        print "Le test est vrai"
    print "car la variable nb vaut", nb
```

Comment expliquez-vous ce résultat ?

6.3 Tests multiples

Les tests multiples permettent de tester plusieurs conditions en même temps en utilisant des opérateurs booléens. Les deux opérateurs les plus couramment utilisés sont le **OU** et le **ET**. Voici un petit rappel du mode de fonctionnement de ces opérateurs :

Condition 1	Opérateur	Condition 2	Résultat
Vrai	OU	Vrai	Vrai
Vrai	OU	Faux	Vrai
Faux	OU	Vrai	Vrai
Faux	OU	Faux	Faux
Vrai	ET	Vrai	Vrai
Vrai	ET	Faux	Faux
Faux	ET	Vrai	Faux
Faux	ET	Faux	Faux

En Python, on utilise le mot réservé `and` pour l'opérateur **ET** et le mot réservé `or` pour l'opérateur **OU**. Respectez bien la casse, `and` et `or` s'écrivent en minuscule. En voici un exemple d'utilisation :

```
>>> x = 2
>>> y = 2
>>> if x == 2 and y == 2:
...     print "le test est vrai"
...
le test est vrai
```

Notez que le même résulat serait obtenu en utilisant deux instructions `if` imbriquées :

```
>>> x = 2
>>> y = 2
>>> if x == 2:
...     if y == 2:
...         print "le test est vrai"
...
le test est vrai
```

Vous pouvez aussi tester directement l'effet de ces opérateurs à l'aide de `True` et `False` (attention à respecter la casse).

```
>>> True or False
True
```

Enfin, on peut utiliser l'opérateur logique de négation `not` qui inverse le résultat d'une condition :

```
>>> not True
False
>>> not False
True
>>> not (True and True)
False
```

6.4 Instructions break et continue

Ces deux instructions permet de modifier le comportement d'une boucle (`for` ou `while`) avec un test.

L'instruction `break` stoppe la boucle.

```
>>> for i in range(5):
...     if i > 2:
...         break
...     print i
...
0
1
2
```

L'instruction `continue` saute à l'itération suivante.

```
>>> for i in range(5):
...     if i == 2:
...         continue
...     print i
```

```
...
0
1
3
4
```

6.5 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

1. Constituez une liste `semaine` contenant les sept jours de la semaine. En utilisant une boucle, écrivez chaque jour de la semaine ainsi que les messages suivants :
   – `Au travail` s'il s'agit du lundi au jeudi
   – `Chouette c'est vendredi` s'il s'agit du vendredi
   – `Repos ce week-end` s'il s'agit du week-end.
   (Les messages ne sont que des suggestions, vous pouvez laisser aller votre imagination.)
2. La liste ci-dessous représente une séquence d'ADN :

   ```
   ["A","C","G","T","T","A","G","C","T","A","A","C","G"]
   ```

   Écrivez un script qui transforme cette séquence en sa séquence complémentaire. Rappel : la séquence complémentaire s'obtient en remplaçant A par T, T par A, C par G et G par C.
3. La fonction `min()` de Python, renvoie l'élément le plus petit d'une liste. Sans utiliser cette fonction, écrivez un script qui détermine le plus petit élément de la liste `[8, 4, 6, 1, 5]`.
4. La liste ci-dessous représente une séquence d'acides aminés

   ```
   ["A","R","A","W","W","A","W","A","R","W","W","R","A","G","A","R"]
   ```

   Calculez la fréquence en alanine (A), arginine (R), tryptophane (W) et glycine (G) dans cette séquence.
5. Voici les notes d'un étudiant `[14, 9, 13, 15, 12]`. Écrivez un script qui affiche la note maximum (fonction `max()`), la note minimum (fonction `min()`) et qui calcule la moyenne. Affichez la valeur de la moyenne avec deux décimales. Affichez aussi la mention obtenue sachant que la mention est passable si la moyenne est entre 10 inclus et 12 exclus, assez-bien entre 12 inclus et 14 exclus et bien au-delà de 14.
6. Faites une boucle qui parcourt les nombres de 0 à 20 et qui affiche les nombres pairs inférieurs ou égaux à 10 d'une part, et les nombres impairs strictement supérieur à 10 d'autre part. Pour cet exercice, vous pourrez utiliser l'opérateur modulo `%` qui retourne le reste de la division entière entre deux nombres.
7. Exercice +++. **Conjecture de Syracuse.**
   La conjecture de Syracuse est une conjecture mathématique qui reste improuvée à ce jour et qui est définie de la manière suivante.
   Soit un entier positif n. Si n est pair, alors le diviser par 2. Si il est impair, alors le multiplier par 3 et lui ajouter 1. En répétant cette procédure, la suite de nombres atteint la valeur 1 puis se prolonge indéfiniment par une suite de trois valeurs triviales appelée cycle trivial. Jusqu'à présent, la conjecture de Syracuse, selon laquelle depuis n'importe quel entier positif la suite de Syracuse atteint 1, n'a pas été mise en défaut.
   Par exemple, les premiers éléments de la suite de Syracuse pour un entier de départ de 10 sont : 10, 5, 16, 8, 4, 2, 1...
   Écrivez un script qui, partant d'un entier positif n, crée une liste des nombres de la suite de Syracuse. Avec différents points de départ (n), la conjecture de Syracuse est-elle toujours vérifiée ? Quels sont les nombres qui constituent le cycle trivial ?

Remarque 1 : pour cet exercice, vous avez besoin de faire un nombre d'itérations inconnu pour que la suite de Syracuse atteigne 1 puis entame son cycle trivial. Vous pourrez tester votre algorithme avec un nombre arbitraire d'itérations, typiquement 20 ou 100, suivant votre nombre *n* de départ.

Remarque 2 : un nombre est pair lorsque le reste de sa division entière (opérateur modulo `%`) par 2 est nul.

8. Exercice +++. **Attribution simple de structure secondaire.**

   Les angles dièdres phi/psi d'une hélice alpha parfaite ont une valeur de -57 degrés et -47 degrés respectivement. Bien sûr, il est très rare que l'on trouve ces valeurs parfaites dans une protéine, par conséquent il est couramment accepté de tolérer une déviation de +/- 30 degrés sur celles-ci. Ci-dessous vous avez une liste de listes contenant les valeurs de phi/psi de la première hélice de la protéine 1TFE. À l'aide de cette liste, écrivez un programme qui teste, pour chacun des résidus, s'ils sont ou non en hélice.

```
[[48.6,53.4],[-124.9,156.7],[-66.2,-30.8],[-58.8,-43.1], \
[-73.9,-40.6],[-53.7,-37.5],[-80.6,-16.0],[-68.5,135.0], \
[-64.9,-23.5],[-66.9,-45.5],[-69.6,-41.0],[-62.7,-37.5], \
[-68.2,-38.3],[-61.2,-49.1],[-59.7,-41.1],[-63.2,-48.5], \
[-65.5,-38.5],[-64.1,-40.7],[-63.6,-40.8],[-66.4,-44.5], \
[-56.0,-52.5],[-55.4,-44.6],[-58.6,-44.0],[-77.5,-39.1], \
[-91.7,-11.9],[48.6,53.4]]
```

9. Exercice +++. **Détermination des nombres premiers inférieurs à 100.**

   Voici un extrait de l'article sur les nombres premiers tiré de l'encyclopédie en ligne wikipédia.

   « *Un nombre premier est un entier naturel qui admet exactement deux diviseurs distincts entiers et positifs (qui sont alors 1 et lui-même). Cette définition exclut 1, qui n'a qu'un seul diviseur entier positif. Par opposition, un nombre non nul produit de deux nombres entiers différents de 1 est dit composé. Par exemple 6 = 2 × 3 est composé, tout comme 21 = 3 × 7, mais 11 est premier car 1 et 11 sont les seuls diviseurs de 11. Les nombres 0 et 1 ne sont ni premiers ni composés.* »

   Déterminez les nombres premiers inférieurs à 100. Combien y a-t-il de nombres premiers entre 0 et 100 ? Pour vous aider, nous vous proposons deux méthodes.

   Méthode 1 (peu optimale mais assez intuitive)
   Pour chaque nombre de 2 à 100, calculez le reste de la division entière (avec l'opérateur modulo `%`) depuis 1 jusqu'à lui-même. Si c'est un nombre premier, il aura exactement deux nombres pour lesquels le reste de la division entière est égal à 0 (1 et lui-même). Si ce n'est pas un nombre premier, il aura plus de deux nombres pour lesquels le reste de la division entière est égal à 0.

   Méthode 2 (plus optimale et plus rapide, mais un peu plus compliquée)
   Vous pouvez parcourir tous les nombres de 2 à 100 et vérifier si ceux-ci sont composés, c'est-à-dire qu'ils sont le produit de deux nombres premiers. Pratiquement, cela consiste à vérifier que le reste de la division entière (opérateur modulo `%`) entre le nombre considéré et n'importe quel nombre premier est nul. Le cas échéant, ce nombre n'est pas premier.

7 Fichiers

7.1 Lecture dans un fichier

Dans la plupart des travaux de programmation, on doit lire ou écrire dans un fichier. Python possède pour cela tout un tas d'outils qui vous simplifient la vie. Avant de passer à un exemple concret, créez un fichier dans un éditeur de texte que vous enregistrerez dans votre répertoire avec le nom `zoo.txt`, par exemple :

```
girafe
tigre
singe
souris
```

Ensuite, testez cet exemple :

```
>>> filin = open('zoo.txt', 'r')
>>> filin
<open file 'zoo.txt', mode 'r' at 0x7f8ea16efc00>
>>> filin.readlines()
['girafe\n', 'tigre\n', 'singe\n', 'souris\n']
>>> filin.close()
>>> filin
<closed file 'zoo.txt', mode 'r' at 0x7f8ea16efc00>
```

- La première commande ouvre le fichier `zoo.txt` en lecture seule (ceci est indiqué avec la lettre `r`). Remarquez que le fichier n'est pas encore lu, mais simplement ouvert (*un peu comme lorsqu'on ouvre un livre, mais qu'on ne l'a pas encore lu*). Lorsqu'on affiche la valeur de la variable `filin`, vous voyez que Python la considère comme un objet de type fichier
  `<open file 'zoo.txt', mode 'r' at 0x7f8ea16efc00>`
  Et oui, Python est un langage orienté **objet**. Retenez seulement que l'on peut considérer chaque variable comme un objet sur lequel on peut appliquer des **méthodes**.
- À propos de méthode, on applique la méthode `readlines()` sur l'objet `filin` dans l'instruction suivante (remarquez la syntaxe du type objet.méthode). Ceci nous retourne une liste contenant toutes les lignes du fichier (*dans notre analogie avec un livre, ceci correspondrait à lire les lignes du livre*).
- Enfin, on applique la méthode `close()` sur l'objet `filin`, ce qui vous vous en doutez, va fermer le fichier (*ceci correspondrait bien sûr à fermer le livre*). On pourra remarquer que l'état de l'objet a changé
  `<closed file 'zoo.txt', mode 'r' at 0x7f8ea16efc00>`

Vous n'avez bien-sûr pas à retenir ces concepts d'objets pour pouvoir programmer avec Python, nous avons juste ouvert cette parenthèse pour attirer votre attention sur la syntaxe.

Remarque : n'utilisez jamais le mot `file` comme nom de variable pour un fichier car `file` est un mot réservé de Python.

Voici maintenant un exemple complet de lecture d'un fichier avec Python.

```
>>> filin = open('zoo.txt', 'r')
>>> lignes = filin.readlines()
>>> lignes
['girafe\n', 'tigre\n', 'singe\n', 'souris\n']
>>> for ligne in lignes:
...     print ligne
...
```

```
girafe

tigre

singe

souris

>>> filin.close()
```

Vous voyez qu'en cinq lignes de code, vous avez lu et parcouru le fichier.

Remarques :
- Notez que la liste `lignes` contient le caractère `\n` à la fin de chacun de ses éléments. Ceci correspond au saut à la ligne de chacune d'entre elles (ceci est codé par un caractère spécial que l'on symbolise par `\n`). Vous pourrez parfois rencontrer également la notation octale `\012`.
- Remarquez aussi que lorsque l'on affiche les différentes lignes du fichier à l'aide de la boucle `for` et de l'instruction `print`, Python saute à chaque fois une ligne.

Méthode read()

Il existe d'autres méthodes que `readlines()` pour lire (et manipuler) un fichier. Par exemple, la méthode `read()` lit tout le contenu d'un fichier et renvoie une chaîne de caractères unique.

```
>>> filin = open('zoo.txt', 'r')
>>> filin.read()
'girafe\ntigre\nsinge\nsouris\n'
>>> filin.close()
```

Méthode readline()

La méthode `readline()` (sans `s`) lit une ligne d'un fichier et la renvoie sous forme d'une chaîne de caractères. À chaque nouvel appel de `readline()`, la ligne suivante est renvoyée. Associée à la boucle `while`, cette méthode permet de lire un fichier ligne par ligne.

```
>>> filin = open('zoo.txt', 'r')
>>> ligne = filin.readline()
>>> while ligne != "":
...     print ligne
...     ligne = filin.readline()
...
girafe

tigre

singe

souris

>>> filin.close()
```

Méthodes seek() et tell()

Les méthodes `seek()` et `tell()` permettent respectivement de se déplacer au n^{ième} caractère (plus exactement au n^{ième} octet) d'un fichier et d'afficher où en est la lecture du fichier, c'est-à-dire quel caractère (ou octet) est en train d'être lu.

```
>>> filin = open('zoo.txt', 'r')
>>> filin.readline()
'girafe\n'
>>> filin.tell()
7
>>> filin.seek(0)
>>> filin.tell()
0
>>> filin.readline()
'girafe\n'
>>> filin.close()
```

On remarque qu'à l'ouverture d'un fichier, le tout premier caractère est considéré comme le caractère 0 (tout comme le premier élément d'une liste). La méthode `seek()` permet facilement de remonter au début du fichier lorsque l'on est arrivé à la fin ou lorsqu'on en a lu une partie.

Itérations directement sur le fichier

Python essaie de vous faciliter la vie au maximum. Voici un moyen à la fois simple et élégant de parcourir un fichier.

```
>>> filin = open('zoo.txt', 'r')
>>> for ligne in filin:
...     print ligne
...
girafe

tigre

singe

souris

>>> filin.close()
```

La boucle `for` va demander à Python d'aller lire le fichier ligne par ligne. Privilégiez cette méthode par la suite.

7.2 Écriture dans un fichier

Écrire dans un fichier est aussi simple que de le lire. Voyez l'exemple suivant :

```
>>> animaux2 = ['poisson', 'abeille', 'chat']
>>> filout = open('zoo2.txt', 'w')
>>> for animal in animaux2:
...     filout.write(animal)
...
>>> filout.close()
```

Le contenu du fichier `zoo2.txt` est `poissonabeillechat`.
Quelques commentaires sur cet exemple :
- Après avoir initialisé la liste `animaux2`, nous avons ouvert un fichier mais cette fois-ci en mode écriture (avec le caractère `w`).
- Ensuite, on a balayé cette liste à l'aide d'une boucle. À chaque itération, nous avons écrit chaque élément de la liste dans le fichier. Remarquez à nouveau la méthode `write()` qui s'applique sur l'objet `filout`.
- Enfin, on a fermé le fichier avec la méthode `close()`.

Vous voyez qu'il est extrêmement simple en Python de lire ou d'écrire dans un fichier.

Remarque. Si votre programme produit uniquement du texte, vous pouvez l'écrire sur la sortie standard (avec l'instruction `print`). L'avantage est que dans ce cas l'utilisateur peut bénéficier de toute les potentialités d'Unix (redirection, tri, *parsing*...). S'il veut écrire le résultat du programme dans un fichier, il pourra toujours le faire en redirigeant la sortie.

7.3 Méthode optimisée d'ouverture et de fermeture de fichier

Depuis la version 2.5, Python introduit le mot-clé `with` qui permet d'ouvrir et fermer un fichier de manière commode. Si pour une raison ou une autre l'ouverture conduit à une erreur (problème de droits, etc), l'utilisation de `with` garantit la bonne fermeture du fichier (ce qui n'est pas le cas avec l'utilisation de la méthode `open()` invoquée telle quelle). Voici un exemple :

```
>>> with open('zoo.txt', 'r') as filin:
...     for ligne in filin:
...         print ligne
...
girafe

tigre

singe

souris

>>>
```

Vous remarquez que `with` introduit un bloc d'indentation. C'est à l'intérieur de ce bloc que nous effectuons toutes les opérations sur le fichier. Une fois sorti, Python fermera **automatiquement** le fichier. Vous n'avez donc plus besoin d'invoquer la fonction `close()`.
Pour ceux qui veulent appronfondir, la commande with est plus générale et utilisable dans d'autres contextes (méthode compacte pour gérer les exceptions).

7.4 Exercices

Conseil : pour les exercices 1 et 2, utilisez l'interpréteur Python. Pour les exercices suivants, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

1. Dans l'exemple `'girafe'`, `'tigre'`, etc, ci-dessus, comment expliquer vous que Python saute une ligne à chaque itération ? Réécrivez les instructions *ad-hoc* pour que Python écrive le contenu du fichier sans sauter de ligne.

2. En reprenant le dernier exemple sur l'écriture dans un fichier, vous pouvez constater que les noms d'animaux ont été écrits les uns à la suite des autres, sans retour à la ligne. Comment expliquez-vous ce résultat ? Modifiez les instructions de manière à écrire un animal par ligne.
3. Dans cet exercice, nous allons utiliser une sortie partielle de DSSP (*Define Secondary Structure of Proteins*), qui est un logiciel d'extraction des structures secondaires. Ce fichier contient 5 colonnes correspondant respectivement au numéro de résidu, à l'acide aminé, sa structure secondaire et ses angles phi/psi.
   – Téléchargez le fichier first_helix_1tfe.txt sur le site de notre cours et sauvegardez-le dans votre répertoire de travail (jetez-y un oeil en passant).
   – Chargez les lignes de ce fichier en les plaçant dans une liste puis fermez le fichier.
   – Écrivez chaque ligne à l'écran pour vérifier que vous avez bien chargé le fichier.
   – Écrivez dans un fichier `output.txt` chacune des lignes. N'oubliez pas le retour à la ligne pour chaque acide aminé.
   – Écrivez dans un fichier `output2.txt` chacune des lignes suivies du message `line checked`. Encore une fois, n'oubliez pas les retours à la ligne.

8 Modules

8.1 Définition

Les modules sont des programmes Python qui contiennent des fonctions que l'on est amené à réutiliser souvent (on les appelle aussi bibliothèques ou *libraries*). Les développeurs de Python ont mis au point de nombreux modules qui effectuent une quantité phénoménale de tâches. Pour cette raison, prenez le réflexe de vérifier si une partie du code que vous souhaitez écrire n'existe pas déjà sous forme de module. La plupart de ces modules sont déjà installés dans les versions standards de Python. Vous pouvez accéder à une documentation exhaustive pour python 2.7 sur le site de Python (ici pour python 3.*). Surfez un peu sur ce site, la quantité de modules est impressionante.

8.2 Importation de modules

Jusqu'à maintenant, nous avons rencontré une fois cette notion de module lorsque nous avons voulu tirer un nombre aléatoire.

```
>>> import random
>>> random.randint(0,10)
4
```

Regardons de plus près cet exemple :

- L'instruction `import` permet d'accéder à 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 (contrairement à `range()` par exemple). Remarquez la notation objet `random.randint()` où la fonction `randint()` peut être considérée comme une méthode de l'objet `random`.

Il existe un autre moyen d'importer une ou des fonctions d'un module :

```
>>> from random import randint
>>> randint(0,10)
7
```

À l'aide du mot-clé `from`, vous pouvez importer une fonction spécifique d'un module donné. Remarquez que dans ce cas il est inutile de répéter le nom du module, seul le nom de ladite fonction est requis.

On peut également importer toutes les fonctions d'un module :

```
>>> from random import *
>>> x = [1, 2, 3, 4]
>>> shuffle(x)
>>> x
[2, 3, 1, 4]
>>> shuffle(x)
>>> x
[4, 2, 1, 3]
>>> randint(0,50)
46
>>> uniform(0,2.5)
0.64943174760727951
```

Comme vous l'avez deviné, l'instruction `from random import *` importe toutes les fonctions du module `random`. On peut ainsi utiliser toutes ses fonctions directement, comme par exemple `shuffle()`, qui permute une liste aléatoirement.

Dans la pratique, plutôt que de charger toutes les fonctions d'un module en une seule fois (`from random import *`), nous vous conseillons de charger le module (`import random`) puis d'appeler explicitement les fonctions voulues (`random.randint(0,2)`).

Enfin, si vous voulez vider de la mémoire un module déjà chargé, vous pouvez utiliser l'instruction `del` :

```
>>> import random
>>> random.randint(0,10)
2
>>> del random
>>> random.randint(0,10)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'random' is not defined
```

Vous pouvez constater qu'un rappel d'une fonction du module `random` après l'avoir vidé de la mémoire retourne un message d'erreur.

Enfin, il est également possible de définir un alias (un nom plus court) pour un module :

```
>>> import random as rand
>>> rand.randint(1, 10)
6
```

Dans cet exemple, les fonctions du module `random` sont accessibles via l'alias `rand`.

8.3 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)
...
```

On peut se promener dans l'aide avec les flèches ou les touches `page-up` et `page-down` (comme dans les commandes Unix `man`, `more` ou `less`). Il est aussi possible d'invoquer de l'aide sur une fonction particulière d'un module de la manière suivante `help(random.randint)`.

La commande `help()` est en fait une commande plus générale permettant d'avoir de l'aide sur n'importe quel objet chargé en mémoire.

```
>>> x = range(2)
>>> help(x)
Help on list object:

class list(object)
 |  list() -> new list
 |  list(sequence) -> new list initialized from sequence's items
 |
```

```
 |  Methods defined here:
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
...
```

Enfin, si on veut connaître en seul coup d'oeil toutes les méthodes ou variables associées à un objet, on peut utiliser la commande `dir` :

```
>>> import random
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'System
'TWOPI', 'WichmannHill', '_BuiltinMethodType', '_MethodType', '__all__', '__buil
'__doc__', '__file__', '__name__', '_acos', '_ceil', '_cos', '_e', '_exp', '_hex
'_inst', '_log', '_pi', '_random', '_sin', '_sqrt', '_test', '_test_generator',
'_urandom', '_warn', 'betavariate', 'choice', 'expovariate', 'gammavariate', 'ga
'getrandbits', 'getstate', 'jumpahead', 'lognormvariate', 'normalvariate',
'paretovariate', 'randint', 'random', 'randrange', 'sample', 'seed', 'setstate',
'shuffle', 'uniform', 'vonmisesvariate', 'weibullvariate']
>>>
```

8.4 Modules courants

Il existe une série de modules que vous serez probablement amenés à utiliser si vous programmez en Python. En voici une liste non exhaustive. Pour la liste complète, reportez-vous à la page des modules sur le site de Python :

- math : fonctions et constantes mathématiques de base (sin, cos, exp, pi...).
- sys : passage d'arguments, gestion de l'entrée/sortie standard...
- os : dialogue avec le système d'exploitation (e.g. permet de sortir de Python, lancer une commande en *shell*, puis de revenir à Python).
- random : génération de nombres aléatoires.
- time : permet d'accéder à l'heure de l'ordinateur et 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).
- urllib2 : permet de récupérer des données sur internet depuis python.
- Tkinter : interface python avec Tk (permet de créer des objets graphiques ; nécessite d'installer Tk).
- re : gestion des expressions régulières.
- pickle : écriture et lecture de structures Python (comme les dictionnaires par exemple).

Nous vous conseillons vivement d'aller surfer sur les pages de ces modules pour découvrir toutes leurs potentialités.

Vous verrez plus tard comment créer votre propres modules lorsque vous êtes amenés à réutiliser souvent vos propres fonctions.

Enfin, notez qu'il existe de nombreux autres modules qui ne sont pas installés de base dans Python mais qui sont de grand intérêt en bioinformatique (au sens large). Citons-en quelques-uns : `numpy` (algèbre linéaire, transformée de Fourier), `biopython` (recherche dans les banques de données biologiques), `rpy` (dialogue R/Python)...

8.5 Module sys : passage d'arguments

Le module sys contient (comme son nom l'indique) des fonctions et des variables spécifiques au système, ou plus exactement à l'interpréteur lui-même. Par exemple, il permet de gérer l'entrée (*stdin*) et la sortie standard (*stdout*). Ce module est particulièrement intéressant pour récupérer les arguments passés à un script Python lorsque celui-ci est appelé en ligne de commande. Dans cet exemple, oublions l'interpréteur et écrivons le script suivant que l'on enregistrera sous le nom `test.py` (n'oubliez pas de le rendre exécutable) :

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

import sys
print sys.argv
```

Ensuite lancez `test.py` suivi de plusieurs arguments. Par exemple :

```
poulain@cumin> python test.py salut girafe 42
['test.py', 'salut', 'girafe', '42']
```

Dans l'exemple précédent, `poulain@cumin>` représente l'invite du *shell*, `test.py` est le nom du script Python, `salut`, `girafe` et `42` sont les arguments passés au script.

La variable `sys.argv` est une liste qui représente tous les arguments de la ligne de commande, y compris le nom du script lui même qu'on peut retrouver dans `sys.argv[0]`. On peut donc accéder à chacun de ces arguments avec `sys.argv[1]`, `sys.argv[2]`...

On peut aussi utiliser la fonction `sys.exit()` pour quitter un script Python. On peut donner comme argument un objet (en général une chaîne de caractères) qui sera renvoyé au moment ou Python quittera le script. Par exemple, si vous attendez au moins un argument en ligne de commande, vous pouvez renvoyer un message pour indiquer à l'utilisateur ce que le script attend comme argument :

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

import sys

if len(sys.argv) != 2:
    sys.exit("ERREUR : il faut exactement un argument.")

#
# suite du script
#
```

Puis on l'exécute sans argument :

```
poulain@cumin> python test.py
ERREUR : il faut exactement un argument.
```

Notez qu'ici on vérifie que le script possède deux arguments car le nom du script lui-même est le premier argument et `file.txt` constitue le second.

8.6 Module os

Le module `os` gère l'interface avec le système d'exploitation.

Une fonction pratique de ce module permet de gérer la présence d'un fichier sur le disque.

```
>>> import sys
>>> import os
>>> if os.path.exists("toto.pdb"):
...     print "le fichier est présent"
... else:
...     sys.exit("le fichier est absent")
...
le fichier est absent
```

Dans cet exemple, si le fichier n'est pas présent sur le disque, on quitte le programme avec la fonction `exit()` du module `sys`.

La fonction `system()` permet d'appeler n'importe quelle commande externe.

```
>>> import os
>>> os.system("ls -al")
total 5416
drwxr-xr-x 2 poulain dsimb    4096 2010-07-21 14:33 .
drwxr-xr-x 6 poulain dsimb    4096 2010-07-21 14:26 ..
-rw-r--r-- 1 poulain dsimb  124335 2010-07-21 14:31 1BTA.pdb
-rw-r--r-- 1 poulain dsimb 4706057 2010-07-21 14:31 NC_000913.fna
-rw-r--r-- 1 poulain dsimb  233585 2010-07-21 14:30 NC_001133.fna
-rw-r--r-- 1 poulain dsimb  463559 2010-07-21 14:33 NC_001133.gbk
0
```

La commande externe `ls -al` est introduite comme une chaîne de caractères à la fonction `system()`.

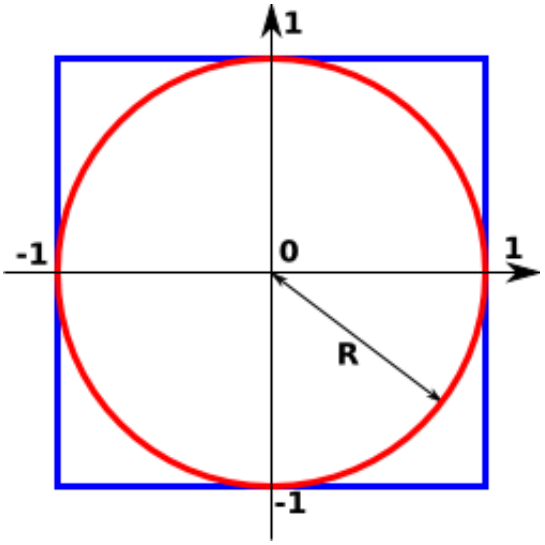
8.7 Exercices

Conseil : pour les trois premiers exercices, utilisez l'interpréteur Python. Pour les exercices suivants, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

1. Affichez sur la même ligne les nombres de 10 à 20 (inclus) ainsi que leur racine carrée avec 3 décimales (module math). Exemple :

   ```
   10 3.162
   11 3.317
   12 3.464
   13 3.606
   ```

2. Calculez le cosinus de pi/2 (module math).
3. Affichez la liste des fichiers du répertoire courant avec le module `os`. N'utilisez pas la fonction `os.system()` mais la fonction `os.listdir()` (lisez la documentation pour comprendre comment l'utiliser).
4. Écrivez les nombres de 1 à 10 avec 1 seconde d'intervalle (module time).
5. Générez une séquence aléatoire de 20 chiffres, ceux-ci étant des entiers tirés entre 1 et 4 (module random).
6. Générez une séquence aléatoire de 20 bases de deux manières différentes (module random).
7. Déterminez votre jour (lundi, mardi...) de naissance (module calendar).
8. Exercice +++. **Évaluation du nombre π par la méthode Monte Carlo**.
   Soit un cercle de rayon 1 (en rouge) inscrit dans un carré de coté 2 (en bleu).



L'aire du carré vaut $(2R)^2$ soit 4. L'aire du cercle vaut πR^2 soit π.

En choisissant N points aléatoires (à l'aide d'une distribution uniforme) à l'intérieur du carré, la probabilité que ces points se trouvent aussi dans le cercle est

$$p = \frac{\text{aire du cercle}}{\text{aire du carré}} = \frac{\pi}{4}$$

Soit n, le nombre points effectivement dans le cercle, il vient alors

$$p = \frac{n}{N} = \frac{\pi}{4},$$

d'où

$$\pi = 4 \times \frac{n}{N}.$$

Déterminez une approximation de π par cette méthode. Pour cela, vous allez, pour N itérations, choisir aléatoirement les coordonnées d'un point entre -1 et 1 (fonction `uniform()` du module `random`), calculer la distance entre le centre du cercle et ce point et déterminer si cette distance est inférieure au rayon du cercle. Le cas échéant, le compteur `n` sera incrémenté.

Que vaut l'approximation de π pour 100 itérations ? 500 ? 1000 ?

Conseil : pour les premiers exercices, utilisez l'interpréteur Python.

9 Plus sur les chaînes de caractères

9.1 Préambule

Nous avons déjà abordé les chaînes de caractères dans le chapitre *variables et écriture*. Ici nous allons un peu plus loin notamment avec les méthodes associées aux chaînes de caractères. Notez qu'il existe un module `string` mais qui est maintenant considéré comme obsolète depuis la version 2.5 de Python.

9.2 Chaînes de caractères et listes

Les chaînes de caractères peuvent être considérées comme des listes.

```
>>> animaux = "girafe tigre"
>>> animaux
'girafe tigre'
>>> len(animaux)
12
>>> animaux[3]
'a'
```

Nous pouvons donc utiliser certaines propriétés des listes comme les tranches :

```
>>> animaux = "girafe tigre"
>>> animaux[0:4]
'gira'
>>> animaux[9:]
'gre'
>>> animaux[:-2]
'girafe tig'
```

A contrario des listes, les chaînes de caractères présentent toutefois une différence notable, ce sont **des listes non modifiables**. Une fois définie, vous ne pouvez plus modifier un de ses éléments. Le cas échéant, Python renvoie un message d'erreur :

```
>>> animaux = "girafe tigre"
>>> animaux[4]
'f'
>>> animaux[4] = "F"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Par conséquent, si vous voulez modifier une chaîne, vous êtes obligés d'en construire une nouvelle. Pour cela, n'oubliez pas que les opérateurs de concaténation (`+`) et de duplication (`*`) (cf chapitre *variables et écriture*) peuvent vous aider. Vous pouvez également générer une liste, qui elle est modifiable, puis revenir à une chaîne.

9.3 Caractères spéciaux

Il existe certains caractères spéciaux comme le `\n` que nous avons déjà vu (pour le retour à la ligne). Le `\t` vous permet d'écrire une tabulation. Si vous voulez écrire un guillemet simple ou double (et que celui-ci ne soit pas confondus avec les guillemets de déclaration de la chaîne de caractères), vous pouvez utiliser `\'` ou `\"` ou utiliser respectivement des guillements doubles ou simple pour déclarer votre chaîne de caractères.

```
>>> print "Un retour a la ligne\npuis une tabulation\t, puis un guillemet\""
Un retour a la ligne
puis une tabulation     , puis un guillemet"
>>> print 'J\'imprime un guillemet simple'
J'imprime un guillemet simple
>>> print "Un brin d'ADN"
Un brin d'ADN
>>> print 'Python est un "super" langage'
Python est un "super" langage
```

Lorsqu'on souhaite écrire un texte sur plusieurs lignes, il est très commode d'utiliser les guillemets triples permettant de conserver le formatage (notamment les retours à la ligne) :

```
>>> x = '''souris
... chat
... abeille'''
>>> x
'souris\nchat\nabeille'
>>> print x
souris
chat
abeille
```

9.4 Méthodes associées aux chaînes de caractères

Voici quelques méthodes spécifiques aux objets de type `string` :

```
>>> x = "girafe"
>>> x.upper()
'GIRAFE'
>>> x
'girafe'
>>> 'TIGRE'.lower()
'tigre'
```

Les fonctions `lower()` et `upper()` passent un texte en minuscule et en majuscule respectivement. On remarque que l'utilisation de ces fonctions n'altèrent pas la chaîne de départ mais renvoie la chaîne transformée.

Pour mettre en majuscule la première lettre seulement, vous pouvez faire :

```
>>> x[0].upper() + x[1:]
'Girafe'
```

ou encore plus simple avec la fonction Python adéquate :

```
>>> x.capitalize()
'Girafe'
```

Il existe une méthode associée aux chaînes de caractères qui est particulièrement pratique, la fonction `split()` :

```
>>> animaux = "girafe tigre singe"
>>> animaux.split()
['girafe', 'tigre', 'singe']
```

```
>>> for animal in animaux.split():
...     print animal
...
girafe
tigre
singe
```

La fonction `split()` découpe la ligne en champs, en utilisant comme séparateur les espaces ou les tabulations. Il est possible de modifier le séparateur de champs, par exemple :

```
>>> animaux = "girafe:tigre:singe"
>>> animaux.split(":")
['girafe', 'tigre', 'singe']
```

La fonction `find()` recherche une chaîne de caractères passée en argument.

```
>>> animal = "girafe"
>>> animal.find('i')
1
>>> animal.find('afe')
3
>>> animal.find('tig')
-1
```

Si l'élément recherché est trouvé, alors l'indice du début de l'élément dans la chaîne de caractères est renvoyé. Si l'élément n'est pas trouvé, alors la valeur -1 est renvoyée.

Si l'élément recherché est trouvé plusieurs fois, seul l'indice de la première occurence est retourné :

```
>>> animaux = "girafe tigre"
>>> animaux.find("i")
1
```

On trouve aussi la fonction `replace()`, qui serait l'équivalent de la fonction de substitution de la commande Unix **sed** :

```
>>> animaux = "girafe tigre"
>>> animaux.replace("tigre", "singe")
'girafe singe'
>>> animaux.replace("i", "o")
'gorafe togre'
```

Enfin, la fonction `count()` compte le nombre d'occurences d'une chaîne de caractères passée en argument :

```
>>> animaux = "girafe tigre"
>>> animaux.count("i")
2
>>> animaux.count("z")
0
>>> animaux.count("tigre")
1
```

9.5 Conversion de types

Dans tout langage de programmation, on est souvent amené à convertir les types, c'est-à-dire passer d'un type numérique à une chaîne de caractères ou vice-versa. En Python, rien de plus simple avec les fonctions `int()`, `float()` et `str()`. Pour vous en convaincre, regardez ces exemples :

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

Ces conversions sont essentielles lorsqu'on lit ou écrit des nombres dans un fichier. En effet, les nombres dans un fichier sont considérés comme du texte par la fonction `readlines()`, par conséquent il faut les convertir si on veut effectuer des opérations numériques dessus.

9.6 Conversion d'une liste de chaînes de caractères en une chaîne de caractères

La conversion d'une liste de chaînes de caractères en une chaîne de caractères est un peu particulière puisqu'elle fait appelle à la fonction `join()`.

```
>>> seq = ["A", "T", "G", "A", "T"]
>>> seq
['A', 'T', 'G', 'A', 'T']
>>> "-".join(seq)
'A-T-G-A-T'
>>> " ".join(seq)
'A T G A T'
>>> "".join(seq)
'ATGAT'
```

Les éléments de la liste initiale sont concaténés les uns à la suite des autres et intercalés par un séparateur qui peut être n'importe quelle chaîne de caractères (ici, nous avons utilisé un tiret, un espace et rien).

Attention, la fonction `join()` ne s'applique qu'à une liste de chaînes de caractères.

```
>>> maliste = ["A", 5, "G"]
>>> " ".join(maliste)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 1: expected string, int found
```

Nous espérons qu'après ce petit tour d'horizon vous serez convaincu de la richesse des méthodes associées aux chaînes de caractères. Pour avoir une liste exhaustive de l'ensemble des méthodes associées à une variable particulière, vous pouvez utiliser la commande `dir()`.

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

Pour l'instant vous pouvez ignorer les méthodes qui commencent et qui se terminent par deux tirets bas (*underscores*) « __ ».

Vous pouvez ensuite accéder à l'aide et à la documentation d'une fonction particulière avec `help()` :

```
>>> help(animaux.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)
```

9.7 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

1. Soit la liste `['girafe', 'tigre', 'singe', 'souris']`. Avec une boucle, affichez chaque élément ainsi que sa taille (nombre de caractères).
2. Soit la séquence nucléique `ATATACGGATCGGCTGTTGCCTGCGTAGTAGCGT`. Calculez la fréquence de chaque base dans cette séquence.
3. Soit la séquence protéique `ALA GLY GLU ARG TRP TYR SER GLY ALA TRP`. Transformez cette séquence en une chaîne de caractères en utilisant le code une lettre pour les acides aminés.
4. **Distance de Hamming**.
   La distance de Hamming mesure la différence entre deux séquences de même taille en sommant le nombre de positions qui, pour chaque séquence, ne correspondent pas au même acide aminé. Calculez la distance de Hamming pour les séquences `AGWPSGGASAGLAIL` et `IGWPSAGASAGLWIL`.
5. **Palindrome**.
   Un palindrome est un mot ou une phrase dont l'ordre des lettres reste le même si on le lit de gauche à droite ou de droite à gauche. Par exemple, « ressasser » et « Engage le jeu que je le gagne » sont des palindromes.
   Écrivez un script qui détermine si une chaîne de caractères est un palindrome. Pensez à vous débarasser des majuscules et des espaces. Testez si les expressions suivantes sont des palindromes : « Radar », « *Never odd or even* », « Karine alla en Iran », « Un roc si biscornu ».

6. **Mot composable**.

   Un mot est composable à partir d'une séquence de lettres si la séquence contient toutes les lettres du mot. Comme au Scrabble, chaque lettre de la séquence ne peut être utilisée qu'une seule fois. Par exemple, *coucou* est composable à partir de *uocuoceokzefhu*.

   Écrivez un script qui permet de savoir si un mot est composable à partir d'une séquence de lettres. Testez le avec différents mots et séquences.

   Remarque : dans votre script, le mot et la séquence de lettres seront des chaînes de caractères.

7. **Alphabet et pangramme**.

   Les codes ASCII des lettres minuscules de l'alphabet vont de 97 (lettre a) à 122 (lettre z). La fonction `chr()` prend en argument un code ASCII sous forme d'une entier et renvoie le caractère correspondant. Ainsi `chr(97)` renvoie `'a'`, `chr(98)` renvoie `'b'` et ainsi de suite.

   Écrivez un script qui construit une chaîne de caractères contenant toutes les lettres de l'alphabet.

   Un pangramme est une phrase comportant au moins une fois chaque lettre de l'alphabet. Par exemple, « Portez ce vieux whisky au juge blond qui fume » est un pangramme.

   Modifiez le script précédent pour déterminer si une chaîne de caractères est un pangramme ou non. Pensez à vous débarasser des majuscules le cas échéant. Testez si les expressions suivantes sont des pangrammes : « Monsieur Jack vous dactylographiez bien mieux que votre ami Wolf », « Buvez de ce whisky que le patron juge fameux ».

8. Téléchargez le fichier pdb 1BTA dans votre répertoire. Faites un script qui récupère seulement les carbones alpha et qui les affiche à l'écran.

9. En vous basant sur le script précédent, affichez à l'écran les carbones alpha des deux premiers résidus. Toujours avec le même script, calculez la distance inter atomique entre ces deux atomes.

10. En vous basant sur le script précédent, calculez les distances entre carbones alpha consécutifs. Affichez ces distances sous la forme

    ```
    numero_calpha_1 numero_calpha_2 distance
    ```

    À la fin du script, faites également afficher la moyenne des distances. La distance inter carbone alpha dans les protéines est très stable et de l'ordre de 3,8 angströms. Observez avec attention les valeurs que vous avez obtenues. Expliquez l'anomalie détectée.

10 Plus sur les listes

10.1 Propriétés des listes

Jusqu'à maintenant, nous avons toujours utilisé des listes qui étaient déjà remplies. Nous savons comment modifier un de ses éléments, mais il est aussi parfois très pratique d'en remanier la taille (*e. g.* ajouter, insérer ou supprimer un ou plusieurs éléments, etc.). Les listes possèdent à cet effet des méthodes qui leurs sont propres. Observez les exemples suivants :

append() pour ajouter un élément à la fin d'une liste.

```
>>> x = [1,2,3]
>>> x.append(5)
>>> x
[1, 2, 3, 5]
```

qui est équivalent à

```
>>> x = [1,2,3]
>>> x = x + [5]
>>> x
[1, 2, 3, 5]
```

insert() pour insèrer un object dans une liste avec un indice déterminé.

```
>>> x.insert(2,-15)
>>> x
[1, 2, -15, 3, 5]
```

del pour supprimer un élément d'une liste à une indice déterminé.

```
>>> del x[1]
>>> x
[1, -15, 3, 5]
```

remove() pour supprimer un élément d'une liste à partir de sa valeur

```
>>> x.remove(5)
>>> x
[1, -15, 3]
```

sort() pour trier une liste.

```
>>> x.sort()
>>> x
[-15, 1, 3]
```

reverse() pour inverser une liste.

```
>>> x.reverse()
>>> x
[3, 1, -15]
```

count() pour compter le nombre d'éléments (passé en argument) dans une liste.

```
>>> l=[1, 2, 4, 3, 1, 1]
>>> l.count(1)
3
>>> l.count(4)
1
>>> l.count(23)
0
```

Remarque 1 : attention, une liste remaniée n'est pas renvoyée ! Pensez-y dans vos utilisations futures des listes.

Remarque 2 : attention, certaines fonctions ci-dessus décalent les indices d'une liste (par exemple `insert()`, `del` etc).

La méthode `append()` est particulièrement pratique car elle permet de construire une liste au fur et à mesure des itérations d'une boucle. Pour cela, il est commode de définir préalablement une liste vide de la forme `maliste = []`. Voici un exemple où une chaîne de caractères est convertie en liste :

```
>>> seq = 'CAAAGGTAACGC'
>>> seq_list = []
>>> seq_list
[]
>>> for base in seq:
...   seq_list.append(base)
...
>>> seq_list
['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
```

Remarquez que vous pouvez directement utiliser la fonction `list()` qui prend n'importe quel objet séquentiel (liste, chaîne de caractères, tuples, etc.) et qui renvoie une liste :

```
>>> seq = 'CAAAGGTAACGC'
>>> list(seq)
['C', 'A', 'A', 'A', 'G', 'G', 'T', 'A', 'A', 'C', 'G', 'C']
```

Cette méthode est certes plus simple, mais il arrive parfois que l'on doive utiliser les boucles tout de même, comme lorqu'on lit un fichier.

10.2 Test d'appartenance

L'inscription `in` permet de tester si un élément fait partie d'une liste.

```
liste = [1, 3, 5, 7, 9]
>>> 3 in liste
True
>>> 4 in liste
False
>>> 3 not in liste
False
>>> 4 not in liste
True
```

La variation avec `not` permet, *a contrario*, de vérifier qu'un élement n'est pas dans une liste.

10.3 Copie de listes

Il est très important de savoir que l'affectation d'une liste (à partir d'une liste préexistante) crée en réalité une **référence** et non une **copie** :

```
>>> x = [1,2,3]
>>> y = x
>>> y
[1, 2, 3]
>>> x[1] = -15
>>> y
[1, -15, 3]
```

Vous voyez que la modification de `x` modifie `y` aussi. Rappelez-vous de ceci dans vos futurs programmes car cela pourrait avoir des effets désastreux ! Techniquement, Python utilise des pointeurs (comme dans le langage C) vers les mêmes objets et ne crée pas de copie à moins que vous n'en ayez fait la demande explicitement. Regardez cet exemple :

```
>>> x = [1,2,3]
>>> y = x[:]
>>> x[1] = -15
>>> y
[1, 2, 3]
```

Dans l'exemple précédent, `x[:]` a créé une copie « à la volée » de la liste `x`. Vous pouvez utiliser aussi la fonction `list()` qui renvoie explicitement une liste :

```
>>> x = [1,2,3]
>>> y = list(x)
>>> x[1] = -15
>>> y
[1, 2, 3]
```

Attention, les deux techniques précédentes ne fonctionnent que pour les listes à une dimension, autrement dit les listes qui ne contienent pas elles-mêmes d'autres listes.

```
>>> x = [[1,2],[3,4]]
>>> x
[[1, 2], [3, 4]]
>>> y = x[:]
>>> y[1][1] = 55
>>> y
[[1, 2], [3, 55]]
>>> x
[[1, 2], [3, 55]]
>>> y = list(x)
>>> y[1][1] = 77
>>> y
[[1, 2], [3, 77]]
>>> x
[[1, 2], [3, 77]]
```

La méthode de copie qui **marche à tous les coups** consiste à appeler la fonction `deepcopy()` du module `copy`.

```
>>> import copy
>>> x = [[1,2],[3,4]]
>>> x
[[1, 2], [3, 4]]
>>> y = copy.deepcopy(x)
>>> y[1][1] = 99
>>> y
[[1, 2], [3, 99]]
>>> x
[[1, 2], [3, 4]]
```

10.4 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

1. Soit la liste de nombres `[8, 3, 12.5, 45, 25.5, 52, 1]`. Triez les nombres de cette liste par ordre croissant, sans utiliser la fonction `sort()` (les fonctions `min()`,`append()` et `remove()` vous seront utiles).
2. Générez aléatoirement une séquence nucléique de 20 bases en utilisant une liste et la méthode `append()`.
3. Transformez la séquence nucléique `TCTGTTAACCATCCACTTCG` en sa séquence complémentaire inverse. N'oubliez pas que la séquence complémentaire doit être inversée, pensez aux méthodes des listes !
4. Soit la liste de nombres `[5, 1, 1, 2, 5, 6, 3, 4, 4, 4, 2]`. Enlevez les doublons de cette liste, triez-là et affichez-là.
5. Générez aléatoirement une séquence nucléique de 50 bases contenant 10 % de A, 50 % de G, 30 % de T et 10 % de C.
6. Exercice +++. **Triangle de Pascal**
   Voici le début du triangle de Pascal :

   ```
   1
   11
   121
   1331
   14641
   ...
   ```

   Comprenez comment une ligne est construite à partir de la précente. À partir de l'ordre 1 (ligne 2, `11`), générez l'ordre suivant (`121`). Vous pouvez utiliser une liste préalablement générée avec `range()`. Généralisez à l'aide d'une boucle. Écrivez dans un fichier `pascal.out` les lignes du triangle de Pascal de l'ordre 1 jusqu'à l'ordre 10.

11 Dictionnaires et tuples

11.1 Dictionnaires

Les **dictionnaires** se révèlent très pratiques lorsque vous devez manipuler des structures complexes à décrire et que les listes présentent leurs limites. Les dictionnaires sont des collections non ordonnées d'objets, c-à-d qu'il n'y a pas de notion d'ordre (*i. e.* pas d'indice). On accède aux **valeurs** d'un dictionnaire par des **clés**. Ceci semble un peu confus ? Regardez l'exemple suivant :

```
>>> ani1 = {}
>>> ani1['nom'] = 'girafe'
>>> ani1['taille'] = 5.0
>>> ani1['poids'] = 1100
>>> ani1
{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}
>>> ani1['taille']
5.0
```

En premier, on définit un dictionnaire vide avec les symboles `{}` (tout comme on peut le faire pour les listes avec `[]`). Ensuite, on remplit le dictionnaire avec différentes clés auxquelles on affecte des valeurs (une par clé). Vous pouvez mettre autant de clés que vous voulez dans un dictionnaire (tout comme vous pouvez ajouter autant d'éléments que vous voulez dans une liste). Pour récupérer la valeur d'une clé donnée, il suffit d'utiliser une syntaxe du style `dictionnaire['cle']`.

Méthodes keys() et values()

Les méthodes `keys()` et `values()` renvoient, comme vous vous en doutez, les clés et les valeurs d'un dictionnaire (sous forme de liste) :

```
>>> ani1.keys()
['nom', 'poids', 'taille']
>>> ani1.values()
['girafe', 1100, 5.0]
```

On peut aussi initialiser toutes les clés d'un dictionnaire en une seule opération :

```
>>> ani2 = {'nom':'singe', 'poids':70, 'taille':1.75}
```

Liste de dictionnaires

En créant une liste de dictionnaires possédant les mêmes clés, on obtient une structure qui ressemble à une base de données :

```
>>> animaux = [ani1, ani2]
>>> animaux
[{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}, {'nom': 'singe', 'poids': 70,
>>>
>>> for ani in animaux:
...     print ani['nom']
...
girafe
singe
```

Existence d'une clef

Enfin, pour vérifier si une clé existe, vous pouvez utiliser la propriété `has_key()` :

```
>>> if ani2.has_key('poids'):
...     print "La clef 'poids' existe pour ani2"
...
La clef 'poids' existe pour ani2
```

Python permet même de simplifier encore les choses :

```
>>> if "poids" in ani2:
...     print "La clef 'poids' existe pour ani2"
...
La clef 'poids' existe pour ani2
```

Vous voyez que les dictionnaires permettent de gérer des structures complexes de manière plus explicite que les listes.

11.2 Tuples

Les **tuples** correspondent aux listes à la différence qu'ils sont **non modifiables**. On a vu à la section précédente que les listes pouvaient être modifiées par des références ; les tuples vous permettent de vous affranchir de ce problème. Pratiquement, ils utilisent les parenthèses au lieu des crochets :

```
>>> x = (1,2,3)
>>> x
(1, 2, 3)
>>> x[2]
3
>>> x[0:2]
(1, 2)
>>> x[2] = 15
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

L'affectation et l'indiçage fonctionne comme avec les listes, mais si l'on essaie de modifier un des éléments du tuple, Python renvoie un message d'erreur. Si vous voulez ajouter un élément (ou le modifier), vous devez créer un autre tuple :

```
>>> x = (1,2,3)
>>> x + (2,)
(1, 2, 3, 2)
```

Remarquez que pour utiliser un tuple d'un seul élément, vous devez utiliser une syntaxe avec une virgule `(element,)`, ceci pour éviter une ambiguïté avec une simple expression. Autre particularité des tuples, il est possible d'en créer de nouveaux sans les parenthèses, dès lors que ceci ne pose pas d'ambiguïté avec une autre expression :

```
>>> x = (1,2,3)
>>> x
(1, 2, 3)
>>> x = 1,2,3
>>> x
(1, 2, 3)
```

Toutefois, nous vous conseillons d'utiliser systématiquement les parenthèses afin d'éviter les confusions.

Enfin, on peut utiliser la fonction `tuple(sequence)` qui fonctionne exactement comme la fonction `list`, c-à-d qu'elle prend en argument un objet séquentiel et renvoie le tuple correspondant :

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> tuple(range(10))
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> tuple("ATGCCGCGAT")
('A', 'T', 'G', 'C', 'C', 'G', 'C', 'G', 'A', 'T')
```

Remarque : les listes, dictionnaires, tuples sont des objets qui peuvent contenir des collections d'autres objets. On peut donc construire des listes qui contiennent des dictionnaires, des tuples ou d'autres listes, mais aussi des dictionnaires contenant des tuples, des listes, etc.

11.3 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

1. En utilisant un dictionnaire et la fonction `has_key()`, déterminez le nombre d'occurences de chaque acide aminé dans la séquence `AGWPSGGASAGLAILWGASAIMPGALW`. Le dictionnaire ne doit contenir que les acides aminés présents dans la séquence.
2. Soit la séquence nucléotidique suivante :

   `ACCTAGCCATGTAGAATCGCCTAGGCTTTAGCTAGCTCTAGCTAGCTG`

   En utilisant un dictionnaire, faites un programme qui répertorie tous les mots de 2 lettres qui existent dans la sequence (AA, AC, AG, AT, etc.) ainsi que leur nombre d'occurences puis qui les affiche à l'écran.
3. Faites de même avec des mots de 3 et 4 lettres.
4. En vous basant sur les scripts précédents, extrayez les mots de 2 lettres et leur occurence sur le génome du chromosome I de la levure du boulanger *Saccharomyces cerevisiae* NC_001133.fna. Attention, le génome complet est fourni au format fasta.
5. Créez un script `extract-words.py` qui prend en arguments un fichier genbank suivi d'un entier compris entre 1 et 4. Ce script doit extraire du fichier genbank tous les mots (ainsi que leur nombre d'occurences) du nombre de lettres passées en option.
6. Appliquez ce script sur le génome d'*Escherichia coli* : NC_000913.fna (au format fasta). Cette méthode vous paraît-elle efficace sur un génome assez gros comme celui d'*E. Coli* ? Comment pourrait-on en améliorer la rapidité ?
7. À partir du fichier PDB 1BTA, construisez un dictionnaire qui contient 4 clés se référant au premier carbone alpha : le numéro du résidu, puis les coordonnées x, y et z.
8. Sur le même modèle que ci-dessus, créez une liste de dictionnaires pour chacun des carbones alpha de la protéine.
9. À l'aide de cette liste, calculez les coordonnées x, y et z du barycentre de ces carbones alpha.

12 Fonctions

12.1 Principe

En programmation, les fonctions sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme. Elles permettent également de rendre le code plus lisible et plus clair en le fractionnant en blocs logiques.

Vous connaissez déjà certaines fonctions Python, par exemple `math.cos(angle)` du module `math` renvoie le cosinus de la variable `angle` exprimée en radian. Vous connaissez aussi des fonctions internes à Python comme `range()` ou `len()`. Pour l'instant, une fonction est à vos yeux une sorte de « boîte noire » :

- À laquelle vous passez une (ou zero ou plusieurs) valeur(s) entre parenthèses. Ces valeurs sont appelées arguments.
- Qui effectue une action. Par exemple `random.shuffle()` permute aléatoirement une liste.
- Et qui renvoie éventuellement un résultat.

Par exemple si vous appelez la fonction `range()` en lui passant la valeur `5` (`range(5)`), celle-ci vous renvoie une liste de nombres entiers de 0 à 4 (`[0, 1, 2, 3, 4]`).

Au contraire, aux yeux du programmeur une fonction est une portion de code effectuant une action bien particulière. Avant de démarrer sur la syntaxe, revenons sur cette notion de « boîte noire » :

1. Une fonction effectue une tâche. Pour cela, elle reçoit éventuellement des arguments et renvoie éventuellement un résultat. Ce qui se passe en son sein n'intéresse pas directement l'utilisateur. Par exemple, il est inutile de savoir comment la fonction `math.cos()` calcule un cosinus, on a juste besoin de savoir qu'il faut lui passer en argument un angle en radian et qu'elle renvoie le cosinus de cet angle. Ce qui se passe au sein de la fonction ne regarde que le programmeur (c'est-à-dire vous dans ce chapitre).
2. Chaque fonction effectue en général une tâche **unique et précise**. Si cela se complique, il est plus judicieux d'écrire plusieurs fonctions (qui peuvent éventuellement s'appeler les unes les autres). Cette **modularité** améliore la qualité générale et la lisibilité du code. Vous verrez qu'en Python, les fonctions présentent une grande flexibilité.

12.2 Définition

Pour définir une fonction, Python utilise le mot-clé `def` et si on veut que celle-ci renvoie une valeur, il faut utiliser le mot-clé `return`. Par exemple :

```
>>> def carre(x):
...     return x**2
...
>>> print carre(2)
4
```

Remarquez que la syntaxe de `def` utilise les `:` comme les boucles `for`, `while` ainsi que les tests `if`, un bloc d'instrutions est donc attendu. De même que pour les boucles et les tests, l'**indentation** de ce bloc d'instructions (*i.e.* le corps de la fonction) est **obligatoire**.

Dans l'exemple précédent, nous avons passé un argument à la fonction `carre()` qui nous a retourné une valeur que nous avons affichée à l'écran. Que veut dire valeur retournée ? Et bien cela signifie que cette dernière est stockable dans une variable :

```
>>> res = carre(2)
>>> print res
4
```

Ici, le résultat renvoyé par la fonction est stockée dans la variable `res`. Notez qu'une fonction ne prend pas forcément un argument et ne renvoie pas forcément une valeur, par exemple :

```
>>> def hello():
...     print "bonjour"
...
>>> hello()
bonjour
```

Dans ce cas la fonction `hello()` se contente d'imprimer la chaîne de caractères `"hello"` à l'écran. Elle ne prend aucun argument et ne renvoie aucun résultat. Par conséquent, cela n'a pas de sens de vouloir récupérer dans une variable le résultat renvoyé par une telle fonction . Si on essaie tout de même, Python affecte la valeur `None` qui signifie « rien » en anglais :

```
>>> x = hello()
bonjour
>>> print x
None
```

12.3 Passage d'arguments

Le nombre d'argument(s) que l'on peut passer à une fonction est variable. Nous avons vu ci-dessus des fonctions auxquelles on passait 0 ou 1 argument. Dans les chapitres précédentes, vous avez vu des fonctions internes à Python qui prenaient au moins 2 arguments, pour rappel souvenez-vous de `range(1,10)` ou encore `range(1,10,2)`. Le nombre d'argument est donc laissé libre à l'initiative du programmeur qui est en train de développer une nouvelle fonction.

Une particularité des fonctions en Python est que vous n'êtes pas obligé de préciser le type des arguments que vous lui passez, dès lors que les opérations que vous effectuez avec ces arguments sont valides. Python est en effet connu comme étant un langage au *typage dynamique*, c'est-à-dire qu'il reconnaît pour vous le type des variables au moment de l'exécution, par exemple :

```
>>> def fois(x,y):
...     return x*y
...
>>> fois(2,3)
6
>>> fois(3.1415,5.23)
16.430045000000003
>>> fois('to',2)
'toto'
```

L'opérateur $*$ reconnait plusieurs types (entiers, réels, chaînes de caractères), notre fonction est donc capable d'effectuer plusieurs tâches !

Un autre gros avantage de Python est que ses fonctions sont capables de renvoyer plusieurs valeurs à la fois, comme dans cette fraction de code :

```
>>> def carre_cube(x):
...     return x**2,x**3
...
```

```
>>> carre_cube(2)
(4, 8)
```

Vous voyez qu'en réalité Python renvoie un objet séquentiel qui peut par conséquent contenir plusieurs valeurs. Dans notre exemple Python renvoie un objet `tuple` car on a utilisé une syntaxe de ce type. Notre fonction pourrait tout autant renvoyer une liste :

```
>>> def carre_cube2(x):
...     return [x**2,x**3]
...
>>> carre_cube2(3)
[9, 27]
```

Enfin, il est possible de passer un ou plusieurs argument(s) de manière facultative et de leur attribuer une valeur par défaut :

```
>>> def useless_fct(x=1):
...     return x
...
>>> useless_fct()
1
>>> useless_fct(10)
10
```

Notez que si on passe plusieurs arguments à une fonction, le ou les arguments facultatifs doivent être situés après les arguments obligatoires. Il faut donc écrire `def fct(x, y, z=1):`.

12.4 Portée des variables

Il est très important lorsque l'on manipule des fonctions de connaître la portée des variables. Premièrement, on peut créer des variables au sein d'une fonction qui ne seront pas visibles à l'extérieur de celle-ci ; on les appelle **variables locales**. Observez le code suivant :

```
>>> def mafonction():
...     x = 2
...     print 'x vaut',x,'dans la fonction'
...
>>> mafonction()
x vaut 2 dans la fonction
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
```

Lorsque Python exécute le code de la fonction, il connait le contenu de la variable `x`. Par contre, de retour dans le module principal (dans notre cas, il s'agit de l'interpréteur Python), il ne la connaît plus d'où le message d'erreur. De même, une variable passée en argument est considérée comme **locale** lorsqu'on arrive dans la fonction :

```
>>> def mafonction(x):
...     print 'x vaut',x,'dans la fonction'
...
>>> mafonction(2)
x vaut 2 dans la fonction
```

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

Deuxièmement, lorsqu'une variable déclarée à la racine du module (c'est comme cela que l'on appelle un programme Python), elle est visible dans tout le module.

```
>>> def mafonction():
...     print x
...
>>> x = 3
>>> mafonction()
3
>>> print x
3
```

Dans ce cas, la variable `x` est visible dans le module principal et dans toutes les fonctions du module. Toutefois, Python ne permet pas la modification d'une variable globale dans une fonction :

```
>>> def mafonction():
...     x = x + 1
...
>>> x=1
>>> mafonction()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fct
UnboundLocalError: local variable 'x' referenced before assignment
```

L'erreur renvoyée montre que Python pense que `x` est une variable locale qui n'a pas été encore assignée. Si on veut vraiment modifier une variable globale dans une fonction, il faut utiliser le mot-clé `global` :

```
>>> def mafonction():
...     global x
...     x = x + 1
...
>>> x=1
>>> mafonction()
>>> x
2
```

Dans ce dernier cas, le mot-clé `global` a forcé la variable `x` a être globale plutôt que locale au sein de la fonction.

12.5 Portée des listes

Soyez extrêmement attentifs avec les types modifiables (tels que les listes) car vous pouvez les changer au sein d'une fonction :

```
>>> def mafonction():
...     liste[1] = -127
...
```

```
>>> liste = [1,2,3]
>>> mafonction()
>>> liste
[1, -127, 3]
```

De même que si vous passez une liste en argument, elle est tout autant modifiable au sein de la fonction :

```
>>> def mafonction(x):
...     x[1] = -15
...
>>> y = [1,2,3]
>>> mafonction(y)
>>> y
[1, -15, 3]
```

Si vous voulez éviter ce problème, utilisez des tuples, Python renverra une erreur puisque ces derniers sont non modifiables ! Une autre solution pour éviter la modification d'une liste lorsqu'elle est passée en tant qu'argument, est de la passer explicitement (comme nous l'avons fait pour l'affectation) afin qu'elle reste intacte dans le programme principal.

```
>>> def mafonction(x):
...     x[1] = -15
...
>>> y = [1,2,3]
>>> mafonction(y[:])
>>> y
[1, 2, 3]
>>> mafonction(list(y))
>>> y
[1, 2, 3]
```

Dans ces deux derniers exemples, une copie de `y` est créée à la volée lorsqu'on appelle la fonction, ainsi la liste `y` du module principal reste intacte.

12.6 Règle LGI

Lorsque Python rencontre une variable, il va traiter la résolution de son nom avec des priorités particulières : d'abord il va regarder si la variable est **locale**, puis si elle n'existe pas localement, il vérifiera si elle est **globale** et enfin si elle n'est pas globale, il testera si elle est **interne** (par exemple la fonction `len()` est considérée comme une fonction interne à Python, *i.e.* elle existe à chaque fois que vous lancez Python). On appelle cette règle la règle **LGI** pour locale, globale, interne. En voici un exemple :

```
>>> def mafonction():
...     x = 4
...     print 'Dans la fonction x vaut', x
...
>>> x = -15
>>> mafonction()
Dans la fonction x vaut  4
>>> print 'Dans le module principal x vaut',x
Dans le module principal x vaut -15
```

Vous voyez que dans la fonction, `x` a pris la valeur qui lui était définie localement en priorité sur sa valeur définie dans le module principal.

Conseil : même si Python peut reconnaître une variable ayant le même nom que ses fonctions ou variables internes, évitez de les utiliser car ceci rendra votre code confus !

12.7 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

1. Prédisez le comportement de ce code sans le recopier dans un script ni dans l'interpréteur Python :

```
def hello(prenom):
    print "Bonjour", prenom

hello("Patrick")
print x
```

2. Prédisez le comportement de ce code sans le recopier dans un script ni dans l'interpréteur Python :

```
x = 10

def hello(prenom):
    print "Bonjour", prenom

hello("Patrick")
print x
```

3. Prédisez le comportement de ce code sans le recopier dans un script ni dans l'interpréteur Python :

```
x = 10

def hello(prenom):
    print "Bonjour", prenom
    print x

hello("Patrick")
print x
```

4. Prédisez le comportement de ce code sans le recopier dans un script ni dans l'interpréteur Python :

```
x = 10

def hello(prenom):
    x = 42
    print "Bonjour", prenom
    print x

hello("Patrick")
print x
```

5. Créez une fonction qui prend une liste de bases et qui renvoie la séquence complémentaire d'une séquence d'ADN sous forme de liste.
6. À partir d'une séquence d'ADN `ATCGATCGATCGCTGCTAGC`, renvoyez le brin complémentaire (n'oubliez pas que la séquence doit être inversée).
7. Créez une fonction `distance()` qui calcule une distance euclidienne en 3 dimensions entre deux atomes. En reprenant l'exercice sur le calcul de la distance entre carbones alpha consécutifs de la protéine 1BTA (chapitre sur les *chaînes de caractères*), refaites la même chose en utilisant votre fonction `distance()`.

13 Expressions régulières et parsing

Le module re vous permet d'utiliser des expressions régulières au sein de Python. Les expressions régulières sont aussi appelées en anglais *regular expressions* ou *regex*. Elles sont incontournables en bioinformatique lorsque vous voulez récupérer des informations dans un fichier.

Cette action de recherche de données dans un fichier est appelée plus généralement *parsing* (qui signifie littéralement « analyse syntaxique » en anglais). Le *parsing* fait partie du travail quotidien du bioinformaticien, il est sans arrêt en train de « fouiller » dans des fichiers pour en extraire des informations d'intérêt comme par exemple récupérer les coordonnées 3D des atomes d'une protéines dans un fichier PDB ou alors extraire les gènes d'un fichier genbank.

Dans ce chapitre, nous ne ferons que quelques rappels sur les expressions régulières. Pour un documentation plus complète, référez-vous à la page d'aide des expressions régulières sur le site officiel de Python.

13.1 Définition et syntaxe

Une expression régulière est une suite de caractères qui a pour but de décrire un fragment de texte. Elle est constitué de deux types de caractères :

1. Les caractères dits *normaux*.
2. Les *métacaractères* ayant une signification particulière, par exemple `^` signifie début de ligne et non pas le caractère « chapeau » littéral.

Certains programmes Unix comme `egrep`, `sed` ou encore `awk` savent interpréter les expressions régulières. Tous ces programmes fonctionnent généralement selon le schéma suivant :

1. Le programme lit un fichier ligne par ligne.
2. Pour chaque ligne lue, si l'expression régulière passée en argument est présente alors le programme effectue une action.

Par exemple, pour le programme `egrep` :

```
[fuchs@rome cours_python]$ egrep "^DEF" herp_virus.gbk
DEFINITION  Human herpesvirus 2, complete genome.
[fuchs@rome cours_python]$
```

Ici, `egrep` renvoie toutes les lignes du fichier genbank du virus de l'herpès (`herp_virus.gbk`) qui correspondent à l'expression régulière `^DEF` (*i.e.* `DEF` en début de ligne).

Avant de voir comment Python gère les expressions régulières, voici quelques éléments de syntaxe des métacaractères :

`^`	début de chaîne de caractères ou de ligne Exemple : l'expression `^ATG` correspond à la chaîne de caractères `ATGCGT` mais pas à la chaîne `CCATGTT`.
`$`	fin de chaîne de caractères ou de ligne Exemple : l'expression `ATG$` correspond à la chaîne de caractères `TGCATG` mais pas avec la chaîne `CCATGTT`.
`.`	n'importe quel caractère (mais un caractère quand même) Exemple : l'expression `A.G` correspond à `ATG`, `AtG`, `A4G`, mais aussi à `A-G` ou à `A G`.
`[ABC]`	le caractère A ou B ou C (un seul caractère) Exemple : l'expression `T[ABC]G` correspond à `TAG`, `TBG` ou `TCG`, mais pas à `TG`.
`[A-Z]`	n'importe quelle lettre majuscule Exemple : l'expression `C[A-Z]T` correspond à `CAT`, `CBT`, `CCT`...
`[a-z]`	n'importe quelle lettre minuscule
`[0-9]`	n'importe quel chiffre
`[A-Za-z0-9]`	n'importe quel caractère alphanumérique
`[^AB]`	n'importe quel caractère sauf A et B Exemple : l'expression `CG[^AB]T` correspond à `CG9T`, `CGCT`... mais pas à `CGAT` ni à `CGBT`.
`\`	caractère d'échappement (pour protéger certains caractères) Exemple : l'expression `\+` désigne le caractère + sans autre signification particulière. L'expression `A\.G` correspond à `A.G` et non pas à `A` suivi de n'importe quel caractère, suivi de `G`.
`*`	0 à n fois le caractère précédent ou l'expression entre parenthèses précédente Exemple : l'expression `A(CG)*T` correspond à `AT`, `ACGT`, `ACGCGT`...
`+`	1 à n fois le caractère précédent ou l'expression entre parenthèses précédente Exemple : l'expression `A(CG)+T` correspond à `ACGT`, `ACGCGT`... mais pas à `AT`.
`?`	0 à 1 fois le caractère précédent ou l'expression entre parenthèses précédente Exemple : l'expression `A(CG)?T` correspond à `AT` ou `ACGT`.
`{n}`	n fois le caractère précédent ou l'expression entre parenthèses précédente
`{n,m}`	n à m fois le caractère précédent ou l'expression entre parenthèses précédente
`{n,}`	au moins n fois le caractère précédent ou l'expression entre parenthèses précédente
`{,m}`	au plus m fois le caractère précédent ou l'expression entre parenthèses précédente
`(CG|TT)`	chaînes de caractères `CG` ou `TT` Exemple : l'expression `A(CG|TT)C` correspond à `ACGC` ou `ATTC`.

13.2 Module re et fonction search

Dans le module `re`, la fonction `search()` permet de rechercher un motif (*pattern*) au sein d'une chaîne de caractères avec une syntaxe de la forme `search(motif, chaine)`. Si `motif` existe dans `chaine`, Python renvoie une instance `MatchObject`. Sans entrer dans les détails propres au langage orienté objet, si on utilise cette instance dans un test, il sera considéré comme vrai. Regardez cet exemple dans lequel on va rechercher le motif `tigre` dans la chaîne de caractères `"girafe tigre singe"` :

```
>>> import re
```

```
>>> animaux = "girafe tigre singe"
>>> re.search('tigre', animaux)
<_sre.SRE_Match object at 0x7fefdaefe2a0>
>>> if re.search('tigre', animaux):
...     print "OK"
...
OK
```

Fonction match()

Il existe aussi la fonction `match()` dans le module `re` qui fonctionne sur le modèle de `search()`. La différence est qu'elle renvoie une instance `MatchObject` seulement lorsque l'expression régulière correspond (*match*) au début de la chaîne (à partir du premier caractère).

```
>>> animaux = "girafe tigre singe"
>>> re.search('tigre', animaux)
<_sre.SRE_Match object at 0x7fefdaefe718>
>>> re.match('tigre', animaux)
```

Nous vous recommandons plutôt l'usage de la fonction `search()`. Si vous souhaitez avoir une correspondance avec le début de la chaîne, vous pouvez toujours utiliser l'accroche de début de ligne `^`.

Compilation d'expressions régulières

Il est aussi commode de préalablement compiler l'expression régulière à l'aide de la fonction `compile()` qui renvoie un objet de type expression régulière :

```
>>> regex = re.compile("^tigre")
>>> regex
<_sre.SRE_Pattern object at 0x7fefdafd0df0>
```

On peut alors utiliser directement cet objet avec la méthode `search()` :

```
>>> animaux = "girafe tigre singe"
>>> regex.search(animaux)
>>> animaux = "tigre singe"
>>> regex.search(animaux)
<_sre.SRE_Match object at 0x7fefdaefe718>
>>> animaux = "singe tigre"
>>> regex.search(animaux)
```

Groupes

Python renvoie un objet `MatchObject` lorsqu'une expression régulière trouve une correspondance dans une chaîne pour qu'on puisse récupérer des informations sur les zones de correspondance.

```
>>> regex = re.compile('([0-9]+)\.([0-9]+)')
>>> resultat = regex.search("pi vaut 3.14")
>>> resultat.group(0)
'3.14'
>>> resultat.group(1)
'3'
```

```
>>> resultat.group(2)
'14'
>>> resultat.start()
8
>>> resultat.end()
12
```

Dans cet exemple, on recherche un nombre composé
- de plusieurs chiffres `[0-9]+`,
- suivi d'un point `\.` (le point a une signification comme métacaractère, donc il faut l'échapper avec `\` pour qu'il ait une signification de point),
- suivi d'un nombre à plusieurs chiffres `[0-9]+`.

Les parenthèses dans l'expression régulière permettent de créer des groupes qui seront récupérés ultérieurement par la fonction `group()`. La totalité de la correspondance est donné par `group(0)`, le premier élément entre parenthèse est donné par `group(1)` et le second par `group(2)`.

Les fonctions `start()` et `end()` donnent respectivement la position de début et de fin de la zone qui correspond à l'expression régulière. Notez que la fonction `search()` ne renvoie que la première zone qui correspond à l'expression régulière, même s'il en existe plusieurs :

```
>>> resultat = regex.search("pi vaut 3.14 et e vaut 2.72")
>>> resultat.group(0)
'3.14'
```

Fonction findall()

Pour récupérer chaque zone, vous pouvez utiliser la fonction `findall()` qui renvoie une liste des éléments en correspondance.

```
>>> regex = re.compile('[0-9]+\.[0-9]+')
>>> resultat = regex.findall("pi vaut 3.14 et e vaut 2.72")
>>> resultat
['3.14', '2.72']
>>> regex = re.compile('([0-9]+)\.([0-9]+)')
>>> resultat = regex.findall("pi vaut 3.14 et e vaut 2.72")
>>> resultat
[('3', '14'), ('2', '72')]
```

Fonction sub()

Enfin, la fonction `sub()` permet d'effectuer des remplacements assez puissants. Par défaut la fonction `sub(chaine1,chaine2)` remplace toutes les occurrences trouvées par l'expression régulière dans `chaine2` par `chaine1`. Si vous souhaitez ne remplacer que les *n* premières occurrences, utilisez l'argument `count=n` :

```
>>> regex.sub('quelque chose',"pi vaut 3.14 et e vaut 2.72")
'pi vaut quelque chose et e vaut quelque chose'
>>> regex.sub('quelque chose',"pi vaut 3.14 et e vaut 2.72", count=1)
'pi vaut quelque chose et e vaut 2.72'
```

Nous espérons que vous êtes convaincus de la puissance du module `re` et des expressions régulières, alors à vos expressions régulières !

13.3 Exercices : extraction des gènes d'un fichier gbk

Pour les exercices suivants, vous utiliserez le module d'expressions régulières `re` et le fichier genbank du chromosome I de la levure du boulanger *Saccharomyces cerevisiae* NC_001133.gbk.

1. Écrivez un script qui extrait l'organisme du fichier genbank `NC_001133.gbk`.
2. Modifiez le script précédent pour qu'il affiche toutes les lignes qui indiquent l'emplacement du début et de la fin des gènes, du type :

   ```
   gene            58..272
   ```

3. Faites de même avec les gènes complémentaires :

   ```
   gene            complement(55979..56935)
   ```

4. Récupérez maintenant la séquence nucléique du génome (dans une chaine de caractères) et affichez-la à l'écran. Vérifiez que vous n'avez pas fait d'erreur en comparant la taille de la séquence extraite avec celle indiquée dans le fichier genbank.
5. A partir de la séquence du génome, récupérez les deux premiers gènes (en les affichant à l'écran). Attention, le premier gène est un gène complémentaire, n'oubliez pas de prendre le complémentaire inverse de la séquence extraite.
6. À partir de toutes ces petites opérations que vous transformerez en fonctions, concevez un programme `genbank2fasta.py` qui extrait tous les gènes d'un fichier genbank fourni en argument et les affiche à l'écran. Pour cela vous pourrez utiliser tout ce que vous avez vu jusqu'à présent (fonctions, listes, modules, etc.).
7. À partir du script précédent, refaites le même programme (`genbank2fasta.py`) en écrivant chaque gène au format fasta dans un fichier.

   Pour rappel, l'écriture d'une séquence au format fasta est le suivant :

   ```
   >ligne de commentaire
   sequence sur une ligne de 80 caractères maxi
   suite de la séquence ......................
   suite de la séquence ......................
   ```

   Vous utiliserez comme ligne de commentaire le nom de l'organisme (avec éventuellement le nom de la souche), suivi du numero du gène, suivi des positions de début et de fin du gène, et enfin une indication si le gène est sur le brin direct (direct) ou complémentaire (compl), comme dans cet exemple :

   ```
   >Saccharomyces cerevisiae S288c 1 1807 2169 compl
   ```

   Les noms des fichiers fasta seront de la forme `gene1.fasta`, `gene2.fasta`, etc, les numéros de gènes étant croissant selon leur position dans le génome, et ce quel que soit leur brin d'appartenance (on ne veut pas une liste pour les gènes sur le brin direct et une autre à part pour ceux du brin complémentaire).

NB : si vous tombez sur des gènes contenant un symbole < ou > à côté de leur position, ignorez ce symbole (après avoir soigneusement vérifié ce qu'il signifiait sur le site de la genbank).

14 Création de modules

14.1 Création

Vous pouvez créez votres propres modules très simplement en Python. Il vous suffit d'écrire un ensemble de fonctions (et/ou de variables) dans un fichier, puis d'enregistrer celui-ci avec une extension `.py` (comme n'importe quel script Python). À titre d'exemple, voici le contenu du module `message.py`.

```
"""Module inutile qui affiche des messages :-)
"""

def Bonjour(nom):
    """Affiche bonjour !
    """
    return "bonjour " + nom

def Ciao(nom):
    """Affiche ciao !
    """
    return "ciao " + nom

def Hello(nom):
    """Affiche hello !
    """
    return "hello " + nom + " !"

date=16092008
```

14.2 Utilisation

Pour appeler une fonction ou une variable de ce module, il faut que le fichier `message.py` soit dans le répertoire courant (dans lequel on travaille) ou bien dans un répertoire indiqué par la variable d'environnement Unix `PYTHONPATH`. Ensuite, il suffit d'importer le module et toutes ses fonctions (et variables) vous sont alors accessibles.

La première fois qu'un module est importé, Python crée un fichier avec une extension `.pyc` (ici `message.pyc`) qui contient le bytecode (code précompilé) du module.

L'appel du module se fait avec la commande `import message`. Notez que le fichier est bien enregistré avec une extension `.py` et pourtant on ne la précise pas lorsqu'on importe le module. Ensuite on peut utiliser les fonctions comme avec un module classique.

```
>>> import message
>>> message.Hello("Joe")
'hello Joe !'
>>> message.Ciao("Bill")
'ciao Bill'
>>> message.Bonjour("Monsieur")
'bonjour Monsieur'
>>>  message.date
16092008
```

Les commentaires (entre triple guillemets) situés en début de module et sous chaque fonction permettent de fournir de l'aide invoquée ensuite par la commande `help()` :

```
>>> help(message)
NAME
    message - Module inutile qui affiche des messages :-)

FILE
    /home/cumin/poulain/message.py

FUNCTIONS
    Hello(nom)
        Affiche hello !

    Bonjour(nom)
        Affiche bonjour !

    Ciao(nom)
        Affiche ciao !

DATA
    date = 16092008
```

Remarques :
- Les fonctions dans un module peuvent s'appeler les unes les autres.
- Les fonctions dans un module peuvent appeler des fonctions situées dans un autre module s'il a été préalablement importé avec la commande `import`.

Vous voyez que les modules sont d'une simplicité enfantine à créer. Si vous avez des fonctions que vous serez amenés à utiliser souvent, n'hésitez plus !

14.3 Exercices

Conseil : pour cet exercice, écrivez un script dans un fichier, puis exécutez-le dans un *shell*.

1. Reprenez l'ensemble des fonctions qui gèrent le traitement de séquences nucléiques et incluez-les dans un module `adn.py`. Testez-les au fur et à mesure.

15 Autres modules d'intérêt

Nous allons voir dans cette section deux modules qui peuvent se révéler très pratiques. Le premier (`urllib2`) permet de télécharger n'importe quel fichier depuis internet. Le second (`pickle`) permet d'enregistrer des objets python pour pouvoir les retrouver tels quels plus tard. Ces deux modules sont présents par défaut dans n'importe quelle distribution Python.

15.1 Module urllib2

Les modules urllib et urllib2 permettent de rapatrier, directement avec Python, des fichiers depuis internet. Nous n'aborderons ici que des exemples simples, mais libre à vous d'approfondir si besoin. Il existe des différences subtiles entre `urllib` et `urllib2` que nous n'aborderons pas ici mais qui sont consultables sur la documentation officielle de Python. Dans ce cours, nous avons choisi de vous présenter `urllib2` car une des fonctions principales (`urlopen()`) est considérée comme obsolète dans `urllib` depuis la version 2.6 de Python.

Prenons un exemple simple, supposons que vous souhaitiez rapatrier le fichier PDB de la protéine barstar. En Unix vous utiliseriez probablement la commande `wget`, regardez comment faire en Python :

```
>>> import urllib2
>>> u = urllib2.urlopen("http://www.pdb.org/pdb/files/1BTA.pdb")
>>> pdb_lines = u.readlines()
>>> u.close()
>>> for line in pdb_lines:
...     print line,
...
HEADER    RIBONUCLEASE INHIBITOR                  09-MAY-94   1BTA
TITLE     THREE-DIMENSIONAL SOLUTION STRUCTURE AND 13C ASSIGNMENTS OF
TITLE    2 BARSTAR USING NUCLEAR MAGNETIC RESONANCE SPECTROSCOPY
COMPND    MOL_ID: 1;
COMPND   2 MOLECULE: BARSTAR;
COMPND   3 CHAIN: A;
COMPND   4 ENGINEERED: YES
SOURCE    MOL_ID: 1;
SOURCE   2 ORGANISM_SCIENTIFIC: BACILLUS AMYLOLIQUEFACIENS;
SOURCE   3 ORGANISM_TAXID: 1390
[...]
>>>
```

Comme vous le voyez, il suffit d'utiliser la fonction `urlopen()` et de lui passer en argument une URL (*i.e.* adresse internet) sous forme de chaîne de caractères. Dès lors, les méthodes associées aux fichiers (cf chapitre 6) sont accessibles (`read()`, `readline()`, `readlines()`, etc.).

Il est également possible d'enregistrer le fichier téléchargé :

```
>>> import urllib2
>>> u = urllib2.urlopen("http://www.pdb.org/pdb/files/1BTA.pdb")
>>> pdb_file_content = u.read()
>>> u.close()
>>> f = open("1BTA.pdb","w")
>>> f.write(pdb_file_content)
>>> f.close()
```

Dans ce dernier exemple, nous récupérons le fichier complet sous la forme d'une chaîne de caractères unique (avec la fonction `read()`). Le fichier est ensuite enregistré dans le répertoire courant (duquel vous avez lancé l'interpréteur Python). Nous vous laissons imaginer comment faire sur une série de plusieurs fichiers (cf. exercices).

15.2 Module pickle

Le module `pickle` permet d'effectuer une sérialisation et désérialisation des données. Il s'agit en fait d'encoder (d'une manière particulière) les objets que l'on peut créer en Python (variables, listes, dictionnaires, fonctions, classes, etc.) afin de les stocker dans un fichier, puis de les récupérer, sans devoir effectuer une étape de *parsing*.

15.2.1 Codage des données

Pour encoder des données avec `pickle` et les envoyer dans un fichier, on peut utiliser la fonction `dump(obj,file)` (où `file` est un fichier déjà ouvert) :

```
>>> import pickle
>>> fileout = open("mydata.dat","w")
>>> maliste = range(10)
>>> mondico = {'year': 2000, 'name': 'Far beyond driven', 'author': 'Pantera',
'style': 'Trash Metal'}
>>> pickle.dump(maliste,fileout)
>>> pickle.dump(mondico,fileout)
>>> pickle.dump(3.14,fileout)
>>> fileout.close()
```

Si on observe le fichier créé, on s'aperçoit que les données sont codées :

```
>>> import os
>>> os.system("cat mydata.dat")
(lp0
I0
aI1
aI2
aI3
aI4
aI5
aI6
aI7
aI8
aI9
a.(dp0
S'style'
p1
S'Trash Metal'
p2
sS'author'
p3
S'Pantera'
p4
sS'name'
p5
```

```
S'Far beyond driven'
p6
sS'year'
p7
I2000
s.F3.1400000000000001
.0
>>>
```

15.2.2 Décodage des données

Pour décoder les données, rien de plus simple, il suffit d'utiliser la fonction `load(file)` où `file` est un fichier ouvert en lecture :

```
>>> filein = open("mydata.dat")
>>> pickle.load(filein)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> pickle.load(filein)
{'style': 'Trash Metal', 'year': 2000, 'name': 'Far beyond driven',
'author': 'Pantera'}
>>> pickle.load(filein)
3.1400000000000001
```

Attention à ne pas utiliser la fonction `load` une fois de trop, sinon une erreur est générée :

```
>>> pickle.load(filein)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.5/pickle.py", line 1370, in load
    return Unpickler(file).load()
  File "/usr/lib/python2.5/pickle.py", line 858, in load
    dispatch[key](self)
  File "/usr/lib/python2.5/pickle.py", line 880, in load_eof
    raise EOFError
EOFError
>>>
```

Il est à noter qu'il existe un module équivalent mais beaucoup plus rapide, le module cpickle.

15.3 Exercices

1. Concevez un script qui rapatrie de manière automatique le fichier `1MSE.pdb` depuis la *Protein Data Bank* (à l'aide du module `urllib2`).
2. On souhaite récupérer les structure 3D de la partie N-terminale de la protéine HSP90 en présence de différents co-cristaux, les codes PDB sont `3K97`, `3K98` et `3K99`. Écrivez un script Python qui télécharge de manière automatique les fichiers PDB (à l'aide du module `urllib2` et d'une boucle) et les enregistre dans le répertoire courant.
3. Reprenez le dictionnaire des mots de 4 lettres du génome du chromosome I de la levure S. cerevisiae (*exercice 7 sur les dictionnaires et tuples*). Encodez le dictionnaire dans un fichier avec le module `pickle`. Quittez Python et regardez attentivement le fichier généré avec un éditeur de texte. Relancez Python, et décodez le dictionnaire.

16 Modules d'intérêt en bioinformatique

Nous allons voir dans cette section quelques modules très importants en bioinformatique. Le premier `numpy` permet notamment de manipuler des vecteurs et des matrices en Python. Le module `biopython` permet de travailler sur des données biologiques type séquence (nucléique et protéique) ou structure (fichier PDB). Le module `matplotlib` permet de dessiner des graphiques depuis Python. Enfin, le module `rpy` permet d'interfacer n'importe quelle fonction du puissant programme R.

Ces modules ne sont pas fournis avec le distribution Python de base (contrairement à tous les autres modules vus précédemment). Nous ne nous étendrons pas sur la manière de les installer. Consultez pour cela la documentation sur les sites internet des modules en question. Sachez cependant que ces modules existent dans la plupart des distributions Linux récentes.

Dans ce chapitre, nous vous montrererons quelques exemples d'utlisation de ces modules pour vous convaincre de leur pertinence.

16.1 Module numpy

Le module numpy est incontournable en bioinformatique. Il permet d'effectuer des calculs sur des vecteurs ou des matrices, élément par élément, via un nouveau type d'objet appelé *array*. Ce module contient des fonctions de base pour faire de l'algèbre linéaire, des transformées de Fourier ou encore des tirages de nombre aléatoire plus sophistiqués qu'avec le module `random`. Vous pourrez trouver les sources de numpy à cette adresse. Notez qu'il existe un autre module scipy que nous n'aborderons pas dans ce cours. `scipy` est lui même basé sur `numpy`, mais il en étend considérablement les possibilités de ce dernier (*e.g.* statistiques, optimisation, intégration numérique, traitement du signal, traitement d'image, algorithmes génétiques, etc.).

16.1.1 Objets de type array

Les objets de type *array* correspondent à des tableaux à une ou plusieurs dimensions et permettent d'effectuer du calcul vectoriel. La fonction `array()` permet la conversion d'un objet séquentiel (type liste ou tuple) en un objet de type *array*. Voici un exemple simple de conversion d'une liste à une dimension en objet *array* :

```
>>> import numpy
>>> a = [1,2,3]
>>> numpy.array(a)
array([1, 2, 3])
>>> b = numpy.array(a)
>>> type(b)
<type 'numpy.ndarray'>
>>> b
array([1, 2, 3])
```

Nous avons converti la liste `a` en *array*, mais cela aurait donné le même résultat si on avait converti le tuple `(1,2,3)`. Par ailleurs, vous voyez que lorsqu'on demande à Python le contenu d'un objet *array*, les symboles `([])` sont utilisés pour le distinguer d'une liste `[]` ou d'un tuple `()`.

Notez qu'un objet *array* ne peut contenir que des valeurs numériques. Vous ne pouvez pas, par exemple, convertir une liste contenant des chaînes de caractères en objet de type *array*.

La fonction `arange()` est équivalente à `range()` et permet de construire un *array* à une dimension de manière simple.

```
>>> numpy.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> numpy.arange(10.0)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> numpy.arange(10,0,-1)
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
>>>
```

Un des avantages de la fonction `arange()` est qu'elle permet de générer des objets *array* qui contiennent des entiers ou de réels selon l'argument qu'on lui passe.

La différence fondamentale entre un objet *array* à une dimension et une liste (ou un tuple) est que celui-ci est considéré comme un **vecteur**. Par conséquent on peut effectuer des opérations **élément par élément** dessus, ce qui est bien commode lorsqu'on analyse de grandes quantités de données. Regardez ces exemples :

```
>>> v = numpy.arange(4)
>>> v
array([0, 1, 2, 3])
>>> v + 1
array([1, 2, 3, 4])
>>> v + 0.1
array([ 0.1,  1.1,  2.1,  3.1])
>>> v * 2
array([0, 2, 4, 6])
>>> v * v
array([0, 1, 4, 9])
```

Notez bien sur le dernier exemple de multiplication que l'*array* final correspond à la multiplication **élément par élément** des deux *array* initiaux. Avec les listes, ces opérations n'auraient été possibles qu'en utilisant des boucles ! Nous vous encourageons donc à utiliser dorénavent les objets *array* lorsque vous aurez besoin de faire des opérations élément par élément.

Il est aussi possible de construire des objets *array* à deux dimensions, il suffit de passer en argument une liste de listes à la fonction `array()` :

```
>>> numpy.array([[1,2,3],[2,3,4],[3,4,5]])
array([[1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])
>>>
```

Attention, plus complexe encore ! On peut aussi créer des tableaux à trois dimensions en passant à la fonction `array()` une liste de listes de listes :

```
>>> numpy.array([[[1,2],[2,3]],[[4,5],[5,6]]])
array([[[1, 2],
        [2, 3]],

       [[4, 5],
        [5, 6]]])
>>>
```

La fonction `array()` peut créer des tableaux à n'importe quel nombre de dimensions. Toutefois ça devient vite compliqué lorsqu'on dépasse trois dimensions. Retenez qu'un objet *array* à une dimension peut être considéré comme un **vecteur** et un *array* à deux dimensions comme une **matrice**.

Pour récupérer un ou plusieurs élément(s) d'un objet *array*, vous pouvez utiliser l'indiçage ou les tranchage, de la même manière que pour les listes.

```
>>> a = numpy.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[5:]
array([5, 6, 7, 8, 9])
>>> a[::2]
array([0, 2, 4, 6, 8])
>>> a[1]
1
```

Dans le cas d'un objet *array* à deux dimensions, vous pouvez récupérer une ligne, une colonne ou bien un seul élément.

```
>>> a = numpy.array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a[:,0]
array([1, 3])
>>> a[0,:]
array([1, 2])
>>> a[1,1]
4
```

La syntaxe `a[m,:]` récupère la ligne m-1, et `a[:,n]` récupère la colonne n-1. Les tranches sont évidemment aussi utilisables sur un tableau à deux dimensions.

Il peut être parfois pénible de construire une matrice (*array* à deux dimensions) à l'aide d'une liste de listes. Le module `numpy` contient quelques fonctions commodes pour construire des matrices à partir de rien. Les fonctions `zeros()` et `ones()` permettent de construire des objets *array* contenant des 0 ou de 1, respectivement. Il suffit de leur passer un tuple indiquant la dimensionnalité voulue.

```
>>> numpy.zeros((3,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> numpy.zeros((3,3),int)
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
>>> numpy.ones((3,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Par défaut, les fonctions `zeros()` et `ones()` génèrent des réels, mais vous pouvez demander des entiers en passant l'option `int` en second argument.

Enfin il existe les fonctions `reshape()` et `resize()` qui permettent de remanier à volonté les dimensions d'un *array*. Il faut pour cela, leur passer en argument l'objet *array* à remanier ainsi qu'un tuple indiquant la nouvelle dimensionnalité.

```
>>> a = numpy.arange(9)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> numpy.reshape(a,(3,3))
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Toutefois, `reshape()` attend un tuple dont la dimension est compatible avec le nombre d'éléments contenus dans l'*array* de départ, alors que `resize()` s'en moque et remplira le nouvel objet *array* généré même si les longueurs ne coincident pas.

```
>>> a = numpy.arange(9)
>>> a.reshape((2,2))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> numpy.resize(a,(2,2))
array([[0, 1],
       [2, 3]])
>>> numpy.resize(a,(4,4))
array([[0, 1, 2, 3],
       [4, 5, 6, 7],
       [8, 0, 1, 2],
       [3, 4, 5, 6]])
```

Dans l'exemple précédent, la fonction `reshape()` renvoie une erreur si les dimensions ne coïncident pas. La fonction `resize()` duplique ou coupe la liste initiale s'il le faut jusqu'à temps que le nouvel *array* soit rempli.

Si vous ne vous souvenez plus de la dimension d'un objet *array*, la fonction `shape()` permet d'en retrouver la taille.

```
>>> a = numpy.arange(3)
>>> numpy.shape(a)
(3,)
```

Enfin, la fonction `transpose()` renvoie la transposée d'un *array*. Par exemple pour une matrice :

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> numpy.transpose(a)
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

16.1.2 Un peu d'algèbre linéaire

Après avoir manipulé les vecteurs et les matrices, voici quelques fonctions pour faire de l'algèbre linéaire.

La fonction `dot()` vous permet de faire une multiplication de matrices.

```
>>> a = numpy.resize(numpy.arange(4),(2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> numpy.dot(a,a)
array([[ 2,  3],
       [ 6, 11]])
>>> a * a
array([[0, 1],
       [4, 9]])
```

Notez bien que `dot(a,a)` renvoie le **produit matriciel** entre deux matrices, alors que `a*a` renvoie le produit **élément par élément**[1].

Pour toutes les opérations suivantes, il faudra utiliser des fonctions dans le sous-module `numpy.linalg`. La fonction `inv()` renvoie l'inverse d'une matrice carrée, `det()` son déterminant, `eig()` ses vecteurs et valeurs propres.

```
>>> a
array([[0, 1],
       [2, 3]])
>>> numpy.linalg.inv(a)
array([[-1.5,  0.5],
       [ 1. ,  0. ]])
>>> numpy.linalg.det(a)
-2.0
>>> numpy.linalg.eig(a)
(array([-0.56155281,  3.56155281]), array([[-0.87192821, -0.27032301],
       [ 0.48963374, -0.96276969]]))
>>> numpy.linalg.eig(a)[0]
array([-0.56155281,  3.56155281])
>>> numpy.linalg.eig(a)[1]
array([[-0.87192821, -0.27032301],
       [ 0.48963374, -0.96276969]])
```

Notez que la fonction `eig()` renvoie un tuple dont le premier élément correspond aux valeurs propres et le second élément aux vecteurs propres.

16.1.3 Un peu de transformée de Fourier

La transformée de Fourier est très utilisée pour l'analyse de signaux, notamment lorsqu'on souhaite extraire des périodicités au sein d'un signal bruité. Le module `numpy` possède la fonction `fft()` (dans le sous-module `fft`) permettant de calculer des transformées de Fourier.

Voici un petit exemple sur la fonction cosinus de laquelle on souhaite extraire la période à l'aide de la fonction `fft()` :

1. Dans numpy, il existe également des objets de type *matrix* pour lesquels les multiplications de matrices sont différents, mais nous ne les aborderons pas ici.

```
# 1) on definit la fonction y = cos(x)
import numpy
debut = -2 * numpy.pi
fin = 2 * numpy.pi
pas = 0.1
x = numpy.arange(debut,fin,pas)
y = numpy.cos(x)

# 2) on calcule la TF de la fonction cosinus
TF=numpy.fft.fft(y)
ABSTF = numpy.abs(TF)
# abcisse du spectre en radian^-1
pas_xABSTF = 1/(fin-debut)
x_ABSTF = numpy.arange(0,pas_xABSTF * len(ABSTF),pas_xABSTF)
```

Plusieurs commentaires sur cet exemple.

- Vous constatez que numpy redéfinit certaines fonctions ou constantes mathématiques de base, comme `pi` (nombre π), `cos()` (fonction cosinus) ou `abs()` (valeur absolue, ou module d'un complexe). Ceci est bien pratique car nous n'avons pas à appeler ces fonctions ou constantes depuis le module `math`, le code en est ainsi plus lisible.
- Dans la partie 1), on définit le vecteur `x` représentant un angle allant de -2π à 2π radians par pas de 0,1 et le vecteur `y` comme le cosinus de `x`.
- En 2) on calcule la transformée de Fourier avec la fonction `fft()` qui renvoie un vecteur (objet *array* à une dimension) de nombres complexes. Eh oui, le module `numpy` gère aussi les nombres complexes ! On extrait ensuite le module du résultat précédent avec la fonction `abs()`.
- La variable `x_ABSTFL` représente l'abscisse du spectre (en radian^{-1}).
- La variable `ABSTF` contient le spectre lui même. L'analyse de ce dernier nous donne un pic à 0,15 radian^{-1}, ce qui correspond bien à 2π (plutôt bon signe de retrouver ce résultat). Le graphe de ce spectre est présenté dans la partie dédiée à `matplotlib` (section 16.3).

Notez que tout au long de cette partie, nous avons toujours utilisé la syntaxe `numpy.fonction()` pour bien vous montrer quelles étaient les fonctions propres à `numpy`. Bien sûr dans vos futurs scripts il sera plus commode d'importer complètement le module `numpy` avec l'instruction `from numpy import *`. Vous pourrez ensuite appeler les fonctions de `numpy` directement (sans le préfixe `numpy.`).

Si vous souhaitez quand même spécifier pour chaque fonction `numpy` son module d'appartenance, vous pouvez définir un alias pour `numpy` avec l'instruction `import numpy as np`. Le module `numpy` est alors connu sous le nom `np`. Par l'appel de la fonction `array()` se fera par `np.array()`.

16.2 Module biopython

Le module biopython propose de nombreuses fonctionnalités très utiles en bioinformatique. Le tutoriel est particulièrement bien fait, n'hésitez pas à le consulter.

Voici quelques exemples d'utilisation.

Définition d'une séquence.

```
>>> import Bio
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> ADN = Seq("ATATCGGCTATAGCATGCA", IUPAC.unambiguous_dna)
>>> ADN
Seq('ATATCGGCTATAGCATGCA', IUPACUnambiguousDNA())
```

`IUPAC.unambiguous_dna` signifie que la séquence entrée est bien une séquence d'ADN.

Obtention de la séquence complémentaire et complémentaire inverse.

```
>>> ADN.complement()
Seq('TATAGCCGATATCGTACGT', IUPACUnambiguousDNA())
>>> ADN.reverse_complement()
Seq('TGCATGCTATAGCCGATAT', IUPACUnambiguousDNA())
```

Traduction en séquence protéique.

```
>>> ADN.translate()
Seq('ISAIAC', IUPACProtein())
```

16.3 Module matplotlib

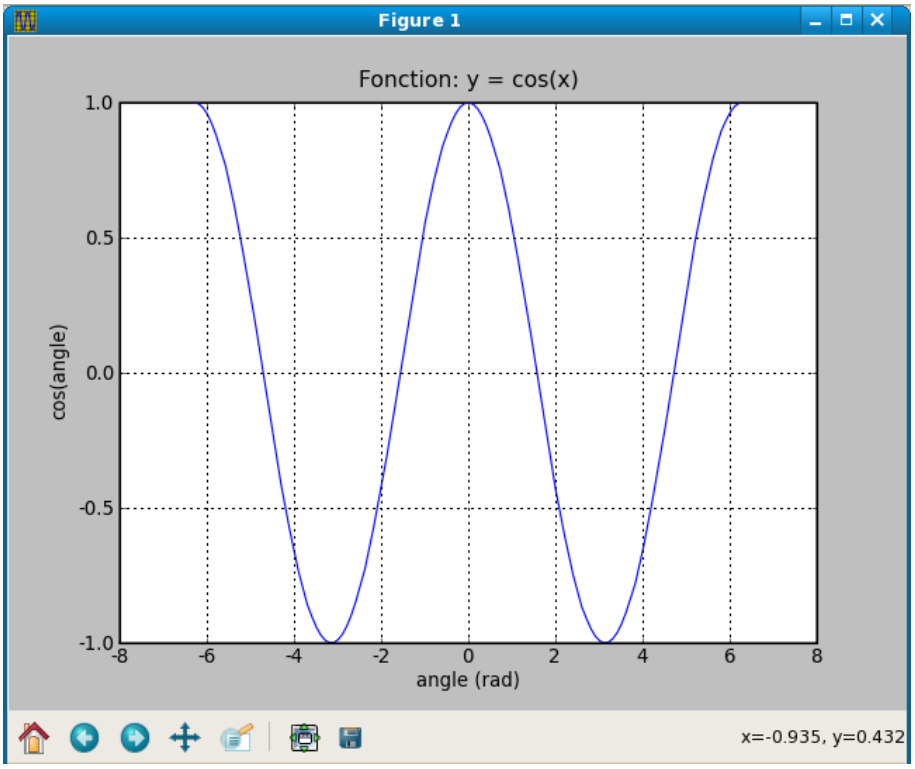
Le module matplotlib permet de générer des graphes interactifs depuis Python. Il est l'outil complémentaire de `numpy` et `scipy` lorsqu'on veut faire de l'analyse de données.

Nous ne présenterons ici qu'un petit exemple traitant de la représentation graphique de la fonction cosinus et de la recherche de sa période par transformée de Fourier (voir également la section 16.1.3).

Plutôt qu'un long discours, regardez cet exemple que nous commenterons après.

```
# define cosine function
from pylab import *
debut = -2 * pi
fin = 2 * pi
pas = 0.1
x = arange(debut,fin,pas)
y = cos(x)

# draw the plot
plot(x,y)
xlabel('angle (rad)')
ylabel('cos(angle)')
title('Fonction: y = cos(x)')
grid()
show()
```

Vous devriez obtenir une image comme celle-ci :



Vous constatez que le module `matplotlib` génère un fenêtre graphique **interactive** permettant à l'utilisateur de votre script de manipuler le graphe (enregistrer comme image, zoomer, etc.).

Revenons maintenant sur le code. Tout d'abord, vous voyez qu'on importe le module s'appelle `pylab` (et non pas `matplotlib`). Le module `pylab` importe lui-même toutes les fonctions (et variables) du module `numpy` (*e.g.* `pi`, `cos`, `arange`, etc.). Il est plus commode de l'importer par `from pylab import *` que par `import pylab`.

La partie qui nous intéresse (après la ligne `# draw the plot`) contient les parties spécifiques à `matplotlib`. Vous constatez que les commandes sont très intuitives.

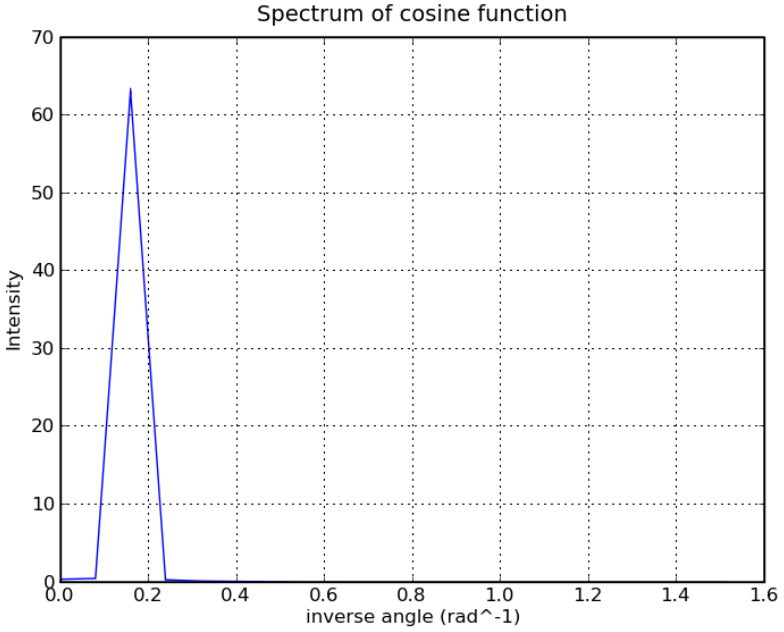
- La fonction `plot()` va générer un graphique avec des lignes et prend comme valeurs en abscisse (`x`) et en ordonnées (`y`) des vecteurs de type *array* à une dimension.
- Les fonctions `xlabel()` et `ylabel()` sont utiles pour donner un nom aux axes.
- `title()` permet de définir le titre du graphique.
- `grid()` affiche une grille en filligrane.
- Jusqu'ici, aucun graphe n'est affiché. Pour activer l'affichage à l'écran du graphique, il faut appeler la fonction `show()`. Celle-ci va activer une boucle dite *gtk* qui attendra les manipulations de l'utilisateur.
- Les commandes Python éventuellement situées après la fonction `show()` seront exécutées seulement lorsque l'utilisateur fermera la fenêtre graphique (petite croix en haut à droite).

Voici maintenant l'exemple complet sur la fonction cosinus et sa transformée de Fourier.

```
from pylab import *

# define cosine function
x = arange(-2*pi,2*pi,0.1)
y = cos(x)

# calculate TF of cosine function
TF=fft(y)
```

```
ABSTF = abs(TF)
pas_xABSTF = 1/(4*pi)
x_ABSTF = arange(0,pas_xABSTF * len(ABSTF),pas_xABSTF)

# draw cos plot
plot(x,y)
xlabel('angle (rad)')
ylabel('cos(angle)')
title('Fonction: y = cos(x)')
grid()
show()

# plot TF of cosine
plot(x_ABSTF[:20],ABSTF[:20])
xlabel('inverse angle (rad^-1)')
ylabel('Intensity')
title('Spectrum of cosine function')
grid()
show()
```

Le premier graphe doit afficher la fonction cosinus comme ci-dessus et le second doit ressembler à cela :



On retrouve bien le pic à 0,15 radian^{-1} correspondant à 2π radians. Voilà, nous espèrons que ce petit exemple vous aura convaincu de l'utilité du module `matplotlib`. Sachez qu'il peut faire bien plus, par exemple générer des histogrammes ou toutes sortes de graphiques utiles en analyse de données.

16.4 Module rpy

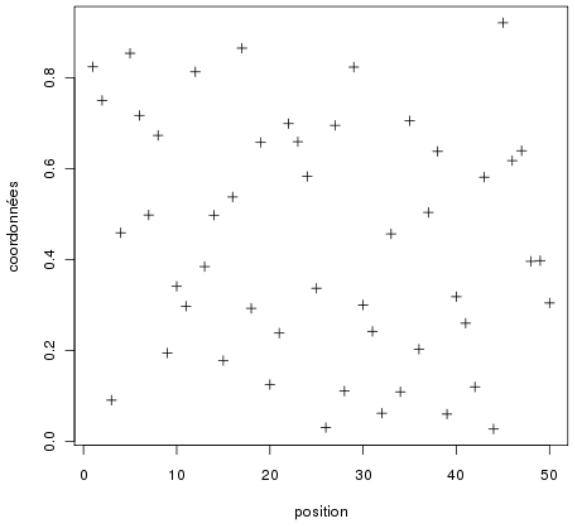
R est un programme extrêmement puissant permettant d'effectuer des analyses statistiques. Il contient tout un tas de fonctions permettant de générer divers types de graphiques. Nous

ne rentrerons pas dans les détails de R, mais nous vous montrerons quelques exemples pour générer des graphiques avec R depuis Python.

Les vecteurs de R peuvent être remplacés par des listes en Python. En interne, `rpy` manipule des variables de types *array* car il est basé sur le module `numpy` vu précédemment. Dans cet exemple, nous allons tracé les coordonnées aléatoires de 50 points.

```
import random
import rpy
# construction d'une liste de 50 éléments croissants
x = range(1, 51)
# construction d'une liste de 50 éléments aléatoires
y = []
for i in x:
    y.append( random.random() )

# enregistrement du graphique dans un fichier png
rpy.r.png("rpy_test1.png")
# dessin des points
rpy.r.plot(x, y, xlab="position", ylab="coordonnées", col="black", pch=3)
# fin du graphique
rpy.r.dev_off()
```

Voici le fichier `rpy_test1.png` obtenu :



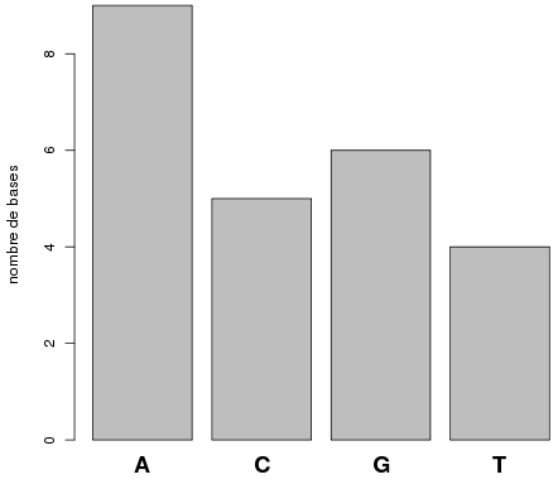
Les fonctions R sont accessibles par le sous-module `r` du module `rpy`. Les fonctions `png()` et `plot()` sont utilisables comme en R. Les fonctions qui contiennent un point dans leur nom en R (`dev.off()`) sont utilisables avec un caractère souligné « _ » à la place (ici `dev_off()`).

Voyons maintenant un exemple plus intéressant, pour lequel on calcule la distribution des différentes bases dans une séquence nucléique :

```
from rpy import r as R
seq = "ACGATCATAGCGAGCTACGTAGAA"
seq2 = list( seq )
```

```
R.png("rpy_test2.png")
# tri des bases car unique() n'ordonne pas les données
# alors que table() le fait
seq3 = R.sort( seq2 )
# listes des bases présentes
bases = R.unique( seq3 )
# effectif de chaque base
effectifs = R.table( seq3 )
# dessin du barplot et sauvegarde de la position des abscisses
coords = R.barplot( effectifs, ylab="nombre de bases")
# ajout du texte pour l'axe des abscisses
R.text(coords, -0.5, bases, xpd = True, cex = 1.5, font = 2 )
R.dev_off()
```

Voici le fichier `rpy_test2.png` obtenu :



Pour plus de concision, le module `rpy.r` est renomé en `R`. Les booléens `TRUE` et `FALSE` en R (dans la fonction `text()`) doivent être remplacés par leurs équivalents en Python (`True` et `False` ; attention à la casse).

16.5 Exercice numpy

Cet exercice présente le calcul de la distance entre deux carbones alpha consécutifs de la barstar. Il demande quelques notions d'Unix et nécessite le module `numpy` de Python.

1. Téléchargez le fichier `1BTA.pdb` sur le site de la PDB.
2. Pour que le séparateur décimal soit le point (au lieu de la virgule, par défaut sur les systèmes d'exploitation français), il faut au préalable redéfinir la variable `LC_NUMERIC` en bash :

   ```
   export LC_NUMERIC=C
   ```

   Extrayez les coordonnées atomiques de tous les carbones alpha de la barstar dans le fichier `1BTA_CA.txt` avec la commande Unix suivante :

   ```
   awk '$1=="ATOM" && $3=="CA" {printf "%.3f %.3f %.3f ", $7, $8, $9}' 1BTA.pdb > 1BTA_CA.txt
   ```

   Les coordonnées sont toutes enregistrées sur une seul ligne, les unes après les autres, dans le fichier `1BTA_CA.txt`.

3. Ouvrez le fichier `1BTA_CA.txt` avec Python et créez une liste contenant toutes les coordonnées sous forme de réels avec les fonctions `split()` et `float()`.
4. Avec la fonction `array()` du module `numpy`, convertissez cette liste en matrice.
5. Avec la fonction `reshape()` de `numpy`, et connaissant le nombre d'acides aminés de la barstar, construisez une matrice à deux dimensions contenant les coordonnées des carbones alpha.
6. Créez une matrice qui contient les coordonnées des n-1 premiers carbones alpha et une autre qui contient les coordonnées des n-1 derniers carbones alpha.
7. En utilisant les opérateurs mathématiques habituels (-, **2, +) et les fonctions `sqrt()` et `sum()` du module `numpy`, calculez la distance entre les atomes n et n+1.
8. Affichez les distances entre carbones alpha consécutifs et repérez la valeur surprenante.

16.6 Exercice rpy

1. Soit la séquence protéique WVAAGALTIWPILGALVILG. Représentez avec le module `rpy` la distribution des différents acides aminés.
2. Reprenez l'exercice du calcul de la distance entre deux carbones alpha consécutifs de la barstar avec `numpy` et représentez cette distance avec `rpy`.

17 Avoir la classe avec les objets

Une classe permet de définir des objets qui sont des représentants (des instances) de cette classe. Les objets peuvent posséder des attributs (variables associées aux objets) et des méthodes ($\sim$ fonctions associées aux objets).

Exemple de la classe `Rectangle` :

```
class Rectangle:
    "ceci est la classe Rectangle"
    # initialisation d'un objet
    # définition des attributs avec des valeurs par défaut
    def __init__(self, long = 0.0, larg = 0.0, coul = "blanc"):
        self.longueur = long
        self.largeur = larg
        self.couleur = coul
    # definition de la méthode qui calcule la surface
    def calculSurface(self):
        print "surface = %.2f m2" %(self.longueur * self.largeur)
    # definition de la méthode qui transforme un rectangle en carré
    def changeCarre(self, cote):
        self.longueur = cote
        self.largeur = cote
```

Ici, `longueur`, `largeur` et `couleur` sont des attributs alors que `calculPerimetre()`, `calculSurface()` et `changeCarre()` sont des méthodes. Tous les attributs et toutes les méthodes se réfèrent toujours à `self` qui désigne l'objet lui même. Attention, les méthodes prennent toujours au moins `self` comme argument.

Exemples d'utilisation de la classe `Rectangle` :

```
# création d'un objet Rectangle avec les paramètres par défaut
rect1 = Rectangle()
print rect1.longueur, rect1.largeur, rect1.couleur
# sans surprise :
rect1.calculSurface()
# on change le rectangle en carré
rect1.changeCarre(30)
rect1.calculSurface()
# création d'un objet Rectangle avec des paramètres imposés
rect2 = Rectangle(2, 3, "rouge")
rect2.calculSurface()
```

17.1 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

1. Entraînez-vous avec la classe `Rectangle`. Créez la méthode `calculPerimetre()` qui calcule le périmère d'un objet rectangle.
2. Créez une nouvelle classe `Atome` avec les attributs `x`, `y`, `z` (qui contiennent les coordonnées atomique) et la méthode `calculDistance()`. Testez cette classe sur plusieurs exemples.
3. Améliorez la classe `Atome` avec de nouveaux attributs (par ex. : `masse`, `charge`, etc.) et de nouvelles méthodes (par ex. : `calculCentreMasse()`,`calculRayonGyration()`).

18 Gestion des erreurs

La gestion des erreurs permet d'éviter que votre programme plante en prévoyant vous même les sources d'erreurs éventuelles.

Voici un exemple dans lequel on demande à l'utilisateur d'entrer un nombre, puis on affiche ce nombre.

```
>>> nb = int(raw_input("Entrez un nombre: "))
Entrez un nombre: 23
>>> print nb
23
```

La fonction `raw_input()` permet à l'utilisateur de saisir une chaîne de caractères. Cette chaîne de caractères est ensuite transformée en nombre entier avec la fonction `int()`.

Si l'utilisateur ne rentre pas un nombre, voici ce qui se passe :

```
>>> nb = int(raw_input("Entrez un nombre: "))
Entrez un nombre: ATCG
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'ATCG'
```

L'erreur provient de la fonction `int()` qui n'a pas pu convertir la chaîne de caractères `"ATCG"` en nombre, ce qui est normal.

Le jeu d'instruction `try / except` permet de tester l'exécution d'une commande et d'intervenir en cas d'erreur.

```
>>> try:
...     nb = int(raw_input("Entrez un nombre: "))
... except:
...     print "Vous n'avez pas entré un nombre !"
...
Entrez un nombre: ATCG
Vous n'avez pas entré un nombre !
```

Dans cette exemple, l'erreur renvoyée par la fonction `int()` (qui ne peut pas convertir `"ATCG"` en nombre) est interceptée et déclenche l'affichage du message d'avertissement.

On peut ainsi redemander sans cesse un nombre à l'utilisateur, jusqu'à ce que celui-ci en rentre bien un.

```
>>> while 1:
...     try:
...             nb = int(raw_input("Entrez un nombre: "))
...             print "Le nombre est", nb
...             break
...     except:
...             print "Vous n'avez pas entré un nombre !"
...             print "Essayez encore"
...
Entrez un nombre: ATCG
Vous n'avez pas entré un nombre !
Essayez encore
Entrez un nombre: toto
```

```
Vous n'avez pas entré un nombre !
Essayez encore
Entrez un nombre: 55
Le nombre est 55
```

Notez que dans cet exemple, l'instruction `while 1` est une boucle infinie (car la condition 1 est toujours vérifiée) dont l'arrêt est forcé par la commande `break` lorsque l'utilisateur a effectivement bien rentré un nombre.

La gestion des erreurs est très utile dès lors que des données extérieures entrent dans le programme, que ce soit directement par l'utilisateur (avec la fonction `raw_input()`) ou par des fichiers.

Vous pouvez par exemple vérifier qu'un fichier a bien été ouvert.

```
>>> nom = "toto.pdb"
>>> try:
...     f = open(nom, "r")
... except:
...     print "Impossible d'ouvrir le fichier", nom
```

Si une erreur est déclenchée, c'est sans doute que le fichier n'existe pas à l'emplacement indiqué sur le disque ou que (sous Unix), vous n'ayez pas les droits d'accès pour le lire.

Il est également possible de spécifier le type d'erreur à gérer. Le premier exemple que nous avons étudié peut s'écrire :

```
>>> try:
...     nb = int(raw_input("Entrez un nombre: "))
... except ValueError:
...     print "Vous n'avez pas entré un nombre !"
...
Entrez un nombre: ATCG
Vous n'avez pas entré un nombre !
```

Ici, on intercepte une erreur de type `ValueError`, ce qui correspond bien à un problème de conversion avec `int()`. Il existe d'autres types d'erreurs comme `RuntimeError`, `TypeError`, `NameError`, `IOError`, etc.

Enfin, on peut aussi être très précis dans le message d'erreur. Observez la fonction `downloadPage()` qui, avec le module `urllib2`, télécharge un fichier sur internet.

```
import urllib2

def downloadPage(address):
    error = ""
    page = ""
    try:
        data = urllib2.urlopen(address)
        page = data.read()
    except IOError, e:
        if hasattr(e, 'reason'):
            error =  "Cannot reach web server: " + str(e.reason)
        if hasattr(e, 'code'):
            error = "Server failed %d" %(e.code)
    return page, error
```

La variable `e` est une instance (un représentant) de l'erreur de type `IOError`. Certains de ces attributs sont testés avec la fonction `hasattr()` pour ainsi affiner le message renvoyé (ici contenu dans la variable `error`).

19 Trucs et astuces

19.1 Shebang et /usr/bin/env python

Lorsque vous programmez sur un sytème Unix, le shebang correspond aux caractères `#!` qui se trouve au début de la première ligne d'un script. Le shebang est suivi du chemin complet du programme qui interprète le script.

En Python, on trouve souvent la notation

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

Cependant, l'exécutable `python` ne se trouve pas toujours dans le répertoire `/usr/bin`. Pour maximiser la portabilité de votre script Python sur plusieurs systèmes Unix, utilisez plutôt cette notation :

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

Dans le cas présent, on appelle le programme d'environnement `env` (qui se situe toujours dans le répertoire `/usr/bin`) pour lui demander où se trouve l'exécutable `python`.

19.2 Python et utf-8

Si vous utilisez des caractères accentués dans des chaînes de caractères ou bien même dans des commentaires, cela occasionnera une erreur lors de l'exécution de votre script.

Pour éviter ce genre de désagrément, ajoutez la ligne suivante à la deuxième ligne (la position est importante) de votre script :

```
# -*- coding: utf-8 -*-
```

En résumé, tous vos scripts Python devraient ainsi débuter par les lignes :

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
```

19.3 Vitesse d'itération dans les boucles

La vitesse d'itération (de parcours) des élements d'une liste peut être très différente selon la structure de boucle utilisée. Pour vous en convaincre, copiez les lignes suivantes dans un script Python (par exemple `boucles.py`) puis exécutez-le.

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import time

# création d'une liste de 5 000 000 d'éléments
# (à adapter suivant la vitess de vos machines)
taille = 5000000
print "Création d'une liste avec %d élements" %( taille )
toto = range( taille )

# la variable 'a' accède à un élément de la liste

# méthode 1
start = time.time()
```

```
for i in range( len(toto) ):
    a = toto[i]
print "méthode 1 (for in range) : %.1f secondes" %( time.time() - start )

# méthode 2
start = time.time()
for ele in toto:
    a = ele
print "méthode 2 (for in) : %.1f secondes" %( time.time() - start )

# méthode 3
start = time.time()
for i in xrange( len(toto) ):
    a = toto[i]
print "méthode 3 (for in xrange) : %.1f secondes" %( time.time() - start )

# méthode 4
start = time.time()
for idx, ele in enumerate( toto ):
    a = ele
print "méthode 4 (for in enumerate): %.1f secondes" %( time.time() - start )
```

Vous devriez obtenir une sortie similaire à celle-ci :

```
poulain@cumin> ./boucles.py
Création d'une liste avec 5000000 élements
méthode 1 (for in range) : 1.8 secondes
méthode 2 (for in) : 1.0 secondes
méthode 3 (for in xrange) : 1.2 secondes
méthode 4 (for in enumerate): 1.4 secondes
```

La méthode la plus rapide pour parcourir une liste est donc d'itérer directement sur les éléments (`for element in liste`). Cette instruction est à privilégier le plus possible.

La méthode `for i in range(len(liste))` est particulièrement lente car la commande `range(len(liste))` génère une énorme liste avec tous les indices des éléments (la création de liste est assez lente en Python). Si vous voulez absolument parcourir une liste avec les indices des élements, utilisez plutôt la commande `for i in xrange(len(liste))` car l'instruction `xrange()` ne va pas créer une liste mais incrémenter un compteur qui correspond à l'indice des éléments successifs de la liste.

Enfin, la commande `for indice, element in enumerate(liste)` est particulièrement efficace pour récupérer en même temps l'élément et son indice.

19.4 Liste de compréhension

Une manière originale et très puissante de générer des listes est la compréhension de liste. Pour plus de détails, consultez à ce sujet le site de Python et celui de wikipédia.

Voici quelques exemples :

– Nombres pairs compris entre 0 et 99
  `print [i for i in range(99) if i%2 == 0]`
  Le même résultat est obtenu avec
  `print range(0,99,2)`
  ou

```
print range(0,99)[::2]
```

– Jeu sur la casse des mots d'une phrase.

```
message = "C'est sympa la BioInfo"
msg_lst = message.split()
print [[m.upper(), m.lower(), len(m)] for m in msg_lst]
```

– Formatage d'une séquence avec 60 caractères par ligne

```
# exemple d'une séquence de 150 alanines
seq = "A"*150
width= 60
print "\n".join( [seq[i:i+width] for i in range(0,len(seq),width)] )
```

Formatage fasta d'une séquence (avec la ligne de commentaire)

```
commentaire = "mon commentaire"
# exemple d'une séquence de 150 alanines.
seq = "A"*150
width= 60
print "> "+commentaire+"\n"+"\n".join( [seq[i:i+width] for i in range(0,len(seq),width)] )
```

– Sélection des lignes correspondantes aux carbones alpha dans un fichier pdb

```
# ouverture du fichier pdb (par exemple 1BTA.pdb)
pdb = open("1BTA.pdb", "r")
# sélection des lignes correspondantes aux C alpha
CA_line = [line for line in pdb if line.split()[0] == "ATOM" and line.split()[2] == "CA"]
# fermeture du fichier
pdb.close()
# éventuellement affichage des lignes
print CA_line
```

19.5 Sauvegardez votre historique de commandes

Vous pouvez sauvegarder l'historique des commandes utilisées dans l'interpréteur Python avec le module `readline`.

```
>>> print "hello"
hello
>>> a = 22
>>> a = a + 11
>>> print a
33
>>> import readline
>>> readline.write_history_file()
```

Quittez Python. L'historique de toutes vos commandes est dans votre répertoire personnel, dans le fichier `.history`. Relancez l'interpréteur Python.

```
>>> import readline
>>> readline.read_history_file()
```

Vous pouvez accéder aux commandes de la session précédente avec la flèche du haut de votre clavier.

```
>>> print "hello"
hello
>>> a = 22
>>> a = a + 11
>>> print a
33
```