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.