Always use tab-completion to prevent spelling mistakes: you never know when you have made mistakes

emboss

```
seqret -help
```
http://emboss.sourceforge.net/apps/release/6.1/emboss/apps/seqret.html

```
seqret -sequence test-query.fa.cowrument.out.m9.head10.fa.l -outseq test-query.fa.cowrument.out.m9.head10.fa.l.aln -sformat fasta -osformat aln
```

```
infoseq -help
```
http://emboss.sourceforge.net/apps/release/6.2/emboss/apps/infoseq.html

```
infoseq -sequence test-query.fa.cowrument.out.m9.head10.fa -name -only -length
```

More command examples:

```
needle -help
water -help
fuzznuc -help
pepstats -help
pepinfo -help
plotorf  -help
transeq  -help
garnier -help
prettyseq -help
est2genome -help
```

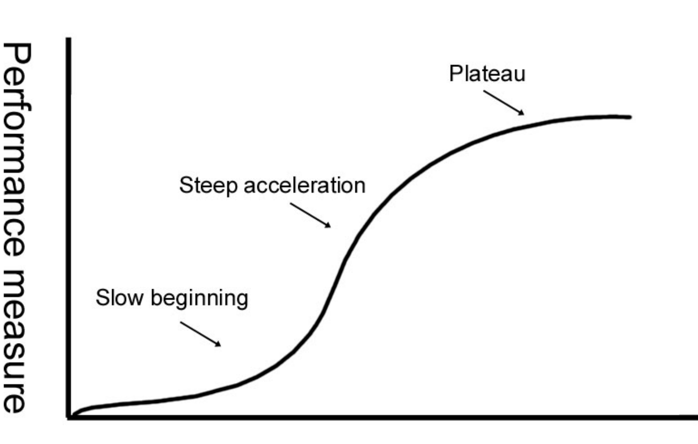
In the remaining classes

Do expect you:

- Get Familiarized with Linux commands
- Be able to read example Perl scripts
- Know how to run given perl scripts
- Practice examples on projects
- Be able to finish the two course projects

Do NOT expect you (or not all of you):

- Be able to write complex Perl scripts
- Become a professional programmer
- Become a professional bioinformatian



Things you should know about programming

Learning programming has to go through the hands-on practice, a lot of practice

Hearing what I describe about a command or a program helps, but you will not be able to do it unless you type in the codes and run it to see what happens

Although painful and frustrating, trouble-shooting is normal and part of the learning experience (ask experienced people or google)

To avoid errors, you have to follow rules; most errors occurred in programming are because of not knowing rules or forgetting rules

Reading others' codes helps but often is harder than writing it by yourself from scratch

Use comments in case you forget what you've written means

Edit -> run -> errors -> revise -> errors -> ………………………………….. -> run -> success

Good news: finished scripts could be reused or edited for later use

What we will cover in the remaining classes:

Perl basic concepts
Example Perl scripts
Bioperl concepts
Example scripts using bioperl modules

Perl Basics

- Variable, scalar, array, hash, hash of arrays
- Control flow: if/elsif/else, for loop, foreach loop, while loop
- Input and output
- Numeric operators and logic operators
- Subroutines and modules
- Regular expression

What is Perl?

- Scripting language by Larry Wall, 1985
- Born of AWK
- Practical Extraction and Reporting Language

http://www.bsi.umn.edu/resources/perl1.pdf

Why Perl?!

- An easy language to use, though sometimes hard to learn. Some choices were made to make things easier for the programmer at the expense of the student.
- Fast cross platform text processing.
- Good pattern matching. (regex)
- Many extensions for Life Sciences data types. (BioPerl)
- Many biologists already know Perl.
- Powerful

http://www.bsi.umn.edu/resources/perl1.pdf

Edit -- Run -- Revise (and Save)

- As a programmer, most of your time will be spent planning, testing, and revising your program.
- Running is often incidental on today's hardware.
- Carefully written programs can be productive tools for years.
- **Programming is a method of communication**: your code must be readable by both the computer and your users.

http://www.bsi.umn.edu/resources/perl1.pdf

Pseudocode

- An informal program in which there are no details and formal syntax is not followed.
- A quick and informal way to collect your ideas about solving the problem at hand.

Programming Strategies

Break down into two major approaches:

1. Find a program written by someone else.
2. Write one yourself.

The reality is usually somewhere in between.

The Process

1. Identify the inputs, data, and specifications from the user.
2. Design the solution as a series of steps toward the desired result.
3. Decide on the output(s). Does the result print to the screen or to a file? How will this output be used? Does format matter?
4. Refine the design with increasing detail. (pseudocode)
5. Do appropriate code modules exist? (CPAN)
6. Write the program.

http://www.bsi.umn.edu/resources/perl1.pdf

Now create a file called hello.pl

```
vi hello.pl
```

Or

```
nano hello.pl
```

Type in the following :

```
#!/usr/bin/perl -w
#
# a program to do the obvious
#
print "Hello, world!\n";
```

After exit do:

```
perl hello.pl
```

If use vi

Press i to the edit mode

Esc then :x to save and exit

Esc then :q! to exit without save

A simple program

```
#!/usr/bin/perl -w
#
# a program to do the obvious
#
print "Hello, world!\n";
```

This is a special line called command interpretation that tells the computer that this is a Perl program

You can have empty lines

Every line of codes must have ; in the end

Special formatting characters:

\n	new line
\t	Tab
\s	Space

Notice that the first line of code uses a flag -w. The "w" stands for warnings, and it causes Perl to print messages in case of an error. Very often the error message suggests the line number where it thinks the error began.

How does it work?

```
#!/usr/bin/perl -w
#
# a program to do the ob[illegible]us
#
print “Hello, world!\n”;
```

Every Perl program begins with this line.

Comments

The ‘print’ function sends the quoted text to the default output device, the screen.

Working with strings: double quotes

The type of quotation mark around the string makes a difference as to how Perl treats it. A string enclosed in double quotes undergoes a process called **interpolation**, and anything that Perl recognizes as a variable gets replaced by the value of that variable

If you want to print the following:

Today's "Blue-Plate Special" costs $5.99.

```
print "Today's "Blue-Plate Special" costs $5.99."
```

```
print "Today\'s \"Blue-Plate Special\" costs \$5.99."
```

Backslash (\) is used to escape interpolation of special characters.

Create hello2.pl to define a variable to hold the word string

```
#!/usr/bin/perl -w
#
# assign a value to $message
my $message = "Hello, world!\n";

# print the $message
print $message;
```

Store the value "Hello, world!" in a container called a **variable**.

A variable is a named reference to a memory location. Variables provide an easy handle for programmers to keep track of data stored in memory

Perl has three basic types of variables.
- Scalar variables hold the basic building blocks of data: numbers and characters.
- Array variables and hash variables hold lists.
- The three types are differentiated by the first character in the variable name: '$ ', '@ ', and '%', respectively. For example, $a, @a, %a

Make a few deleterious mutations to your program hello2.pl

What happens when?:
1. You remove a semicolon?
2. You remove a dollar sign?
3. You change the shebang (#!)?
4. Can you change the shebang to something else that works?

Observe the error messages. One of the most important aspects of programming is debugging. Probably more time is spent debugging than programming, so it's a good idea to start recognizing errors now.

- Scalar: a variable quantity that cannot be resolved into components, e.g. $a

- List or array: a collection of items, often stored in an array, indexing item with a number, e.g. @a, when referring to a particular item: $a[1]

- Hash: like an array, but instead of indexing values by number, values are accessed by name. Think of them as name-value pairs, e.g. %a, when referring to a particular item: $a{“name”}

Scalar

```
$y = 1;           # integer
$x = 3.14;        # floating point
$w = 2.75E-6;     # scientific/engineering notation
$t = 0377;        # octal
$u = 0xffff;      # hexadecimal
```

```
$dna = "ATGCAGTGA";         # one space either side of the '=' sign

$dna="ATGCAGTGA";           # no spaces either side of the '=' sign

$dna     =     "ATGCAGTGA"; # lots of spaces!
```

Variable Name	Comment
$a	valid
$apple_g4_computer_counter	valid: names can be any length with most alpha numeric characters
$my invalid variable name	X invalid: spaces are one type of characters which aren't allowed (use underscores)
$my(invalid[variable{name}])	X invalid: parens, brackets, and braces are allowed, but do something different that you might be intending (see Chapter 3)
$1 through $9	X valid: "special" reserved variables
$_	X valid: "special" reserved variable

Mathematical operations

```
$x = 3
$y = 2
```

Space or not and how many, does not matter

```
print "$x plus $y is ", $x + $y, "\n";
print "$x minus $y is ", $x - $y, "\n";
print "$x times $y is ", $x * $y, "\n";
print "$x divided by $y is ", $x / $y, "\n";
print "$x modulo $y is ", $x % $y, "\n";
print "$x to the power of $y is ", $x ** $y, "\n";
```

```
print "the absolute value of -$x is ", abs(-$x), "\n";
print "the natural log of $x is ", log($x), "\n";
print "the square root of $x is ", sqrt($x), "\n";
print "the sin of $x is ", sin($x), "\n";
print "a random number up to $y is ", rand($y), "\n";
print "a random integer up to $x x $y is ", int(rand($x * $y)), "\n";
```

http://korflab.ucdavis.edu/Unix_and_Perl/unix_and_perl_v3.1.1.html

Table 2.2 Perl operators

Operator	Description
++	Autoincrement
--	Autodecrement
**	Exponentiation
*	Multiply
/	Divide
%	Modulus
+	Add
-	Subtract
cos()	Cosine
sin()	Sine
sqrt()	square root
=	Assign
+=	assign add
-=	assign subtract

$x++
$x=$x+1

$x+=1

Array and hash are VERY useful to hold text data in the memory and are <u>often created using loops</u>

- Array:

```
@fruit_list = ('apple', 'orange', 'banana');
```

=

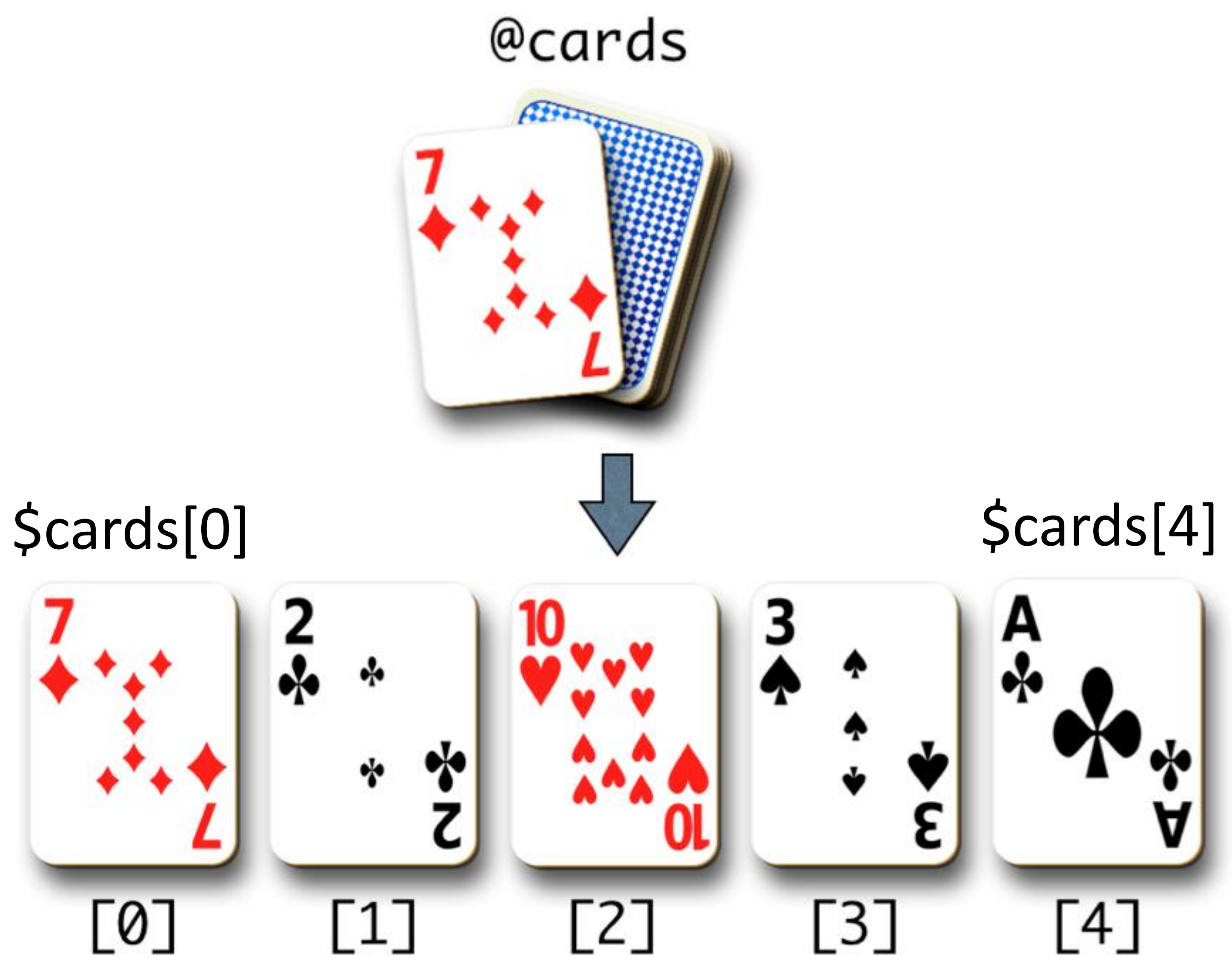
```
$fruit_list[0]='apple';
$fruit_list[1]='orange';
$fruit_list[2]='banana';
```

- Hash:

```
%ip2hostname = (
"glu" => "131.156.41.196",
"gly" => "131.156.41.193",
"cys" => "131.156.41.195"
);
```

=

```
$ip2hostname{"glu"}='131.156.41.196';
$ip2hostname{"gly"}='131.156.41.193';
$ip2hostname{"cys"}='131.156.41.195';
```



http://korflab.ucdavis.edu/Unix_and_Perl/unix_and_perl_v3.1.1.html

Create an array from a tabular format file using perl

```
vi array-from-file.pl
```

```
#!/usr/bin/perl

open (IN,$ARGV[0]);

@a=<IN>;   # assign to an array

foreach $line (@a){
   print $line;
}
```

File handle: IN

@ARGV: special variable to capture command line arguments

$ARGV[0]: the first element

Save and exit vi

```
perl  array-from-file.pl test-query.fa.cowrumen.out.m9 | less
```

Perl has many predefined special variables that contain default values designed to make life easier for programmers. Most special variables are a combination of punctuation marks and obscure characters, and a programmer following the good coding practice of creating meaning variable names will never accidentally run into them.

Table 2.4 Special variables

Variable	Function
$_	default input and regexp search space
$/ and $\	input and output record separator
$,	output field separator
@ARGV	array with the command line arguments for the current script

```
#!/usr/bin/perl

open (IN,$ARGV[0]);

@a=<IN>;  # assign to an array

foreach $line (@a){
   print $line;
}
```

||

```
#!/usr/bin/perl

while($line=<>){
  print $line;
}
```

=

```
#!/usr/bin/perl

while(<>){
  print $_;
}
```

The "diamond operator", <> is used when a program is expecting input; <> means standard input <STDIN>

```
perl  array-from-file.pl test-query.fa.cowrumen.out.m9 | less
```

```
#!/usr/bin/perl

open (IN,$ARGV[0]);

@a=<IN>;

foreach $line (@a){
   @col=split(/\t/,$line);
   print $col[0];
}
```

```
perl  array-from-file.pl test-query.fa.cowrumen.out.m9 | less
```