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
```