1 Un exemple assez subtil...

Obtenir d'un utilisateur qu'il entre une valeur entière en le faisant recommencer autant que nécessaire :

```
>>> def getInteger (message) :
...     OK = False
...     while not OK :
...         print("N'entrez que des chiffres SVP")
...         reponse = input(message + " : ")
...         i = 0
...         while i < len(reponse) and reponse[i] in "0123456789":
...             i = i + 1
...         OK = (i == len(reponse))
...     return(int(reponse))
```

On donne en argument de la fonction la demande faite à l'utilisateur :

```
>>> getInteger ("Donnez votre date de naissance")
N'entrez que des chiffres SVP
Donnez votre date de naissance : année 1980
N'entrez que des chiffres SVP
Donnez votre date de naissance : 1980
1980
```

2 Parcours de chaîne de caractères avec `for`

La primitive `for` permet de parcourir une chaîne de caractères du début à la fin en énumérant les caractères les uns après les autres, de la gauche vers la droite.

Reprenons l'exemple de `brinCompl` et commençons par remarquer que cette autre version, qui utilise toujours un `while`, est équivalente à la précédente parce qu'elle concatène à gauche de `resultat` :

```
>>> def brinCompl (c) :
...     resultat = ""
...     i = 0
...     while i < len(c) :
...         resultat = complementaire(c[i]) + resultat
...         i = i + 1
...     return(resultat)
```

Cette nouvelle version est d'un certain point de vue plus simple que la première version car elle parcourt la chaîne de caractères « dans le sens normal » de gauche à droite, ainsi, l'indice `i` part de 0 et augmente de 1 en 1 jusqu'à la longueur de `c` moins 1.

Ce qui reste pénible dans cette version, c'est qu'on passse du temps à réfléchir comment gérer proprement l'indice `i` : ne pas oublier d'initialiser l'indice `i` ; faut-il commencer à 0 ou à 1 ? faut-il finir les tours de boucle `while` avec `i=len(c)` ou `i=len(c)-1` ? faut-il mettre « inférieur ou égal » ou un « inférieur strict » ?

Dans les cas où l'on doit parcourir la chaîne de caractère *du début à la fin* en prenant les caractères les uns après les autres, on peut utiliser la primitive « `for` » qui évite de gérer l'indice `i`. En effet, la seule chose qui nous intéresse dans le programme précédent, ce n'est pas vraiment la valeur de `i`, c'est le *caractère* `c[i]`de la chaîne `c`. Une boucle `for` permet justement de ne pas gérer l'indice `i` du tout : l'ordinateur le fera de manière cachée afin de nous fournir directement les caractères les uns après les autres.

Ainsi, par définition, la version ci-dessous est équivalente à la précédente :

```
>>> def brinCompl (c) :
...     resultat = ""
...     for n in c :
...         resultat = complementaire(n) + resultat
...     return(resultat)
```

La variable `n` dans cette version remplace avantageusement le nucléotide qu'était précédemment `c[i]` : on n'a plus besoin de faire appel à un entier (`i`) pour obtenir chaque caractère (`c[i]`). La variable `n` vaut successivement, à chaque tour de la boucle, les caractères de la chaîne `c` du début à la fin. Il y a donc autant de tour de boucle `for` que de caractères dans `c` et le programmeur n'a plus à réfléchir sur les questions à propos de l'un indice `i`.
Nota : en revanche, l'indice `i` n'existe plus dans une boucle `for`, de sorte que si on en a besoin, il faut continuer à gérer une boucle `while` « à la main » !

3 Encore des exemples

Calcul du pourcentage de `G` ou `C` dans un brin d'ADN :

```
>>> def teneurGC(b) :
...     if len(b) == 0 :
...         print("Le brin est vide!")
...     else :
...         nbGC = 0
...         for n in b :
...             if n in "GC" :
...                 nbGC = nbGC + 1
...         return( (nbGC * 100) // len(b) )
```

Ne garder que les voyelles minuscules non accentuées d'une chaîne de caractères :

```
>>> def extraitvoyelles(s):
...   v=""
...   for c in s:
...     if c in "aeiouy":
...       v = v + c
...   return(v)
...
>>> extraitvoyelles("Oui ce truc va marcher")
'uieuaae'
```

Tester si une chaîne de caractères représente de l'ADN, de l'ARN ou aucun des deux :

```
>>> def nature(b):
...   possibleADN=True
...   possibleARN=True
...   for n in b:
...     if not(n in "ATGC"):
...       possibleADN=False
...     if not(n in "AUGC"):
...       possibleARN=False
...   if possibleADN:
...     if possibleARN:
...       return("ADN ou ARN")
...     else:
...       return("ADN")
...   elif possibleARN:
...     return("ARN")
...   else:
...     return("ni ADN, ni ARN")
```

```
...
>>> nature("AAATTGGCCAA")
'ADN'
>>> nature("AUUGCGGCCAA")
'ARN'
>>> nature("AGCGGCCAA")
'ADN ou ARN'
>>> nature("AGCGGUT")
'ni ADN, ni ARN'
```

4 Les listes

Une *liste* en Python est une suite finie d'élements quelconques. Le type des listes est appelé `list`.

L'ensemble des listes est constitué des expressions de la forme

$$[e_1, \ldots, e_n]$$

où les e_i sont des données absolument quelconques (elles peuvent même être aussi elles-mêmes des listes).

```
>>> [2+3,"toto",2.5]
[5, 'toto', 2.5]
>>> type([5,"toto",2.5])
<type 'list'>
>>> [5,[9,5.5],"toto"]
[5, [9, 5.5], 'toto']
>>> type([5,[9,5.5],"toto"])
<type 'list'>
```

Il se peut qu'il n'y ait aucun élément dans la liste ; il s'agit alors de la *liste vide*, notée `[]`.

Beaucoup d'opérations travaillent sur les listes, à commencer par les opérations d'accès direct que nous venons de voir sur les chaînes de caractères. Ces dernières se transcrivent sur les listes de manière naturelle :

```
>>> ["toto",5,"tutu",5.5][1]
5
>>> ["toto",5,"tutu",5.5][1:3]
[5, 'tutu']
```

Le test d'appartenance ne fonctionne que pour un unique élémentdans une liste, et non pas pour une sous-liste contrairement aux chaînes de caractères.

```
>>> "glop" in ["toto",5,"tutu",5.5]
False
>>> "tutu" in ["toto",5,"tutu",5.5]
True
>>> "tu" in ["toto",5,"tutu",5.5]
False
>>> [5,"tutu"] in ["toto",5,"tutu",5.5]
False
>>> [5,"tutu"] in ["toto",[5,"tutu"],5.5]
True
```

Comme pour les chaînes de caractères, on trouve également les opérations : `len` pour la longueur (nombre d'éléments dans la liste) et `+` pour la concaténation de deux listes.

```
>>> def homogene (l) :
```

```
...     if len(l) == 0 :
...         return(True)
...     else :
...         t = type(l[0])
...         i = 1
...         vu = True
...         while vu and i < len(l) :
...             vu = vu and t == type(l[i])
...             i = i + 1
...         return(vu)
...
>>> homogene (["toto",5,"tutu",5.5])
False
>>> homogene (["toto","glop","titi"])
True
>>> homogene ([])
True
```

Lorsqu'une liste `l` est homogène, les opérations `min(l)` et `max(l)` fournissent respectivement le plus petit et le plus grand élément de la liste.

On peut également énumérer les éléments d'une liste homogène avec`for` :

```
>>> def somme (l) :
...     s = 0
...     for n in l :
...         s = s + n
...     return(s)
...
>>> somme ([])
0
>>> somme ([2,2,5,3])
12
>>> min ([2,2,5,3])
2
>>> max ([2,2,5,3])
5
```

5 Instructions de modification de liste

> ***Nota :*** *cette section du poly n'est pas développée en cours et ne sera pas au programme de l'examen. En revanche ces opérations sur les listes peuvent être utiles en pratique.*

Contrairement aux chaînes de caractères, dont les caractères ne peuvent pas être modifiés individuellement, une variable de type liste peut être partiellement modifiée sans réaffecter toute la variable. Par exemple :

```
>>> coursesAfaire=["beurre","pain","artichaut","vin rouge"]
>>> coursesAfaire
['beurre', 'pain', 'artichaut', 'vin rouge']
>>> coursesAfaire[2]="avocat"
>>> coursesAfaire
['beurre', 'pain', 'avocat', 'vin rouge']
```

Ainsi, si v est une variable de type liste, alors l'instruction « v`[`i`]=`*expr* » remplace l'élément d'indice i dans la liste par la valeur du calcul de l'expression *expr*.

En revanche, avec des chaînes de caractères c'est impossible :

```
>>> nom="Samon"
```

```
>>> nom[1]
'a'
>>> nom[1]='u'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

On peut aussi remplacer une portion entière de liste par une autre liste :

```
>>> coursesAfaire[1:3]=["baguette","chou","vinaigre"]
>>> coursesAfaire
['beurre', 'baguette', 'chou', 'vinaigre', 'vin rouge']
```

Remarquez que là aussi, dans l'instruction « v`[`i`:`j`]=`*expr* » on remplace les indices de i à $(j-1)$...

Il est possible de supprimer un élément ou une tranche d'éléments d'une variable de type liste avec `del` :

```
>>> coursesAfaire = ['beurre', 'chou', 'vinaigre', 'crème', 'pommes', 'vin rouge']
>>> del coursesAfaire[2]
>>> coursesAfaire
['beurre', 'vinaigre', 'crème', 'pommes', 'vin rouge']
>>> del coursesAfaire[2:4]
>>> coursesAfaire
['beurre', 'vinaigre', 'vin rouge']
```

Évitez à tout prix de prendre des indices hors des bornes, le comportement n'est pas garanti ; cela ne produit pas forcément une erreur !

```
>>> del coursesAfaire[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>> del coursesAfaire[1:6]
>>> coursesAfaire
['beurre']
```

En revanche, il est possible de donner des indices négatifs : cela signifie « en partant de la fin ». Ainsi « v`[-`i`]=`*expr* » est équivalent à « v`[len(`v`)-`i`]=`*expr* » :

```
>>> coursesAfaire = ["beurre","pain","artichaut","vin rouge"]
>>> coursesAfaire[-1]
'vin rouge'
>>> coursesAfaire[1:-1]
['pain', 'artichaut']
>>> del coursesAfaire[1:-1]
>>> coursesAfaire
['beurre', 'vin rouge']
```

Enfin pour les tranches d'éléments (avec le « : »), un indice manquant va jusqu'au bout de la liste :

```
>>> coursesAfaire = ["beurre","pain","artichaut","vin rouge"]
>>> coursesAfaire[1:]
['pain', 'artichaut', 'vin rouge']
>>> coursesAfaire[2:]
['artichaut', 'vin rouge']
>>> coursesAfaire[:2]
['beurre', 'pain']
>>> coursesAfaire[:]
['beurre', 'pain', 'artichaut', 'vin rouge']
```

6 Exemples de programme sur les listes

Ecrire une fonction `combien` qui prend en entrée deux arguments : le premier est une valeur `e` de type quelconque et le second est une liste `l`. Cette fonction doit retourner en résultat le nombre de fois où l'élément `e` apparaît dans la liste `l` (en particulier 0 fois si l'élément n'apparaît jamais dans la liste).

```
>>> def combien (e,l) :
...     compteur = 0
...     for x in l :
...         if x == e :
...             compteur = compteur + 1
...     return(compteur)
```

Extraire la liste des codons (entiers) d'un brin d'ADN pour une phase donnée :

```
>>> def codons (b,p):
...   if p <= 0 or p > 3:
...     print("Phase entre 1 et 3 SVP !")
...   else:
...     l = []
...     i = p - 1
...     while i + 3 <= len(b) :
...       l = l + [b[i:i+3]]
...       i = i + 3
...     return(l)
```