Introduction to Perl Programming for Bioinformatics

```
use Bio::Seq;
my $io = Bio::SeqIO->new( -file => "input.fasta",
                          -format => "fasta" );

my $firstSeq = $io->next_seq();

print "The sequence id is: " . $firstSeq->display_id . "\n";
print "The sequence is: " . $firstSeq->seq . "\n";
print "The sequence length is: " . $firstSeq->length . "\n";
print "The first 10 bases of the sequence is: " .
      $firstSeq->subseq(1,10) . "\n";
my $revComplement = $firstSeq->revcom();
print "The reverse complement of the sequence is: " .
      $revComplement->seq . "\n";
```

IDeA Network of Biomedical Research Excellence
INBRE II
Hawaii State Research and Education Partnership

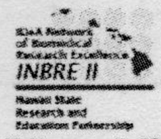
Pacific Center for Emerging Infectious Diseases Research

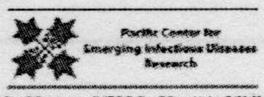
RMATRIX
RCMI Multidisciplinary And Translational Research Infrastructure eXpansion

Workshop sponsored by COBRE P20RR018727, INBRE P20RR016467, RCMI G12RR003061-26 and RMATRIX U54RR026136

Introduction to Perl Programming for Bioinformatics

Mahdi Belcaid – mahdi@hawaii.edu
David Schanzenbach – davidls@hawaii.edu





Day 1: Outline

- Introduction to Perl
- Working with scalars
- Conditional blocks and loops
- Lists and arrays
- Input, output and hashes

Day 2: Outline

- Subroutines
- Regular expressions
- Perl references
- Introduction to Bioperl
- Practical exercises and Open Questions

Quick Introduction to Perl

Meet Perl

- Perl stands for **P**ractical **E**xtraction and **R**eport **L**anguage
- Designed by Larry Wall and first released in 1987
- Borrows from many languages, such as C, awk, Lisp and many others
- Originally developed for text manipulation (System admin tasks)
- Now used for a wide range of tasks including web and network programming, astronomy, bioinformatics, etc.

What is Perl Good For

- Text processing
- Rapid prototyping
- Task automation
- Make portable code: write once, run anywhere
- Making large libraries of useful modules, such as in bioinformatics
- "Making easy things easy and complicated things possible"
  - There is more than one way to do it
  - From baby Perl to advanced perlish constructs

CPAN

What is Perl Bad For?

- Tasks where complex data-structures are needed
  - Very complex relationship between the data elements.
- Performance-critical applications
  - Complex, long-lived software projects with more than one author
- Doing important things in more than one way
  - Should there be one – and preferably only one – obvious way to do it?

Remember

- "Perl is an empirical Science. You learn about it but trying things out." Larry Wall
- You have to be diligent in testing what you are unsure of.
  - Or else you might introduce some nasty bugs

Hello World in Perl

```
#!/usr/bin/perl
# This is a sample program

use strict;
use warnings;

Print "Hello world!\n";
```

- Let's execute the program

```
mandi@mandi-laptop:~/Desktop$ perl helloWorld.pl
Hello World!
```

- A Perl "interpreter" is used to interpret the script.
- Scripts are interpreted from top to bottom
  - You can also run the script by executing it.
- Perl is **case sensitive.**
  - Use Warning is different from use warnings

Anatomy of a Perl Script

```
1: #!/usr/bin/perl
2: # This is a sample program

3: use strict;
4: use warnings;

5: Print "Hello world!\n";
```

```
Hello world
```

- **Line 1** is a directive to the shell.
  - Tells the shell which interpreter to use.
- **Line 2** is a comment.
  - Except for ***#!*** (hashbang), anything that starts with # is a comment.
- **Lines 3** and **4** are **pragmas**.
  - Pragmas are instructions to the interpreter
- **Line 5** Is a statement.
  - Calls the `print` function on a *string literal* and ends with ';' character. "\n" instructs Perl to print a new line.

Running Perl: Demonstration

- Download the workshop archive from:
  - http://dl.dropbox.com/u/28270/perl_workshop.zip
- Running a program from the command line.
- Using the IDE
  - Open the Komodo and create a new project
  - Add the helloWorld.pl file to the project
  - Run the file
  - Run with error "Missing end string delimiter"

Scalars

The Elementary Variable Type in Perl

How Is Information Handled?

- In Perl information is classified into three categories:
  - One dimensional
    - Ex. The string *"ATGTGA"*
  - Multi-dimensional
    - Ex. The x,y coordinates *(10, 12)*
  - Dictionary like
    - Ex. The side chain charge of different a.a
    - Alanine: Neutral, Arginine: positive, Asparagine: neutral, etc.
- In programming, variables are used to store information
  - Able to refer to information using meaningful names
  - Allows the modification of stored information
  - Ex. Having a variable called MolType, with a value of DNA or RNA, allows you to use it anywhere in the program.

Variables in Perl

- Variable names have a specific requirements
  - Start with alphabetical character
  - Contain any character [Aa-Zz] or [1-9] or _
  - Variable names are case sensitive
  - Should not contain any spaces
- Perl has three major variable types: **scalars**, **arrays** and **hashes**.
  - They can be distinguished with the "funny characters" prefixing the variable name.
  - Scalars have **$**, array have **@** and hashes have **%**
- A variable in Perl needs to be declared using the keyword "**my**" when it is first introduced.

one value

> one value

dictionary

Scalars

- Scalars are the simplest data-types that Perl manipulates
  - Store one dimensional information
  - Start with the "$" funny character
- To assign a value to a scalar, the "=" sign is used
- Scalar can hold a number or a string literal

```
my $length = 12;
my $evalue_threshold = -12e-16;
my $randomString = "ACGAGTGA";
my $stop_RNA = 'UAG';
```

- You can use single or double quotes to define a string.

Basic Scalar Operations

- Many operations available out of the box in Perl
  - They go in the middle of two operands
  - Can be applied to literals or variables
- For example, Perl provides basic arithmetic and string operations

```
2 + 3                    # addition
$lengths - $buffer       # subtraction
$length * 2              # multiplication
10 / 3                   # division
"2". "3"                 # concatenation of 2 and 3 -> "23"
$value++                 # auto increment. Eq. to $value = $value + 1
$value--                 # auto decrement. Eq. to $value = $value - 1
```

- If you do not assign a value to a variable, the variable is said to be undefined or **undef**
  - When using strict, Perl will complain if you try to do anything with an undef variable

Interpolation

important for later debugging

- In certain contexts, Perl replaces a variable with its content. It is called **interpolation**
- Interpolation takes place inside double quotes but not inside single quotes

```
my $x = "ACTG";
Print "1- The four letters coding for DNA are $x";
Print '2- The four letters coding for DNA are $x';
```

```
1- The four letters coding for DNA are ACGT
2- The four letters coding for DNA are $x
```

- We can use a "\" for **escaping**
  - literally printing a special character
- This works with all the special characters, including "\"

```
 my $myDNA = "ACTG";
 my $statement = "The value of \$myDNA is $myDNA";
print $statement;
```

```
The value of $myDNA is ACGT
```

Functions

- Functions in Perl are similar to Mathematical functions
  - Ex. f(x) = 1/(2x+10)
  - f(x) takes a value of x and computes (returns) 1/(2x+10)
- Functions in Perl do "almost" the same thing
  - The input, called argument, is placed between parentheses
    - Ex. f(3)
  - They execute some logic and may or may not return a value
- `print` is an example of a Perl function
  - Takes as input one or more arguments and prints it/them to the screen
  - Returns 1 (True) if the print went successfully
  - The parentheses are not always mandatory

Perl Functions to Quote Strings

- We use double quotes to delimit strings
- The **qq()** acts as a double quotes substitute (**q**uote **q**uote)

```
print "This is a polyA "AAAAAA" "          # Wrong bare-word AAAAAA
print qq(This is a polyA "AAAAAA")          # correct
```

- Similarly the q( ) function is a single quote function
  - Works the same way as qq( )

```
$x = 2;
print q($x)."\n";
Print qq($x)."\n";
---
$x
2
```

Mahdi doesn't use this

word not Known by Perl language

can also do this by escaping double quotes

Exercise

- Open the file **correctMe.pl**
- This program contains 3 errors. Find and correct them.
- Without running the program, what does it print?
- Run the program in Komodo and compare the results to your answer

```
#!/usr/bin/perl
use strict;
use warnings;

my $string = "ACG"

print q(my string is $string \n);

print "ACGT" + "$string\n";
```

Working with Strings

This is what perl is Known for

String Manipulation Overview

- Perl is ideal for working with strings
- A plethora of functions and perlish constructs to execute complicated operations
- We will examine the following operations
  - String length
  - Deleting the end of line character
  - Getting and replacing substrings
  - Changing the case of strings
- We will see other operations in later sections

String Length - 1

- The function `length()` returns the length of a string passed in as an argument
  - Counts all the characters (chars), including the end of line delimiter
  - Could be a source of bugs, be careful!
- Escaped chars, such as "`\$`" count as one char.
- To grab user input, we use the standard input handle `<STDIN>`

```
#!/usr/bin/perl
use strict;
use warnings;

print "Please enter your sequence:\n";

#We grab user input using <STDIN>
my $sequence = <STDIN>;
print "Sequence length before chomps is: " . Length($sequence) . "\n";
```

```
Please enter your sequence:
ACGTACGT
Sequence length before chomps is: 9
```

String Length - 2

- `chomp(STRING)` is another function from the Perl standard library
- Takes a string as input
- Deletes the end of line delimiter "\n"
  - Chomp won't do anything if there is not end of line char
  - Be careful not to use `chop(STRING)`
    - Removes the last char of the `STRING`

```
print "Please enter your sequence: ";
my $sequence = <STDIN>;
chomp($sequence);
print "\$sequence is $sequence and its length is: " .
        length($sequence) . "\n";
my $char = chop($sequence);
print "print \$sequence is $sequence and \$char is $char \n";
```

```
Please enter your sequence: ACGTACGT
$sequence is ACGTACGT and its length is: 8
print $sequence is ACGTACG and $char is T
```

chomp only removes \n

↓

use this whenever a user inputs its sequence

variables within qq get interpolated

String Concatenation

- We have seen concatenation with "."
- We can also use qq(STRING) to concatenate strings

```
my $x = "UAG";
my $y = "means";
my $z = "STOP";
my $s = " ";
print $x . $s . $y . $s . $z."\n";

$phrase = qq($x$s$y$s$z);
print $phrase . "\n";

$phrase = qq($x $y $z);
print $phrase . "\n";
```

```
UAG means STOP
UAG means STOP
UAG means STOP
```

- Can we use q(STRING)?

String Lists

- A list is an ordered group of scalars enclosed in parenthesis and separated by commas.
  - Ex (1,2,3)
- Instead of declaring each variable alone, we can use a list to assign values to more that one at a time.

```
my ($x, $y, $z, $s) = ("UAG", "means" "STOP", " ")
```

join concatenation function

Concatenation Function

- Given a list of strings, we can use join(EXP,LIST) to glue them together
  - EXP: Expression used to join the strings
  - LIST: List of strings to be joined

```
# We are assigning a list of literals into a list of variables
my ($x, $y, $z, $s) = ("UAG", "mean", "STOP", " ");
my $phrase = join($s, $x, $y, $z);
print $phrase . "\n";

my ($x, $y, $z, $s) = ("Perl", "is", "cool", ":");
my $phrase = join($s, $x, $y, $z);
print $phrase . "\n";
```

```
Perl is cool
Perl:is:cool
```

useful for creating a (eg. comma separated file)

Split, The Dual of Join

- Split is used to create a list by splitting a string according to a specific delimiter
- `split(/DELIMITER/, STRING)`
- `DELIMITER`: Can be a char, a string or a regular expression
- Return a list of literals

```
# Splits the string along colon
my ($x, $y, $z) = split(/:/, "AUG:means:STOP")

# Splits the string along colon
my ($x, $y, $z) = split(/ /, "AUG means STOP")
```

- Be careful when using / /
  - Splits across one single space. Use " " or regex when in doubt of the number of spaces used.

```
split(/ /, "Perl is   cool");
#returns ("Perl", "is", "", "", "", "cool")

split(" ", "Perl is   cool");
#returns ("Perl", "is", "cool")
```

- "\t" splits across tabs (key for splitting tsv files)

tab based delimiter function

Working With Substrings - 1

- In Perl, each character in a string has a position
  - The first char is at position (pos) 0, second at pos 1, etc.
  - Indices can also be negative if we count from right to left

```
         AUG is a start codon
Index +  01234567891111111111
                   0123456789
         AUG is a start codon
         21111111111987654321 - Index
         09876543210
```

- The `substr()` function allows us to extract/modify a substring
  - Flexible function that can take 2, 3, 4 arguments
- Basic usage is `substr(STRING,START)`

useful for parsing the end of a large string

Working with Substrings - 2

- `substr(STRING,START)`
  - Returns the substring starting at position `START`

```
my $phrase = "ATG is a start codon";
substr($phrase, 1);          # TG is a start codon
substr($phrase, 10);         # tart codon
substr($phrase, -8);         # rt codon
```

- `substr(STRING,START,LEN)`
  - Returns the substring of length `LEN` starting at position `START`

```
my $phrase = "ATG is a start codon";
substr($phrase, 1, 4);       # TG i
substr($phrase, 10, 10);     # tart codon
substr($phrase, -8, 1);      # t
# different than when a positive length is specified
substr($phrase, 2, -2);      # G is a start cod
```

- Useful to test one position

Substitution of a Substring

- `substr()` can also be used for replacing a substring
  - Used as an l-value in an expression

```
my $phrase = "Perl is cool";
substr($phrase, 8, 4) = "confusing sometimes";
print $phrase;
```

```
Perl is confusing sometimes
```

  - The right value does not have to be the same length as the substring

- The right side of the expression can be replaced by a fourth argument in the `substr()` expression

```
my $phrase = "Perl is cool";
substr($phrase, 8, 4, "confusing sometimes");
print $phrase;
```

```
Perl is confusing sometimes
```

will be used on a regular basis

Other Useful Methods

- `reverse(STRING)` – Returns a reversed a string
- `lc(STRING)` – Returns a string converted to lower case
- `uc(STRING)` – Returns a string converted to upper case
- `ucfirst(STRING)` – Returns string with the first char converted to upper case
- `lcfirst(STRING)` – Returns string with the first char converted to lower case
- `index(STRING,SUBSTR)` – Returns the index of the first occurrence of SUBSTR in STRING or -1 if SUBSTR is not present in STRING

```
my $dnaSeq = "ACGTacgt";
print reverse($dnaSeq);                    # tgcaTGCA
print lc($dnaSeq);                         # acgtacgt
print uc($dnaSeq);                         # ACGTACGT
print lcfirst($dnaSeq);                    # aCGTacgt

my $myStatement = "COG0004 is Metabolism";
print index($myStatement, "is");           # 8
print index($myStatement, "Signaling");    # -1
```

lower & upper case nucleotides can be weighed differently in BLAST e-value calculation

Exercise 1

- Open the program substrings.pl in your editor
- Complete the program by writing the code that
  - Gets the sequence name
  - Gets the sequence length
  - Chops off the polyA tail (last 6 bases)

Exercise 2

- The program "Blast" has an option to filter any region that is in lower case.
  - Mask Lower Case –U
- You want to take advantage of that option to filter the adapter in your sequences.
- Your adapter has the sequence: "**ATTGATTG**".
- You need to find it in the sequence and lower case it as well as anything before it.
- For example:
  - ATTGATTGTTATTAAAT becomes attgattgTTATTAAAT
  - TATGATTGATTGAATGTA becomes tatgattgattgAATGTA
  - Etc.
- A program structure is provided in maskAdaptor.pl

Conditional Blocks and Loops

Making Decisions - 1

- There are a number of useful ways to make decisions in Perl
- `if(CONDITION)` do something
  - Is a statement that can chose (or not) to do something.
- If `CONDITION` evaluates to `TRUE`, then the block is executed.
- You can group a set of statements between { }
  - Similar to a paragraph
- An `else{}` statement can be added to execute an alternative block if the condition is `FALSE`.

To test spacing, place -B-$seq-E-
want print

Example

```
if ($sequence eq "ATG") {
        print "This is a DNA codon";
        print "This is a methionine";
} else {
        print "This is not a methioine";
}
```

- The `if(CONDITION)` evaluates to TRUE whenever $sequence is equal to "**ATG**"
- When the CONDITION is FALSE, the `else` block is executed.
- If the outcome of `if` is one line, you can use the shortened construct (Perlish construct)

```
print "Codon is a methionine" if ($sequence  eq "ATG");
```

Booleans

- Conditional expressions can be any expressions that produce a **boolean** value.
  - They evaluate to True or False (1 or 0).
  - Ex. 'eq' and '==' are boolean expressions

all ← numbers only

- These expression can be combined using boolean operators
  - The "&&" operator refers to AND
  - The "||" operator refers to OR
  - The "!" operator refers to NOT

==

eq <=

ne >=

!=

Making Decisions - 2

- `unless(CONDITION)` statement is used to negate a condition
- `unless(CONDITION)` is equivalent to `if(NOT CONDITION)`

```
print "Codon is a methionine" if ($sequence  eq "ATG");
print "Codon is not methionine" unless ($sequence  eq "ATG");
```

- Ex.

```
if ( ($codon eq "ATG") ||
     ($codon eq "TAG") ||
     ($codon eq "TAA") ||
     ($codon eq "TGA"))
{
   print "This is a stop codon";
}
if ( (len($dnaSeq) > 8) && (index($sequence, "ATTGATTG") != -1 ) )
{
   print "This sequence does not contain the adapter and";
   print "the sequence is at least 10 bases long \n";
}
```

← return value if sequence found

if/not can be used instead of unless

Truth in Perl

- What makes a CONDITION TRUE or FALSE in Perl
  - undef, "" (empty string), 0, "0" are all False
  - All else is True

```
my $seq = "";
if($seq){
   print "sequence contains something";
}
if(len($seq)){
   print "sequence has length > 0";
}
# To avoid confusion, explicitly state the test your are trying to accomplish
if("" != $seq)
# and
if(len($seq) > 0  )
```

While Loops

- `while(CONDITION){}` allows you to keep on executing a block of code, for as long as a CONDITION is TRUE
- `while(CONDITION)` is a "sort of" iterated `if`

```
while( my $input = <STDIN>){
   print "Enter your text: ";
   print "You have entered: $input";
}
```

- The previous program creates an infinite loop

While Loop

- The following program creates an infinite loop
  - An **infinite loop** runs indefinitely (Unless some condition terminates it)

```
while(1){
   print "I am stuck here\n";
}
```

- You can break out of any loop using `last`
- You can also skip an iteration using `next`

```
my $i = 0;
while(1){
   $nbIterations = $nbIterations + 1;
   next if (3 == $nbIterations);
   print " I am in iteration $nbIterations\n"
   last if (5 == $nbIterations);
}
```

For Loop

- A Perlish way of implementing loops.
- Uses a **range**
  - (1..5) generates a list of all the integers between 1 and 5

```
For my $i (1..5){
   print "I've counted to: $i\n";
}
```

```
> perl for_loopv2.pl
I've counted to: 1
I've counted to: 2
I've counted to: 3
I've counted to: 4
I've counted to: 5
```

- `last` and `next` can be used with the for loop as well

Nested blocks

- **Loops, if statements** and **blocks** can be nested
- You can put a loop within a loop
  - A loop is not repeated until all the code within it has completed running

my $i

```
for (1..3){
   for my $j (4..6){
        print "i is: $i \t j is: $j \n";
   }
}
```

- For every value of $i, the internal loop is executed 3 times (for the value 4, 5 and 6)
- In total, the print statement will be executed 9 times.

Interactive Example

- We will write a program that reverse complements a sequence
- The complete program is in reverseComplement.pl
- We read a line from the user (standard input)
- We reverse the sequence
- We loop along the sequence
- For every base, we print its reverse complement.
- Using if, **elsif**, else (elsif used to groups if statement having the same outcome)
- Using die()

example of removing 3 different primers from tons of sequences

Using random()

- `rand(MAX)` returns a random fractional number greater than or equal to 0 and less than the value of MAX
- Perl has a function called `int()`
  - This function takes a fractional number and returns an integer
  - The function `int` is used for converting between fractional (`floats`) and integers (`ints`).
- We can use `int` and `rand` conjointly to generate random integers

```
my $fractionalRand = rand(10);
my $integerRandom = int($fractionalRand);
# or just
my $integerRandom = int(rand(10));
```

Exercise

- Print a list of random DNA fasta sequences
- Each DNA sequence is of size 10.
- Ex.
  >1
  GCTGATCGTA
  >2
  GCTGATCGTA
  >3
  GCTGATCGTA
  ...
- A program structure is given in randomFasta.pl

Lists and Arrays

Lists

- Remember that a list is a group of ordered elements
- Lists are elements enclosed in parenthesis
  - Ex.: (A, B, C), (1, 2, 3), (John, 23, Honolulu), etc.
- A list is a fixed collection literal
  - They are not tied to a variable
  - We can access each element separately, but we cannot modify it.
- Lists can be generated by other functions. Ex.
  - `qw(ATT ATG) → ("ATT", "ATG")`
  - `split(//, "ACGT") → ('A', 'T', 'G', 'T')`
  - `range (1..5) → (1, 2, 3, 4, 5)`
- We use an index to access a specific position **[index]**
  - (1,2,3,4)**[0]** → returns 1
  - (1,2,3,4)**[1]** → returns 2
  - etc.

Arrays

- Arrays are similar to a lists
- The array is a variable collection of ordered elements. The variable name starts with the "@" funny character
- Practically, you can think of an array as a list but with a name that can be used to access it.
- An array is initialized with a list

```
my @cellularProcesses = ("Translation", "Transcription", "Replication");
```

- Or using any function that returns a list

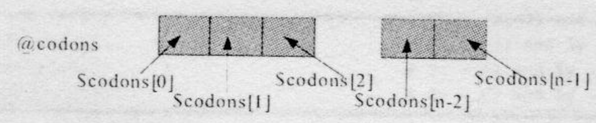
```
my @alphabet = (a..z);
my @tokens = split(//, "fructokinase    128 a.a.    Metabolism")
```

← Single or double-quoted (error)

↑ splitting on lack of space

Array Indexing

- Every element in the array is indexed by a position
- First element of the array is located at position 0
- If the array contains n elements, the last element is then located at position n-1
  - Or position -1 if using the negative index
- **Important**: Every element of the array is a scalar.



Accessing Array Elements

- Array elements can be accessed for display or modification, using their index

```
my @digits = (A..E);

print $digits[1];              # prints B
print $digits[4];              # prints E
print $digits[15];             # generates an error

$digits[20] = 'T';             # does not generate an error
print $digits[20];             # does not generate an error
print $digits[19];             # does generate an error
```

- The example `$digits[20]` does not generate an error.
  - Arrays grow and shrink dynamically
  - Arrays automatically add undef to compensate for gaps
  - Trying to access and undef element will generate an error
- Always, use the scalar funny character to access a single element of the array.

Array Operations - 1

- The `for()` allows us to iterate through the elements of an array, using user defined steps
- The perlish way to loop through an array is through a `foreach`.

```
my @digits = (1..10);
for my $i (0..9){
   print $digits[$i] . "\n";
}
```

```
my @digits = (1..10);
foreach my $element (@digits) {
   print $element . "\n";
}
```

- The first approach generates indexes
  - Requires to specify the size of the array
- The perlish approach iterates through the elements

Index of Last Element

- To find the position of the last element, prefix the array name with `@#`.
- We can use the index of the last element to generate a range and use it to loop through the array.

```
my @digits = (1..10);

# @#digits returns the index of the last element of the array
foreach my $i (0 .. @#digits){
   print $digits[$i];
}
```

Array Operations - 2

- We can use the size of the array to loop through all the elements

```
my @digits = (1..10);
foreach my $i (0 .. @#digits){
   print $digits[$i];
}
```

```
1
2
3
...
```

Adding to an Array

- push() is the most used way to add to an array
  - It can add a single element, a list or even another array

```
my @x = ();
push(@x, 1);              # @x now contains 1
push(@x, 2);              # @x now contains 1, 2
push(@x, 3, 4, 5);        # @x now contains 1, 2, 3, 4, 5
push(@x, qq(6 7) );       # @x now contains 1, 2, 3, 4, 5, 6, 7
my @y = (8, 9, 10);
push(@x, @y);             # @x now contains 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Deleting From an Array

- To delete the last element, use pop()
- To delete the first element, use shift()
  - Both pop() and shift() automatically resize the array and return the element deleted

```
my @x = (1, 2, 3);
my $y = pop(@x);                    # @x = (1,2) and $y = 3
$y = pop(@x);                       # @x = (1) and $y = 2
$y = pop(@x);                       # @x = () and $y = 1
$y = pop(@x);                       # @x = () and $y = undef

@x = (1, 2, 3);
$y = shift(@x);                     # @x = (2, 3) and $y = 1
$y = shift(@x);                     # @x = (3) and $y = 2
$y = shift(@x);                     # @x = () and $y = 3
$y = shift(@x);                     # @x = () and $y = undef
```

Arrays and Context

- Array operation have different meaning in different contexts

```
my @x = ("A", "B", "C", "D");
print "@x\n";                    # prints A B C D
print @x;                         # prints ABCD
print @x . "\n";                  # prints 4
```

```
my @y = @x;
print "@y\n";                     # prints 1 2 3 4
my $value = @x;
print "$value \n";                # prints 4
($value) = @x;
print "$value \n";                # prints 1
print @x - 1;                     # prints 3
```

- Assignments can also be non-intuitive.
- In case 1, an array is being assigned to another array
  - @y will be an exact and independent copy of @x
- In case 2, an array is being assigned to a scalar
  - `$myVar` contains the number of elements in the array

Know arrays well, easy to introduce bugs here

Array Length

- If we put an array in a scalar context, we get the length of the array

```
@x - 1;                           #returns length - 1
```

- Array length can be used as a for loop argument
  - It is not equal to `$#ARRAY`
  - In a `for` loop, `@x-1` is easier to read that `$#x`

```
my @x = (1..5);

foreach my $i (0 .. (@x - 1) ){
   print "element $i is $x[$i] \n";
}
```

Reversing an Array

- The `reverse(LIST)` function returns the reversed list of the passed argument.
  - The returned list is reversed but the argument is **NOT**!
  - If you need to reverse the argument, assign it back to itself

```
my @x = (1, 2, 3);
my @y = reverse(@x);              # @y = (3, 2, 1) and @x = (1, 2, 3)
@x = reverse(@x);                 # @x = (3, 2, 1)
```

Sorting an Array

- The `sort()` function returns a list sorted in ASCII ascending order
  - By default, even digits are sorted using the ASCII order
  - Use `reverse()` if you need descending order
- To sort numerically redefine `sort()`
  - `sort ({$a <=>$b}, LIST)`
  - Compares the predefined variables $b and $a numerically
  - If you change the order of $a and $b, it changes the sorting order.

```
my @x = sort("small", "medium", "large");
print "@x\n";                          # prints large medium small
my @y = (1, 2, 4, 8, 16, 32, 64);
@y = sort(@y);
print "@y\n";                          # prints 1 16 2 32 4 64 8
@y = sort({$a <=> $b} , @y);
print "@y\n";                          # prints 1 2 4 8 16 32 64
```

tells perl to sort numerically

Exercises

- We will write a program that shuffles sequences for a boot strap
- Shuffled sequence contains the same letter composition, but a different order
- Ex:

```
my $dnaSeq = "ACGACGT";
my $shuffledSeq = "TACGCTAG";
```

- The algorithm will be explained first
- A program structure is given in shuffle.pl

Input Output

Input From a File

- Reading from a file is done in 3 steps
  1. Open the file for reading
  2. Grabbing input, we use the <FILEHANDLE> operator to get a line of input from the file
     - Similar to how we were reading from <STDIN>
  3. Closing the file

```
# open a file and create a file handle
open(INPUT, "/tmp/randSeqs.fasta") || die ("could not open the input file");
# fetch the next lie in the file, using the <> operator.
while (my $line = <INPUT>) {
   # do some work …
}
close(INPUT);
```

- File handles are capitalized by convention
- The <> operator returns TRUE while there are more lines to reads
  - Returns FALSE if we reach the end of a file!

Interactive Example

- Open the file or throw an exception if you cannot find it
  - If you are unsure what the relative path is, use the file info and the full file path to avoid any confusion
- Always the same structure
- Grab one line of input at a time
  - Remember that the line contains "\n" at the end
  - Do something useful with it
- Do not forget to close the file
  - Especially if you are opening the file for writing
- readFromFile.pl

Output to a File

- Writing to a file is similar to reading
  - Also done in 3 steps
- Open the file for writing.
  - Overwrite or create: `open(FILEHANDLE,">FILEPATH");`
  - Append or create: `open(FILEHANDLE,">>FILEPATH");`

```
# open a file and create a file handle
open(OUTPUT, "/tmp/someFile.txt") || die ("could not open the file $!");

# print to output file
print OUTPUT "Print this line to the file";

# close the file handle
close(OUTPUT);
```

- Not properly closing the files can cause the buffers not to be flushed
  - The text is not output to the file!

Interactive Example

- We modify the random fasta program to write the fasta sequences to a file
- We get the file name to write to from the user
- And add three modifications
  1. Open the file
  2. Write to the file, instead of the screen
  3. Close the file
- writeRandomFasta.pl

Hashes

What are Hashes?

- A hash is a dictionary-like structure
  - It holds a set of key/value elements
- Every element is indexed by a unique key
  - Position no longer relevant since we index by key
- A Hash variable is identified by the "%" *funny character*
- Every element of the hash is like a list of two elements
  - (key, value)

%codons	ATT	CTT	GTT	GGT	...	CGT	TAA
	ILE	LEU	VAL	GLY		ARG	STOP

There is no predetermined order for the elements of a hash structure

Store amino acid elements in a hash

($somval @anearray) = (1, 2, 3, 4, 5)

Initializing Hashes

- A hash is initialized using a list of elements
- Every element contains a key and a value, separated by the "=>" symbol
  - The key is not quoted but the value is

```
my %codons = (ATT => "ILE", CTT => "LEU", GTT => "VAL");
($codons{CGT}, $codons{TAA}) = ("ARG", "STOP");
```

- To access a hash value, we use $HASH{KEY}
- To get a list of the hash keys, we use keys()

```
my %codons = (ATT => "ILE", CTT => "LEU", GTT => "VAL", => "STOP");
foreach my $k ( keys(%codons) ){
   print "$k codes for $codons{$k} \n";
}
```

```
ATT codes for ILE
CTT codes for LEU
GTT codes for VAL
TAA codes for STOP
```

copy this one

FASTA comparison, any thing that would have a unique identifier (ie. taxonomy of a sequence)

Initializing Hashes

- Every element of the hash is a scalar
  - We use the scalar prefix $ to access an element

```
my %codons = (ATT => "ILE", CTT => "LEU", GTT => "VAL");
print $codons{GTT};                    # prints VAL
print $codons{ATT};                    # prints ILE
```

  - We can add a new value by specifying a key and a value

```
$codons{CGT} = "ARG";
$codons{TAA} = "STOP";
```

- We can use the **keys()** in the same way we have used lists
  - Print, loop, size, get the size, etc...

```
my %codons = (ATT => "ILE", CTT => "LEU", GTT => "VAL", TAA => "STOP");
# we return all the keys in codons
my @keys = keys(%codons);
# the keys are not necessarily in the order they were added
print "@keys \n";
```

Truth and Deleting From a Hash

- To check if a key exists in your hash, use exists(KEY)
- Same notion of "truth" as for scalars
- if(exists(HASH{KEY}))
  - TRUE if KEY exists (value may be undefined)
- The delete(KEY) function is used to remove entries from a hash
  - Deletes the KEY and VALUE from the hash.
  - Remember to use the scalar prefix ($)

```
my %codons = (ATT => "ILE", CTT => "LEU", GTT => "VAL", TAA => "STOP");
delete ($codons{TAA});                 # removes the key/value pair
```

COG functions table

Interactive Example

- Lets test the randomness of `rand()`
  - We generate a 1,000, 3bp sequences

```
int(rand(@array));
```

  - For every 3bp sequence created, we keep track, using a hash, of the number of times this triplet was obtained
  - Using a horizontal count plot, we illustrate the obtained statistics.

```
AAA  ***************
AAC  **************
AAG  ****************
AAT  ********************
ACA  *************
...
...
TTT  *******************
```

- See randomKmers.pl

Exercise

- Putting together annotations using the three following annotation files

```
1 - GeneCOG.txt association file
GeneID     COG
637671327 COG0001
637671102 COG0001
637671101 COG0002
...

2 - COGFunctions.txt association file
[J] COG0001
[A] COG0002
[K] COG0003
[K] COG0004
[L] COG0005
...

3 - COGcodes.txt file
J Translation, ribosomal structure and biogenesis
A RNA processing and modification
```

Exercise (cont'd)

- We need to assign the annotation (cogFunctions.txt) to each geneID.
  - Each geneID is assigned to a COG number, and each COG # has an assigned function
  - For the first example, we would print the following line:

```
637671327 COG0001  J        Translation, ribosomal structure and biogenesis
```

- Program structure is given in buildAnnotations.pl

```
1 - GeneCOG.txt association file
GeneID     COG
637671327 COG0001
...

2 - COGFunctions.txt association file
[J] COG0001
...

3 - COGcodes.txt file
J Translation, ribosomal structure and biogenesis
```

Subroutines in Perl

What Are Subroutines?

- A **subroutine** is a reusable block of code that performs a specific task
  - Similar to the functions we have used so far
  - `length()`, `keys()`, `rand()`
- Subroutines are a good programming practice
  - Add modularity to a script
  - Easier to read, debug and maintain
- A subroutine is defined by its name
  - Should be informative about the functionality

Using Subroutines

- A subroutine is defined with **sub** keyword, followed by the name
- A block of code, delimited by "{" and "}" defines the execution logic of the subroutine
- A return statement is used when the subroutine returns a result.

```
# definition of the subroutine
sub getGcRate{
    my $seq = $_[0];
    my $gcRate = 0;
    # do the necessary work to calculate the GC rate
    return $gcRate;
}

#call the subroutine
my $result = getGcRate("ACTGTAGCTGATCGTGATCGTGTAGCTGTAGCT");
```

recalls this, position zero

saved as @_

my $num = @_;
assigns length of array

Subroutine Arguments

- A subroutine accepts one or more arguments
  - ex. `length(STR)` takes one argument, the string to compute the length of, and returns the length.
- Inside the sub., an automatic array, "`@_`", is created
  - `@_` is an array where each element is a passed argument
  - Remember to use the scalar character prefix (`$`) when accessing one element of the array

```
# subroutine name: getBaseCount
# arguments:      $seq: a sequence
#                 $bt: the base type to count (ex. G)
# return value: the base count in the sequence
sub getBaseCount {
   my $ct = 0;
   ($seq, $bt) = @_;
   #do work
}
my $countType = getBaseCount($sequence, $baseType);
```

Example

- What does the following code print?
  - Hint: it contains a possible bug

```
# some subroutine … that does something!
sub my_square {
   my $num = @_;
   return $num**2;
}

print my_square(5);
```

Argument Validation

- It is always a good practice to verify the validity of the passed arguments
  - Depends on the function logic
- As a good practice, your function should assign default values or raise errors when necessary

```
sub my_square {
   my $num = @_;
   return $num**2;
}

print my_square();          # oops, use of uninitialized value!!!
```

```
# BETTER! … but who wants 0**2 ?
sub my_square {
   die("No arguments passed") if(!defined($_[0]))
   my $num = $_[0];
   return $num**2;
}
```

Return Values

- Your function can return any three of the variable types we have seen so far

```
($scalar, @array, %hash)
```

- Be careful when returning more than one array!
- Be careful when passing non scalars, as arguments are **flattened**
- Create a copy whenever it is possible and not too costly

Exercise – 1 of 2

- Write a subroutine that returns a k-mer
- The size is passed as an argument to the subroutine
- A program structure is provided in returnKmer.pl

Exercise – 2 of 2

- Write a function that parses a fasta file and returns an array of all the ids in the file
- The sequence filename is passed as an argument.
- A program structure is given in parseFastaSub.pl

Regular Expressions

A Brief Definition

- "A regular expression, abbreviated as **regex** or **regexp**, is a string that describes or matches a set of strings, according to certain syntax rules." *Wikipedia*
- A regex is a pattern used to give a concise description of a set, without listing all the elements of that set.
  - A regular expression is case sensitive
- It is used for "*matching*" (specifying and recognizing) strings of text, such as particular characters, words, etc.
- The simplest regular expression would contain the exact searched string
  - A binding site TRYA can be expressed as; TACA or TATA or TGCA or TGTA.

Matching Character

- When "matching", the regular expression is used as a parameter of a match expression `m/REGEX/`
  - The **`m`** is often dropped and only the **`/REGEX/`** `part` is kept
- To request a match, the "=~" operator is used to bind the regular expression to the target sequence
- If the absence of pattern is tested, the "!~" is used

```
$sentence = "A sequence of amino acids";
$sentence =~ m/sequence/;          # returns TRUE
$sentence =~ m/the/;               # returns FALSE
$sentence !~ m/the/;               # returns TRUE
```

Regular Expressions

- We use a character class to represent a set of possible characters
  - Character classes are denoted by "[]" with the possible matches inside
  - It means that at least one of the characters inside "[]" should be matched
- For example, TRYA, can be expressed as T[AG][CT]A

```
# ( ($sequence =~ /TACA/) || ($sequence =~ /TACA/ || … )
$sequence =~ /T[AG][CT]A/;                # matches 'TACA', 'TATA', …
```

Character Classes Abbreviations

- The special character "-" acts as a range indicator, within character classes.
  - When in the beginning or the end, it is considered a normal character

```
$sequence =~ /item[0-9]/;          # matches item0, item1, …, item[9]
```

- Several abbreviations for char classes are used. The most common are:

\d	matches a digit ([0-9])
\s	matches a whitespace
\w	matches a word character (alphanumeric or _)
\D	matches any non digit character. It is a negation of \d;
\S	matches any non-whitespace character. It is a negation of \s
\W	matches any non-word character (such as $, %, &, etc...). It is a negation of \w.

Some Useful Meta-characters

- The "^" and "$" meta-characters are used to specify the beginning and the end of a string, respectively
- To specify alternate choices, the special char "|" can be used.
- Ex.: To match AAA or TTT use the AAA|TTT regular expression
  - At every position in the sequence, AAA is tested first and if it doesn't match, the second alternate, TTT, is tested
  - The first alternative allowing a hit is chosen
  - By default, the match stops right after we get a hit

. means ~~any~~ wildcard

Matching Repetition

a?	match 'a' 0 or 1 times
a*	match 'a' 0 or more times
a+	match 'a' at least once
a{n,m}	match 'a' at least n times, but not more than m times.
a{n,}	match 'a' at least n or more times
a{n}	match 'a' exactly n times

```
/[a-z]+\s+\d*/;          # match a lower-case word of minimum size 1,
                         # at least on 1 or more spaces, and any
                         # number of digits

$year =~ /\d{2,4}/;      # make sure year is at least 2 but not more
                         # than 4 digits
```

- What is the problem with the year validation in the example above?
  - What would be a better regular expression?

Examples

```
$sequence =~ /A[ACGTN]A/;      # matches A followed by any other base
                               # Followed by A such as ATA, AGA, etc.
$sequence =~ /^gi\|\d*$/;      # matches the refseq id format
                               # >gi|46110415

"AAACCCTTT" =~ /AAA|TTT/;      # matches AAA
"AAACCCTTT" =~ /TTT|AAA/;      # matches TTT
"ACGT" =~ /A|AC|ACG|ACGT/;     # matches A
"ACGT" =~ /ACGT|ACG|AC|A/;     # matches ACGT

$sequence =! /CDS\s*\d*\.\.\d*/  # matches the coding sequence format of
                                 # the NCBI CDS 150..980
```

escaped dots

Grouping and Extracting Matches

- Grouping allows the creation of a sub-regex
- Implemented by enclosing the desired "unit" between parenthesis. "()"

```
#matches VERSION refseq of VERSION uniport
$domain =~ /VERSION (refseq|uniprot)/;
```

- The parenthesis are also used to capture the "hit"
  - The variables $1, $2, $3, etc. contain the hits of a regular expression, enclosed by parenthesis

```
my $version = "VERSION: refseq";
#matches VERSION refseq of VERSION uniport
$version =~ /VERSION: (refseq|uniprot)/;
print "The version is: $1";
```

```
The version is: refseq
```

master this for pulling out specific fields of a fasta file

Global Matching - 1

- We have seen that, by default, a regular expression finds and reports the first hit it finds
- Global matching is the option to find and report all matches obtained by a regular expression

```
my $sequence =~ m/REGEX/g;
```

  – We can assign the hits directly to an array

```
my $sequence = "A AAC AAG AAT ACA ACC ACG ACT AGA AGC AGG AGT";
my @hits = ($sequence =~ /A.A/g);
for my $hit (@hits){
   print "-$hit \n";
}
```

```
-A A
-ACA
-AGA
```

Global Matching - 2

- In a scalar context, the global search would return TRUE or FALSE, based on the fact that a hit was found or not
  – We can use this fact to loop through the found hits

```
my $sequence = "AAA AAC AAG AAT ACA ACC ACG ACT AGA AGC AGG AGT";
while($sequence =~ /(A.A)/g) {
   print "$1 \n";
}
```

- What do you think is going to happen if we remove the global (g) modifier

/g need to make a regular expression greedy to find all motifs within a longer sequence

Hit Position

- To determine the position of a hit, use the "pos()" function on the searched sequence.
  – pos() returns the position of the first mismatch occurring after the last match
    - The result is 0 indexed

```
my $sequence = "AAA AAC AAG AAT ACA ACC ACG ACT AGA AGC AGG AGT";
while($sequence =~ /(A.A)/g) {
   print "hit found $1 at position " . (pos($sequence) -3) . "\n";
}
```

```
hit found AAA at position 0
hit found ACA at position 16
hit found AGA at position 32
```

$1 $2 $3

(COG0001 T Transcription

/(^COG\d+)\s+(.)\s+(.+)$/

$1 $2 $3

String Substitution

- When the position of the desired text is known, we can use `substr()` to substitute a substring.
- An easier way to do that is through the substitution expression **s/REGEX/REPLACEMENT/**
  - Bound to a string using the =~ operator
  - **REPLACEMENT**: Exact text used to replace a substring
  - Substitution can be **global** or **minimal**
  - The operation returns the number of substitutions made

Example

```
my $sequence = "AAA AAC AAG AAT ACA ACC ACG ACT AGA AGC AGG AGT";

$sequence =~ s/(A\SA)/NNN/;
print "The sequence after using the substitution " .
      "in a minimal context is: \n$sequence\n\n";

$sequence =~ s/(A\SA)/NNN/g;
print "The sequence after using the substitution " .
      "in a global context is: \n$sequence\n";
```

```
The sequence after using the substitution in a minimal context is:
NNN AAC AAG AAT ACA ACC ACG ACT AGA AGC AGG AGT

The sequence after using the substitution in a minimal context is:
NNN AAC AAG AAT NNN ACC ACG ACT NNN AGC AGG AGT
```

quick search and replace

String Translation

- To substitute a string of characters for another, the "`tr///`" operator is used.
- The expression allows character for character translation "`tr/TN/UX/`"
  - Replaces T by U and N for X
  - Returns the number of substitutions made
- Can sometimes be used to count the number of occurrences in a string.

```
my $count = ($sequence =~ tr/A/A/);
```

  - Assigns to `$count` the number of occurrences of A in `$sequence`.
- If used with the **c** modifier, reverses the search list. (Anything but the search list)

Exercise

- Rewrite the script reverseComplement.pl
- You script should replace all the non-standard DNA letter entered by the character **N**
- A program structure is given in betterReverseComplement.pl

compliment /c

eg. $seq=~ tr/ACGT/N/c;

Running a Shell Script From Perl

- Perl offers different ways to run a program from within the code.
  - Programmatically running programs is ideal for automation and building pipelines
- The easiest way is to simply enclose the command to run by back-ticks
  - Variables are interpolated within back-ticks (similar to a double quote context)

```
my $inputFile = "/tmp/input.fasta";
my $outputFile = "/tmp/output.fasta";

my $database = "/tmp/database.fasta";

#run blast
`blastall -p blastn -i $inputFile -o $outputFile -d $database -F F -U -e 1e-05`
```

Returning Output From a Shell Command

- If the command returns an output, you can assign the output to a variable.

```
my $directory = "/tmp/";

# get the files in a directory
my @filesInDir = `ls $directory`;

chomp($filesInDir[0]);
print $filesInDir[0] . "\n";
chomp($filesInDir[1]);
print $filesInDir[1] . "\n"
```

```
myRead.fasta
myRead.fastq
```

- runCommand.pl

Very Brief Intro to Objects

- BioPerl use the concept of Objects, another paradigm for code design
  - Procedural programming: Code arranged in subroutines and programs flows like a lab protocol (or a recipe)
  - Object oriented programming: Encapsulates the logic within entities called objects
- Each class represents an entity and all the code within one class has a common purpose (self contained)
- Choosing one approach or another depends on many factors.
  - The size of the project
  - The efficiency
  - The desired level of code reuse.
  - Etc.

Object Attributes and Methods

- Object oriented programming models the program using objects, classes.
- A class is a set of attributes and actions
  - Attributes are the fields that describe the object
  - Actions (Methods) are the tasks that can be applied to the object
    Methods are the way the user manipulate the objects
- For example, we can think of a Fasta Sequence as an Object, instead of a string
  - Attributes can be: Type(DNA|RNA|Protein), Length, Name,
  - Methods can be:
    - `complement()`
    - `rev_complement()`
    - `substrting()`,
    - etc.

A Brief Introduction to Bioperl

What is Bioperl?

- Bioperl is a collection of modules (libraries of code) that facilitates the development of bioinformatic applications
- Maintained by a dedicated group of volunteer developers.
- Latest developments can be found on www.bioperl.org website
- It gives access to a variety of functionality, such as plotting phylogenic trees, modeling sequences, alignments, communicating with databases, parsing popular sequence files, doing alignments,....
- The purpose is to interactively see some of the available functionality by looking at some simple examples

Objects in BioPerl

- Most concepts in BioPerl are objects
  - Sequence Objects(Seq, RichSeq, LargeSeq, SeqWithQuality, etc...),
  - Alignment objects (SimpleAlign, Stockholm)
  - Trees(Tree)
  - Sequence variations (SNP, DNAMutation)
  - Etc.
- Offers ways to create those objects and methods to operate on them

Parsing Bioinformatic Files

- Most objects in BioPerl have their parsers
  - SeqIO(Fasta, Fastq, genbank, Interpro, ace, etc...),
  - TreeIO(Newick, Nexus, Phyloxml, ...),
  - SearchIO(Blast, hmmer, fasta, etc..),
  - FeatureIO(Bed, GFF, GTF)
- Parsers open the file and build iterators
  - Iterators iterate through the objects in the file
  - You can iterate through a Fasta file by getting each sequence independently
- Most of the time, they offer ways to convert between formats
  - Convert fasta to fastq, newick to Nexus, etc.

can change trees formats instantly

Parsing a Fasta File

```
use Bio::SeqIO;

my $io = Bio::SeqIO->new( -file => "/tmp/myReads.fasta", -format => "fasta");
my $firstSeq = $io->next_seq();

print "The sequence id is: " . $firstSeq->display_id . "\n";
print "The sequence is: " . $firstSeq->seq . "\n";
print "The sequence length is: " . $firstSeq->length . "\n";
print "The first 10 bases of the sequence is: " . $firstSeq->subseq(1,10) .
    "\n";
my $revComplement = $firstSeq->revcom();
print "The reverse complement of the sequence is: ".$revComplement->seq .
    "\n";
print "The translated sequence is: ".$firstSeq->translate()->seq."\t";
```

```
The sequence id is: 1
The sequence is: AGATGATGGATAGAGTTACGGCATGATC
The sequence length is: 28
The first 10 bases of the sequence is: AGATGATGGA
The reverse complement of the sequence: GATCATGCCGTAACTCTATCCATCATCT
The translated sequence is: R*WIELRHD
```

firstBioPerl.pl

Parsing a Fasta File

```
my $io = Bio::SeqIO->new( -file => "/tmp/myReads.fasta",
                          -format =>"fasta");
```

- The `$io` object is the iterator
  - When asked for the next_seq, the iterator returns the next seq in line.
  - The sequence returned is a BioPerl Object
  - Returns False, when there no more sequences to return
- The id, seq and length are all attributes of the $firstSeq class.
- `subseq()` and `revcom()` are methods of the object
- Both methods and attributes are called by post fixing the objects name with the "->" characters
- Sequences have many other attributes and methods

Interactive Example

- Modifying the previous script to only print those sequences that have lengths above a user given threshold.
- We can use the same construct to write output files

```
my $seqout = Bio::SeqIO->new( -file => ">output.fa", -format =>"fasta");
```

- we can directly write BioPerl Seq objects to output files using the method write_seq()
- We use these methods to convert fasta-sanger to fasta-illumina
  - convertSangerToIllumina.pl
- We can read convert between many other formats
  - http://search.cpan.org/~cjfields/BioPerl-1.6.901/

Parsing Blast Results

- BioPerl offers the same parsing functionality as that used for parsing sequences
  - http://search.cpan.org/~cjfields/BioPerl-1.6.901/Bio/SearchIO/blast.pm
  - You can generate a parser using
    - `Bio::SearchIO->new()`
  - For get every result using
    - `$searchio->next_result()`
  - Then the hits and the hsps.
- All of these are objects: They have attributes and methods

Example: Parsing Blast Results

parseBlast.pl

```
use Bio::SearchIO

my $fileParser = new Bio::SearchIO( -file => '/tmp/metagenome.blast.out'
                                    -format => 'blast');

while(my $result = $fileParser->next_result) {
  print "----------------START----------------";
  while(my $hit = $result->next_hit){
    while(my $hsp = $hit->next_hsp){
      if($hit->significance < 1e-02){
        print $result->query_name . "\tvs\t" . $hit->name . " evalue: " .
              $hit->significance() . "\n";
        print "\tLength\t" . $hsp->length() .
              " Percent ID\t" .$hsp->percent_identity . "\n";
      }
    }
  }
  print "--------------- END ----------------";
}
```

query, hit, HSB

nested loops need to cover all elements within all elements within all elements

Reading And writing Trees

TreeOperations.pl

```
if ($tree->get_lca(\@nodes) == $tree->get_root_node()) {
    print "The lowest common ancestor is the root\n";
} else {
    print "The lowest common ancestor is NOT the root\n";
}

@nodes = $tree->find_node(-id => 'A');
push(@nodes, $tree->find_node(-id => 'E'));

if ($tree->get_lca(\@nodes) == $tree->get_root_node()) {
    print "The lowest common ancestor is the root\n";
} else {
    print "The lowest common ancestor is NOT the root\n";
}
```

Questions

perldoc