12 Fonctions

12.1 Principe

En programmation, les fonctions sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme. Elles permettent également de rendre le code plus lisible et plus clair en le fractionnant en blocs logiques.

Vous connaissez déjà certaines fonctions Python, par exemple `math.cos(angle)` du module `math` renvoie le cosinus de la variable `angle` exprimée en radian. Vous connaissez aussi des fonctions internes à Python comme `range()` ou `len()`. Pour l'instant, une fonction est à vos yeux une sorte de « boîte noire » :

- À laquelle vous passez une (ou zero ou plusieurs) valeur(s) entre parenthèses. Ces valeurs sont appelées arguments.
- Qui effectue une action. Par exemple `random.shuffle()` permute aléatoirement une liste.
- Et qui renvoie éventuellement un résultat.

Par exemple si vous appelez la fonction `range()` en lui passant la valeur `5` (`range(5)`), celle-ci vous renvoie une liste de nombres entiers de 0 à 4 (`[0, 1, 2, 3, 4]`).

Au contraire, aux yeux du programmeur une fonction est une portion de code effectuant une action bien particulière. Avant de démarrer sur la syntaxe, revenons sur cette notion de « boîte noire » :

1. Une fonction effectue une tâche. Pour cela, elle reçoit éventuellement des arguments et renvoie éventuellement un résultat. Ce qui se passe en son sein n'intéresse pas directement l'utilisateur. Par exemple, il est inutile de savoir comment la fonction `math.cos()` calcule un cosinus, on a juste besoin de savoir qu'il faut lui passer en argument un angle en radian et qu'elle renvoie le cosinus de cet angle. Ce qui se passe au sein de la fonction ne regarde que le programmeur (c'est-à-dire vous dans ce chapitre).
2. Chaque fonction effectue en général une tâche **unique et précise**. Si cela se complique, il est plus judicieux d'écrire plusieurs fonctions (qui peuvent éventuellement s'appeler les unes les autres). Cette **modularité** améliore la qualité générale et la lisibilité du code. Vous verrez qu'en Python, les fonctions présentent une grande flexibilité.

12.2 Définition

Pour définir une fonction, Python utilise le mot-clé `def` et si on veut que celle-ci renvoie une valeur, il faut utiliser le mot-clé `return`. Par exemple :

```
>>> def carre(x):
...     return x**2
...
>>> print carre(2)
4
```

Remarquez que la syntaxe de `def` utilise les **:** comme les boucles `for`, `while` ainsi que les tests `if`, un bloc d'instrutions est donc attendu. De même que pour les boucles et les tests, l'**indentation** de ce bloc d'instructions (*i.e.* le corps de la fonction) est **obligatoire**.

Dans l'exemple précédent, nous avons passé un argument à la fonction `carre()` qui nous a retourné une valeur que nous avons affichée à l'écran. Que veut dire valeur retournée ? Et bien cela signifie que cette dernière est stockable dans une variable :

```
>>> res = carre(2)
>>> print res
4
```

Ici, le résultat renvoyé par la fonction est stockée dans la variable `res`. Notez qu'une fonction ne prend pas forcément un argument et ne renvoie pas forcément une valeur, par exemple :

```
>>> def hello():
...     print "bonjour"
...
>>> hello()
bonjour
```

Dans ce cas la fonction `hello()` se contente d'imprimer la chaîne de caractères `"hello"` à l'écran. Elle ne prend aucun argument et ne renvoie aucun résultat. Par conséquent, cela n'a pas de sens de vouloir récupérer dans une variable le résultat renvoyé par une telle fonction . Si on essaie tout de même, Python affecte la valeur `None` qui signifie « rien » en anglais :

```
>>> x = hello()
bonjour
>>> print x
None
```

12.3 Passage d'arguments

Le nombre d'argument(s) que l'on peut passer à une fonction est variable. Nous avons vu ci-dessus des fonctions auxquelles on passait 0 ou 1 argument. Dans les chapitres précédentes, vous avez vu des fonctions internes à Python qui prenaient au moins 2 arguments, pour rappel souvenez-vous de `range(1,10)` ou encore `range(1,10,2)`. Le nombre d'argument est donc laissé libre à l'initiative du programmeur qui est en train de développer une nouvelle fonction.

Une particularité des fonctions en Python est que vous n'êtes pas obligé de préciser le type des arguments que vous lui passez, dès lors que les opérations que vous effectuez avec ces arguments sont valides. Python est en effet connu comme étant un langage au *typage dynamique*, c'est-à-dire qu'il reconnaît pour vous le type des variables au moment de l'exécution, par exemple :

```
>>> def fois(x,y):
...     return x*y
...
>>> fois(2,3)
6
>>> fois(3.1415,5.23)
16.430045000000003
>>> fois('to',2)
'toto'
```

L'opérateur $*$ reconnait plusieurs types (entiers, réels, chaînes de caractères), notre fonction est donc capable d'effectuer plusieurs tâches !

Un autre gros avantage de Python est que ses fonctions sont capables de renvoyer plusieurs valeurs à la fois, comme dans cette fraction de code :

```
>>> def carre_cube(x):
...     return x**2,x**3
...
```

```
>>> carre_cube(2)
(4, 8)
```

Vous voyez qu'en réalité Python renvoie un objet séquentiel qui peut par conséquent contenir plusieurs valeurs. Dans notre exemple Python renvoie un objet `tuple` car on a utilisé une syntaxe de ce type. Notre fonction pourrait tout autant renvoyer une liste :

```
>>> def carre_cube2(x):
...     return [x**2,x**3]
...
>>> carre_cube2(3)
[9, 27]
```

Enfin, il est possible de passer un ou plusieurs argument(s) de manière facultative et de leur attribuer une valeur par défaut :

```
>>> def useless_fct(x=1):
...     return x
...
>>> useless_fct()
1
>>> useless_fct(10)
10
```

Notez que si on passe plusieurs arguments à une fonction, le ou les arguments facultatifs doivent être situés après les arguments obligatoires. Il faut donc écrire `def fct(x, y, z=1):`.

12.4 Portée des variables

Il est très important lorsque l'on manipule des fonctions de connaître la portée des variables. Premièrement, on peut créer des variables au sein d'une fonction qui ne seront pas visibles à l'extérieur de celle-ci ; on les appelle **variables locales**. Observez le code suivant :

```
>>> def mafonction():
...     x = 2
...     print 'x vaut',x,'dans la fonction'
...
>>> mafonction()
x vaut 2 dans la fonction
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
```

Lorsque Python exécute le code de la fonction, il connait le contenu de la variable `x`. Par contre, de retour dans le module principal (dans notre cas, il s'agit de l'interpréteur Python), il ne la connaît plus d'où le message d'erreur. De même, une variable passée en argument est considérée comme **locale** lorsqu'on arrive dans la fonction :

```
>>> def mafonction(x):
...     print 'x vaut',x,'dans la fonction'
...
>>> mafonction(2)
x vaut 2 dans la fonction
```

```
>>> print x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
```

Deuxièmement, lorsqu'une variable déclarée à la racine du module (c'est comme cela que l'on appelle un programme Python), elle est visible dans tout le module.

```
>>> def mafonction():
...     print x
...
>>> x = 3
>>> mafonction()
3
>>> print x
3
```

Dans ce cas, la variable `x` est visible dans le module principal et dans toutes les fonctions du module. Toutefois, Python ne permet pas la modification d'une variable globale dans une fonction :

```
>>> def mafonction():
...     x = x + 1
...
>>> x=1
>>> mafonction()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in fct
UnboundLocalError: local variable 'x' referenced before assignment
```

L'erreur renvoyée montre que Python pense que `x` est une variable locale qui n'a pas été encore assignée. Si on veut vraiment modifier une variable globale dans une fonction, il faut utiliser le mot-clé `global` :

```
>>> def mafonction():
...     global x
...     x = x + 1
...
>>> x=1
>>> mafonction()
>>> x
2
```

Dans ce dernier cas, le mot-clé `global` a forcé la variable `x` a être globale plutôt que locale au sein de la fonction.

12.5 Portée des listes

Soyez extrêmement attentifs avec les types modifiables (tels que les listes) car vous pouvez les changer au sein d'une fonction :

```
>>> def mafonction():
...     liste[1] = -127
...
```

```
>>> liste = [1,2,3]
>>> mafonction()
>>> liste
[1, -127, 3]
```

De même que si vous passez une liste en argument, elle est tout autant modifiable au sein de la fonction :

```
>>> def mafonction(x):
...     x[1] = -15
...
>>> y = [1,2,3]
>>> mafonction(y)
>>> y
[1, -15, 3]
```

Si vous voulez éviter ce problème, utilisez des tuples, Python renverra une erreur puisque ces derniers sont non modifiables ! Une autre solution pour éviter la modification d'une liste lorsqu'elle est passée en tant qu'argument, est de la passer explicitement (comme nous l'avons fait pour l'affectation) afin qu'elle reste intacte dans le programme principal.

```
>>> def mafonction(x):
...     x[1] = -15
...
>>> y = [1,2,3]
>>> mafonction(y[:])
>>> y
[1, 2, 3]
>>> mafonction(list(y))
>>> y
[1, 2, 3]
```

Dans ces deux derniers exemples, une copie de `y` est créée à la volée lorsqu'on appelle la fonction, ainsi la liste `y` du module principal reste intacte.

12.6 Règle LGI

Lorsque Python rencontre une variable, il va traiter la résolution de son nom avec des priorités particulières : d'abord il va regarder si la variable est **locale**, puis si elle n'existe pas localement, il vérifiera si elle est **globale** et enfin si elle n'est pas globale, il testera si elle est **interne** (par exemple la fonction `len()` est considérée comme une fonction interne à Python, *i.e.* elle existe à chaque fois que vous lancez Python). On appelle cette règle la règle **LGI** pour locale, globale, interne. En voici un exemple :

```
>>> def mafonction():
...     x = 4
...     print 'Dans la fonction x vaut', x
...
>>> x = -15
>>> mafonction()
Dans la fonction x vaut  4
>>> print 'Dans le module principal x vaut',x
Dans le module principal x vaut -15
```

Vous voyez que dans la fonction, `x` a pris la valeur qui lui était définie localement en priorité sur sa valeur définie dans le module principal.

Conseil : même si Python peut reconnaître une variable ayant le même nom que ses fonctions ou variables internes, évitez de les utiliser car ceci rendra votre code confus !

12.7 Exercices

Conseil : pour ces exercices, écrivez des scripts dans des fichiers, puis exécutez-les dans un *shell*.

1. Prédisez le comportement de ce code sans le recopier dans un script ni dans l'interpréteur Python :

```
def hello(prenom):
    print "Bonjour", prenom

hello("Patrick")
print x
```

2. Prédisez le comportement de ce code sans le recopier dans un script ni dans l'interpréteur Python :

```
x = 10

def hello(prenom):
    print "Bonjour", prenom

hello("Patrick")
print x
```

3. Prédisez le comportement de ce code sans le recopier dans un script ni dans l'interpréteur Python :

```
x = 10

def hello(prenom):
    print "Bonjour", prenom
    print x

hello("Patrick")
print x
```

4. Prédisez le comportement de ce code sans le recopier dans un script ni dans l'interpréteur Python :

```
x = 10

def hello(prenom):
    x = 42
    print "Bonjour", prenom
    print x

hello("Patrick")
print x
```

5. Créez une fonction qui prend une liste de bases et qui renvoie la séquence complémentaire d'une séquence d'ADN sous forme de liste.
6. À partir d'une séquence d'ADN `ATCGATCGATCGCTGCTAGC`, renvoyez le brin complémentaire (n'oubliez pas que la séquence doit être inversée).
7. Créez une fonction `distance()` qui calcule une distance euclidienne en 3 dimensions entre deux atomes. En reprenant l'exercice sur le calcul de la distance entre carbones alpha consécutifs de la protéine 1BTA (chapitre sur les *chaînes de caractères*), refaites la même chose en utilisant votre fonction `distance()`.

13 Expressions régulières et parsing

Le module re vous permet d'utiliser des expressions régulières au sein de Python. Les expressions régulières sont aussi appelées en anglais *regular expressions* ou *regex*. Elles sont incontournables en bioinformatique lorsque vous voulez récupérer des informations dans un fichier.

Cette action de recherche de données dans un fichier est appelée plus généralement *parsing* (qui signifie littéralement « analyse syntaxique » en anglais). Le *parsing* fait partie du travail quotidien du bioinformaticien, il est sans arrêt en train de « fouiller » dans des fichiers pour en extraire des informations d'intérêt comme par exemple récupérer les coordonnées 3D des atomes d'une protéines dans un fichier PDB ou alors extraire les gènes d'un fichier genbank.

Dans ce chapitre, nous ne ferons que quelques rappels sur les expressions régulières. Pour un documentation plus complète, référez-vous à la page d'aide des expressions régulières sur le site officiel de Python.

13.1 Définition et syntaxe

Une expression régulière est une suite de caractères qui a pour but de décrire un fragment de texte. Elle est constitué de deux types de caractères :

1. Les caractères dits *normaux*.
2. Les *métacaractères* ayant une signification particulière, par exemple `^` signifie début de ligne et non pas le caractère « chapeau » littéral.

Certains programmes Unix comme `egrep`, `sed` ou encore `awk` savent interpréter les expressions régulières. Tous ces programmes fonctionnent généralement selon le schéma suivant :

1. Le programme lit un fichier ligne par ligne.
2. Pour chaque ligne lue, si l'expression régulière passée en argument est présente alors le programme effectue une action.

Par exemple, pour le programme `egrep` :

```
[fuchs@rome cours_python]$ egrep "^DEF" herp_virus.gbk
DEFINITION  Human herpesvirus 2, complete genome.
[fuchs@rome cours_python]$
```

Ici, `egrep` renvoie toutes les lignes du fichier genbank du virus de l'herpès (`herp_virus.gbk`) qui correspondent à l'expression régulière `^DEF` (*i.e.* `DEF` en début de ligne).

Avant de voir comment Python gère les expressions régulières, voici quelques éléments de syntaxe des métacaractères :

`^`	début de chaîne de caractères ou de ligne Exemple : l'expression `^ATG` correspond à la chaîne de caractères `ATGCGT` mais pas à la chaîne `CCATGTT`.
`$`	fin de chaîne de caractères ou de ligne Exemple : l'expression `ATG$` correspond à la chaîne de caractères `TGCATG` mais pas avec la chaîne `CCATGTT`.
`.`	n'importe quel caractère (mais un caractère quand même) Exemple : l'expression `A.G` correspond à `ATG`, `AtG`, `A4G`, mais aussi à `A-G` ou à `A G`.
`[ABC]`	le caractère A ou B ou C (un seul caractère) Exemple : l'expression `T[ABC]G` correspond à `TAG`, `TBG` ou `TCG`, mais pas à `TG`.
`[A-Z]`	n'importe quelle lettre majuscule Exemple : l'expression `C[A-Z]T` correspond à `CAT`, `CBT`, `CCT`...
`[a-z]`	n'importe quelle lettre minuscule
`[0-9]`	n'importe quel chiffre
`[A-Za-z0-9]`	n'importe quel caractère alphanumérique
`[^AB]`	n'importe quel caractère sauf A et B Exemple : l'expression `CG[^AB]T` correspond à `CG9T`, `CGCT`... mais pas à `CGAT` ni à `CGBT`.
`\`	caractère d'échappement (pour protéger certains caractères) Exemple : l'expression `\+` désigne le caractère + sans autre signification particulière. L'expression `A\.G` correspond à `A.G` et non pas à `A` suivi de n'importe quel caractère, suivi de `G`.
`*`	0 à n fois le caractère précédent ou l'expression entre parenthèses précédente Exemple : l'expression `A(CG)*T` correspond à `AT`, `ACGT`, `ACGCGT`...
`+`	1 à n fois le caractère précédent ou l'expression entre parenthèses précédente Exemple : l'expression `A(CG)+T` correspond à `ACGT`, `ACGCGT`... mais pas à `AT`.
`?`	0 à 1 fois le caractère précédent ou l'expression entre parenthèses précédente Exemple : l'expression `A(CG)?T` correspond à `AT` ou `ACGT`.
`{n}`	n fois le caractère précédent ou l'expression entre parenthèses précédente
`{n,m}`	n à m fois le caractère précédent ou l'expression entre parenthèses précédente
`{n,}`	au moins n fois le caractère précédent ou l'expression entre parenthèses précédente
`{,m}`	au plus m fois le caractère précédent ou l'expression entre parenthèses précédente
`(CG|TT)`	chaînes de caractères `CG` ou `TT` Exemple : l'expression `A(CG|TT)C` correspond à `ACGC` ou `ATTC`.

13.2 Module re et fonction search

Dans le module `re`, la fonction `search()` permet de rechercher un motif (*pattern*) au sein d'une chaîne de caractères avec une syntaxe de la forme `search(motif, chaine)`. Si `motif` existe dans `chaine`, Python renvoie une instance `MatchObject`. Sans entrer dans les détails propres au langage orienté objet, si on utilise cette instance dans un test, il sera considéré comme vrai. Regardez cet exemple dans lequel on va rechercher le motif `tigre` dans la chaîne de caractères `"girafe tigre singe"` :

```
>>> import re
```

```
>>> animaux = "girafe tigre singe"
>>> re.search('tigre', animaux)
<_sre.SRE_Match object at 0x7fefdaefe2a0>
>>> if re.search('tigre', animaux):
...     print "OK"
...
OK
```

Fonction match()

Il existe aussi la fonction `match()` dans le module `re` qui fonctionne sur le modèle de `search()`. La différence est qu'elle renvoie une instance `MatchObject` seulement lorsque l'expression régulière correspond (*match*) au début de la chaîne (à partir du premier caractère).

```
>>> animaux = "girafe tigre singe"
>>> re.search('tigre', animaux)
<_sre.SRE_Match object at 0x7fefdaefe718>
>>> re.match('tigre', animaux)
```

Nous vous recommandons plutôt l'usage de la fonction `search()`. Si vous souhaitez avoir une correspondance avec le début de la chaîne, vous pouvez toujours utiliser l'accroche de début de ligne `^`.

Compilation d'expressions régulières

Il est aussi commode de préalablement compiler l'expression régulière à l'aide de la fonction `compile()` qui renvoie un objet de type expression régulière :

```
>>> regex = re.compile("^tigre")
>>> regex
<_sre.SRE_Pattern object at 0x7fefdafd0df0>
```

On peut alors utiliser directement cet objet avec la méthode `search()` :

```
>>> animaux = "girafe tigre singe"
>>> regex.search(animaux)
>>> animaux = "tigre singe"
>>> regex.search(animaux)
<_sre.SRE_Match object at 0x7fefdaefe718>
>>> animaux = "singe tigre"
>>> regex.search(animaux)
```

Groupes

Python renvoie un objet `MatchObject` lorsqu'une expression régulière trouve une correspondance dans une chaîne pour qu'on puisse récupérer des informations sur les zones de correspondance.

```
>>> regex = re.compile('([0-9]+)\.([0-9]+)')
>>> resultat = regex.search("pi vaut 3.14")
>>> resultat.group(0)
'3.14'
>>> resultat.group(1)
'3'
```

```
>>> resultat.group(2)
'14'
>>> resultat.start()
8
>>> resultat.end()
12
```

Dans cet exemple, on recherche un nombre composé
- de plusieurs chiffres `[0-9]+`,
- suivi d'un point `\.` (le point a une signification comme métacaractère, donc il faut l'échapper avec `\` pour qu'il ait une signification de point),
- suivi d'un nombre à plusieurs chiffres `[0-9]+`.

Les parenthèses dans l'expression régulière permettent de créer des groupes qui seront récupérés ultérieurement par la fonction `group()`. La totalité de la correspondance est donné par `group(0)`, le premier élément entre parenthèse est donné par `group(1)` et le second par `group(2)`.

Les fonctions `start()` et `end()` donnent respectivement la position de début et de fin de la zone qui correspond à l'expression régulière. Notez que la fonction `search()` ne renvoie que la première zone qui correspond à l'expression régulière, même s'il en existe plusieurs :

```
>>> resultat = regex.search("pi vaut 3.14 et e vaut 2.72")
>>> resultat.group(0)
'3.14'
```

Fonction findall()

Pour récupérer chaque zone, vous pouvez utiliser la fonction `findall()` qui renvoie une liste des éléments en correspondance.

```
>>> regex = re.compile('[0-9]+\.[0-9]+')
>>> resultat = regex.findall("pi vaut 3.14 et e vaut 2.72")
>>> resultat
['3.14', '2.72']
>>> regex = re.compile('([0-9]+)\.([0-9]+)')
>>> resultat = regex.findall("pi vaut 3.14 et e vaut 2.72")
>>> resultat
[('3', '14'), ('2', '72')]
```

Fonction sub()

Enfin, la fonction `sub()` permet d'effectuer des remplacements assez puissants. Par défaut la fonction `sub(chaine1,chaine2)` remplace toutes les occurrences trouvées par l'expression régulière dans `chaine2` par `chaine1`. Si vous souhaitez ne remplacer que les *n* premières occurrences, utilisez l'argument `count=n` :

```
>>> regex.sub('quelque chose',"pi vaut 3.14 et e vaut 2.72")
'pi vaut quelque chose et e vaut quelque chose'
>>> regex.sub('quelque chose',"pi vaut 3.14 et e vaut 2.72", count=1)
'pi vaut quelque chose et e vaut 2.72'
```

Nous espérons que vous êtes convaincus de la puissance du module `re` et des expressions régulières, alors à vos expressions régulières !

13.3 Exercices : extraction des gènes d'un fichier genbank

Pour les exercices suivants, vous utiliserez le module d'expressions régulières `re` et le fichier genbank du chromosome I de la levure du boulanger *Saccharomyces cerevisiae* NC_001133.gbk.

1. Écrivez un script qui extrait l'organisme du fichier genbank `NC_001133.gbk`.
2. Modifiez le script précédent pour qu'il affiche toutes les lignes qui indiquent l'emplacement du début et de la fin des gènes, du type :

   ```
   gene            58..272
   ```

3. Faites de même avec les gènes complémentaires :

   ```
   gene            complement(55979..56935)
   ```

4. Récupérez maintenant la séquence nucléique et affichez-la à l'écran. Vérifiez que vous n'avez pas fait d'erreur en comparant la taille de la séquence extraite avec celle indiquée dans le fichier genbank.
5. Mettez cette séquence dans une liste et récupérez les deux premiers gènes (en les affichant à l'écran). Attention, le premier gène est un gène complémentaire, n'oubliez pas de prendre le complémentaire inverse de la séquence extraite.
6. À partir de toutes ces petites opérations que vous transformerez en fonctions, concevez un programme `genbank2fasta.py` qui extrait tous les gènes d'un fichier genbank fourni en argument et les affiche à l'écran. Pour cela vous pourrez utiliser tout ce que vous avez vu jusqu'à présent (fonctions, listes, modules, etc.).
7. À partir du script précédent, refaites le même programme (`genbank2fasta.py`) en écrivant chaque gène au format fasta dans un fichier.

   Pour rappel, l'écriture d'une séquence au format fasta est le suivant :

   ```
   >ligne de commentaire
   sequence sur une ligne de 80 caractères maxi
   suite de la séquence .....................
   suite de la séquence .....................
   ```

   Vous utiliserez comme ligne de commentaire le nom de l'organisme, suivi du numero du gène, suivi des positions de début et de fin du gène, comme dans cet exemple

   ```
   >Saccharomyces cerevisiae 1 1807 2169
   ```

   Les noms des fichiers fasta seront de la forme `gene1.fasta`, `gene2.fasta`, etc.