Notions de Perl

Daniel Gautheret

2004-2005.1

LIENS UTILES

Les pages d'introduction a Perl qui suivent sont librement adaptées du cours de Bruno Pouliquen, ` l'Université de Rennes.

Livre recommandé:

Introduction à Perl pour la bioinformatique
James D. Tisdall Laurent Mouchard Guénola Ricard
O'reilly France ; 07/2002

1. PREMIER ESSAI

```
#!/usr/bin/perl
print ("hello world\n");
```

Puis taper le nom du programme (si executable) ou

```
perl <nom du programme>
```

2. TYPES DE DONNEES

Constantes

```
1, -12345, 1.6E16 (signifie 160 000 000 000 000 000),
"cerise", "a" , 'a', 'ceci est une chaine de caracteres'
```

Scalaires

Les scalaires sont précédés du caractère $

```
$i = 0;
$c = 'a';
$mon_nom = 'personne';
$pi = 3.14;
$chaine = "mon nom est $nom";
```

=> mon nom est personne

Lorsque la chaine est entourée de guillemets ("), son contenu est interprété par Perl (cad que s'il y a des variables, elles sont remplacées par leur valeur), si la chaine est entourée d'apostrophes ('), elle n'est pas interprétée.

```
$chaine = 'mon nom est $nom';
```

=> mon nom est $nom

Attention: Pas d'accents ni d'espaces dans les noms de variables
Par contre un nom peut être aussi long qu'on le veut.

Booléens: Les variables de type booléen (cf C ou Pascal) n'existent pas en Perl.
En Perl: 0='faux', autres chiffres = 'vrai'. Exemple:

```
if (2+3) {print "vrai";}
```

Tableaux, Listes

En Perl, les tableaux peuvent être utilisés comme des ensembles ou des listes.
Toujours précédés du caractère « @ »

```
@chiffres = (1,2,3,4,5,6,7,8,9,0);
@fruits = ('amande','fraise','cerise');
@alphabet = ('a'..'z');
```

Les deux points signifient de "tant à tant"

```
@a = ('a'); @nul = ();
@cartes = ('01'..'10','Valet','Dame','Roi');
```

on fait référence à un élément du tableau selon son indice par :

```
$chiffres[1]
```

(=> 2)

```
$fruits[0]
```

(=> 'amande')

REMARQUE : En Perl (comme en C) les tableaux commencent à l'indice 0
On peut affecter un tableau à un autre tableau :

```
@ch = @chiffres;
```

Remarques :
On dispose d'un scalaire spécial : $#tableau qui indique le dernier indice du tableau (et donc sa taille - 1) :

```
$fruits[$#fruits]
```

(=> 'cerise')

Tableaux associatifs

Des tableaux dans lesquels les indices sont des chaines de caractères, et non pas des entiers.
Par exemple le tableau « age » :

Clé :	« pierre »	« paul »	« jacques »
Valeur :	23	12	43

Ils sont toujours précédés du caractère %.
Remplissage du tableau :

```
%age = ('pierre',23,'paul',12,'jacques',43);
```

On référence ensuite un élément du tableau par :

```
$age{'pierre'}
```

ou

```
$age {pierre}
```

Exemples:

```
%chiffre = ();
$chiffre{'un'} = 1;
print $chiffre{un};
$var = 'un'; print $chiffre{$var};
```

Tableaux à plusieurs dimensions

Depuis Perl 5:

```
@table_multiplication = (
[ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],# Multiplié par 0
[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9],# Multiplié par 1
[ 0, 2, 4, 6, 8,10,12,14,16,18],# Multiplié par 2
...
[ 0, 9,18,27,36,45,54,63,72,81]);# Multiplié par 9
```

On référencera alors 2*6 par $table_mult[6] [2]

3. OPERATEURS/COMPARAISONS

A. *Opérateurs*

1. Opérateurs arithmétiques

```
$a = 1; $b = $a; les variables a, et b auront pour valeur 1
$c = 53 + 5 - 2*4; => 50
```

Plusieurs notations pour incrémenter une variable
`$a = $a + 1; ou $a += 1; ou encore$a++;addition`
Même chose pour * (multiplication), - (soustraction), / (division), ** (exponentielle)
`$a *= 3; $a /= 2; $a -= $b; ...`
% : modulo (17 % 3=>2)

2. Sur chaînes de caractères

. concaténation
```
$c = 'ce' . 'rise'; (=> $c devient 'cerise')
$c .= 's'; (=> $c devient 'cerises')
```
xréplique
```
$b = 'a' x 5; => 'aaaaa'
$b = 'jacqu' . 'adi' x 3 => 'jacquadiadiadi'
$b = 'assez ! '; $b x= 5; => 'assez ! assez ! assez ! assez
! assez ! assez !'
```

3. Parenthèses

Comme dans tous les langages de programmation, les parenthèses peuvent être utilisées dans les expressions :
```
$x = 2 * (56 - 78);
```

B. *Comparaisons*

1. de chiffres

Ce sont les opérateurs habituels :
>, >=, <, <=, ==, !=
respectivement: supérieur à, supérieur ou égal, inférieur à, inférieur ou égal, égalité, différent

Attention: = est une affectation, == est une comparaison

2. de chaînes

gt, ge, lt, le, eq, ne
respectivement: supérieur à (selon l'ordre alphabétique), supérieur ou égal, inférieur à, inférieur
ou égal, égalité, différent

Attention! Ne pas confondre la comparaison de chaînes et d'entiers

```
'b' == 'a' => évalué comme étant vrai !
```

il faut écrire :

```
'b' eq 'a' => évalué faux bien-sûr
```

3. de booléens

Même si le type booléen n'existe pas en tant que tel, des opérateurs existent :
|| (ou inclusif), &&(et), !(négation)

```
(! 2 < 1)  => vrai
(1 < 2) && (2 < 3) => vrai
($a < 2) || ($a == 2) équivaut à ($a <= 2)
```

Remarque: depuis Perl5 une notation plus agréable existe :
or(au lieu de ||), and(au lieu de &&), not(au lieu de !)

```
if (not ($trop_cher or $trop_mur)) {print "J'achete !";}
```

4. SYNTAXE GENERALE

- Chaque instruction doit être terminée par un point-virgule.
- Les commentaires commencent par un # Tout le reste de la ligne est considéré comme un commentaire.
- Un bloc est un ensemble de commandes entourées par des crochets ({}, chaque commande étant suivie d'un point-virgule.

A. *Expressions conditionnelles*

```
if ($frequence{'AAA'} > 1) {
 print ("erreur, frequence superieure a 1!");
}
if ($frequence{'AAA'} > 1) {
 print ("erreur, frequence superieure a 1!");
```

```
}else{
 print ("tout va bien");
}
```

B. ***Boucles***

1. ***while***

```
$i=100;
while ($i > 0) {
  print "$i\n";
  $i=$i-1;  # mieux: $i--
}
```

2. ***for***

```
for ($i=0; $i < 100; $i++) {
  print "$i \n";
}
```

3. ***foreach***

```
@taille_genome=(2300000, 3450000, 155000000, 3000000000);
foreach $f (@taille_genome) {
  print "$f\n";
}
```

4. ***while..each***

Tres utile pour boucler sur les tableaux associatifs

```
while( ( $nom , $x ) = each %age ){
  print "$nom: $x\n";
}
```

5. FONCTIONS PREDEFINIES

Système

- Print

```
print "bonjour\n";
print "la frequence de $codon est $freq{$codon}\n";
print "la frequence de", $codon, "est", $freq{$codon}*100,
"\n";
```

- exit

```
if ($erreur) {exit;}
```

- die

```
if ($fruit eq 'orange') {die 'Je déteste les oranges !'}
```

- system

```
system "cal 2032";
system "mkdir $x";
```

avec « system », la sortie de la commande UNIX est affichée normalement à l'écran.
Variante : avec des guillemets inversés (backquotes), le sortie de la commande UNIX est envoyée comme un résultat qui peut ensuite être affecté à une variable. Par exemple :

```
$x = `cal 2032`;
print $x;
```

ou :

```
$blastout = `blastall -p blastn -i $query -d $db`;
```

Mathématique

```
$s = cos(0);
$s = log(exp(1));
$i = int(sqrt(8));
$tirage_loto = int(rand(42)) + 1;
$i = abs(-5.6)
```

Chaînes

- concaténation

```
$a="salut ";
$b="les amis";
$c=$a.$b;
```

$c contient "salut les amis"

- chop

Enleve le dernier caractère de la chaine:

```
$ch='cerises';
chop ($ch);
```

$ch contient "cerise"

- chomp

Idem mais enlève seulement les Retour charriots

- length

Renvoie la longueur de la chaine

```
$l= length ('cerises');
```

$l vaut 7

- split('motif', ch)

  Sépare la chaîne 'ch' en plusieurs éléments, le séparateur étant 'motif'
  ```
  @t = split(';', 'pierre; paul ; jacques; jean');
  ```
  @t est un tableau contenant (‘pierre‘, ‘paul‘, ‘jacques‘, ‘jean')

- substr(ch, indicedébut, longueur)

  Retourne la chaîne de caractère contenue dans ch, à partir du caractère 'indicedébut' et sur une longueur 'longueur'.
  ```
  $ch=substr('RKAILVA', 0, 3)
  $ch=substr('AAATTTT', 4)
  ```

- index(ch, recherche)

  Retourne la position de 'recherche' dans la chaîne 'ch'
  ```
  $i=index('GCATGTAGCTAGCTAG','ATG');
  ```
  $i vaut 2.

Tableaux

- join (ch, tableau)

  Regroupe tous les éléments d'un tableau dans une chaîne de caractères (en spécifiant le séparateur 'ch')
  ```
  print join(', ', @fruits)
  ```

- pop (tableau)

  Retourne le dernier élément du tableau (et l'enlève)
  ```
  print pop(@fruits);
  ```

- push (tableau, element)

  Ajoute un élément en fin de tableau (contraire de pop)
  ```
  push(@fruits, "pomme");
  ```

- sort (tableau)

  Trie le tableau par ordre croissant
  ```
  @fruits = sort (@fruits);
  ```

- reverse (tableau)

  Inverse le tableau (le dernier devient premier)
  ```
  @fruits = reverse (@fruits);
  ```

Tableaux associatifs

- keys (tableau)

  Renvoie les clé du tableau
  ```
  foreach $c (keys %codon_usage) {
    print $c, " ";
  }
  ```

- values (tableau)

  Renvoie les valeurs du tableau
  ```
  foreach $v (values %codon_usage) {
    print $v, " ";
  }
  ```

- each (tableau)

  Renvoie les couples (clé,valeur) du tableau
  ```
  while (($cod,$val) = each (%codon_usage)) {
   print "codon $cod : nombre: $val\n";
  }
  ```

- exists (element)

  Renvoie 1 ou 0 selon que l'élément a été défini ou non
  ```
  if (exists $codon_usage{"UAG"}) {
    print "probleme!\n";
  }
  ```

6. GESTION DES FICHIERS

Ouverture

- En lecture
  ```
  open (F, 'e-coli.fasta')
  ```

- En écriture
  ```
  open (F2, '>fichier.sortie')
  open (F2, '>>fichier.sortie')
  ```

- Gestion des erreurs:
  ```
  if (! open (F, "toto.fasta")) {
  die "Problème à l'ouverture du fichier";
  }
  ```

  ou:

```
open (F, "toto.fasta") || die "Pb d'ouverture";
```

Fermeture

```
close F2;
```

Lecture

```
$ligne = <F>;
```
- La fin de ligne (retour-chariot) est lue également. Pour enlever cette fin de ligne il suffit d'utiliser la commande chop, ou son équivalent : chomp(enlève le dernier caractère uniquement si c'est un retour-chariot)

Parcours:

```
open (F, $fichier) ;
while ($ligne = <F>) {
 print $ligne;
}
```

On peut lire toutes les lignes d'un fichier dans un tableau (en une seule instruction)

```
@ligne = <F>;
```

Un fichier spécial: STDIN, le clavier (entrée standard).

Lecture d'une ligne au clavier:

```
$ligne = <STDIN>;
```

Ecriture

Deux fichiers spéciaux: STDOUT, STDERR (respectivement: sortie standard, et sortie erreur), par défaut l'écran.

```
print F2 'DUPONT Jean';
print F3 'Comment fait-on pour se connecter SVP ?';
print STDOUT "Bonjour\n";
print STDERR 'Je déteste les oranges !';
```

7. EXPRESSIONS REGULIERES

Perl offre la même puissance de manipulation d'expressions régulières que la commande "egrep" d'Unix. On utilise l'opérateur conditionnel =~ qui signifie en gros "contient". Syntaxe: chaîne =~/expression/

Caractères spéciaux des expressions régulières

.	tout caractère sauf fin de ligne
*****	répétition du caractère précédent de 0 à l'infini
+	répétition du caractère précédent de 1 à l'infini
?	répétition du caractère précédent de 0 ou 1 fois
{x,y}	répétition du caractère précédent de x à y fois
{x,}	répétition du caractère précédent au moins x fois
[abc]	soit a, soit b soit c
[a-z]	tout caractère alphabétique
[aeiouy]	toute voyelle
[a-zA-Z0-9]	tout caractère alphanumérique
$	fin de ligne
^	début de ligne
	mais : le caractère ^ au début d'un ensemble signifie « tout sauf »
[^0-9]	tout caractère non numérique
(toto|titi)	toto ou titi
\n	retour-chariot
\t	tabulation
\w	un mot,
\s	un espace
\W	tout sauf un mot
\S	tout sauf un espace
\d	un chiffre
\D	tout sf un chiffre

Remarque :Si l'on veut qu'un caractère spécial apparaisse tel quel, il faut le précéder d'un « anti- slash » (\), les caractères spéciaux sont :

```
« ^ | ( ) [ ] { } \ / $ + * ? .»
```

Exemple:

```
if ($nom =~ /^[Dd]upon/) {print "OK !";}
```

Ok si nom est 'dupont', 'dupond', 'Dupont-Lassoeur'
^ signifie « commence par »
On peut rajouter « i » derrière l'expression pour signifier qu'on ne différencie pas les majuscules des minuscules.

Le contraire de l'opérateur =~ est !~ (ne contient pas ...)

```
if ($nom !~ /^dupon/i) {print "Non...";}
```

Options:

```
chaîne =~/expression/i ;
```

insensible aux majuscules/minuscules

```
chaîne =~/expression/g ;
```

…recherche toutes les occurences, et non pas seulement la premiere. Avec l'option « g » la commande « =~ » renvoie en résultat un tableau contenant toutes les instances trouvées.

Ainsi, on peut faire :

```
@result = (chaîne =~/expression/g) ;
```

Et on obtient dans le tableau @result toutes les instance de « expression » dans « chaine ».

Exemples

```
print 'Etes vous d\'accord?';
$reponse = <STDIN>
if ($reponse =~ /^O/i{
 print "alors on contine...\n";
}

while ($ligne = <F>) {
 if ($ligne =~ /([0-9][0-9]\.)+) /{ #si la ligne est de la
forme 04.91...
  print $ligne;
 }
}
```

Les parenthèses permettent de récupérer des parties de l'expression dans les variables $1, $2, etc... Par exemple:

```
$chaine = " ORGANISM Mycoplasma genitalium";
$chaine =~ /ORGANISM\s+(\S+)\s+(\S+)/
print "genre: $1, espece: $2\n"
```

Remplacements

Comme le fait la commande « sed » en Unix, Perl permet de faire des remplacements sur une chaîne de caractère, en utilisant la syntaxe : $chaîne =~ s/motif/remplacement/; où motifest une expression régulière et remplacementce qui remplace.
Exemples:

```
$seq =~ s/U/T/; #remplace les U par des T

$tel =~ s/^91\./04\.91\./;
```

Options :

```
s/exp/remp/i;
```
=> Indifférenciation minuscules/majuscules

```
s/exp/remp/g;
```
=> Remplace toute occurrence (pas seulement la première)

8. VARIABLES ET TABLEAUX SPECIAUX

Ce sont les variables sous la forme $c, c étant un caractère non alphabétique.

- $_ La dernière ligne lue (au sein d'une boucle + while ;)
- $! La dernière erreur, utilisé dans les détections d'erreurs
- $& La dernière chaine trouvé par une recherche d'expression régulière (avec =~)
  `open(F,'fichier') || die "erreur $!"`
- $1, $2, ...le contenu de la parenthèse numéro dans la dernière expression régulière
- $0 Le nom du programme
- @_ contient les paramètres passés à une procédure.
- @ARGV contient les paramètres passés au programme

9. PASSAGE DE PARAMETRES

En Unix (ou Windows) on peut appeler un programme Perl en lui donnant des paramètres, comme on le fait pour les commandes Unix

Les paramètres sont stockés dans un tableau spécial : @ARGV Le premier paramètre est donc $ARGV[0], le second $ARGV[1], etc.

Exemple de passage de parametres:

```
if ($#ARGV != 0) {
print "Program <Fasta sequence file> \n";
exit;
}

open(F, $ARGV[0]) || die ("file not found\n");
```

10. PROCEDURES

Déclaration d'une procédure:

```
sub monprint {
  print $_[0], "\n"
}
```

Appel d'une procédure:

```
monprint ($seq);
```

- Fonctions

  Une fonction est une procédure qui retourne une valeur (résultat d'un calcul, vrai ou faux, etc..)

  ```
  sub pluriel {
    $s = $_[0]."s";
    return ($s);
  }
  ```

- "My"

  Contrairement à notre attente, le programme suivant imprime "portes, portes" (et non pas "porte, portes").

  ```
  $a = "porte";
  $b = pluriel ($a);
  print "$a, $b \n";

  sub pluriel {
    $a = $_[0]."s";
    return ($a);
  }
  ```

  La raison est que les variables employées dans les procédures sont des variables GLOBALES (ici $a). C'est à dire que si elles sont modifiées dans la procédure, elle le sont aussi dans la programme principal.

  Pour ne pas modifier les variables par inadvertance, il faut les déclarer dans la procédure comme LOCALES, avec l'instruction "My":

  ```
  sub pluriel {
    my $a;
    $a = $_[0]."s";
    return ($a);
  }
  ```

  Employée dans le programme ci-dessus, cette procédure affichera le résultat escompté ("porte, portes").

11. BIOPERL ET LES OBJETS PERL

Bioperl est une collection de modules Perl qui facilitent le développement de scripts Perl pour des applications en Bioinformatique. Bioperl est un projet communautaire Open Source en constant développement auquel contribuent des dizaines de bioinformaticiens. Il comprend des scripts pratiques et réutilisables pour :

- la manipulation de séquences
- l'accès aux bases de données selon de multiples formats de séquence
- des parsers perrmettant d'analyser les sorties de nombreux programmes de bioinfo

Bioperl permet de réaliser des analyses sur de grandes quantité de séquence qui seraient impossible à réaliser via des interfaces Web.

Le site officiel de Bioperl est www.bioperl.org.

Le concept d'objet en Perl

Extrait cours de Sylvain Lhullier (http://lhullier.org/publications/intro_perl) avec mes commentaires :

La programmation orientée objet est un type de programmation qui se concentre principalement sur les données. La question qui se pose en programmation OO (orientée objet) est "quelles sont les données du problème ?" à l'instar de la programmation procédurale par exemple, qui pose la question "quelles sont les fonctions/actions à faire ?". En programmation OO, on parle ainsi d'objets, auxquels on peut affecter des variables/attributs (propriétés) et des fonctions/actions (méthodes). Par exemple , un objet de la classe `Vehicule` peut avoir une couleur, un poids, une puissance, une forme etc. Ce sont ses propriétés. On peut le créer, le dessiner, le détruire, le modifier, etc. Ce sont ses méthodes.

En Perl, une classe correspond module et un objet (instance de cette classe) n'est autre qu'une référence associée à cette classe. Par exemple :

Le fichier `vehicule.pm` contient la description de la classe `Vehicule`. Pour utiliser le module:

```
use Vehicule;
```

Puis, par exemple :

```
my $v = Vehicule->new( "bleu" );
my $v2 = Vehicule->new( "rouge" );
```

Généralement, les classes contiennent un constructeur `new`, qui permet de créer de nouvelles instances de cette classe. Nous venons ici de créer deux instances de la classe `Vehicule`. On dit que ces deux variables `$v` et `$v2` sont des `Vehicule`. Ces deux objets sont donc indépendants l'un de l'autre ; ils sont de la même classe `Vehicule`, mais en constituent des instances autonomes, la modification de l'un ne modifiant pas l'autre.

Attention, si on essaye d'imprimer un objet, on obtient pas ce qu'on pense :

```
print "$v\n";
```

On obtient:

```
Vehicule=HASH(0x80f606c)
```

En fait, l'objet est codé en Perl par un tableau associatif (hash). Pour vraiment afficher les propriétés de l'objet, il faut passer par une méthode spécifique (programmée par les concepteurs de la classe), ou alors accéder à chaque propriété, comme ceci :

```
print "$v->couleur\n";
print "$v->poids\n";
```

Le module Seq

L'objet central utilisé par BIOPERL est l'objet `seq`. Cet objet permet de stocker la séquence elle même, ses identificateurs, ainsi que toutes les annotations associées.

```
use Bio::Seq;
$seq = Bio::Seq->new(-seq              => 'actgtggcgtcaact',
                     -desc             => 'Exemple Bioperl:',
                     -display_id       => 'qqchose',
                     -accession_number => 'numacc',
                     -alphabet         => 'dna' );
```

Puis, pour imprimer la séquence :

```
print $seq->seq(), "\n";
```

En fait, `seq()` est ici une méthode. Il existe plusieurs méthodes associées aux objets `seq` :

Ces méthodes retournent des chaînes :

```
$seq->display_id();       # the human read-able id of the sequence
$seq->seq();              # string of sequence
$seq->subseq(5,10);       # part of the sequence as a string
$seq->accession_number(); # when there, the accession number
$seq->alphabet();         # one of 'dna','rna','protein'
$seq->primary_id();       # a unique id for this sequence irregardless
                          # of its display_id or accession number
$seq->desc();             # a description of the sequence
```

Ces méthodes retournent des objets `seq`.

```
$seq->trunc(5,10);  # truncation from 5 to 10 as new object
$seq->revcom;       # reverse complements sequence
$seq->translate;    # translation of the sequence
```

Ces méthodes retournent d'autres objets:

```
$seq->get_SeqFeatures;      # The 'top level' sequence features
$seq->get_all_SeqFeatures;  # All sequence features, including sub-
                            # seq features
```

Par exemple :

```
$s2=$seq->revcom();
print $s2->seq()."\n";
```

Ou :

```
$s2=$seq->translate();
print $s2->seq()."\n";
```

La fonction `translate` admet plusieurs arguments (cadre de lecture etc.)

Autres classes de séquences :

L'objet `PrimarySeq` n'a que les propriétés séquence, Id et alphabet, et occupe donc beaucoup moins de mémoire quand il s'agit de stocker des millions de séquences. Il existe également `LocatableSeq`, `RelSegment`, `LiveSeq`, `LargeSeq`, `RichSeq`, `SeqWithQuality`, `SeqI`.

Le module SeqIO

L'objet `SeqIO` est un ensemble de séquences, ou plus exactement un flux de séquences. Permet de lire ou d'écrire un flux de séquences et d'efefctuer des conversions de format. Par exemple :

```
use Bio::SeqIO;

$in = Bio::SeqIO->new(-file => "toto.gb",
                      -format => 'Genbank');
$seq2 = $in->next_seq();
print $seq2->seq()."\n";
```

Notez l'importance de `next_seq` qui va chercher la prochaine séquence du flux (retourne un object `Seq`).
Effectuons maintenant une conversion de format :

```
$in  = Bio::SeqIO->new(-file => "inputfilename",
                       -format => 'Fasta');
$out = Bio::SeqIO->new(-file => ">outputfilename",
                       -format => 'EMBL');
while ( $seq = $in->next_seq() ) {
  $out->write_seq($seq);
}
```

Ici, `write_seq` écrit dans le flux `$out`.

Utilisé en combinaison avec `SeqIO` pour aller lire des enregistrments Genbank ou EMBL, le module `Seq` comprend de nombreuses fonctions pour extraire les features (table ci-contre). Par exemple:

DESTROY	No description	Code
_retrieve_subSeqFeature	No description	Code
accession	No description	Code
accession_number	Description	Code
add_SeqFeature	Description	Code
all_SeqFeatures	Description	Code
alphabet	Description	Code
annotation	Description	Code
can_call_new	Description	Code
desc	Description	Code
display_id	Description	Code
end	Description	Code
feature_count	Description	Code
flush_SeqFeature	No description	Code
flush_SeqFeatures	Description	Code
id	Description	Code
length	Description	Code
new	Description	Code
primary_id	Description	Code
primary_seq	Description	Code
seq	Description	Code
species	Description	Code
start	Description	Code
strand	Description	Code
subseq	Description	Code
top_SeqFeatures	Description	Code
validate_seq	Description	Code

```
$in = Bio::SeqIO->new(-file => "test.gb",
                      -format => 'Genbank');

$seq = $in->next_seq();

print "longeur:",  $seq->length(), " \n";

foreach my $f ( $seq->top_SeqFeatures ) {
  printf("%s %s(%s..%s)\n",$f->primary_tag,
         $f->strand, $f->start, $f->end);
}
```

Il existe également un module `AlignIO` pour convertir les fichiers d'alignement (Fasta, Clustal etc.) et un module `SimpleAlign` qui permet de manipuler les fichiers d'alignement : création de séquence consensus, découpage de l'alignement, calcul ed conservation, etc.

Le module DB

```
use Bio::DB::GenBank;

$gb = new Bio::DB::GenBank;

$seq = $gb->get_Seq_by_id('MUSIGHBA1'); # Unique ID

print ">Sequence de Genbank\n", $seq->seq(), "\n";
```

Bioperl supporte pour l'instant l'accès aux banques genbank, genpept, RefSeq, swissprot et EMBL.

Le module SeqStats

Permet de calculer le poids moléculaire d'une séquence, la fréquence en nt/aa, la fréquence en codons.
Attention, les tableaux retournés par les différentes méthodes – `get_mol_wt();` `count_monomers();` `count_codons()` – sont en fait des pointeurs sur des tableaux :

Example :

```
use Bio::Tools::SeqStats;
```
(charge également le module `PrimarySeq`)

```
$seqobj = Bio::PrimarySeq->new(-seq=>'ACTGTGGCGTCAACTG',
                               -alphabet=>'dna',
                               -id=>'test');
$seq_stats  =  Bio::Tools::SeqStats->new(-seq=>$seqobj);

$monomers = $seq_stats->count_monomers();

foreach $base (sort keys %$monomers) {
    print "Bases de type ", $base, "= ", %$monomers->{$base},"\n";
}
```

Si l'on ne veut faire qu'un calcul sans plus, il n'est pas nécessaire de créer un objet:

```
$weight = Bio::Tools::SeqStats->get_mol_wt($seqobj);
```

Le module RemoteBlast

Permet de lancer un Blast à distance au NCBI et de récupérer le résultat. (Notre configuration de TP ne permet pas de le tester).

```
use Bio::Tools::Run::RemoteBlast;

$rblast = Bio::Tools::Run::RemoteBlast->new (
    -prog => 'blastp',-data => 'ecoli',-expect => '1e-10' );

$r = $rblast->submit_blast("toto.fa");

while (@rids = $rblast->each_rid ) {
    foreach $rid ( @rids ) {$rc = $rblast->retrieve_blast($rid);}
}
```

La méthode `SubmitBlast` retourne un objet de type rapport Blast. On peut également changer certains paramètres de Blast, par exemple, la matrice de substitution :

```
 $Bio::Tools::Run::RemoteBlast::HEADER{'MATRIX_NAME'} = 'BLOSUM25';
```

Le module SearchIO

Permet de lire et d'analyser des sorties de Blast, Fasta, PSI-Blast, Hmmer.

```
use Bio::SearchIO;

  # Get the report
  $blastout = new Bio::SearchIO (-format => 'blast',
                                 -file   => "blastout");
$result = $blastout->next_result;
$algorithm_type =  $result->algorithm;

$hit = $result->next_hit;   # info about first hit
```

```
$hit_name = $hit->name ;

$hsp = $hit->next_hsp; # info about first hsp of first hit
$hsp_start = $hsp->query->start;

print $algorithm_type, " ", $hit_name, " ", $hsp_start , "\n";
```