1 Les expressions conditionnelles

Abandonnons quelques instants 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).

2 Les définitions de fonctions

Lorsque certains calculs sont assez bien identifiés pour porter un nom, on a intérêt à leur donner un nom, 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 cela devrait 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' :` ».
– Si l'on veut programmer une fonction avec plusieurs entrées, il suffit de les séparer avec des virgules à l'intérieur de la parenthèse.

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`...

3 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` est appelé dans chacune des deux commandes précédentes parce qu'il 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 (IHM = Interface Homme-Machine).

4 Variables locales et variables globales

Lorsqu'une fonction ou une procédure modifie une variable, elle est toujours « locale » à cette fonction ou procédure. Cela signifie que sitôt que l'on est sorti de la fonction, on a « oublié » la variable.

```
>>> def triple (n) :
...     resultat = n * 3
...     return resultat
...
>>> triple (4)
12
>>> resultat
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'resultat' is not defined
```

Mieux : si la variable était définie préalablement, elle n'est pas modifiée par la fonction :

```
>>> resultat = 128
>>> triple(2)
6
>>> resultat
128
```

En revanche, une variable *globale* qui n'est pas modifiée par la fonction est visible dans la fonction :

```
>>> def teste (n) :
...     if n > resultat :
...         print("plus grand")
...     else :
...         print("plus petit")
...
>>> teste (4)
plus petit
>>> teste (200)
plus grand
```

MAIS, par défaut, en Python une variable ne peut pas être *à la fois* globale et locale :

```
>>> def ajuste (n) :
...     if n > resultat :
```

```
...         resultat = n
...
>>> ajuste (200)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in ajuste
UnboundLocalError: local variable 'resultat' referenced before assignment
```

Moralité : par défaut en Python, une fonction ou une procédure ne peut pas modifier une variable globale. Les autres langages l'autorisent par défaut mais c'est de toute façon un style de programmation dangeureux, puisque l'état des variables serait alors modifié implicitement. Mieux vaut écrire :

```
>>> def ajuste (n) :
...     if n > resultat :
...         return n
...     else :
...         return resultat
...
>>> resultat = ajuste(200)
>>> resultat
200
```

Si l'on tient absolument à modifier une variable globale dans une fonction, il faut utiliser une *déclaration* dans la fonction qui change le comportement par défaut pour la variable en question. On utilise alors le mot `global` :

```
>>> nbUsage=0
>>> def triple(n) :
...   global nbUsage
...   resultat = 3 * n
...   nbUsage = nbUsage + 1
...   return resultat
...
>>> nbUsage
0
>>> triple(2)
6
>>> nbUsage
1
>>> triple(6)
18
>>> nbUsage
2
```

5 Compléments sur les chaînes de caractères, et `while`

Revenons un peu aux structures de données et commençons par des moyens complémentaires d'accéder aux caractères dans une chaîne de caractères.

Il est souvent pratique de pouvoir 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 liste (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). Il faut alors parcourir 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'
```

À cette occasion, on rencontre 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.

Encore une opération qui est parfois utile sur les chaînes de caractères : s_0 `in` s prend en entrée deux chaînes de caractères et 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.

6 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 » !