1 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 » !

2 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'
```

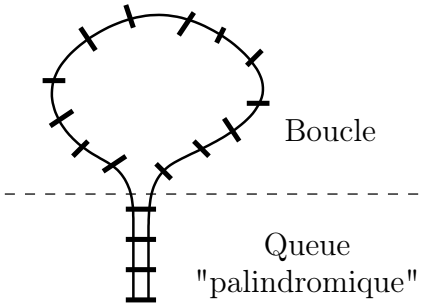
Tester si une chaîne de caractères représente de l'ADN, de l'ARN ou aucun des deux :

```
>>> 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'
```

Une fonction `palindrome` qui prend en entrée une chaîne de caractères et fournit en sortie un booléen qui dit si c'est un palindrome.

```
def palindrome(s):
   d=0
   f=len(s) - 1
   while d < f :
     if s[d] != s[f] :
       return(False)
     d=d+1
     f=f-1
   return(True)
```

une fonction `boucle` qui prend en entrée une séquence d'ARN et qui fournit en sortie la boucle maximale obtenue en supprimant aux extrémités les sous-séquences qui se replient l'une sur l'autre selon le dessin suivant :




```
def comp(n):
   if n == 'A' :
     return('U')
   elif n == 'U' :
     return('A')
   elif n == 'G' :
     return('C')
   else :
     return('G')

def boucle(b):
   d = 0
   f = len(b) -1
   while d < f :
     if b[d] != comp(b[f]) :
       return(b[d:f+1])
     d=d+1
     f=f-1
   return("")
```

En fait, il faut arrêter à 3 nucléotides pour des raisons d'élasticité maximale de l'ARN. Donc par exemple `while d+3 < f` et `return(b[d:f+1])` à la fin au lieu de `return("")`. *Nota* : on aura une meilleure solution pour `comp` avec les tableaux plus tard.