Bio-Programming

Perl:

Practical Extraction and Report Language (PERL) was developed in 1987 by Larry wall, having the features from a variety of other languages including C, shell scripting (sh), AWK, sed and Lisp. Perl was widely adopted for its strengths in text processing and lack of the arbitrary limitations of many scripting languages at the time. If Perl is ornamented with some additional biological modules for the analysis, management and manipulation of biological data, to solve complex biological problems then it is referred as Bioperl.

Bioperl**:**

Bioperl is a collection of Perl modules that facilitate the development of Perl scripts for bioinformatics applications.

Installing Bioperl on windows:

To install Bioperl, the version of Perl is needed to be installed on the system (Active Perl 5.8.8.819) which is supported by the Bioperl team. The reason for this requirement is that this can be used with Perl package manager 4 (PPM4). PPM4 is a superior to earlier versions and also includes graphical user interface (GUI).

To install Activeperl:

1) Download the Activeperl MSI from Active state (http://www.activestate.com/activeperl)
2) Run the Activeperl installer (accepting all defaults in fine)

Installation using the Perl Package Manager:

GUI Installation:-

1) Click on Perl package Manager GUI from the Start menu.
2) Then go Edit >> Perl package manager Preferences and click the Repositories tab., “add a new Repository” for each of the following:-

    Repositories to add Name Location
    Name:-BioPerl-Release candidates
    Location:- http://bioperl.org/DIST/RC
    Name:-BioPerl-Regular Releases
    Location: - http://bioperl.org/DIST
    Name:-Kobes
    Location:-http://theoryx5.uwinnipeg.ca/ppms
    Name:-Bribes
    Location:-http://www.Bribes.org/perl/ppm

3) Select View >> All Packages.
4) In the search box type bioperl.
5) Right click the latest version of Bioperl available and choose install. (Note for users of previous Bioperl releases: you should not have to use the Bundle-BioPerl package anymore.)
6) From bioperl 1.5.2 onward, all 'optional' pre-requisites will be marked for installation.
7) Click the green arrow (Run marked actions) to complete the installation.

Command-line Installation

1) Follow step1 and step 2 from 'GUI Installation' above, if you haven't done so already.
2) Open a cmd window by going to Start >> Run and typing 'cmd' and pressing return.
3) Type the following into the cmd window:

```
 C:> ppm-shell
>ppm
```

Make sure it should have the module PPM Repositories

```
ppm>install PPM-Repositories
```

BioPerl 1.6.1 require atleast the following repositories:

```
ppm > repo add http://bioperl.org/DIST
ppm > repo add unwinnipeg
ppm > repo add trouchelle
```

Install BioPerl("not bioperl")

```
ppm>install bioperl
```

Get the index number for your active repositories:

```
ppm>repo
```

Execute the following commands:

```
rem -turn off ActiveState, trouchelle repos
ppm> repo off 1
ppm> repo off 4
rem -to get SOAP-Lite-0.69 from uwinnipeg...
ppm> install SOAP-Lite
rem -turn ActiveState, trouchelle back on...
ppm> repo on 1
ppm> repo on 4
rem -now try...
ppm> install BioPerl
```

Variable:

Variables play an important role in computer programming because they enable programmers to write flexible programs. Rather than entering data directly into a program, a programmer can use variables to represent the data. Then, the variables are replaced with real data when the program is executed. This makes it possible for the same program to process different sets of data. Every variable has a name, called the *variable name,* and a data type. A variable's data type indicates what sort of value the variable represents, such as whether it is an integer, a floating-point number, or a character.

Constant:

In programming, constants are variables that allow their value to be set once, in the definition, and never changed after that. So it's like a normal variable, only that value can be assigned to it when it is defined, however value can not be changed later in the program. The opposite of a *variable* is a constant. Constants are values that never change. Because of their inflexibility, constants are used less often than variables in programming.

Variable declaration:

A variable is declared by………..

the ($) symbol (scalar),
the (@) symbol (arrays),
or the (%) symbol (hashes).

i.e. $variable, @variable, %variable;

Types of variable:

1. *Scalar variables:*

   Scalar variables are simple variables containing only one element--a string or a number or a character. Strings may contain any symbol, letter, or number. Numbers may contain exponents, integers, or decimal values. The bottom line here with scalar variables is that they contain only one single piece of data. What you see is what you get with scalar variables.

   For example:

   ```
   $number = "5";
   $exponent = 2**8;
   ```

```
$string = "Hello,
PERL!";     $linebreak
= "\n";
```

2. Array variables

Arrays contain a list of scalar data (single elements). A list can hold an unlimited number of elements. In PERL, arrays are defined with (@) symbol.

Array declaration:

```
@DNA = ("a", "g", "c", "t")
@stop = ("taa", "tag", "tga");
```

3. Hash Variables:

Hashes are complex lists with both a key and a value part for each element of the list. Hash is defined using the percent symbol (%).

Example:

```
%triple = (“methionine”, “met”,  “ Leucine”, “leu”, “ Valine”, “val”, “Alanine”, “ala,”);
%single = (“methionine” , “M”, “ Isoleucin”, “I”, “ Leucine”, “L”, “ Valine”, “V”);
```

Hashes are very complex data types, for now just understand the syntax of how to define one. Later we will take a closer look at these complex variables

Laxial variables:

Lexical variables are other called as private variables because they're private. They're also sometimes called my variables because they're always declared with my. It's tempting to call them `local variables', because their effect is confined to a small part of the program, but the actual meaning of my and local is different.

The declaration of my is:

```
Sub a {
        my $x;
    }
```

Local is an operator, the actual difference between my and local can be well illustrated by the following example.

```
$lo='global';
$m  = 'global';
     A();
     subA{
       local $lo = 'AAA';
             my   $m  = 'AAA';
             print "$lo\n\n";
             print "$m\n\n";
             B();
     }
    sub B {
      print "B ", ($lo eq 'AAA' ? 'can' : 'cannot') ," see the value of lo set by A.\n";
      print "B ", ($m  eq 'AAA' ? 'can' : 'cannot') ," see the value of m  set by A.\n";
      print "$lo\n\n";
      print "$m\n\n";
       }
       print "$lo\n\n";
       print "$m\n\n";
```

Standard Input:

<STDIN> stands for standard input. It can be abbreviated by using simple <>. By declaring a scalar variable and setting it equal to <STDIN> we set the variable equal to whatever will be typed by our user at the command prompt.

Standard Output:

The print Command:

The print command tells the program to print out something. Here is the format for the print statement:

```
print “Your text and other things go here”;
or
print ‘Your text and other things go here’;
```

This statement prints the content between the quote marks to standard output. As follows….
Your text and other things go here

To print a single string without special characters single quote is used, if special character are inserted then it will print the characters as such…

```
print ‘Your text and \n other things go here’;
Your text and \n other things go here
```

If double quote will be used then…….

```
print “Your text and \n other things go here”;
Your text and other things go here
```

die command

You noticed that most of our open () statements are followed by or die "some sort of message". The difference between print and die is ……..

The statement executed and the program proceeds incase of print but incase of die the program stops just after the execution of die statement.

For example:

```
open (OUTPUT, ">$outfile") or die "Can't write to $outfile: $!";
```

The die statement ends your program with an error message.

Special Characters:

The characters used for formatting the output are called as special characters.

Character	**Description**
\L	Transform all letters to lowercase
\l	Transform the next letter to lowercase
\U	Transform all letters to uppercase
\u	Transform the next letter to uppercase
\n	Begin on a new line
\t	Applies a tab to the string
\b	Backspace
\a	Bell
\E	Ends \U, \L

Example for special characters:

```
print"\nEnter a DNA Sequence:\t";
$a=<STDIN>;
print"\nEnter a second DNA Sequence:\t";
```

```
$b=<STDIN>;
chomp($a);
chomp($b);
$c="\U$a"."\u$b";
$e="\nat\Uatgcgc\Etg\a";
print $c;
print "\n\n\l$c";
print "\n\n\L$c\n";
print  $e;
print "\n\nnew\bdelhi";
```

Operators:

The *operators* in a computer language tell the computer what actions to perform. Perl has more operators than most languages. Operators are instructions, that is given to the computer so that it can perform some task or operation. All operators cause actions to be performed on *operands*. An operand can be anything on which operations are performed. In practical terms, any particular operand will be a literal, a variable, or an expression.
The operators are several types, but there are most common operators are given below:-

1. Arithmetic operator
2. Logical operator
3. Conditional operator
4. Range operator
5. String operator and
6. Assignment operator.

1. *Arithmetic operator:*

There are six *arithmetic* operators: addition(+), subtraction(-), multiplication(*), and exponentiation (**), division, and modulus (%).
For example,

```
# op1.pl:
$x = 3 + 1;
$y = 6 - $x;
$z = $x * $y;
$w = 2**3;   # 2 to the power of 3 = 8
```

The modulus operator is useful when a program needs to run down a list and do something on every few items.
Example:

```
for ($index = 0; $index <= 100; $index++)
 {

    if ($index % 10 == 0)
    {
     print("$index ");
    }
}
```

2. *Logical operator:-*

Logical operators are mainly used to control program flow. These are: -

op1 && op2
-- Performs a logical AND of the two operands.

op1 || op2
-- Performs a logical OR of the two operands.

!op1
-- Performs a logical NOT of the operand.

The && operator is used to determine whether both operands and conditions are true .

For example:

```
if ($firstVar == 10 && $secondVar == 9) {   print("Error!");}
```

If either of the two conditions is false or incorrect, then the print command will not be executed.
The || operator is used to determine whether either of the conditions is true.

For example:

```
if ($firstVar == 9 || $firstVar == 10) {  print("Error!");}
```

If the condition is not true.

For example:

```
$var=<STDIN>;
if(!$var){print "x";}
else {print "y";}
```

3.Conditional Operators:-

The conditional operators are of 2 types: equality and relational.

Equality	**Numeric**	**String**
Equal	==	eq
Not Equal	!=	ne
Comparison	<=>	cmp
Relational	**Numeric**	**String**
Less than	<	lt
Greater than	>	gt
Less than or equal	<=	le
Greater than or equal	>=	ge

4. *Assignment Operators:*

Assignment operators perform an arithmetic operation and then assign the value to the existing variable. In this example, set a variable ($x) equal to 5. Using assignment operators replace that value with a new number after performing some type of mathematical operation.

Operator	**Definition**	**Example**
+=	Addition	($x += 10)
-=	Subtraction	($x -= 10)
*=	Multiplication	($x *= 10)
/=	Division	($x /= 10)
%=	Modulus	($x %= 10)
*=	Exponent	($x **= 10)

5. *The Range Operator (..)*

The range operator is used as a shorthand way to set up arrays. When used with arrays, the range operator simplifies the process of creating arrays with contiguous sequences of numbers and letters. For example to create an array with ten elements that include 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 simply write:

@array = (1..10);

Array of contiguous letters can also be created, for example: an array with ten elements that include A, B, C, D, E, F, G, H, I , and J. is written

```
@array = ("A".."J");
```

An array can also be created by including the following AAA, 1, 2, 3, 4, 5, A, B, C, D, and ZZZ:

```
@array = ("AAA", 1..5, "A".."D", "ZZZ");
```

Range operator can also be used to create a list with zero-filled numbers. To create an array with ten elements that include the strings 01, 02, 03, 04, 05, 06, 07, 08, 09, and 10 :

```
@array = ("01".."10");
```

And variables can also be used as operands for the range operator. To assign a string literal to $firstVar, create an array with ten elements that include the strings 01, 02, 03, 04, 05, 06, 07, 08, 09, and 10:

```
$firstVar = "10";
@array = ("01"..$firstVar);
```

Like wise….

```
@array = ("aa" .. "af");
@array = ("ay" .. "bf");
```

6. *The String Operators (. and x)*

Perl has two different string operators-the concatenation (.) operator and the repetition (x) operator. These operators make it easy to manipulate strings in certain ways.
Strings can be concatenated by the (.) operator.

For example:

```
$upstream = "atggcgt";
$downstream = "ggcct";
$sequence = $upstream.$downstream;
Strings can be repeated with (x) operator
For example:
$polyA = "A";
$full_sequence = $sequence.$polyA x 10;
print $full_sequence;
Which gives " atggcgtggcctaaaaaaaaaa ".
```

Conversion between numbers and Strings

This is a useful facility in Perl. Scalar variables are converted automatically to string and number values according to context.
For example:

```
$x = "40";
$y = "11";
$z = $x + $y;  # answer 51
$w = $x . $y;  # answer "4011"
```

Note: if a string contains any trailing non-number characters they will be ignored. *i.e.* “123.45abc" would get converted to 123.45 for numeric work. If no number is present in a string it is converted to 0.

If Conditional Statements:

Conditional statements may involve logical operators and usually test equality or compare one value to another. Here's a look at some common conditional statements. Remember that the code to be executed will only execute if the condition in parentheses is met.

If Else Statements:

The else statement is a perfect compliment to a if statement. The idea behind them is that if the conditional statement is not met then do this. In hypothetical, plain English, "If this is true do this, if not do this".

For Example:

```
print "Enter First DNA sequence:";
$first=<STDIN>;
print "Enter Second DNA sequence:";
$sec=<STDIN>;
if($first eq $sec)
{
  Print "Equal";
}
else
{
   print "Not Equal!!!!!";
}
```

Elseif Statements:

Elseif statements test another conditional statement before executing code. In this way multiple conditions can be tested before the script continues.

For Example:

```
print "Enter a Protein sequence:";
$prot=<STDIN>;
if($prot eq "\n")
{
  print "No sequence Found!!!!!";
}
elsif($prot eq " \n")
{
  print "No sequence Found!!!!!";
}
elsif($prot eq "\t\n")
{
  print "No sequence Found!!!!!";
}
else
{
  print "You Can continue.......";
}
```

Using unless Statements:

The unless condition checks for a certain condition and executes it every time unless the condition is true. It’s like the opposite of the if statement:

```
print "Enter a DNA sequence:";
$Seq=<STDIN>;
$seq="\U$Seq";
unless($seq eq "ATGCGTA\n" )
{
  print "U entered a wrong string !!!!!!!! ";
}
```

Switch:

Multiple conditions can be checked.

```
        use Switch;
        print "Enter 1 nucleotide base:";
        val=<stdin>;
        chomp $val;
        $Base="\U$val";
        print "$Base";
        switch ($Base)
        {
                case 'A'  { print ":Adinine" }
                case 'T'  { print ":Thiamine"}
                case 'G'  { print ":Guanine" }
                case 'C'  { print ":Cytosine" }
        else  { print ":Its Not a nucleotide base!!!!!!!!!" }
        }
```

Arrays:
An array is a list of elements i.e. it as a collection of scalars combined to one name. Though its primary use is being called with @, it shares with scalars and hashes the symbol $. After all, an array is a bunch of scalars linked together.

Array declaration/initialization:

```
@array = ();
my @aminoAcids = ('S','H','L','W','Q');
```

To print an entire array:

```
print @aminoAcids;
```

To print a single position of array: we must treat it as a single element and use it's scalar sign $.

```
print $numbers[1];
```

Perl begin their numbering system at zero so for the three elements we have, the results would be [0] = 1, [1] = 2, [2] = 3.

Array assignments:

```
my @aminoAcids = ('S','H','L','W','Q');
$aminoAcid[6] = 'T';
$ aminoAcid[999999] = 'S';
```

Here there are 999996 undefined values because the array had to build enough room for 9999999.

Array Functions:
*Pop and Push***:**
POP and PUSH are methods we use to add or remove the last element of our array.
For Example:

```
my @aminoAcids = ('S','H','L','W','Q');
my $last_residue =pop(@aminoAcids);
print "Array left: @aminoAcids \n";
print "Amino Acid Removed : $last_residue";
Output: Array left : S H L W
Amino Acid Removed: Q
my @aminoAcids = ('S','H','L','W','Q');
push(@array,  'K');
print "Array left: @aminoAcid \n";
```

```
Output: Array left : S H L W Q K
```

Shift and Unshift:

Push and Pop work at the end of your list, similarily shift and unshift work at the beginning. Unshift will remove the first object from your list and shift adds an object to the beginning.

```
my @aminoAcids = (‘S’,’H’,’L’,’W’,’Q’);
unshift(@aminoAcids,’K’);
print “Array left: @aminoAcid \n”;

Output: Array left :K S H L W Q
```

```
my @aminoAcids = (‘S’,’H’,’L’,’W’,’Q’);
my $first_residue = shift (@aminoAcids);
print “Array left: @aminoAcid \n”;
print “Amino Acid Removed : $first_residue”;

Output: Array left : H L W Q
        Amino Acid Removed: S
```

Reversing an array:
Reversing the printing order of the array.

```
my @aminoAcids = (‘S’,’H’,’L’,’W’,’Q’);
print (reverse (@aminoAcids));
print join(" ", reverse (@aminoAcids));
Output: Q W L H S
```

Counting indexes:
$#array returns the number of indexes (last index) or in the array.

```
my @aminoAcids = (‘S’,’H’,’L’,’W’,’Q’);
print $#aminoAcids;
Output: 4
```

Counting length:
Find the number of elements in the list rather than worrying about indexes .Scalar takes any expression and returns the scalar value.

```
my @aminoAcids = (‘S’,’H’,’L’,’W’,’Q’);
print  scalar@aminoAcids;
Output: 5
```

Merging:
Merge two or more arrays to combine one larger array.

```
my @aminoAcids1 = (‘S’,’H’,’L’,’W’,’Q’);
my @aminoAcids2= (‘R’,’I’,’D’,’M’,’E’);
my @peptide =(@aminoAcids1, @aminoAcids2);
print “New Peptide: @peptide”;
Output: New Peptide: S H L W Q R I D M E
```

@peptide now contains all the amino acids from aminoAcids1 & aminoAcids2 with aminoAcids1's data first.

Array splicing:
Perl enables us to be able to remove (and optionally return) more than one element at a time. We can do this using array *splices*. A splice takes a *list*, an *offset* and a *length.*
Syntax: splice (@array, offset, length);

```
my @aminoAcids = (‘S’,’H’,’L’,’W’,’Q’);
splice(@aminoAcids, 2, 1);
print “New Peptide: @aminoAcids”;

Output: New Peptide: S H  W Q
```

Sorting:
Sort an entire list of array alphabetically.

```
my @aminoAcids =(‘S’,’H’,’L’,’W’,’Q’);
@aminoAcids=sort(@aminoAcids);
print “Sorted Peptide: @aminoAcids”;

Output: Sorted Peptide: H L Q S W
```

Deleting:
Sometimes you will want to remove a certain element from your list. This can be done using *delete*.

```
my @aminoAcids1 = (‘S’,’H’,’L’,’W’,’Q’);
delete $animals[0];
print “New Peptide: @aminoAcids”;
Output: New Peptide:  H  L W Q
```

To delete an entire array, the quickest way would be to assign it a null value.

Hashes:
A hash variable contains a collection of key/value pairs, arranged such that you can easily use any key to find its associative value, so hashes are often called *associative arrays*. Hashes are very adaptable when it comes to data holding just like arrays. They are able to contain regular data (as seen above), scalar variables, other hashes, arrays, and so on...

Hash Declaration:
Hash variable name start with a % sign.
Example:

```
%basename=("Adenine",'A', "Guanine" ,'G', "Cytosine",'C', "Thyamine",'T',);
Or
%basename=("Adenine" =>'A',
                 "Guanine" =>'G',
                  "Cytosine" => 'C',
                   "Thyamine"=>'T',);
```

Printing an Hash:

```
foreach  $key(keys  %basename)
      {
              print $key . " " . $basename{$key}."\n";
      }
```

For Loops:
A for loop counts through a range of numbers, reiterating lines of code each time through the loop. The syntax is for ($start_num,, Range, $increment) {code to execute}. A for loop needs 3

items placed inside of the conditional statement to be successful. First a starting point, then a range operator, and finally the incrementing value. Below is the example.

```
# print an array
        print "Enter a DNA sequence:";
        $d=<STDIN>;
        $seq="\U$d";
        chomp$seq;      # removes the \n character from end of the sequence
        @seq=split(//,$seq);
        $len=@seq;
        for($i=0;$i<$len;$i++)
        {
          print "$seq[$i]";
        }
```

Foreach Loops:

Foreach is designed to work with arrays. Say you want to execute some code foreach element within an array. Here's how you might go about it.

```
# print an array
        print "Enter a DNA sequence:";
        $d=<STDIN>;
        $seq="\U$d";
        chomp$seq;
        @seq=split(//,$seq);
        $f=0;
        foreach $s(@seq)
        {
          print "";
        }
```

While:

While loops continually reiterate as long as the conditional statement remains true. It is very easy to write a conditional statement that will run forever especially at the beginner level of coding. While loops are probably the easiest to understand. The syntax is:

```
        while (conditional statement)
         {
        execute code;
        }
# print an array
        print "Enter a DNA sequence:";
        $d=<STDIN>;
        $seq="\U$d";
        chomp$seq;
        @seq=split(//,$seq);
        $len=@seq;
        $i=0;
        While($i<$len){
          print "$seq[$i]";
          $i++;
        }
```

Next, Last, and Redo:

Outlined below are several interrupts that can be used to redo or even skip reiterations of code. These functions allow you to control the flow of your while loops.

->*Next* is a block of code that is executed after the loop is finished, but before the loop is evaluated for the next iteration
->*Last* stops the looping immediately (like break)
->*Redo* will execute the same iteration over again.

Next:

```
print "Enter a DNA sequence:";
$d=<STDIN>;
$seq="\U$d";
chomp$seq;
@seq=split(//,$seq);
$len=@seq;
$i=0;
While($i<$len){
  if($seq[$i] eq " "){
     next;                  #Skips the printing if any gap found in sequence
    }
  print "$seq[$i]";
  $i++;
}
```

Last:

```
print "Enter a DNA sequence:";
$d=<STDIN>;
$seq="\U$d";
chomp$seq;
@seq=split(//,$seq);
$len=@seq;
$i=0;
do{
  if($i==$len/2) {
        $F=1;
        last; # terminates the loop printing in middle of the sequence
    }
  print "$seq[$i]";
  $i++;
} While($i<$len);
if($f==1)
    {
       print"Loop ended in mid of the sequence!!!!!!!!!"
    }
```

Redo:

```
print "Enter The sequence:";
$dnaSeq=<STDIN>;
chomp($dnaSeq);
 if (! length($dnaSeq)) {
        print("Msg: Zero length input. Please try again\n");
        redo;
    }
 print "Thank you!!!!!!! \n Sequence entered is: $dnaSeq \n";
```

do–While:

Perl's do .. while loop is almost exactly the same as the while loop with one crucial difference - the code is executed before the expression is evaluated. It is used to loop through a designated block of code while a specific condition is evaluated as true.

```
do
{
...
} while (expression);
```

Perl starts by executing the code inside the do .. while block, then the expression inside the parenthesis is evaluated. If the expression evaluates as true the code is executed again, and will continue to execute in a loop until the expression evaluates as false. Let's look at an example of Perl's while loop in action and break down exactly how it works, step by step.

# print an array

```
print "Enter a DNA sequence:";
$d=<STDIN>;
$seq="\U$d";
chomp$seq;
@seq=split(//,$seq);
$len=@seq;
$i=0;
do{
    print "$seq[$i]";
   $i++;
}while($i<$len);
```

Bioinformatics deals with strings i.e. sequences. Sequences are set of defined character having some biological meaning thus string manipulating functions have great biological relevance. The manipulation of string or sequences are performed by certain functions and regular expression which are discussed below:

Perl provides several functions to perform various operations on strings.

1) *chop( )* :- Removes last character of a string.

   Eg:
   ```
   $a=ATGC;
   Chop($a);
   Output: ATGC
   ```

2) *chomp( )*:-Removes last character, if its only '\n'.ie any newline character.

   Eg:
   ```
   $a=CGTA\n;
   Chomp($a);
   Output: CGTA
   ```

3) *length( ) :*- Returns the length in characters of an expression evaluated in a scalar context.

   Eg.
   ```
   $a=TTAGCG;
   $x=length($a);
   Output: $x=6
   ```

4) *index( ) :*- Returns the position (starting at 0)of the first (leftmost) occurrence of a substring in a string. If substring is not found, -1 is returned.

   Eg.
   ```
   index(STRING, SUBSTRING, POSITION);
                  Or
   index(STRING, SUBSTRING)
   $x=index ("testing","t");    #$x = 0
   $x=index ("testing","hi");   #$x = -1
   $x=index ("testing","t",2);   #$x = 3
   ```

5) *rindex( ) :-* Returns the position (starting at 0) of the last (rightmost) occurrence of a substring in a string. If substring not found, -1 is returned.

Eg:

```
rindex(STRING, SUBSTRING, POSITION)
rindex(STRING, SUBSTRING)
$x=rindex(“testing”, “t”);  # $x is 3
$x=rindex(“testing”,”t”,2); # $x is 0
```

6) *substr( ):-* Extracts a substring from a string. If length is not specified, everything to the end of the string is extracted.

Eg:

```
$x= substr(“testing”,2) ;  #$x is “sting”
$x= substr(“testing”, 2, 3);  # $x is “sti”
```

7) *reverse( ):-* Reverses the whole string.

Eg :

```
$x= “ATTGCTGATG”;
reverse($x);
output: GTAGTCGTTA
```

8) *hex( ):-* Returns the decimal value of an expression interpreted as a hex string. Hex function can handle string with or without a leading 0x or 0X.

Eg:

```
$x=hex (“0xa2” );   # $x is 162
$x=hex (“a2”);      # $x is 162
```

9) *oct( ):-* Returns the decimal value of an expression interpreted as an octal string.

Eg:

```
$x=oct (“042”);  # $x is 34
$x=oct (“42”);  # $x is 34
```

Regular Expressions:

Common metacharacters :

^	beginning of string
$	end of string
.	any character except new line
*	match 0 or more times
+	match 1 or more times

Examples:

1)
```
$string1 =”ACGT ACGG\n”;
if($string1=~m/^AC/){
 print “$string starts with character ‘AC’ \n”;
}
```

2)
```
$string1 =”GGTA CGTA\n”;
if($string1=~m/GT$/){
  print “$string starts ends character ‘GT’ \n”;
}
```

3) $string1 =”ACGG GTGG\n”;
   if($string1=~m/C..G/)   {
     print “$string contains 2 characters b/w ‘C’ and ‘G’ \n”;
   }

4) $string1 =”ACCG TGTG\n”;
   if($string1=~m/l+/){
     print “$string contains one or more ‘l’ \n”;
   }

Substitution:

A slight variation of the match operator can be used to search and replace. Put an "s" in front of the pattern and follow the match pattern with a replacement pattern.

```
$seq= “TTGATAGCAGTACCGTAGC”
$seq =~ s/T/A/ig;      # replaces all T’s with A.
```

Character Translate –tr:

The tr// operator goes through a string and replaces characters with other characters.

```
$string=’acbtaab efgbacta h’;
$string =~ tr/a/b/;                                # change all a's to b's
$string =~ tr/A-Z/a-z/;         # change uppercase to lowercase (actually lc() is better for this)
$cnt=($string=~tr/a//);         #counts the number of a in string
print “$cnt”;
```

Modifiers:

Matching	
i	match should not be case sensitive
g	do the replacement repeatedly in the target up to complete length
c	Complements the process
s	Squeezes
o	Match once

Examples:

1) $dna=’atgcatt’;
   $dna=~s/T/U/i;   # replaces 1st ‘t’ with ‘u’
   print”$dna” ;
2) $dna=’atgcagtcatacgtgactgacgtttagca’;
   $dna=~s/t/u/g;   # replaces every ‘ t’ with u
   print”$dna” ;
3) $dna=’ATGCCATACXXNNNNXXATCATGAA’;
   $dna=~tr/ATGC/ /cs; # It’ll replace characters other than ATGC and squeezes multiple spaces
   Print ”Now the Sequence is: $dna”;

File Handling:

In every language file handling is a very important task. Opening a file in perl is straightforward: open FILE, "filename.txt" or die $!; A file is opened in 3 types of modes Read, write or append

mode	operand		
read	<		
write	>	✓	✓
append	>>	✓	

Each of the above modes can also be prefixed with the + character to allow for simultaneous reading and writing.

mode	operand	create	truncate
read/write	+<		
read/write	+>	✓	✓
read/append	+>>	✓	

Reading files:
Example-

```
print "This program will open an example file. ";
open (example, "genbank_file.txt");
@text = <example>;
print "The first line of the file reads : $text ";
foreach $line(@text)
{
  print "$line\n";
}
close (example);
```

Writing files:
Example-

```
open (example, "genbank_file.txt") ;
open (backup, ">backup.txt") ;
@copy_this = <example>;
print backup (@copy_this);
close (example);
close (backup);
```

Appending to a file:
It can be accomplished in exactly the same manner - apart from specifying the appropriate (>>) mode of course.
Instead write is used to write formatted records to file, a subject outside the scope of this article.

```
print "write the sequence u want to append in fasta file:  ";
$new_info =<STDIN> ;
open (example,">>fasta.txt") ;
print example ($new_info);
print "the sequence is appended in file!!!";
close (example);
```

Closing files:
Once reading and writing is completed the file should be closed. If we forget to close a file handle Perl will do it for us before our script exists but it is good practice to close yourself what we have opened.

Subroutine:
Perl allows the user to define their own functions, called *subroutines*. They may be placed anywhere in your program but it's probably best to put them all at the beginning or all at the end. A subroutine has the form.

```
sub mysubroutine
{
   print "Not a very interesting routine\n";
   print "This does the same thing every time\n";
}
```

regardless of any parameters that we may want to pass to it. All of the following will work to call this subroutine. Notice that a subroutine is called with an & character in front of the name:

```
&mysubroutine;                  # Call the subroutine
&mysubroutine($_);             # Call it with a parameter
&mysubroutine(1+2, $_);         # Call it with two parameters
```

Parameters:
In the above case the parameters are acceptable but ignored. When the subroutine is called any parameters are passed as a list in the special @_ list array variable. This variable has absolutely nothing to do with the $_ scalar variable. The following subroutine merely prints out the list that it was called with. It is followed by a couple of examples of its use.

```
sub printargs
{
   print "@_\n";
}

&printargs("gene", "protein");     # Example prints "gene protein"
&printargs("dna", "rna", "protein"); # Prints "dna rna protein"
```

Just like any other list array the individual elements of **@_** can be accessed with the square bracket notation:

```
sub printfirsttwo
{
   print "Your first argument was $_[0]\n";
   print "and $_[1] was your second\n";
}
```

Again it should be stressed that the indexed scalars **$_[0]** and **$_[1]** and so on have nothing to with the scalar **$_** which can also be used without fear of a clash.

Returning values:
Result of a subroutine is always the last thing evaluated. This subroutine returns the maximum of two input parameters. An example of its use follows.

```
sub maximum
{
  if ($_[0] > $_[1])
  {
         $_[0];
  }

  else
  {
         $_[1];
  }
}

$biggest = &maximum(37, 24);     # Now $biggest is 37
```

The &printfirsttwo subroutine above also returns a value, in this case 1. This is because the last thing that subroutine did was a **print** statement and the result of a successful **print** statement is always 1.

Modules:

A *module* or a *package* is a collection of subroutines, usually stored in a separate file with a ".pm" suffix (Perl Module).An example is given bellow for creation of module

```
MyModule.pm
      package MyModule;
      use strict;
      use Exporter;
      sub func1  { print "ATGCGTCGT";  }
      sub func2  { print  "AUGCGUGC" }

      1;
```

To get a namespace firstly declare a package name. This helps ensure our module's functions and variables remain separate from any script that uses it. Use strict is a very good idea for modules to restrict the use of global variables.Exporter module need to be used to export our functions from the MyModule:: namespace into the main:: namespace to make them available to scripts that 'use' MyModule.

MyScript.pl (A simple script to use MyModule)

```
      #!/usr/bin/perl -w

       use strict;
       use MyModule;
       print func1(),"\n";
       print func2(),"\n";
```

Tools of Bioperl:

The modules are the main tools of Bioperl, A *module* or a *package* is a collection of subroutines, usually stored in a separate file with a ".pm" suffix (Perl Module). The subroutines of a module should deal with a *well-defined task*. Subroutine can be invoked from within the *namespace* of that package: PACKAGE::SUBROUTINE(...)

Writing a Bioperl module:

A module is usually written in a separate file with a ".pm" suffix.

The name of the module is defined by a "package" line at the beginning of the file:

```
 package FileHandle;
 use strict;
 sub readDirectory {
 ...
 ...  The last line of the module must be a true value, so usually we just add:
 1;
```

Commonly used modules in Bioperl are listed below the details of all the modules can be retrieved from http://doc.bioperl.org/releases/bioperl-1.4/. These are………..

Bio::Align	**Bio:: AlignIO**	**Bio::Annotation**
Bio::Assembly	**Bio::Assembly::IO**	**Bio::Biblio**
Bio::Cluster	**Bio::ClusterIO**	**Bio::CodonUsage**
Bio::Coordinate	**Bio::Coordinate::Result**	**Bio::DB**
Bio::DB::Biblio	**Bio::DB::Flat**	**Bio::DB::Flat::BDB**
Bio::DB::GFF	**Bio::DB::GFF::Adaptor**	**Bio::DB::GFF::Adaptor::dbi**
bio::DB::GFF::Aggregator	**Bio::DB::GFF::Util**	**Bio::DB::Query**
Bio::DB::Taxonomy	**Bio::Das**	**Bio::Event**
Bio::Expression	**Expression::FeatureGroup**	**Bio::Expression::FeatureSet**
Bio::Factory	**Bio::Graphics**	**Bio::Graphics::FeatureFile**
Bio::Graphics::Glyph	**Bio::Index**	**Bio::LiveSeq::IO**
Bio::Location	**Bio::Map**	**Bio::MapIO**
Bio::Matrix	**Bio::Matrix::IO**	**Bio::Matrix::PSM**
Bio::Matrix::PSM::IO	**Bio::Ontology**	**Bio::OntologyIO**
Bio::OntologyIO::Handlers	**Bio::Phenotype**	**Bio::Phenotype::MeSH**
Bio::Phenotype::OMIM	**Bio::PopGen**	**Bio::PopGen::IO**
Bio::PopGen::Simulation	**Bio::Restriction**	**Bio::Restriction::Enzyme**
Bio::Restriction::IO	**Bio::Root**	**Bio::Search**
Bio::Search::HSP	**Bio::Search::Hit**	**Bio::Search::Iteration**
Bio::Search::Result	**Bio::SearchIO**	**Bio::SearchIO::Writer**
Bio::Seq	**Bio::Seq::Meta**	**Bio::SeqFeature**
Bio::SeqFeature::Gene	**Bio::SeqFeature::SiRNA**	**Bio::SeqFeature::Tools**
Bio::SeqIO	**Bio::SeqIO::game**	**Bio::Structure**
Bio::Structure::IO	**Bio::Structure::SecStr**	**Bio::Structure::SecStr::DSSP**
Bio::Structure::SecStr::STRIDE	**Bio::Symbol**	**Bio::Taxonomy**
Bio::Tools	**Bio::Tools::Alignment**	**Bio::Tools::Analysis**
Bio::Tools::Analysis::DNA	**Bio::Tools::Analysis::Protein**	**Bio::Tools::BPlite**
Bio::Tools::Blast	**Bio::Tools::EMBOSS**	**Bio::Tools::HMMER**
Bio::Tools::Phylo	**Bio::Tools::Phylo::Molphy**	**Bio::Tools::Phylo::PAML**
Bio::Tools::Phylo::Phylip	**Bio::Tools::Prediction**	**Bio::Tools::Primer**
Bio::Tools::Primer::Assessor	**Bio::Tools::Run**	**Bio::Tools::Sim4**
Bio::Tree	**Bio::TreeIO**	**Bio::Variation**
Bio::Variation::IO		

Bioperl objects:

The Bioperl objects play the key role in Bioperl. These are generally of 3 types: Sequence objects, location objects, interface or implementation objects

Sequence objects:

Seq :Seq is the central sequence object in bioperl. This is probably the object that is used to describe a DNA, RNA or protein sequence in bioperl. Most common sequence manipulations can be performed with Seq. Seq objects may be created automatically while reading a file containing sequence data using the SeqIO object. In addition to storing its identification labels and the sequence itself, a Seq object can store multiple annotations and associated ``sequence features", such as those contained in most Genbank and EMBL sequence files. This capability can be very useful - especially in development of automated genome annotation systems

PrimarySeq : Incase of handling hundreds or thousands sequences at a time, then the overhead of adding annotations to each sequence can be significant. For such applications PrimarySeq object is used. PrimarySeq is basically a stripped-down version of Seq. It contains just the sequence data itself and a few identifying labels (id, accession number, alphabet = dna, rna, or protein), and no features. For applications with hundreds or thousands or sequences, using PrimarySeq objects can significantly speed up program execution and decrease the amount of RAM the program requires.

RichSeq :RichSeq objects store additional annotations beyond those used by standard Seq objects. RichSeq objects are created automatically when Genbank, EMBL, or Swissprot format files are read by SeqIO.

SeqWithQuality: These objects are set to manipulate sequences with quality data, like those produced by phred.

LocatableSeq: It is a Seq object which is part of a multiple sequence alignment. It has start and end positions indicating from where in a larger sequence it may have been extracted. It also may have gap symbols corresponding to the alignment to which it belongs. It is used by the alignment object SimpleAlign and other modules that use SimpleAlign objects. These are created automatically while creating an alignment.

RelSegment : RelSegment objects are useful to manipulate the origin of the genomic coordinate system. This situation may occur when looking at a sub-sequence (e.g. an exon) which is located on a longer underlying sequence such as a chromosome or a contig. Such manipulations may be important, for example when designing a graphical genome browser,If the code may need such a capability

LargeSeq :This object is a special type of Seq object used for handling very long sequences (e.g. > 100 MB).

LiveSeq: LiveSeq addresses the problem of features whose location on a sequence changes over time. This can happen, for example, when sequence feature objects are used to store gene locations on newly sequenced genomes - locations which can change as higher quality sequencing data becomes available. Although a LiveSeq object is not implemented in the same way as a Seq object, LiveSeq does implement the SeqI interface

SeqI: These objects are Seq ``interface objects'' They are used to ensure bioperl's compatibility with other software packages. SeqI and other interface objects are not likely to be relevant to the casual Bioperl user.

Location objects

A Location object is designed to be associated with a Sequence Feature object in order to show where the feature is on a longer sequence. Location objects can also be standalone objects used to described positions. The reason why these simple concepts have evolved into a collection of rather complicated objects is that:
1) Some objects have multiple locations or sub-locations (e.g. a gene's exons may have multiple start and stop locations)
2) In unfinished genomes, the precise locations of features is not known with certainty.

Interface objects and implementation objects:

An interface is solely the definition of what methods one can call on an object, without any knowledge of how it is implemented. An implementation is an actual, working implementation of an object. In languages like Java, interface definition is part of the language. In bioperl, the interface objects usually have names like Bio::MyObjectI, with the trailing I indicating it is an interface object. The interface objects mainly provide documentation on what the interface is, and how to use it, without any implementations (though there are some exceptions). Although interface objects are not of much direct utility to the casual Bioperl user, being aware of their existence is useful since they are the basis to understand how bioperl programs can communicate with other bioinformatics projects and computer languages such as Ensembl and biopython and biojava.

Commonly used Bio::perl Functions/subroutines:

- **get_sequence** - gets a sequence from standard, internet accessible databases
- **read_sequence** - reads a sequence from a file
- **read_all_sequences** - reads all sequences from a file

- **new_sequence** - makes a Bioperl sequence just from a string
- **write_sequence** - writes a single or an array of sequence to a file
- **translate** - provides a translation of a sequence
- **translate_as_string** - provides a translation of a sequence, returning back just the sequence as a string
- **blast_sequence** - BLASTs a sequence against standard databases at NCBI
- **write_blast** - writes a blast report out to a file
- **reverse_complement/revcom** - find out reverse complement of nucleotide sequence
- **reverse_complement_as_string/revcom_as_string**- provides a reverse complement of a sequence, returning back just the sequence as a string

Applications and utility of Bioperl:
Bioperl provides software modules for many of the typical tasks of bioinformatics programming. These include:

- Accessing sequence data from local and remote databases
- Transforming formats of database/ file records
- Manipulating individual sequences
- Searching for similar sequences
- Creating and manipulating sequence alignments
- Searching for genes and other structures on genomic DNA
- Developing machine readable sequence annotations

Few examples are cited below:

Accessing sequence data from local and remote databases:
One can directly enter data sequence data into a bioperl Seq object, eg:
```
  $seq = Bio::Seq->new(-seq              => 'actgtggcgtcaact',
                       -description      => 'Sample Bio::Seq object',
                       -display_id       => 'something',
                       -accession_number => 'accnum',
                       -alphabet         => 'dna' );
```

Accessing remote databases (Bio::DB::GenBank, etc):
Accessing sequence data from the principal molecular biology databases is straightforward in bioperl. Data can be accessed by means of the sequence's accession number or id. Batch mode access is also supported to facilitate the efficient retrieval of multiple sequences. For retrieving data from genbank, for example, the code could be as follows:
```
   $gb = new Bio::DB::GenBank();
   # this returns a Seq object :
   $seq1 = $gb->get_Seq_by_id('MUSIGHBA1');
   # this also returns a Seq object :
   $seq2 = $gb->get_Seq_by_acc('AF303112');
   # this returns a SeqIO object,  which can be used to get a Seq
object :
   $seqio = $gb->get_Stream_by_id(["J00522", "AF303112", "2981014"]);
   $seq3 = $seqio->next_seq;
```

Transforming sequence files (SeqIO):
A common and tedious bioinformatics task is that of converting sequence data among the many widely used data formats. The following example are -
```
   use Bio::SeqIO;
   $in  = Bio::SeqIO->new(-file => "inputfilename",
                          -format => 'Fasta');
   $out = Bio::SeqIO->new(-file => ">outputfilename",
                         -format => 'EMBL');
  while ( my $seq = $in->next_seq() ) {$out->write_seq($seq); }
```
In addition, the Perl ``tied filehandle" syntax is available to SeqIO, allowing to use the standard <> and print operations to read and write sequence objects, eg:
```
   $in  = Bio::SeqIO->newFh(-file => "inputfilename" ,
                           -format => 'fasta');
   $out = Bio::SeqIO->newFh(-format => 'embl');
   print $out $_ while <$in>;
```

Transforming alignment files (AlignIO)
Data files storing multiple sequence alignments also appear in varied formats. AlignIO is the Bioperl object for conversion of alignment files.
```
   use Bio::AlignIO;
   my $io = Bio::AlignIO->new(-file   => "receptors.aln",
                             -format => "clustalw" );
```
Like wise…
```
   use Bio::AlignIO;
   $in  = Bio::AlignIO->new(-file => "inputfilename" ,
                           -format => 'fasta');
   $out = Bio::AlignIO->new(-file => ">outputfilename",
                           -format => 'pfam');
   while ( my $aln = $in->next_aln() ) { $out->write_aln($aln); }
```

Manipulating sequence data with Seq methods:
These methods return strings or may be used to set values:
```
   $seqobj->display_id();       # the human read-able id of the
                                  sequence
   $seqobj->seq();              # string of sequence
   $seqobj->subseq(5, 10);       # part of the sequence as a string
   $seqobj->accession_number(); # when there,  the accession number
   $seqobj->alphabet();         # one of 'dna', 'rna', 'protein'
   $seqobj->primary_id();  #a unique id for this sequence irregardless
                          # of its display_id or accession number
  $seqobj->description();      # a description of the sequence
```

The following methods return an array of Bio::SeqFeature objects:
```
   $seqobj->get_SeqFeatures;      # The 'top level' sequence features
   $seqobj->get_all_SeqFeatures;   # All sequence features,  including
sub-
                                  # seq features
```
For a comment annotation, use:
```
   use Bio::Annotation::Comment;
   $seq->annotation->add_Annotation('comment',
      Bio::Annotation::Comment->new(-text => 'some description');
```
For a reference annotation, you can use:
```
   use Bio::Annotation::Reference;
   $seq->annotation->add_Annotation('reference',
      Bio::Annotation::Reference->new(-authors  => 'author1, author2',
                                      -title    => 'title line',
                                      -location => 'location line',
                                      -medline => 998122);
```
The following methods returns new sequence objects, but do not transfer the features from the starting object to the resulting feature:
```
   $seqobj->trunc(5, 10);  # truncation from 5 to 10 as new object
   $seqobj->revcom;       # reverse complements sequence
   $seqobj->translate;    # translation of the sequence
```

Many of these methods are self-explanatory. However, the flexible `translation ()` method needs some explanation. Translation in bioinformatics can mean two slightly different things:

1. Translating a nucleotide sequence from start to end.
2. Translate the actual coding regions in mRNAs or cDNAs.

The Bioperl implementation of sequence translation does the first of these tasks easily. Any sequence object which is not of alphabet 'protein' can be translated by simply calling the method which returns a protein sequence object:

```
        $prot_obj = $my_seq_object->translate;
```

All codons will be translated, including those before and after any initiation and termination codons. For example, **ttttttatgccctaggggg** will be translated to **FFMP*G**
However, the `translate()` method can also be passed several optional parameters to modify its behavior. For example, you can tell `translate()` to modify the characters used to represent terminator (default '*') and unknown amino acids (default 'X').

```
        $prot_obj = $my_seq_object->translate(-terminator => '-');
        $prot_obj = $my_seq_object->translate(-unknown => '_');
```

The frame of the translation can also be determine. The default frame starts at the first nucleotide (frame 0). To get translation in the next frame, we would write:

```
        $prot_obj = $my_seq_object->translate(-frame => 1);
```

The codontable_id argument to `translate()` makes it possible to use alternative genetic codes.. For example, for mitochondrial translation:

```
        $prot_obj = $seq_obj->translate(-codontable_id => 2);
```

To translate full coding regions (CDS) the way major nucleotide databanks EMBL, GenBank and DDBJ do it, the `translate()` method has to perform more checks. Specifically, `translate()` needs to confirm that the sequence has appropriate start and terminator codons at the very beginning and the very end of the sequence and that there are no terminator codons present within the sequence in frame 0. In addition, if the genetic code being used has an atypical (non-ATG) start codon, the `translate()` method needs to convert the initial amino acid to methionine. These checks and conversions are triggered by setting ``complete'' to 1:

```
        $prot_obj = $my_seq_object->translate(-complete => 1);
```

If ``complete'' is set to true and the criteria for a proper CDS are not met, the method, by default, issues a warning. By setting ``throw'' to 1, one can instead instruct the program to die if an improper CDS is found, e.g.

```
        $prot_obj = $my_seq_object->translate(-complete => 1,
                                               -throw => 1);
```

Custom codon table can also be created and pass this object to translate:

```
        $prot_obj    =    $my_seq_object->translate(-codontable    =>
      $table_obj);
```

`translate()` can also find the open reading frame (ORF) starting at the 1st initiation codon in the nucleotide sequence, regardless of its frame, and translate that:

```
        $prot_obj = $my_seq_object->translate(-orf => 1);
```

Most of the codon tables used by `translate()` have initiation codons in addition to ATG, including the default codon table, NCBI ``Standard''. To tell `translate()` to use only ATG, or atg, as the initiation codon set -start to ``atg'':

```
        $prot_obj = $my_seq_object->translate(-orf => 1,
                                               -start => "atg" );
```

The -start argument only applies when -orf is set to 1.
Last trick. By default `translate()` will translate the termination codon to some special character (the default is *, but this can be reset using the -terminator argument). When -complete is set to 1 this character is removed. So, with this:

```
        $prot_obj = $my_seq_object->translate(-orf => 1,
                                              -complete => 1);
```

the sequence **tttttatgccctaggggg** will be translated to **MP**, not **MP***.

Obtaining basic sequence statistics (SeqStats, SeqWord):
SeqStats object provides methods for obtaining the molecular weight of the sequence as well the number of occurrences of each of the component residues (bases for a nucleic acid or amino acids for a protein.) For nucleic acids, SeqStats also returns counts of the number of codons used. For example:

```
        use SeqStats;
        $seq_stats  = Bio::Tools::SeqStats->new($seqobj);
        $weight = $seq_stats->get_mol_wt();
        $monomer_ref = $seq_stats->count_monomers();
        $codon_ref = $seq_stats->count_codons();  # for nucleic acid
        sequence.
```

Identifying restriction enzyme sites (Bio::Restriction):
Another common sequence manipulation task for nucleic acid sequences is locating restriction enzyme cutting sites. Bioperl provides the Bio::Restriction::Enzyme, Bio::Restriction::EnzymeCollection, and Bio::Restriction::Analysis objects for this purpose. These modules replace the older module Bio::Tools::RestrictionEnzyme. A new collection of enzyme objects would be defined like this:

```
         use Bio::Restriction::EnzymeCollection;
         my $all_collection = Bio::Restriction::EnzymeCollection;
```

Bioperl's default Restriction::EnzymeCollection object comes with data for more than 500 different Type II restriction enzymes. A list of the available enzyme names can be accessed using the available_list() method, but these are just the names, not the functional objects. You also have access to enzyme subsets. For example to select all available Enzyme objects with recognition sites that are six bases long one could write:

```
        my $six_cutter_collection = $all_collection->cutters(6);
        for my $enz ($six_cutter_collection){
           print    $enz->name,    "\t",    $enz->site,    "\t",    $enz-
      >overhang_seq, "\n";
           # prints name,  recognition site,  overhang
        }
```

There are other methods that can be used to select sets of enzyme objects, such as `unique_cutters()` and blunt_enzymes(). You can also select a Enzyme object by name, like so:

```
        my $ecori_enzyme = $all_collection->get_enzyme('EcoRI');
```

Once an appropriate enzyme has been selected,the sites for that enzyme on a given nucleic acid sequence can be obtained using the `fragments()` method. The syntax for performing this task is:

```
         use Bio::Restriction::Analysis;
         my $analysis = Bio::Restriction::Analysis->new(-seq => $seq);
         # where $seq is the Bio::Seq object for the DNA to be cut
         @fragments =  $analysis->fragments($enzyme);
         # and @fragments will be an array of strings
```

To get information on isoschizomers, methylation sites, microbe source, vendor or availability, EnzymeCollection need to be created directly from a REBASE file, like this:

```
        use Bio::Restriction::IO;
        my $re_io = Bio::Restriction::IO->new(-file => $file,
                                              -format=>'withrefm');
        my $rebase_collection = $re_io->read;
```

Running BLAST (using RemoteBlast.pm):

Bioperl supports remote execution of blasts at NCBI by means of the RemoteBlast object.

A skeleton script to run a remote blast might look as follows:

```
        $remote_blast = Bio::Tools::Run::RemoteBlast->new (
                 -prog  =>  'blastp',   -data  =>  'ecoli',   -expect  =>
      '1e-10' );
        $r = $remote_blast->submit_blast("t/data/ecolitst.fa");
        while (@rids = $remote_blast->each_rid ) {
            for    $rid    (    @rids    )    {$rc    =    $remote_blast-
      >retrieve_blast($rid);}
        }
```