1 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 par un exemple :

```
>>> 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.

> Nota : contrairement à une *fonction*, une *procédure* ne renvoit jamais de résultat, donc n'utilise pas `return`. Néanmoins il ne faut pas croire qu'une fonction renvoit toujours un résultat. Certaines valeurs d'entrée d'une fonction peuvent être non valides (comme une date de naissance postérieure à la date courante pour une fonction qui calcule les âges par exemple) et dans ce cas la fonction ne doit pas renvoyer de résultat mais produire un message d'erreur (simplement en utilisant `print` dans le cadre de ce cours).

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).

2 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
```

3 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 rappelant 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.

4 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 brinsd'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`).