16 Modules d'intérêt en bioinformatique

Nous allons voir dans cette section quelques modules très importants en bioinformatique. Le premier `numpy` permet notamment de manipuler des vecteurs et des matrices en Python. Le module `biopython` permet de travailler sur des données biologiques type séquence (nucléique et protéique) ou structure (fichier PDB). Le module `matplotlib` permet de dessiner des graphiques depuis Python. Enfin, le module `rpy` permet d'interfacer n'importe quelle fonction du puissant programme R.

Ces modules ne sont pas fournis avec le distribution Python de base (contrairement à tous les autres modules vus précédemment). Nous ne nous étendrons pas sur la manière de les installer. Consultez pour cela la documentation sur les sites internet des modules en question. Sachez cependant que ces modules existent dans la plupart des distributions Linux récentes.

Dans ce chapitre, nous vous montrererons quelques exemples d'utlisation de ces modules pour vous convaincre de leur pertinence.

16.1 Module numpy

Le module numpy est incontournable en bioinformatique. Il permet d'effectuer des calculs sur des vecteurs ou des matrices, élément par élément, via un nouveau type d'objet appelé *array*. Ce module contient des fonctions de base pour faire de l'algèbre linéaire, des transformées de Fourier ou encore des tirages de nombre aléatoire plus sophistiqués qu'avec le module `random`. Vous pourrez trouver les sources de numpy à cette adresse. Notez qu'il existe un autre module scipy que nous n'aborderons pas dans ce cours. `scipy` est lui même basé sur `numpy`, mais il en étend considérablement les possibilités de ce dernier (*e.g.* statistiques, optimisation, intégration numérique, traitement du signal, traitement d'image, algorithmes génétiques, etc.).

16.1.1 Objets de type array

Les objets de type *array* correspondent à des tableaux à une ou plusieurs dimensions et permettent d'effectuer du calcul vectoriel. La fonction `array()` permet la conversion d'un objet séquentiel (type liste ou tuple) en un objet de type *array*. Voici un exemple simple de conversion d'une liste à une dimension en objet *array* :

```
>>> import numpy
>>> a = [1,2,3]
>>> numpy.array(a)
array([1, 2, 3])
>>> b = numpy.array(a)
>>> type(b)
<type 'numpy.ndarray'>
>>> b
array([1, 2, 3])
```

Nous avons converti la liste `a` en *array*, mais cela aurait donné le même résultat si on avait converti le tuple `(1,2,3)`. Par ailleurs, vous voyez que lorsqu'on demande à Python le contenu d'un objet *array*, les symboles `([])` sont utilisés pour le distinguer d'une liste `[]` ou d'un tuple `()`.

Notez qu'un objet *array* ne peut contenir que des valeurs numériques. Vous ne pouvez pas, par exemple, convertir une liste contenant des chaînes de caractères en objet de type *array*.

La fonction `arange()` est équivalente à `range()` et permet de construire un *array* à une dimension de manière simple.

```
>>> numpy.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> numpy.arange(10.0)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> numpy.arange(10,0,-1)
array([10,  9,  8,  7,  6,  5,  4,  3,  2,  1])
>>>
```

Un des avantages de la fonction `arange()` est qu'elle permet de générer des objets *array* qui contiennent des entiers ou de réels selon l'argument qu'on lui passe.

La différence fondamentale entre un objet *array* à une dimension et une liste (ou un tuple) est que celui-ci est considéré comme un **vecteur**. Par conséquent on peut effectuer des opérations **élément par élément** dessus, ce qui est bien commode lorsqu'on analyse de grandes quantités de données. Regardez ces exemples :

```
>>> v = numpy.arange(4)
>>> v
array([0, 1, 2, 3])
>>> v + 1
array([1, 2, 3, 4])
>>> v + 0.1
array([ 0.1,  1.1,  2.1,  3.1])
>>> v * 2
array([0, 2, 4, 6])
>>> v * v
array([0, 1, 4, 9])
```

Notez bien sur le dernier exemple de multiplication que l'*array* final correspond à la multiplication **élément par élément** des deux *array* initiaux. Avec les listes, ces opérations n'auraient été possibles qu'en utilisant des boucles ! Nous vous encourageons donc à utiliser dorénavent les objets *array* lorsque vous aurez besoin de faire des opérations élément par élément.

Il est aussi possible de construire des objets *array* à deux dimensions, il suffit de passer en argument une liste de listes à la fonction `array()` :

```
>>> numpy.array([[1,2,3],[2,3,4],[3,4,5]])
array([[1, 2, 3],
       [2, 3, 4],
       [3, 4, 5]])
>>>
```

Attention, plus complexe encore ! On peut aussi créer des tableaux à trois dimensions en passant à la fonction `array()` une liste de listes de listes :

```
>>> numpy.array([[[1,2],[2,3]],[[4,5],[5,6]]])
array([[[1, 2],
        [2, 3]],

       [[4, 5],
        [5, 6]]])
>>>
```

La fonction `array()` peut créer des tableaux à n'importe quel nombre de dimensions. Toutefois ça devient vite compliqué lorsqu'on dépasse trois dimensions. Retenez qu'un objet *array* à une dimension peut être considéré comme un **vecteur** et un *array* à deux dimensions comme une **matrice**.

Pour récupérer un ou plusieurs élément(s) d'un objet *array*, vous pouvez utiliser l'indiçage ou les tranchage, de la même manière que pour les listes.

```
>>> a = numpy.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a[5:]
array([5, 6, 7, 8, 9])
>>> a[::2]
array([0, 2, 4, 6, 8])
>>> a[1]
1
```

Dans le cas d'un objet *array* à deux dimensions, vous pouvez récupérer une ligne, une colonne ou bien un seul élément.

```
>>> a = numpy.array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a[:,0]
array([1, 3])
>>> a[0,:]
array([1, 2])
>>> a[1,1]
4
```

La syntaxe `a[m,:]` récupère la ligne m-1, et `a[:,n]` récupère la colonne n-1. Les tranches sont évidemment aussi utilisables sur un tableau à deux dimensions.

Il peut être parfois pénible de construire une matrice (*array* à deux dimensions) à l'aide d'une liste de listes. Le module `numpy` contient quelques fonctions commodes pour construire des matrices à partir de rien. Les fonctions `zeros()` et `ones()` permettent de construire des objets *array* contenant des 0 ou de 1, respectivement. Il suffit de leur passer un tuple indiquant la dimensionnalité voulue.

```
>>> numpy.zeros((3,3))
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> numpy.zeros((3,3),int)
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
>>> numpy.ones((3,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Par défaut, les fonctions `zeros()` et `ones()` génèrent des réels, mais vous pouvez demander des entiers en passant l'option `int` en second argument.

Enfin il existe les fonctions `reshape()` et `resize()` qui permettent de remanier à volonté les dimensions d'un *array*. Il faut pour cela, leur passer en argument l'objet *array* à remanier ainsi qu'un tuple indiquant la nouvelle dimensionnalité.

```
>>> a = numpy.arange(9)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> numpy.reshape(a,(3,3))
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

Toutefois, `reshape()` attend un tuple dont la dimension est compatible avec le nombre d'éléments contenus dans l'*array* de départ, alors que `resize()` s'en moque et remplira le nouvel objet *array* généré même si les longueurs ne coincident pas.

```
>>> a = numpy.arange(9)
>>> a.reshape((2,2))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> numpy.resize(a,(2,2))
array([[0, 1],
       [2, 3]])
>>> numpy.resize(a,(4,4))
array([[0, 1, 2, 3],
       [4, 5, 6, 7],
       [8, 0, 1, 2],
       [3, 4, 5, 6]])
```

Dans l'exemple précédent, la fonction `reshape()` renvoie une erreur si les dimensions ne coïncident pas. La fonction `resize()` duplique ou coupe la liste initiale s'il le faut jusqu'à temps que le nouvel *array* soit rempli.

Si vous ne vous souvenez plus de la dimension d'un objet *array*, la fonction `shape()` permet d'en retrouver la taille.

```
>>> a = numpy.arange(3)
>>> numpy.shape(a)
(3,)
```

Enfin, la fonction `transpose()` renvoie la transposée d'un *array*. Par exemple pour une matrice :

```
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> numpy.transpose(a)
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

16.1.2 Un peu d'algèbre linéaire

Après avoir manipulé les vecteurs et les matrices, voici quelques fonctions pour faire de l'algèbre linéaire.

La fonction `dot()` vous permet de faire une multiplication de matrices.

```
>>> a = numpy.resize(numpy.arange(4),(2,2))
>>> a
array([[0, 1],
       [2, 3]])
>>> numpy.dot(a,a)
array([[ 2,  3],
       [ 6, 11]])
>>> a * a
array([[0, 1],
       [4, 9]])
```

Notez bien que `dot(a,a)` renvoie le **produit matriciel** entre deux matrices, alors que `a*a` renvoie le produit **élément par élément**[1].

Pour toutes les opérations suivantes, il faudra utiliser des fonctions dans le sous-module `numpy.linalg`. La fonction `inv()` renvoie l'inverse d'une matrice carrée, `det()` son déterminant, `eig()` ses vecteurs et valeurs propres.

```
>>> a
array([[0, 1],
       [2, 3]])
>>> numpy.linalg.inv(a)
array([[-1.5,  0.5],
       [ 1. ,  0. ]])
>>> numpy.linalg.det(a)
-2.0
>>> numpy.linalg.eig(a)
(array([-0.56155281,  3.56155281]), array([[-0.87192821, -0.27032301],
       [ 0.48963374, -0.96276969]]))
>>> numpy.linalg.eig(a)[0]
array([-0.56155281,  3.56155281])
>>> numpy.linalg.eig(a)[1]
array([[-0.87192821, -0.27032301],
       [ 0.48963374, -0.96276969]])
```

Notez que la fonction `eig()` renvoie un tuple dont le premier élément correspond aux valeurs propres et le second élément aux vecteurs propres.

16.1.3 Un peu de transformée de Fourier

La transformée de Fourier est très utilisée pour l'analyse de signaux, notamment lorsqu'on souhaite extraire des périodicités au sein d'un signal bruité. Le module `numpy` possède la fonction `fft()` (dans le sous-module `fft`) permettant de calculer des transformées de Fourier.

Voici un petit exemple sur la fonction cosinus de laquelle on souhaite extraire la période à l'aide de la fonction `fft()` :

1. Dans numpy, il existe également des objets de type *matrix* pour lesquels les multiplications de matrices sont différents, mais nous ne les aborderons pas ici.

```
# 1) on definit la fonction y = cos(x)
import numpy
debut = -2 * numpy.pi
fin = 2 * numpy.pi
pas = 0.1
x = numpy.arange(debut,fin,pas)
y = numpy.cos(x)

# 2) on calcule la TF de la fonction cosinus
TF=numpy.fft.fft(y)
ABSTF = numpy.abs(TF)
# abcisse du spectre en radian^-1
pas_xABSTF = 1/(fin-debut)
x_ABSTF = numpy.arange(0,pas_xABSTF * len(ABSTF),pas_xABSTF)
```

Plusieurs commentaires sur cet exemple.

- Vous constatez que numpy redéfinit certaines fonctions ou constantes mathématiques de base, comme `pi` (nombre π), `cos()` (fonction cosinus) ou `abs()` (valeur absolue, ou module d'un complexe). Ceci est bien pratique car nous n'avons pas à appeler ces fonctions ou constantes depuis le module `math`, le code en est ainsi plus lisible.
- Dans la partie 1), on définit le vecteur `x` représentant un angle allant de -2π à 2π radians par pas de 0,1 et le vecteur `y` comme le cosinus de `x`.
- En 2) on calcule la transformée de Fourier avec la fonction `fft()` qui renvoie un vecteur (objet *array* à une dimension) de nombres complexes. Eh oui, le module `numpy` gère aussi les nombres complexes ! On extrait ensuite le module du résultat précédent avec la fonction `abs()`.
- La variable `x_ABSTFL` représente l'abscisse du spectre (en radian^{-1}).
- La variable `ABSTF` contient le spectre lui même. L'analyse de ce dernier nous donne un pic à 0,15 radian^{-1}, ce qui correspond bien à 2π (plutôt bon signe de retrouver ce résultat). Le graphe de ce spectre est présenté dans la partie dédiée à `matplotlib` (section 16.3).

Notez que tout au long de cette partie, nous avons toujours utilisé la syntaxe `numpy.fonction()` pour bien vous montrer quelles étaient les fonctions propres à `numpy`. Bien sûr dans vos futurs scripts il sera plus commode d'importer complètement le module `numpy` avec l'instruction `from numpy import *`. Vous pourrez ensuite appeler les fonctions de `numpy` directement (sans le préfixe `numpy.`).

Si vous souhaitez quand même spécifier pour chaque fonction `numpy` son module d'appartenance, vous pouvez définir un alias pour `numpy` avec l'instruction `import numpy as np`. Le module `numpy` est alors connu sous le nom `np`. Par l'appel de la fonction `array()` se fera par `np.array()`.

16.2 Module biopython

Le module biopython propose de nombreuses fonctionnalités très utiles en bioinformatique. Le tutoriel est particulièrement bien fait, n'hésitez pas à le consulter.

Voici quelques exemples d'utilisation.

Définition d'une séquence.

```
>>> import Bio
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import IUPAC
>>> ADN = Seq("ATATCGGCTATAGCATGCA", IUPAC.unambiguous_dna)
>>> ADN
Seq('ATATCGGCTATAGCATGCA', IUPACUnambiguousDNA())
```

`IUPAC.unambiguous_dna` signifie que la séquence entrée est bien une séquence d'ADN.

Obtention de la séquence complémentaire et complémentaire inverse.

```
>>> ADN.complement()
Seq('TATAGCCGATATCGTACGT', IUPACUnambiguousDNA())
>>> ADN.reverse_complement()
Seq('TGCATGCTATAGCCGATAT', IUPACUnambiguousDNA())
```

Traduction en séquence protéique.

```
>>> ADN.translate()
Seq('ISAIAC', IUPACProtein())
```

16.3 Module matplotlib

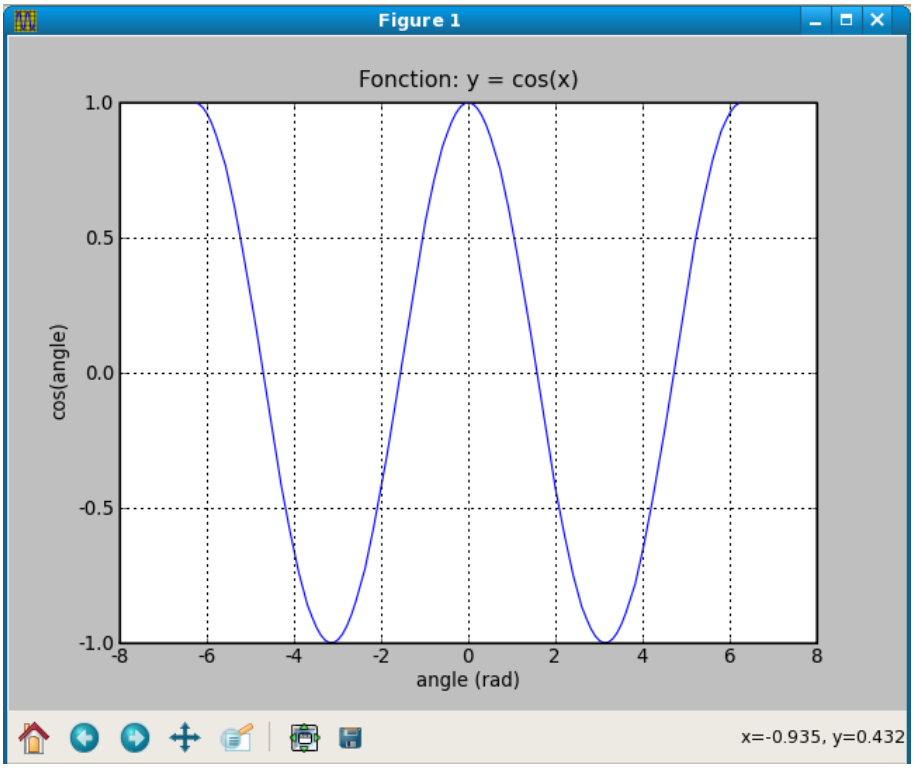
Le module matplotlib permet de générer des graphes interactifs depuis Python. Il est l'outil complémentaire de `numpy` et `scipy` lorsqu'on veut faire de l'analyse de données.

Nous ne présenterons ici qu'un petit exemple traitant de la représentation graphique de la fonction cosinus et de la recherche de sa période par transformée de Fourier (voir également la section 16.1.3).

Plutôt qu'un long discours, regardez cet exemple que nous commenterons après.

```
# define cosine function
from pylab import *
debut = -2 * pi
fin = 2 * pi
pas = 0.1
x = arange(debut,fin,pas)
y = cos(x)

# draw the plot
plot(x,y)
xlabel('angle (rad)')
ylabel('cos(angle)')
title('Fonction: y = cos(x)')
grid()
show()
```

Vous devriez obtenir une image comme celle-ci :



Vous constatez que le module `matplotlib` génère un fenêtre graphique **interactive** permettant à l'utilisateur de votre script de manipuler le graphe (enregistrer comme image, zoomer, etc.).

Revenons maintenant sur le code. Tout d'abord, vous voyez qu'on importe le module s'appelle `pylab` (et non pas `matplotlib`). Le module `pylab` importe lui-même toutes les fonctions (et variables) du module `numpy` (*e.g.* `pi`, `cos`, `arange`, etc.). Il est plus commode de l'importer par `from pylab import *` que par `import pylab`.

La partie qui nous intéresse (après la ligne `# draw the plot`) contient les parties spécifiques à `matplotlib`. Vous constatez que les commandes sont très intuitives.

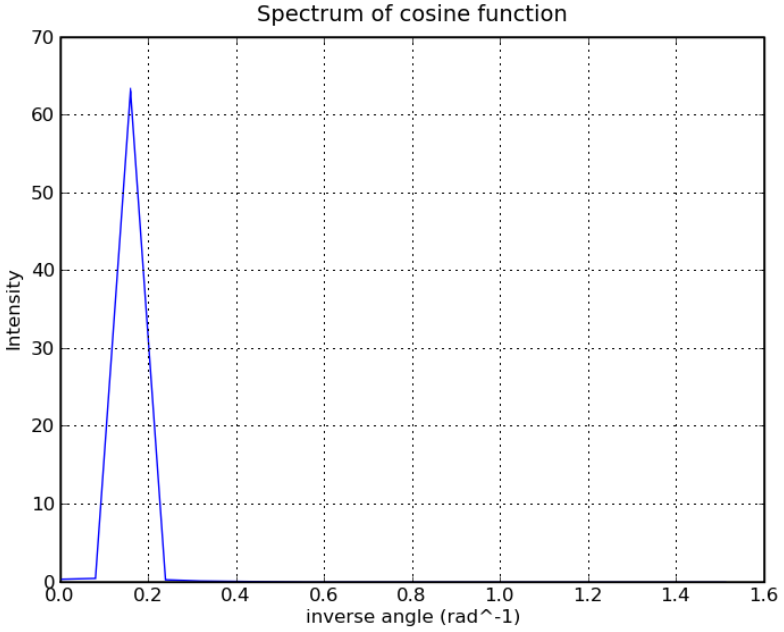
- La fonction `plot()` va générer un graphique avec des lignes et prend comme valeurs en abscisse (`x`) et en ordonnées (`y`) des vecteurs de type *array* à une dimension.
- Les fonctions `xlabel()` et `ylabel()` sont utiles pour donner un nom aux axes.
- `title()` permet de définir le titre du graphique.
- `grid()` affiche une grille en filligrane.
- Jusqu'ici, aucun graphe n'est affiché. Pour activer l'affichage à l'écran du graphique, il faut appeler la fonction `show()`. Celle-ci va activer une boucle dite *gtk* qui attendra les manipulations de l'utilisateur.
- Les commandes Python éventuellement situées après la fonction `show()` seront exécutées seulement lorsque l'utilisateur fermera la fenêtre graphique (petite croix en haut à droite).

Voici maintenant l'exemple complet sur la fonction cosinus et sa transformée de Fourier.

```
from pylab import *

# define cosine function
x = arange(-2*pi,2*pi,0.1)
y = cos(x)

# calculate TF of cosine function
TF=fft(y)
```

```
ABSTF = abs(TF)
pas_xABSTF = 1/(4*pi)
x_ABSTF = arange(0,pas_xABSTF * len(ABSTF),pas_xABSTF)

# draw cos plot
plot(x,y)
xlabel('angle (rad)')
ylabel('cos(angle)')
title('Fonction: y = cos(x)')
grid()
show()

# plot TF of cosine
plot(x_ABSTF[:20],ABSTF[:20])
xlabel('inverse angle (rad^-1)')
ylabel('Intensity')
title('Spectrum of cosine function')
grid()
show()
```

Le premier graphe doit afficher la fonction cosinus comme ci-dessus et le second doit ressembler à cela :



On retrouve bien le pic à 0,15 radian^{-1} correspondant à 2π radians. Voilà, nous espèrons que ce petit exemple vous aura convaincu de l'utilité du module `matplotlib`. Sachez qu'il peut faire bien plus, par exemple générer des histogrammes ou toutes sortes de graphiques utiles en analyse de données.

16.4 Module rpy

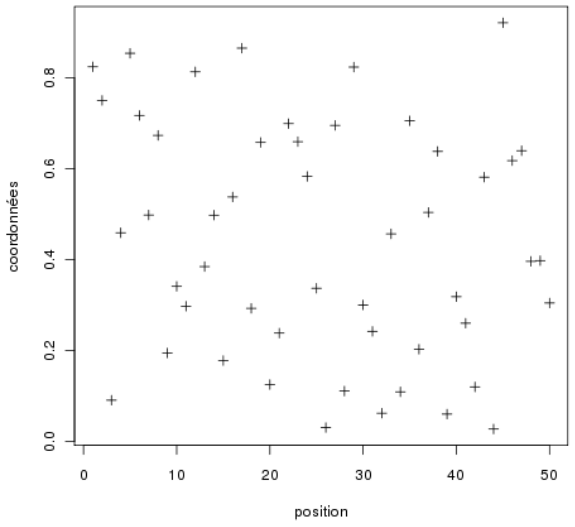
R est un programme extrêmement puissant permettant d'effectuer des analyses statistiques. Il contient tout un tas de fonctions permettant de générer divers types de graphiques. Nous

ne rentrerons pas dans les détails de R, mais nous vous montrerons quelques exemples pour générer des graphiques avec R depuis Python.

Les vecteurs de R peuvent être remplacés par des listes en Python. En interne, `rpy` manipule des variables de types *array* car il est basé sur le module `numpy` vu précédemment. Dans cet exemple, nous allons tracé les coordonnées aléatoires de 50 points.

```
import random
import rpy
# construction d'une liste de 50 éléments croissants
x = range(1, 51)
# construction d'une liste de 50 éléments aléatoires
y = []
for i in x:
    y.append( random.random() )

# enregistrement du graphique dans un fichier png
rpy.r.png("rpy_test1.png")
# dessin des points
rpy.r.plot(x, y, xlab="position", ylab="coordonnées", col="black", pch=3)
# fin du graphique
rpy.r.dev_off()
```

Voici le fichier `rpy_test1.png` obtenu :



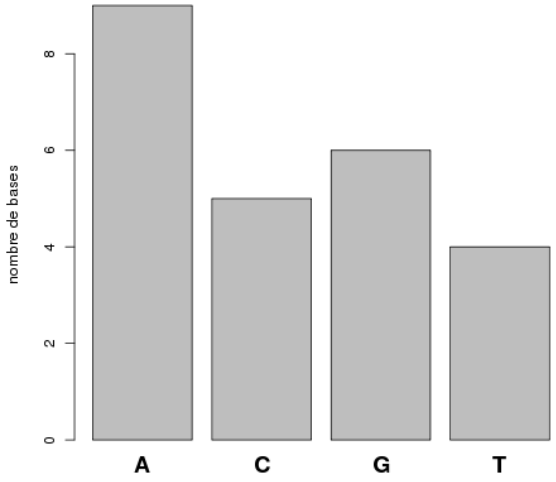
Les fonctions R sont accessibles par le sous-module `r` du module `rpy`. Les fonctions `png()` et `plot()` sont utilisables comme en R. Les fonctions qui contiennent un point dans leur nom en R (`dev.off()`) sont utilisables avec un caractère souligné « _ » à la place (ici `dev_off()`).

Voyons maintenant un exemple plus intéressant, pour lequel on calcule la distribution des différentes bases dans une séquence nucléique :

```
from rpy import r as R
seq = "ACGATCATAGCGAGCTACGTAGAA"
seq2 = list( seq )
```

```
R.png("rpy_test2.png")
# tri des bases car unique() n'ordonne pas les données
# alors que table() le fait
seq3 = R.sort( seq2 )
# listes des bases présentes
bases = R.unique( seq3 )
# effectif de chaque base
effectifs = R.table( seq3 )
# dessin du barplot et sauvegarde de la position des abscisses
coords = R.barplot( effectifs, ylab="nombre de bases")
# ajout du texte pour l'axe des abscisses
R.text(coords, -0.5, bases, xpd = True, cex = 1.5, font = 2 )
R.dev_off()
```

Voici le fichier `rpy_test2.png` obtenu :



Pour plus de concision, le module `rpy.r` est renomé en `R`. Les booléens `TRUE` et `FALSE` en R (dans la fonction `text()`) doivent être remplacés par leurs équivalents en Python (`True` et `False` ; attention à la casse).

16.5 Exercice numpy

Cet exercice présente le calcul de la distance entre deux carbones alpha consécutifs de la barstar. Il demande quelques notions d'Unix et nécessite le module `numpy` de Python.

1. Téléchargez le fichier `1BTA.pdb` sur le site de la PDB.
2. Pour que le séparateur décimal soit le point (au lieu de la virgule, par défaut sur les systèmes d'exploitation français), il faut au préalable redéfinir la variable `LC_NUMERIC` en bash :

   ```
   export LC_NUMERIC=C
   ```

   Extrayez les coordonnées atomiques de tous les carbones alpha de la barstar dans le fichier `1BTA_CA.txt` avec la commande Unix suivante :

   ```
   awk '$1=="ATOM" && $3=="CA" {printf "%.3f %.3f %.3f ", $7, $8, $9}' 1BTA.pdb > 1BTA_CA.txt
   ```

   Les coordonnées sont toutes enregistrées sur une seul ligne, les unes après les autres, dans le fichier `1BTA_CA.txt`.

3. Ouvrez le fichier `1BTA_CA.txt` avec Python et créez une liste contenant toutes les coordonnées sous forme de réels avec les fonctions `split()` et `float()`.
4. Avec la fonction `array()` du module `numpy`, convertissez cette liste en matrice.
5. Avec la fonction `reshape()` de `numpy`, et connaissant le nombre d'acides aminés de la barstar, construisez une matrice à deux dimensions contenant les coordonnées des carbones alpha.
6. Créez une matrice qui contient les coordonnées des n-1 premiers carbones alpha et une autre qui contient les coordonnées des n-1 derniers carbones alpha.
7. En utilisant les opérateurs mathématiques habituels (-, **2, +) et les fonctions `sqrt()` et `sum()` du module `numpy`, calculez la distance entre les atomes n et n+1.
8. Affichez les distances entre carbones alpha consécutifs et repérez la valeur surprenante.

16.6 Exercice rpy

1. Soit la séquence protéique WVAAGALTIWPILGALVILG. Représentez avec le module `rpy` la distribution des différents acides aminés.
2. Reprenez l'exercice du calcul de la distance entre deux carbones alpha consécutifs de la barstar avec `numpy` et représentez cette distance avec `rpy`.

17 Avoir la classe avec les objets

Une classe permet de définir des objets qui sont des représentants (des instances) de cette classe. Les objets peuvent posséder des attributs (variables associées aux objets) et des méthodes ($\sim$ fonctions associées aux objets).

Exemple de la classe `Rectangle` :

```
class Rectangle:
    "ceci est la classe Rectangle"
    # initialisation d'un objet
    # définition des attributs avec des valeurs par défaut
    def __init__(self, long = 0.0, larg = 0.0, coul = "blanc"):
        self.longueur = long
        self.largeur = larg
        self.couleur = coul
    # definition de la méthode qui calcule la surface
    def calculSurface(self):
        print "surface = %.2f m2" %(self.longueur * self.largeur)
    # definition de la méthode qui transforme un rectangle en carré
    def changeCarre(self, cote):
        self.longueur = cote
        self.largeur = cote
```

Ici, `longueur`, `largeur` et `couleur` sont des attributs alors que `calculPerimetre()`, `calculSurface()` et `changeCarre()` sont des méthodes. Tous les attributs et toutes les méthodes se réfèrent toujours à `self` qui désigne l'objet lui même. Attention, les méthodes prennent toujours au moins `self` comme argument.

Exemples d'utilisation de la classe `Rectangle` :

```
# création d'un objet Rectangle avec les paramètres par défaut
rect1 = Rectangle()
print rect1.longueur, rect1.largeur, rect1.couleur
# sans surprise :
rect1.calculSurface()
# on change le rectangle en carré
rect1.changeCarre(30)
rect1.calculSurface()
# création d'un objet Rectangle avec des paramètres imposés
rect2 = Rectangle(2, 3, "rouge")
rect2.calculSurface()
```

17.1 Exercices

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

1. Entraînez-vous avec la classe `Rectangle`. Créez la méthode `calculPerimetre()` qui calcule le périmère d'un objet rectangle.
2. Créez une nouvelle classe `Atome` avec les attributs `x`, `y`, `z` (qui contiennent les coordonnées atomique) et la méthode `calculDistance()`. Testez cette classe sur plusieurs exemples.
3. Améliorez la classe `Atome` avec de nouveaux attributs (par ex. : `masse`, `charge`, etc.) et de nouvelles méthodes (par ex. : `calculCentreMasse()`,`calculRayonGyration()`).