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 :

```
>>> "%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 marge par 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
```

6 Les définitions de procédures

Il est possible de définir des « fonctions » qui ne fournissent aucun résultat ! On appelle cela des *procédures*. Naturellement cela est d'un intérêt moindre et on préfèrera utiliser des fonctions chaque fois que possible.

Pour écrire une procédure, il suffit de ne jamais utiliser `return` dans la programmation. Par exemple, on peut se contenter d'écrire à l'écran le résultat du calcul, sans le fournir pour autant comme résultat « déclaré ».

```
>>> def trans (n) :
...     if n == 'A' :
...         print('U')
...     else :
...         print(complementaire(n))
...
>>> trans ('A')
U
>>> trans ('G')
C
```

La différence ne saute pas aux yeux, si ce n'est que les guillemets n'apparaissent pas dans le « résultat » lors des appels de `trans`. En fait, `trans` imprime lui même `U` ou `C` à l'écran (commande `print`), alors que pour `transcription`, c'est le langage Python qui se chargeait d'écrire le résultat. La différence majeure, c'est que `trans` ne retourne aucun résultat. En voici la preuve :

```
>>> len(transcription('A'))
1
>>> type(transcription('A'))
<type 'str'>
>>> len(trans('A'))
U
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'NoneType' has no len()
>>> type(trans('A'))
U
<type 'NoneType'>
```

On remarque que `trans` imprime « bêtement » à l'écran son résultat, cependant il ne le fournit pas à la fonction qui l'appelle, donc `len` qui attend une chaîne de caractères, produit une erreur, et le « résultat » de `trans` est sans type.

Ainsi donc, on n'utilisera les procédures que pour imprimer divers résultats finaux à l'écran. Plus généralement, les procédures ne devraient être utilisées que pour gérer les dialogues avec les utilisateurs (IHM = Interface Homme-Machine).

7 Dialogues avec un utilisateur

Lorsque l'on veut « récupérer » des informations en les demandant à un utilisateur, on utilise une fonction appelée `input`. La « fonction » `input` prend en entrée une chaîne de caractères et fournit en sortie une autre chaîne de caractère :

1. Elle écrit à l'écran la chaîne qu'on lui donne en argument
2. puis elle attend que l'utilisateur tape une ligne (terminée par un retour chariot c'est-à-dire la touche « Entrée » du clavier)
3. le résultat de la « fonction » `input` est alors la chaîne de caractères qu'a tapée l'utilisateur et est donc *toujours* de type `str`.

```
>>> def bonjour () :
...     prenom = input("Quel est votre prénom ? : ")
...     nom = input("Quel est votre nom de famille ? : ")
...     print("Bonjour %s %s ! bonne journée." % (prenom,nom))
...
>>> bonjour()
Quel est votre prénom ? : Albert
Quel est votre nom de famille ? : Durand
Bonjour Albert Durand ! bonne journée.
```

Il est souvent utile de demander à l'utilisateur une valeur d'un autre type que `str`. Comme `input` fournit toujours une donnée de type `str`, il faut alors gérer « à la main » la conversion de `str` vers le type que l'on souhaite.

Rappel : la fonction `int` convertit en entier (si possible) l'argument qu'on lui donne. Ainsi les nombre flottants (type `float`) sont tronqués par la fonction `int`. Cette fonction `int` marche aussi pour les chaînes de caractères *ne contenant que des chiffres* (au passage, `str` fait donc l'inverse).

```
>>> int(12.7)
12
>>> int("12")
12
>>> int("12.7")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.7'
>>> str(126)
'126'
```

Si l'on veut ignorer les caractères qui ne sont pas des chiffres, il faut le programmer soi-même :

```
>>> def intsouple (s) :
...     r = 0
...     for c in s :
...         if c >= '0' and c <= '9' :
...             r = 10 * r + int(c)
...         #sinon on ignore le caractère
...     return(r)
...
>>> intsouple ("to3to4glop0")
340
```

On reviendra sur l'instruction `for` en Python. Ici elle permet de parcourir l'un après l'autre chaque caractère contenu dans la chaîne `s`. Ensuite, sous la portée du `for`, selon que ce caractère `c` est un chiffre (i.e. compris entre `'0'` et `'9'`) ou non, on le prend en compte dans le résultat `r`.
Et ainsi par exemple :

```
>>> def askint () :
...     s = input("entrez un entier positif : ")
...     n = intsouple(s)
...     if str(n) != s :
...         print("Vous tapez à coté des touches !")
...     return(n)
...
>>> 2 * askint()
entrez un entier positif : 45
90
>>> 2 * askint()
entrez un entier positif : glop9u0
Vous tapez à coté des touches !
180
```

8 Les boucles `while`

On a déjà vu qu'il est pratique d'extraire le n-ième caractère d'une chaîne de caractères. Si s est une chaîne de caractères, alors l'opération d'*accès direct* `s[i]` fournit le (`i`+1)-ième caractère de la chaîne (c'est à dire que le premier caractère est en fait numéroté 0, le deuxième est numéroté 1, *etc*).

```
>>> "abcdef"[0]
'a'
>>> "abcdef"[1]
'b'
>>> "abcdef"[2]
'c'
>>> "abcdef"[5]
'f'
>>> "abcdef"[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

On peut alors par exemple calculer le brin complémentaire d'une portion de génome (et plus seulement d'un seul caractère à la fois comme le fait la fonction `complementaire` programmée au cours précédent) en parcourant le brin d'origine « à l'envers » :

```
>>> def brinCompl (c) :
...     resultat = ""
```

```
...     i = len(c)
...     while i > 0 :
...         i = i - 1
...         resultat = resultat + complementaire(c[i])
...     return(resultat)
...
>>> brinCompl ("AAATCCGT")
'ACGGATTT'
```

C'est l'occasion, de présenter une nouvelle commande de contrôle : `while`, qui s'utilise syntaxiquement comme un `if` sans `else`, mais dont le bloc de programme est exécuté et ré-exécuté tant que la condition reste vraie.

> ATTENTION : si l'on gère mal le programme, une instruction `while` peut boucler indéfiniment (par exemple si l'on oublie `i = i - 1`).

La boucle ci-dessus met donc deux variables locales à jour : l'indice `i` des caractères que l'on traite les uns après les autres et la chaîne de caractères, construite pas à pas, qui fournira le résultat final.

Rappel à propos de `in` sur les chaînes de caractères : s_0 `in` s retourne un booléen qui indique si s_0 est une sous-chaîne (consécutive) de s (test d'appartenance).

```
>>> "to" in "tatetitotu"
True
>>> "io" in "tatetitotu"
False
```

Pour calculer le nombre de voyelles d'une chaîne de caractères, on peut alors programmer :

```
>>> def nbvoyelles (s) :
...     n = 0
...     i = 0
...     while i < len(s) :
...         if s[i] in "aeiouyAEIOUY" :
...             n = n + 1
...         i = i + 1
...     return(n)
...
>>> nbvoyelles("toto")
2
```

(cette fois on a parcouru la chaîne dans l'ordre).

Terminons sur les chaînes de caractères en signalant qu'il est possible d'extraire plus de un caractère à la fois avec la forme suivante : s`[i:j]`. Cette forme fournit la chaîne de caractères commençant à l'indice `i` et se terminant à `j-1` (et non pas `j`, ce serait trop simple...).

```
>>> "abcdef"[2:5]
'cde'
>>> "abcdef"[2:2]
''
```

Cela permet d'extraire n'importe quelle sous-chaîne d'une chaîne de caractères.

9 Quelques exemples

Position du premier nucléotide illisible (X ou N) à l'issue d'un séquençage :

```
>>> def firstbad(b) :
...     i = 0
```

```
...     while i < len(b) :
...         if b[i] in "XN" :
...             return(i)
...         i = i + 1
...     print("Aucun nucléotide douteux dans ce brin")
```

Pourcentage de nucléotides exactement placés pareil sur deux brins d'ADN :

```
>>> def perfectmatch(u,v) :
...     if len(u) < len(v) :
...         lmin = len(u)
...         lmax = len(v)
...     else :
...         lmin = len(v)
...         lmax = len(u)
...     if lmax == 0 :
...         return(100)
...     else :
...         nbOK = 0
...         i = 0
...         while i < lmin :
...             if u[i] == v[i] :
...                 nbOK = nbOK + 1
...             i = i + 1
...         return((nbOK * 100) // lmax)
```

Trouver le préfixe commun de deux chaînes de caractères :

```
>>> def prefixe (a,b):
...   p=""
...   i=0
...   while i < len(a) and i < len(b) and a[i] == b[i] :
...     p = p + a[i]
...     i = i + 1
...   return(p)
...
>>> prefixe("abcd","efg")
''
>>> prefixe("abcde","abcuv")
'abc'
```

... et encore plus élégant avec les « tranches » de chaînes de caractères :

```
>>> def prefixe (a,b):
...   i=0
...   while i < len(a) and i < len(b) and a[i] == b[i] :
...     i = i + 1
...   return(a[0:i])
```

Rappel : pour une chaîne de caractères `s`, la tranche `s[m:n]` retient les caractères dont les indices vont de `m` à `n-1` inclus (et non pas `n`).

10 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
```