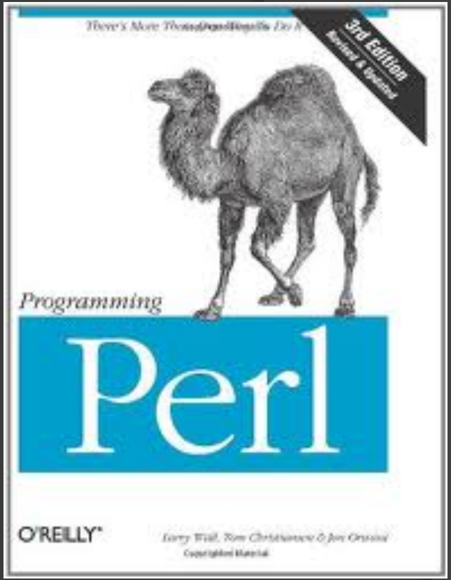
PERL Bioinformatics I

Nicholas E. Navin, Ph.D.
Department of Genetics
Department of Bioinformatics

A Brief History of PERL

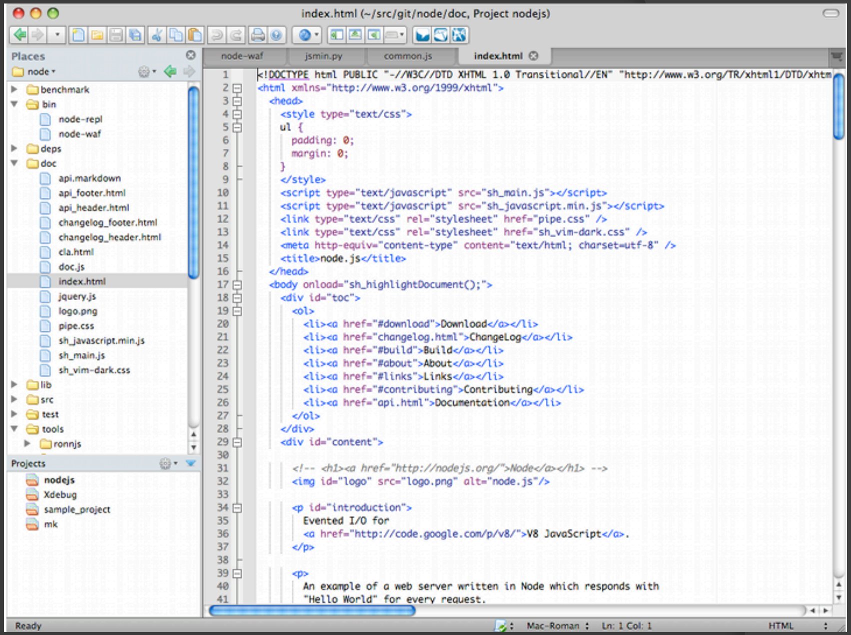
- PERL – ‘Practical Extraction and Reporting Language’
- Developed by Larry Wall in 1987 for the UNIX operating system to overcome the limitations of standard UNIX tools
- Perl is a programming language that was designed for quickly manipulating text files
- Perl was the first major scripting language for the world wide web in the late 1980 – 1990’s and was used for most forms online
- Unlike other programming languages (C, C++) perl is an <u>interpreted</u> language, which means that the code does not need to be *compiled* before execution
- Perl can run on any computer system and is pre-installed in Apple OSX computers
- Perl scripts can be uploaded to servers or HPCCs to run programs on large-scale datasets



Larry Wall

Graphical Interface for Perl Programming

- KOMODO EDIT is a graphical user interface program for PERL with code autocorrection features
- Programs can be edited in KOMODO EDIT and then executed from the UNIX command shell
- Komodo EDIT is an open source editor that is developed by Activestate and can be downloaded for free for APPLE or PC computers: http://www.activestate.com/komodo-edit
- Check your computer to see if KOMODO EDIT is already installed, you should see this icon:



```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtm
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <style type="text/css">
    ul {
      padding: 0;
      margin: 0;
    }
    </style>
    <script type="text/javascript" src="sh_main.js"></script>
    <script type="text/javascript" src="sh_javascript.min.js"></script>
    <link type="text/css" rel="stylesheet" href="pipe.css" />
    <link type="text/css" rel="stylesheet" href="sh_vim-dark.css" />
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>node.js</title>
  </head>
  <body onload="sh_highlightDocument();">
    <div id="toc">
      <ol>
        <li><a href="#download">Download</a></li>
        <li><a href="changelog.html">ChangeLog</a></li>
        <li><a href="#build">Build</a></li>
        <li><a href="#about">About</a></li>
        <li><a href="#links">Links</a></li>
        <li><a href="#contributing">Contributing</a></li>
        <li><a href="api.html">Documentation</a></li>
      </ol>
    </div>
    <div id="content">

      <!-- <h1><a href="http://nodejs.org/">Node</a></h1> -->
      <img id="logo" src="logo.png" alt="node.js"/>

      <p id="introduction">
        Evented I/O for
        <a href="http://code.google.com/p/v8/">V8 JavaScript</a>.
      </p>

      <p>
        An example of a web server written in Node which responds with
        "Hello World" for every request.
```

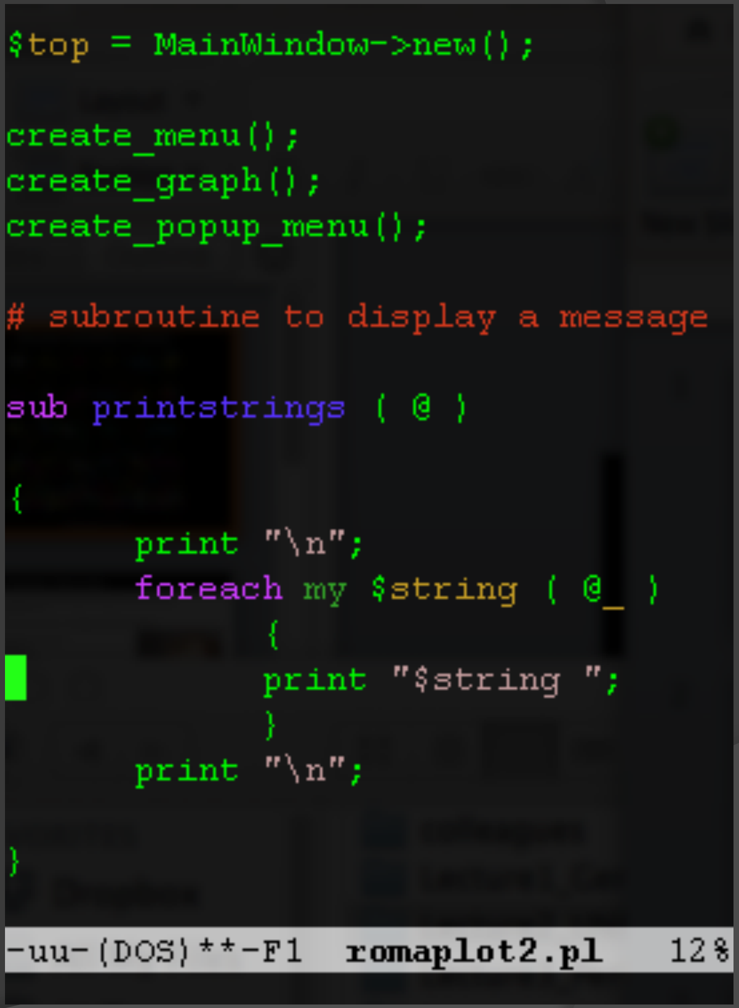
Komodo Edit, a graphical interface for editing perl code

ActiveState
Komodo® Edit 7
The open source editor for Python, PHP, Ruby, Javascript, Perl and Web development.

Emacs: Command Line Editor

- EMACS is a UNIX text editing program that can be used to edit PERL programs
- EMACS has code highlighting features for PERL programs, making it easier to read the code in the UNIX shell
- However, EMACS does not have code autocorrection, so errors cannot be detected until the program is executed from the command line
- Users need to switching back on forth between the EMACS editor and the UNIX shell to run the programs

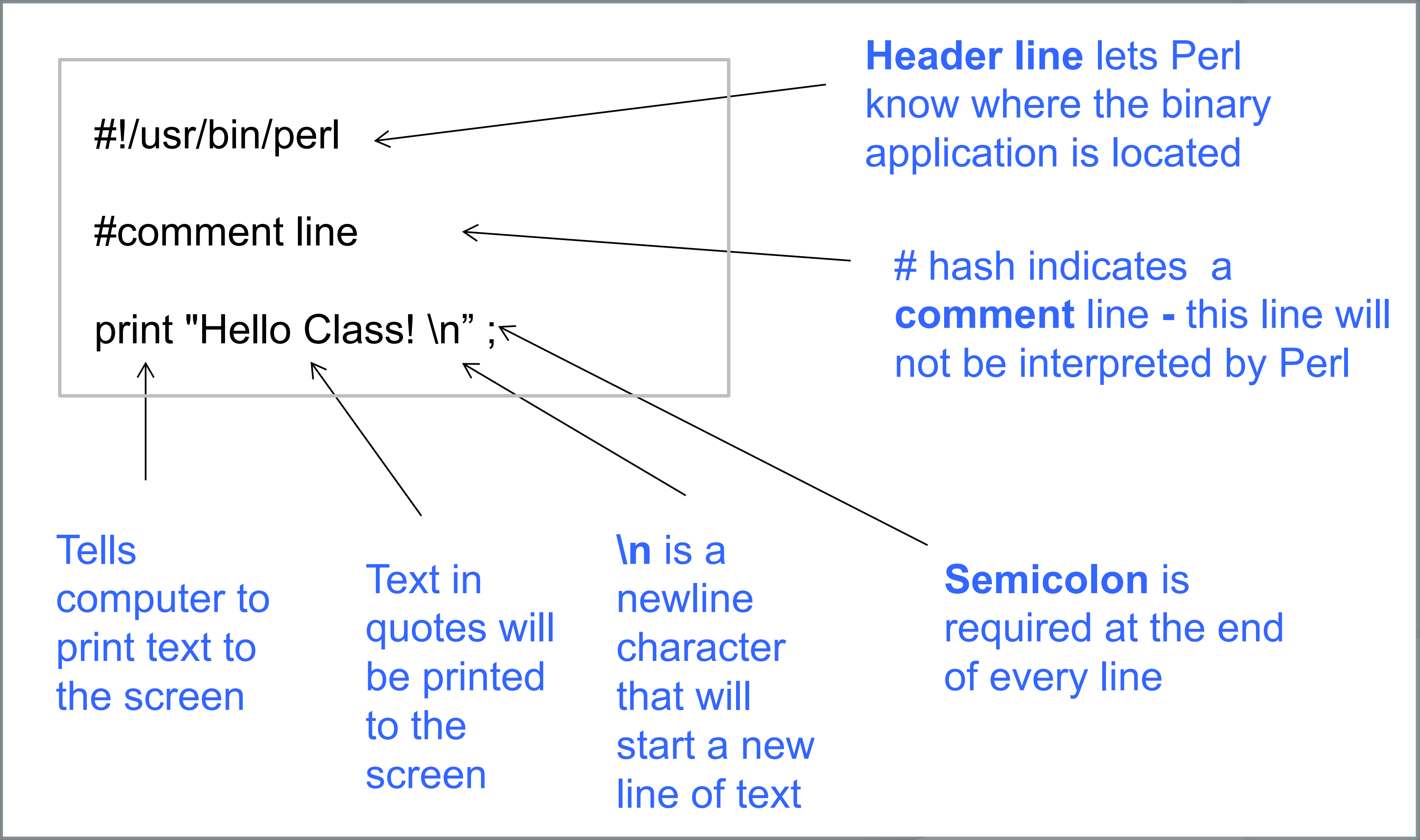
```
>emacs filename
```



```
$top = MainWindow->new();

create_menu();
create_graph();
create_popup_menu();

# subroutine to display a message

sub printstrings ( @ )

{
      print "\n";
      foreach my $string ( @_ )
            {
            print "$string ";
            }
      print "\n";

}
-uu-(DOS)**-F1  romaplot2.pl    12%
```

Emacs

Anatomy of a Simple Perl Program



```
#!/usr/bin/perl

#comment line

print "Hello Class! \n” ;
```

Header line lets Perl know where the binary application is located

# hash indicates a **comment** line **-** this line will not be interpreted by Perl

Tells computer to print text to the screen

Text in quotes will be printed to the screen

\n is a newline character that will start a new line of text

Semicolon is required at the end of every line

Data Types in Perl

Perl has 3 data variable types:

- Scalars $
- Arrays @
- Hashes %

Scalars store integer numbers, floats (decimals numbers) and strings (characters or words)

Arrays are vectors that store a linear series of scalar data. They can be indexed using numbers in brackets

Hashes store collections of data that can be indexed using words or characters

Scalar Variables

- Scalar variables are used to store data
- Scalars are declared using the **$** sign followed by a name and an equal sign to assign the value
- In most computer languages variables must be declared as either an *integer*, *float, character* or *string*, however Perl automatically determines the data type for you

```
$float = 2.7;
```

This is a float variable that can contain decimals

```
$integer = 200;
```

This is an integer value

```
$string = “hello everybody 123”;
```

This is a string which can contain characters and words

```
print $float;
print $integer;
print $string;
```

Print the values out to the screen

Arrays

8	9	3	4	1.99	chr	cat	dog	
0	1	2	3	4	5	6	7	Index

- Arrays are vectors that store a series of data
- Any scalar variable can be stored in an array (integers, floats or strings)

```
@array = (8 , 9 , 3 , 4, 1.99, “chr”, “cat, “dog”  );
```

Initialize an array with some data

```
$length = @array;
```

Determine the length of the array

```
print @array;
```

Print all contents of the array

```
print $array[0];
print $array[5];
```

Print the array index values out to the screen

```
print $length;
```

Print the array length out to the screen

Hash Variables

7.22	cat	9	red	dog	ant	18	ATG	
dec	animal	num	gem	animal2	insect	num2	dna	Hash Keys

- Hashes are data collections that can be indexed with keys (instead of numbers)
- Hashes are initialized with the % character, followed by a key and value

```
%hash= (“dec”, 7.22, “animal”, “cat”, “num”, 9, “gem”, “red”,
“animal”, “dog”, “insect”, “ant”, “num2”, 18, “dna”, “ATG”);
```

Initialize a hash with some data

```
Print $hash{“gem”};

Print $hash{“insect”};

Print $hash{“DNA”};
```

Print the key values of several hash variables

Mathematical Operators

- Mathematical operators can be performed on any integer or float scalar variables

Operator	Operation
+	addition
-	subtraction
*	multiplication
**	exponential
/	division
%	modulus
++	increment
--	decrement

```
$add = 15 + 22;

$sub = 87.43 – 7.43;

$mul = 87 * 54;

$pow = 2 ** 10;

$mod = 10 % 7;

$inc = 5;
$inc++;
$dec = 5;
$dec--;
```

Logical Operators

- Logical operators are used to evaluate expressions
- In computer science 0 = false, and 1 = true
- However in perl any value that is not 0 is considered to be true

Operator	Operation
&&	AND
||	OR
!	NOT

```
$false = 0;
$true = 1;

$ans = !$true;                    FALSE

$ans = $true && $true;            TRUE
$ans = $true && $false;           FALSE
$ans = $false && $false;          FALSE

$ans = $true || $true;            TRUE
$ans = $true || $false;           TRUE
$ans = $false || $false;          FALSE
```

Numerical Comparators

- Comparator operators allow numerical values to be compared
- They return a true (1) or false (0) value when the test is performed

Operator	Test
==	equality
!=	inequality
>	Greater than
<	Less than
>=	Greater than Equal to
<=	Less than Equal to

```
$five = 5;
$ten = 10;

$ans = ($five == $five);    TRUE
$ans = ($five == $ten);     FALSE
$ans = ($five != $five);    FALSE
$ans = ($five != $ten);     TRUE

$ans = ($five < $ten);      TRUE
$ans = ($five >= $ten);     FALSE
```

Strings

- Strings are characters or words that are declared as scalars $

String Operators	Function
$string1.$string2	concatenate
length($string)	Find length of a string
lc($string)	Convert string to lowercase
uc($string)	Convert string to uppercase
index($string1, $string2)	Find location of string2 in string1
substr($string, offset, length)	Find a substring in a string

```
$dna1 = “ATATAATTT”;
$dna2 = “CCCCGCGC”;
$combine = $dna1.$dna2;          ATATAATTTCCCCGCGC
$len_dna1 = length($dna1);       9
$low_dna1 = lc($dna1);           atataattt
$index = index($dna1, “TTT”);    6
$subdna = substr($dna1, 5, 4);   ATTT
```

String Matching with Regular Expression

- Regular Expression is a powerful tool in Perl for matching strings

command	function
$string =~ m/pattern/g	Match pattern in string, return true or false
$string =~s/pattern/replace/g	Replace pattern in string

```
$dna1 = “GAAATTTTAA”;

$dna1 =~ m/TTTT/g          TRUE
$dna1 =~ m/AT+TA/g         TRUE
$dna1 =~ m/AT?TA/g         FALSE
$dna1 =~ m/^G/g            TRUE

$dna1 =~ s/TTTT/CCCC/g     GAAACCCCAA
$dna1 =~ s/T//g            GAAAAA
$dna1 =~ s/[G|T]A/CC/g     CCAATTTCCA
```

RegEx	Test
+	any number of characters
?	Single character
^	Start of line
$	End of line
[A|B|c]	Subset of characters
[A-E]	Series of Letters
/g	global matches
/i	case insensitive

Conditional IF/ELSE Statements

- The IF operator in Perl will evaluate a logical statement and only execute a command if the statement is TRUE

  If (test-expression) { command to execute if true }

- You can also add an ELSE statement **else** { execute if expression is false }

```
$mendel = “monk”;

if ($mendel eq “monk”)                          TRUE
   { print “mendel is a monk”;}                 ‘mendel is a monk’
if ($mendel eq “acrobat”)                       FALSE
   { print “mendel is an acrobat”;}             no output


if ($mendel eq “acrobat”)                       FALSE
   { print “mendel is an acrobat”;}             no output
else
   {print “mendel is a monk”;}                  ‘mendel is a monk’
```

FOR Loops

- The **FOR** loop will execute a command for a specified number of times
- The format is: *for (initializer, condition, increment)*
  { command statement }

```
$up = 0;

for( $i =0; $i<10; $i++)
{

   $up++;

   print "$up \n";

}
```

```
1
2
3
4
5
6
7
8
9
10
```

WHILE / UNTIL Loops

- **While** and **Until** loops will continue to loop indefinitely until a condition is met

  while/until (condition) { command statement }
- While loops assume a condition is TRUE and will exit only when the logical statement becomes FALSE
- Unless loops assume a condition is FALSE and exits only when the logical statement becomes TRUE

```
                                        OUTPUT
$num = 0;                               1
while($num < 10)    TRUE                2
{  $num++;                              3
   print $num;                          4
}                                       5
                                        6
$num = 0;           FALSE               7
until($num == 10)                       8
{  $num++;                              9
   print $num;                          10
}
```

Foreach element in an Array

- **Foreach** is a command in Perl that allows you to traverse all elements within an array, similar to the **FOR** loop, but using less code

```
@arr = (5, 10, 15, 20, 25, 30, 35, 40, 45);

foreach $val (@arr)
{
   $val = $val +1;
   print “$val \n”;
}
```

OUTPUT

```
6
11
16
21
26
31
36
41
46
```

File Input

- Perl has commands for *reading in* and *writing out* text files
- @ARGV is an array that takes input when the program is run
- To input the filename into the program, you run the program followed by the name of the input filename
- Ex.

```
perl program.pl einstein.txt
```

```
$infile= $ARGV[0];

open(TXT, “<$infile”);
@text = <TXT>;
print “$text[2]”;

close(TXT)
```

Output:
“violent opposition from”

einstein.txt

```
Great spirits have
always encountered
violent opposition from
mediocre minds

-albert einstein
```

File Output

- @ARGV can be used for both input and output file names
- Ex.

```
perl program.pl einstein.txt output.txt
```

```
$infile= $ARGV[0];
$outfile = $ARGV[1];

open(TXT, “<$infile”);
open(OUT, “>$outfile”);

@text = <TXT>;

print OUT “$text[3]”;

close(TXT)
```

einstein.txt

```
Great spirits have
always encountered
violent opposition from
mediocre minds

-albert einstein
```

output.txt

```
mediocre minds
```

Subfunctions

- When program code becomes long, sections can be broken down into subfunctions (sub) which are executed using the & character
- Variables can be passed to the subfunction, and returned with the **return** command
- Within the subfunction the **$_[0]** syntax is used to access a passed variable

```
#!/usr/bin/perl

$var = 10;
$result = &calculation($var);
print $result;

sub calculation
{  $num = $_[0] * 66;
    return($num);
}
```

OUTPUT

```
660
```

PERL Bioinformatics Workshop

The workshop for today can be found here:

http://www.navinlab.com/bioperl

Follow the instructions on the website to complete the workshops and ask the TA or instructor for help

There are two options for code editing:

#1
Work locally on your own computer with KOMODO EDIT and PERL (recommended for beginners, since KOMODO edit has autocorrect highlighting)

#2
Work locally or on the server using EMACS and PERL