1 Accès aux caractères d'une chaîne de caractères

Les caractères présents dans une chaîne de caractères `s` sont numérotés de la gauche vers la droite *en partant de 0*. Ainsi par exemple, `"Bon"[0]` est la chaîne réduite à un seul caractère `"B"`, `"Bon"[1]` est la chaîne réduite à un seul caractère `"o"` et `"Bon"[2]` est la chaîne `"o"`. En revanche, `"Bon"[3]` est en dehors de la chaîne de caractères et produit donc une erreur.
Peut importe que la chaîne de caractères soit explicite comme dans l'exemple `"Bon"` précédent ou qu'elle soit dans une variable, pourvu que cette variable soit de type `str`. Par exemple après avoir affecté la variable `nom="Paul"`, `nom[0]` vaudra `"P"`, `nom[1]` vaudra `"a"`, *etc.*

On peut aussi extraire un intervalle de caractères dans une chaîne de caractères. Il faut donner deux entiers séparés par un « : » et par exemple : `"Hello"[1:4]` est la partie de `"Hello"` comprise entre le caractère numéroté 1 inclus et le caractère numéroté 4 exclus, c'est-à-dire la chaîne `"ell"`.

Par exemple, pour extraire le deuxième codon d'un brin d'ADN, on peut écrire la fonction suivante :

```
>>> def deuxieme (b) :
...   if len(b) < 6 :
...     print("brin de longueur insuffisante!")
...   else :
...     return(b[3:6])
...
>>> deuxieme("ATTGCCAAA")
'GCC'
```

2 Opération de substitution de chaînes de caractères

La structure de données des chaînes de caractères a encore une opération qui mérite une section d'explication à elle seule : la *substitution*, notée « `%` ».

Si la chaîne de caractères s_0 contient « `%s` » et si s_1 est une autre chaîne de caractères, alors s_0 `%` s_1 est la chaîne obtenue en remplaçant dans s_0 les deux caractères `%s`par la chaîne s_1. Par exemple :

```
>>> "bonjour %s ; il faut beau aujourd'hui." % "Pierre Dupond"
"bonjour Pierre Dupond ; il faut beau aujourd'hui."
```

Plus généralement, s'il y a plusieurs « `%s` », il faut mettre entre parenthèses autant de chaînes $(s_1,\ldots,s_n)$ après l'opération `%`. Par exemple :

```
>>> nom = "Dupond"
>>> prenom="Pierre"
>>> genre = "homme"
>>> "Ce %s s'appelle %s et c'est un %s" % (prenom,nom,genre)
"Ce Pierre s'appelle Dupond et c'est un homme"
```

Si l'on veut mettre un entier relatif, on utilise « `%i` » au lieu de « `%s` » (`i` comme integer, et `s` comme string).

```
>>> age = 41
>>> "%s %s a %i ans." % (prenom,nom,age)
'Pierre Dupond a 41 ans.'
```

Pour un nombre réel, c'est « `%f` » (`f` comme float).

```
>>> "%s %s mesure %f m." % (prenom,nom,1.85)
'Pierre Dupond mesure 1.850000 m.'
```

et si l'on veut imposer le nombre de chiffres après la virgule, par exemple 2 chiffres après la virgule, on utilise « `%.2f` » :

```
>>> "%s %s mesure %.2f m." % (prenom,nom,1.85)
'Pierre Dupond mesure 1.85 m.'
```

Les différents encodages des substitutions présentés ici sont les plus utiles. Il y en a d'autres, plus rarement utilisés, qu'on trouve aisément dans tous les manuels.

3 Des conversions de type (cast)

Connaître le type d'une valeur ou d'un calcul est *primordial* lorsqu'on programme.
Par exemple `print("solde=" + 10 + "euros")` est mal typé et constitue une erreur. En effet, on veut ici utiliser l'opération « `+` » qui effectue la concaténation des *chaînes de caractères*, par conséquent il faut lui fournir des *chaînes de caractères* en arguments : `print("solde=" + "10" + "euros")`.

La valeur notée `"10"` est une chaîne de caractères constituée de deux caractères consécutifs `"1"` suivi de `"0"` il ne s'agit en aucun cas d'un nombre entier. La chaîne `"10"` est de type `str` et le nombre `10` est de type `int` : ils ne sont pas du tout encodés pareil dans la machine.

Pour passer de l'un à l'autre, il faut une fonction, qui est fournie par Python. Plus généralement, les fonctions qui permettent d'effectuer une conversion d'un type à l'autre de manière « naturelle » sont souvent appelées des *cast* :

- La fonction `str` transforme son argument en chaîne de caractère chaque fois que le type de l'argument est sufisamment simple pour le faire sans ambiguïté.
- La fonction `int` transforme de même son argument en entier relatif chaque fois que possible.
- La fonction `float` transforme de même son argument en réel chaque fois que possible.
- La fonction `bool` existe aussi mais n'a aucun intérêt du fait des opérations de comparaisons, bien plus claires.

```
>>> str(10)
'10'
>>> str(True)
'True'
>>> str(15.33)
'15.33'
>>> int("10")
10
>>> int("toto")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'toto'
>>> int(12.3)
12
>>> int(12.9)
12
>>> int("12.9")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.9'
>>> float("12.6")
12.6
>>> float(12)
12.0
```

4 Les expressions conditionnelles

Abandonnons pour quelques temps la description des structures de données classiques en programmation pour étudier deux éléments cruciaux du contrôle : les expressions conditionnelles et les définitions de fonctions ou de procédures.

Une expression conditionnelle « de base » est de la forme *SI condition ALORS action1 SINON action2*. En python cela donne par exemple, comme on l'a déjà vu :

```
>>> if "a b" == "ab" :
...     print("L'espace ne compte pas.")
... else :
...     print("L'espace compte.")
...
L'espace compte.
```

Dans le cas où l'on ne souhaite rien faire lorsque la condition est fausse, on peut omettre la partie `else`.

```
>>> if len("abc") != 3 :
...     print("YA UN PROBLEME !")
...
>>>
```

Enfin il arrive qu'on ait des « cascades » d'expressions conditionnelles et dans ce cas le mot-clef `elif` est une contraction de `else if` (exemple dans la section suivante).

5 Les définitions de fonctions

La plupart des « calculs » que l'on fait avec les différents types de données sont assez bien identifiés pour porter un nom qui les caractérise. Pour avoir un bon style de programmation on a en fait intérêt à découper tous les « calculs » compliqués en une combinaison de « calculs » plus simples et à donner un nom à chacun d'eux, ce qui simplifiera leur usage ultérieurement. Pour cela on définit des *fonctions*. Par exemple :

```
>>> def complementaire (n) :
...     if n == 'A' :
...         return('T')
...     elif n == 'T' :
...         return('A')
...     elif n == 'G' :
...         return('C')
...     elif n == 'C' :
...         return('G')
...     else :
...         return('N')
...
>>> complementaire ('G')
'C'
>>> complementaire ('N')
'N'
>>> complementaire(4)
'N'  (mais Python devrait normalement retourner une erreur de typage !)
>>> complementaire(complementaire('A'))
'A'
```

Le mot clef `def` signifie que toutes les lignes qui sont *sous sa portée* constituent une définition de la fonction `complémentaire`. Le « (n) » entre parenthèses signifie que par la suite, on devra donner en entrée de cette fonction un seul argument, et que cet argument remplacera toutes les occurrences de `n` dans la définition pour calculer le résultat de la fonction `complementaire`.

- Les 10 lignes qui suivent la ligne « `def complementaire (n) :` » sont *sous la portée* de `def` parce qu'elles ont un décalage de margepar rapport à `def`. La ligne vide qui suit ces 10 lignes indique que la marge revient à zéro, donc la fin de la portée de `def`.
- De la même façon, « `return('T')` » est sous la portée de « `if n == 'A' :` » à cause d'un second décalage de marge, et le fait que le `elif` qui suit revienne au premier décalage de marge signifie la fin de la portée de « `if n == 'A' :` ».

Le mot-clef `return` indique ce que la fonction fournit comme résultat. Dans chacun des cas, il faut donc n'avoir qu'un seul `return` possible. On peut alors réutiliser la fonction à toutes les sauces dans d'autres fonctions, exactement comme si le langage Python la fournissait parmi ses opérations :

```
>>> def nucleotide(n) :
...     return( complementaire(n) != 'N' )
...
>>> nucleotide ('A')
True
>>> nucleotide ('B')
False
>>> def transcription (adn) :
...     if adn == 'A' :
...         return('U')
...     else :
...         return( complementaire(adn) )
...
>>> transcription ('A')
'U'
>>> transcription ('B')
'N'
>>> transcription ('G')
'C'
```

Noter qu'*une variable indique une place* : `adn`, aurait pu être remplacé par `n` ou `glop`...

Si l'on veut programmer une fonction avec plusieurs entrées, il suffit de les séparer avec des virgules à l'intérieur de la parenthèse :

```
>>> def moyenne (x,y):
...     return ( (x+y) / 2 )
...
>>> moyenne(3,4)
3.5
```