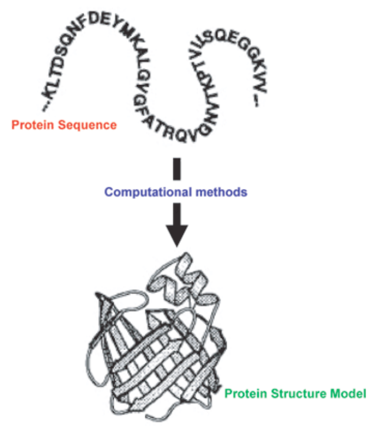
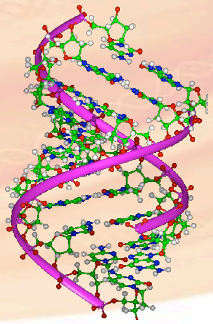


Bioinformatics Computational Methods 1 - BIOL 6308



Million years
3.4 5.1
Million years
7.0 2.2
H
C
G
O
aactggtacg tggtcg tacgaactgg
aaccggttcg cggtcg ttcgaaccgg
aacctgttcg ctgtcg ttcgaacctg
atctggtacg tggtcg tacgatctgg

September 17th 2013

http://155.33.203.128/cleslin/home/teaching6308F2013.php

Last Time

- Intro shell scritps
- Simple shell script syntax
  - Variables
  - Exit status
  - Passing arguments
  - Logic (test)
  - shell special variables
  - Loops, counters, conditional execution, and if-else
- Make sure you know the commands:
  - http://155.33.203.128/teaching/BIOL6308-Fall2013/local/unixCheatSheet.html
- Questions?

Introduction to Perl

So, Now Lets Begin Perl

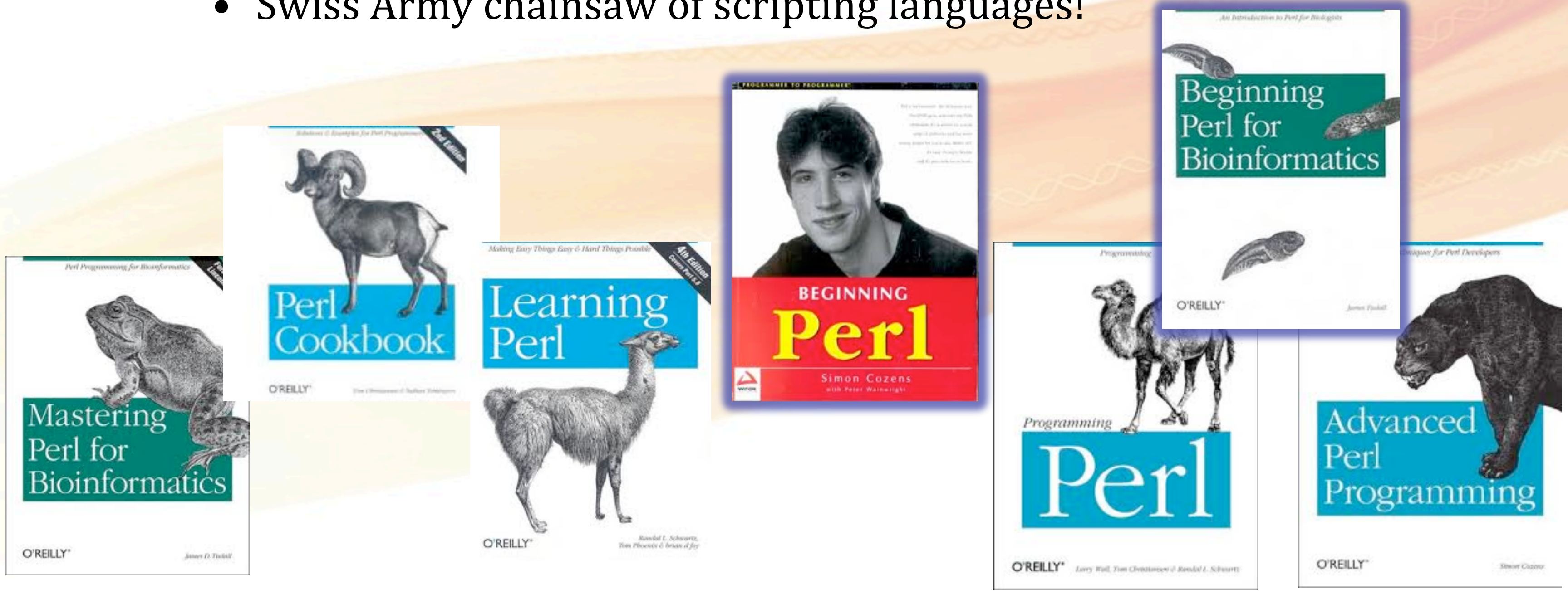
- shell – **clunky** - that should be obvious
  - Simple shell scripting is enough
  - Feel free to learn more, it will only help
- Perl is a **complete programming language**
  - Usage somewhat different from that of shell scripting
  - Shell scripting - more-or-less glue nature, automating, hacking, cracking, etc
- Anything more structured:
  - Must use a "program"
  - Programming languages will fulfill the requirements
  - We will use Perl
    - For a more in-depth Perl class look to BIOL6200 next year!

shell Vs Perl

- Shell
  - limited to basic filesystem functions, no real math of any sort, and very simple parsing
  - Supported in even the tiniest Linux distros – even in embedded systems
  - Quickly automate things
  - Good for small little projects and hacks
  - So its important to know how to write little scripts
  - Great for distributed compute environments
- Perl
  - Better coding syntax and ways to document (perldoc)
  - Faster
  - Much faster development
  - Libraries provide a great deal functionality
  - Easier to port to different platforms
  - No barrier to the language used, like in a shell: bash, tcsh, korn….

What is Perl?

- **P**ractical **E**xtraction and **R**eporting **L**anguage
  - High-level, general-purpose, interpreted, dynamic programming language
    - Developed by Larry Wall in 1987 as tool to parse large text files
      - General-purpose Unix scripting language to make report processing easier
    - Primarily used as a scripting language for batch processing of file
    - Swiss Army chainsaw of scripting languages!


Mastering Perl for Bioinformatics
Perl Cookbook
Learning Perl
BEGINNING Perl
Simon Cozens
Programming Perl
Beginning Perl for Bioinformatics
Advanced Perl Programming
O'REILLY

The Swiss Army Chainsaw of Programming Languages

- Provides powerful text processing facilities
- No arbitrary data length limits of many contemporary Unix tools
- Facilitates easy manipulation of text files
- It is also used for:
  - Graphics programming
  - System administration
  - Network programming
  - Applications that require database access
  - CGI programming on the Web
  - Report generation and analyitcs
  - **Bioinformatics**

How to Get Perl

- Perl is available for most operating systems
  - Usually installed most Unix/Linux boxes by default
  - Mac by default
  - Windows - must be installed
- Recommend using ActivePerl
  - http://www.activestate.com/activeperl

How to Get Help on Perl

- http://www.perl.org/
- Programming Perl" published by O'Reilly (The Camel Book)
- Comprehensive Perl Archive Network: http://www.cpan.org
- Google! Search for "Perl **topic**"
- Look over online tutorials
- Use my Examples
- Code, code, code.....
- Practice, practice, practice

Perl Basics - The program

- Perl programs are **text files**
  - Just like a shell script
  - Run through an interpreter
- If run on UNIX/Linux/OSX, all Perl files **must** contain location of the Perl interpreter on the first line
  - (Well, its not a must, but I don't like to have to call **perl test.pl**)
  - This location is specified in the shebang line
  - **`#!/usr/bin/perl`**
  - **Perl file must be executable**

Perl Basics - A Sample Program

```
#! /usr/bin/perl
use warnings;
use strict;
#setup variables
my $varNum = 1;
my $varString  = "How you doing? ";
#print the variables
print $varString , $varNum , "\n";
```

Always, Always, Always

You can use "." instead of "," – but in certain situations they have a different context. I suggest using the ","

Programming Perl

- Variables
- Datatypes
- Control Structures
- Statements
- Subroutines (user-defined)

Learn these topics and you can program almost any type of project

Datatypes

- Some programming languages have a few main datatypes
  - `character 'c'`
  - `integer 1234`
  - `float 3.214`
  - `string "Bioinformatics is cool"`
- Not in Perl
  - All the above are known as a ***scalar***
  - Perl takes care of the rest for you

Variables

- Why is it called variable? - Because it can change values
- Three kinds of variables Perl
- Each has its own `sigil`
  - Scalar (`$`)
    - Numbers, Strings, and References
    - Contain only one element
  - Arrays (`@`)
    - Lists of **scalars**
    - Indexed by a number, 0,1,2,3…
  - Hashes(`%`)
    - Arrays that use names (`key`) instead of numbers to index element (`value`)
    - Called the `key-value` pair

Example Scripts For This Lecture

- Can be found in **`/data/METHODS/Fall/LECT4`**

- **`Create a directory structure on local machines or fisher`**

```
$ mkdir /home/userName/METHODS/
```

- **`Copy the data above into a here`**

```
$ cp -r /data/METHODS/Fall/LECT4 /home/userName/METHODS/
```

Printing Datatypes

- Interpolation
  - Means “introducing or inserting something”
  - Replacing a variable with value of the variable
  - **`See testScript.pl`****
- Printing caveats
  - If you are:
    - Constructing or printing a string - no variables – can use the single quotes
      - ***simple print statements***
    - Need print a variable in text - use the double quotes
      - ***complex print statements***
  - Sometimes need to use either **backslashes** or **concatenation** operators

****`For all Examples see /data/METHODS/Fall/LECT4  on fisher`**

Formatting Strings

- **See testScript0.5.pl** **

`\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`	Places a tab in the string

```
# STRINGS TO BE FORMATTED
my $mystring = "Welcome to bioinformatics class!";
my $newline = "Welcome to \nbioinformatics class!";
my $capitalLetter = "\uwelcome to bioinformatics class!";
my $allCaps = "\Uwelcome to bioinformatics class!";
my $allLower = "\L$allCaps";
```

Formatting Strings

- **See testScript0.5.pl** **

`\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`	Places a tab in the string

```
# PRINT THE NEWLY FORMATTED STRINGS
print $mystring, "\n";
print $newline ,  "\n";
print $capitalLetter, "\n";
print $allCaps, "\n";
print $allLower , "\n";
```

Formatting Strings

- **See testScript0.5.pl** **

```
– \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      Places a tab in the string
```

```
#say is like print, but no need for \n
say $mystring;
say $newline;
say $capitalLetter;
say $allCaps;
say $allLower;
```

Note to use `say` you must `use feature qw(say);`

Simple Perl - Statements

- Statements - Used in order to process or evaluate expressions
- Perl uses values returned by statements to evaluate or process other statements, and so on
- Generally **assignments** and **operations**

  my `$var = "word";`
- Statements always end with a semicolon
  - Tell interpreter - statement is complete
  - You can have multiple statements per line
    - **Best to only have ONE per line**
      - Makes code more **readable**
      - Helps in code **maintenance**
- Statements use variables for the most part

Remember: Variables

- Why is it called variable?
  - Because it can change values
  - **Three** kinds of variables Perl
- Each has its own `sigil`
  - Scalar (`$`)
    - Numbers, Strings, and References
    - Contain only one element
  - Arrays (`@`)
    - Lists of scalars
    - Indexed by a number, 0,1,2,3…
  - Hashes(`%`)
    - Arrays that use names (`key`) instead of numbers to index element (`value`)
    - Called the `key-value` pair

Scalar Variables

- Can contain any single item
- Integers, real numbers, strings, characters, or *references* to objects
- When accessing contents of any variable - prepended with a `$`
- Imagine you need a variable with a value of Pi
  - What can you do?

```
my $pi = 3.14159265;
```

We will come back to why to use the **my**

Assigning Scalar Variables

- We know Pi ~ 3.14….
- So…

```
my $pi1 = 3.14;
my $pi2 = 3.14;
my $pi3 = 3.14;
```

`or`

```
my $pi3 = my $pi2 = my $pi1;
```

- You can make `3.14` equal a string, and Perl knows to treat it like a #
  - `$pi2 = "3.14";`
  - `$pi1` is a number, but `$pi2` is now a string
    - You can add numbers, but you cannot add strings…. Or can you?

Lets Take a look @ testScript1.pl

```perl
#!/usr/bin/perl
use warnings;
use strict;
use feature qw(say);
#declare variables
my $pi1 = 3.14;
my $pi2;
my $pi3;

#copy values
$pi3 = $pi2 = $pi1;

say "pi3 " , $pi3 , " pi2 " , $pi2;

##pi2 is a string, but perl will think of it as a number
$pi2 = "3.14";
##can we then add them, right?
$pi3 = $pi1 + $pi2;
##Lets see
say "pi3 " , $pi3;


##Now if it's, a number, or is it a string?
if ($pi2 eq "1" ){
        say "1 yes it is, pi2 is equal to 3.14";
}
else{}
```

Initialized $pi1, but not $pi2 or $pi2, they have the value **undef**

Initialized $pi3 and $pi2 = $pi1, no longer undef

what's going to happen here?

```
pi3 3.14 pi2 3.14
pi3 6.28
2 yes it is, pi2 is equal to 3.14
3 yes it is, pi2 is equal to 3.14
4 yes it is, pi2 is equal to pi1
```

OUTSIDE - Assigning Scalar Variables

- If it looks like a number, then Perl can convert it to a number
  - That's why `$pi1 + $pi2 = 6.28`
- Can you add strings?
  - Take a look at **testScript2.pl**
    - Run it and see what happens
    - What was the problem?
    - Place a # in front of the `use warnings` in testScript2.pl, and run agin
      - Why is `use warnings` so important?
        - » http://www.perlmonks.org/?node_id=87628

Declaring Variables

- Combine declaration and initialization:

```
my @dna = qw/A G C T T C C A A A/;
my $protSeq = "MDETTYVALLTYDEG";
```

- `use warnings;`
  - **Use the warnings pragma at the top of your programs**
  - Causes compiler to give error and warning messages for a wide variety of problems
  - The warnings pragma is a replacement for the command line flag -w
    - Very useful for debugging and eliminating bugs
    - Lexically scoped
    - Permits finer control over where warnings can or can't be triggered

Declaring Variables

- **`use strict;`**
  - Use the strict pragma at the top of your programs !!
    - Perl will slap your hands with a fatal error whenever you break certain rules
  - **Requires you to declare all variables**
  - Avoids creating variables by typos
- Declaring variable using `my`:

```
my $dna;                  # value of $dna is undef
my ($dna, $prot, $aa);
# $dna, $prot, $aa are all undef
my @dna;                  # value of @dna is ()
```

Scalar Variables - Caveats

- All variables are preceded by a symbol
  - For scalars (variables with a single value), the sign is "`$`"
  - `$protein =` scalar variable that can hold any single value, numerical or string
  - Later we will see that arrays start with "`@`", etc

Scalar Variables - Caveats

- Perl variables are named by these rules
  - Only letters, numbers, and underscores are allowed
  - A letter or underscore must be the first character after the `$`
  - Case sensitive, `$dna != $DNA`
  - Up to 255 characters long

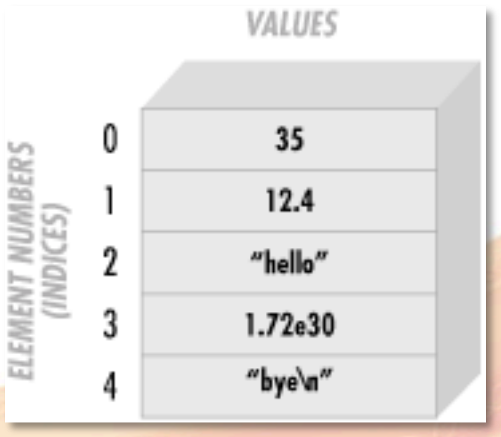
Scalar Variables - Caveats

- Use meaningful variable names
- Don't use single letter names!
  - Loops and counters only time use **single letter variables**
- How should name variables?
  - Use underscores to separate words within a variable name
  - Or capitalize first letter of each word:
  - `$the_dna_motif` and **`$theDnaMotif`** are both good variable names
  - `$r` and `$thednamotif` **are not good**

Arrays in Perl

Array (lists)

- Arrays contain multiple variables
- Can contain a list of:
  - Random numbers
  - List of:
    - strings
    - serial numbers
    - references
    - of *anything*
- Array elements are accessed using indexes
  - Indexing starts at 0
  - There are negative indexes as well



Arrays

`@dna =`

Value	Index	- Index
G	0	-9
C	1	-8
T	2	-7
A	3	-6
A	4	-5
T	5	-4
C	6	-3
A	7	-2
G	8	-1

Arrays are essential for the most useful functions of Perl

They can store more than just a char, like seen to the left

They can store data such as:

- **–the lines of a text file (e.g. primer sequences)**
- **–a list of numbers (e.g. BLAST e values)**
- **–compilation of other data-structures**
- **–etc**

What's My Context: Scalar or List?

- All operations in Perl are evaluated in either **scalar** or **list context**, and may behave differently depending on context

```
my @array = ('cat', 'dog', 'monkey');
my $size = @array;        # scalar context for assignment, return size
say $size;        # prints 3

my ($a) = @array;         # list context for assignment
say $a;           # prints 'cat'

my ($b, $c, $d) = ('', '', '');  # $d is initialized
($a, $b) = @array;
say $a  , ", " ,  $b;             # prints 'cat, dog'
($a, $b, $c, $d) = @array;        # $d is undefined
```

The **left side** of the assignment determines context

testScript3.pl

```
#!/usr/bin/perl
use warnings;
use strict;
use feature qw(say);
#my @days = ("Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday");
my @days = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);

print "These are the days of the week: ";
say "@days";
say @days;
```

Array - Initialization

```
#!/usr/bin/perl
use warnings;
use strict;
use feature qw(say);
my @geneArray = ('EGF1', 'TFEC', 'CFTR', 'LOC1691');
say "@geneArray";
EGF1 TFEC CFTR LOC1691

# there's more than one way to do it
my @geneArray2 = qw(EGF1 TFEC CFTR LOC1691);
say "@geneArray2";
EGF1 TFEC CFTR LOC1691
```

List within double quotes, is a third context - "double quote context" - and it expands list with a space between each element

- Go in and edit the # in testScript3.pl

  ```
  @days = qw(Monday Tuesday Wednesday Thursday Friday
     Saturday Sunday);
  ```

  – `qw` symbolizes a quoted comma delimited list

qw

- A shortcut for eliminating quotes in a list declaration
- Previous example would have array line changed to:
  - edit the # in testScript3.pl

```
@days = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);
```

- `qw` symbolizes a quoted comma delimited list

http://www.perlmeme.org/howtos/perlfunc/qw_function.html

Accessing Arrays

- To assign/retrieve array elements, use [ ] bracket operator
- Remember, an array is a group of data
  - **Scalar items are single data elements**
  - So, to display a single data item, a **scalar symbol** is used

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

my @array;

#assignment
$array[0] = 10;
$array[1] = "20";#do not need "" here, just showing

$array[2] = 20.23;

#accessing elements
my $var1 = $array[0];
my $var2 = $array[1];
```

testScript4.pl

- Negative #'s - used to find information starting at end of an array
  - Last value is -1
  - Second to last value is -2, and so forth (or back as the case may be)

```
say "The 1st mystery value is " , $days[-4] , "\n";
```

- Scalar variables can be used as index values

```
my $i = 2;
say "The 2nd mystery value is " , $days[$i] , "\n";
```

- Assign first value of an array into scalar

```
my ($result) = @days;
say "The 3rd mystery value  is " , $result , "\n";
```

- Assign first two elements of array

```
my ($result1,$result2) = @days;
say "The 4rd mystery values are " , $result1 , " and " ,
  $result2 , "\n";
```

testScript4.pl

- The round brackets are important

```
  my ($result) = @days;
   say "The 3rd mystery value  is " , $result , "\n";
```

- Without the round brackets the length of the array will be passed to the scalar
  - Scalar context

```
    $result = @days;
    say "The 5th mystery values is " , $result , "\n";
```

- Copying the array is easy!

```
    my @days2 = @days;
```

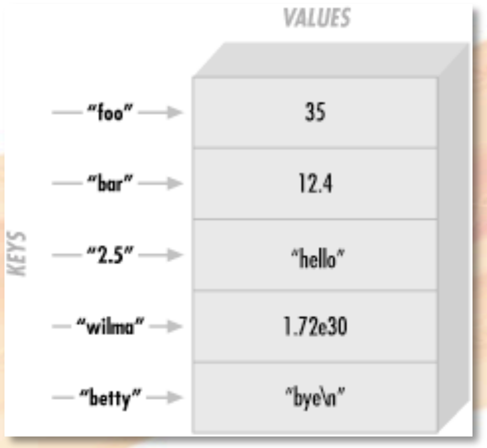
testScript4.pl

- Adding elements to beginning array
```
unshift(@days, $scalar);
@days2 = ($scalar, @days2);
```
- Adding elements to the end array
```
push @days, $scalar
@array2 = (@days2, $scalar);
```
- Remove the first element array
```
my $scalar2 = shift @days;
```
- Remove the last element array
```
$sclars2 = pop @days2;
```
- Combine arrays
```
my @days3 = (@days, @day2);
```

Hashes in Perl

Hashes

- Perl has another **super useful data structure** called a hash
- A hash is an **associative array**
- **Kind of like an** array of variables that are associated with each other
- Complex list data
  - Use scalar as indexes (key)
  - key *associated* with a value
- Use a percent(%) to declare a hash



Hashes

- Hashes
  - Also known as **associative** arrays

```
my %frenchHash = (
       apple  => "pomme",
       pear   => "poivre",
       orange => "Leon Brocard"
);
```

```
my %enzymeHash = (
        EcoRI => "CAATTG",
        BamHI => "GGATCC",
        HindIII => "AAGCTT"
);
```

```
my %three2one = (
        ALA => "A",
        VAL => "V",
        LEU => "L",
        ILE => "I",
        PRO => "P",
        TRP => "W",
        PHE => "F",
        MET => "M",
        GLY => "G",
        SER => "S",
        THR => "T",
        TYR => "Y",
        CYS => "C",
        ASN => "N",
        GLN => "Q",
        LYS => "K",
        ARG => "R",
        HIS => "H",
        ASP => "D",
        GLU => "E"
);
```

testScript6.pl

- Accessing a specific key in the Hash

```
my $var = $french{apple};
say "The word apple is " , $var , " in the French language\n";
```

- Add new key to the Hash

```
$french{quite} = 'tout à fait';
say "The word quiet is " , $french{quite} ,
    " in the French language\n";
```

- Keys must be unique
  - If you use a duplicate key, it will write over that piece of information

```
$french{quite} = 'silence';
say "The word quiet is " , $french{quite}
          , " in the French language\n";
```

testScript6.pl

- Get the number of keys in a Hash

```
my $numKeys = keys %french2;
```

- Copy a Hash

```
my %french2 = %french;
```

- Delete a key/value pair

```
delete $french{apple};
```

Operators Perl

Operators

- Operators are used to operate on variables (it does stuff)
- The important operators are
  - Assignment
  - Comparison
  - Logical

Assignment Operators

- These operators assigning a value (number, string, etc) to a variable

  =

  +=

  -=

  *=

  /=

- testScript7.pl

- Useful for Adding, multiplying, incrementing, etc
- +   -   /   *   %
- ++   --   **
- testScript8.pl

Comparison Operators

- We need to ask things like:
  - Does something equal something else?
  - Is it greater or less than something?
  - Is it not equal?
- Broken into:
  - Number comparisons: `<   >   >=   ==   !=`
  - String comparisons: `lt   gt   le   ge   eq   ne`
- These are used in the Control Structures
- Value can be thought of as True or False / 1 or 0

String Comparison Operators

`eq` Equal to

```
'a' eq  'b' #  0
'b' eq  'a' #  0
'a' eq  'a' #  1
```

`ne` Not-Equal to

```
'a' ne  'b' #  1
'b' ne  'a' #  1
'a' ne  'a' #  0
```

testScript9.pl

`lt` Less than

```
'a' lt  'b' #  1
'b' lt  'a' #  0
'a' lt  'a' #  0
```

`le` Less than or equal to

```
'a' le  'b' #  1
'b' le  'a' #  0
'a' le  'a' #  1
```

`gt` Greater than

```
'a' gt  'b' #  0
'b' gt  'a' #  1
'a' gt  'a' #  0
```

`ge` Greater than or equal to

```
'a' ge  'b' #  0
'b' ge  'a' #  1
'a' ge  'a' #  1
```

Number Comparison Operators

== Equal to

```
1 ==  2 #  0
2 ==  1 #  0
1 ==  1 #  1
```

!= Not-Equal to

```
1 != 2 #  1
2 != 1 #  1
1 != 1 #  0
```

< Less than

```
1 < 2 #  1
2 < 1 #  0
1 < 1 #  0
```

<= Less than or equal to

```
1 <= 2 #  1
2 <= 1 #  0
1 <= 1 #  1
```

> Greater than

```
1 > 2 #  0
2 > 1 #  1
1 > 1 #  0
```

>= Greater than or equal to

```
1 >= 2 #  0
2 >= 1 #  1
1 >= 1 #  1
```

testScript10.pl

Logical Operators

- Allow you to do multiple comparisons
  - Example: Are these two things equal AND is third thing not equal to 1

```
($var1 == $var2) && ($var3 != 1)
```

- See testScript11.pl

Operators are used in Control Structures

Control Structures

Control Structures (Statements)

- So far all of our examples have been completely linear
  - We executed each command in order
  - **Control structures** change program flow
- Useful for making decisions
- Looping, and doing things multiple times
- Basic types:
  - Loops
  - Decisions

Control Structures - Loops

- Need loops
  - Want:
    - to **repeat** certain parts of code
    - something **done** for each item in a list and printed out and …
    - to do something until a **condition** is met
- There are several types of loops, depending on how you want to go about each:

```
for
foreach
while
until
```

Loops - `for`

```
for(initialization; test; increment){
     #Do stuff
}
```

- Know ahead of time exactly how many times want to do something
- `for` statement used to loop through a designated block of code until a specific condition is met
- **Three elements - initialization**, **test** and **increment**
- Initial expression evaluated **once** – at start of the whole process
- Then on test expression, determines whether or not code inside block is executed
- If the test evaluates to:
  - True code is executed
  - False code is not executed and program continues past `for` loop

- See testScript12.pl

Loops - `foreach`

```
foreach my $day (@days){
   say "here is the " , $day;
}

foreach my $word (keys %french){
   say "here is the $word and the value = " , $french{$word};
}
```

- Control structure tailor made process Perl lists and hashes
- **`foreach`** steps through each element of an array using an iterator
- Rather using a scalar as iterator (like the for loop), **`foreach`** uses the array itself

- See testScript12.pl

Loops - `while`

```
while (condition){
    #do some stuff
}
```

- When want to do something **`while`** some condition is true
- Block started by evaluating expression inside `(..)`
- If expression evaluates - true:
  - Code executed &
  - Will continue to execute in a loop **until** expression evaluates **false**
- If expression **initially** evaluates to **false**:
  - Code is never executed &
  - The **`while`** block will be skipped entirely

- See testScript12.pl

Loops - `until`

```
until (condition){
    #do something
}
```

- When want to do something **`until`** something is true
- Block started by evaluating expression inside `(..)`
- If expression evaluates - false:
  - Code executed &
  - Will continue to execute in a loop **until** expression evaluates **tru**e
- If expression **initially** evaluates to **true**:
  - Code is never executed &
  - The **until** block will be skipped entirely

- See testScript12.pl

Control Structures - Decisions - if

- Used to test if a condition if true
- A condition is a test `( $a == 5 )`

`$a <= 5`	`true`
`$a == 6`	`false`
`$a != 7`	`true`

```
if (condition){
    #do some stuff
}
```

Control Structures - Decisions - if-elsif-else

- You may want to do something if something wasn't true…
  - Example: If apples are red, eat them, else discard them
- **See testScript13.pl**

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

use feature qw(say);

my @days = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);

foreach my $day (@days){

           if ($day eq 'Monday'){

               say "Today is " , $day;

           }

           elsif ($day eq 'Tuesday'){

               say "Today is " , $day;

           }

           elsif ($day eq 'Wednesday'){

               say "Today is " , $day;

           }

           else{

               say "Today (" , $day , ") is not a Monday, Tuesday or Wednesday";

           }

}
***
```

What's wrong?

Control Structures - Decisions - if-elsif-else

- There's a better way!
- **Avoid Duplication of code**

```
my @days = qw(Monday Tuesday Wednesday Thursday Friday Saturday Sunday);
foreach my $day (@days){
           if ($day eq 'Monday'    ||
               $day eq 'Tuesday'   ||
               $day eq 'Wednesday' ){
                   say "Today is " , $day";
           }
           else{
                    say "Today (" , $day , ") is not a Monday, Tuesday or Wednesday";
           }

}
```

That's Better!

Open and Closing File

```
#!/usr/bin/perl
use strict;
use warnings;
#get command line argument
my $DnaInfileName = $ARGV[0];

##check for a file given
if (! $DnaInfileName){
    die "You did not provide a input file with a DNA sequence";

}

unless ( open(DNA_INFILE, "<", $DnaInfileName) ) {
    die "Cannot open file " , $DnaInfileName , " ", $!;

}

while (my $line = <DNA_INFILE>){
    chomp $line;
    print $line;
}

close DNA_INFILE;
```

Command Line Arguments

What are we doing here?

Can I open the file? If so get a filehandle.

what's a filehandle?

Process Entire File

Special Perl variable

If you open, then close

```
./testScript15.pl flyUtr.fasta
```

There's a lot going on here, make sure you understand it, if not ask questions!

Filehandles in Perl

- Dealing with files - need something tells Perl which file we're talking about
  - We need a label:
    - Something give us a 'handle' on 'file' want to work with
  - Every Perl script has three FH available at the beginning
    - STDIN - get user input
    - STDOUT - normal print statements
    - STDERR - special print statements for when things go wrong
- If you need an additional FH, then you need to open one
  - First open a FH
  - Then you can print to FH, or read from FH just like we did in last slide
    - Its easy, just remember:
      - "<" for reading

See testScriptFH1.pl

chomp

- `$dnaSeq = <STDIN>` ; #assigns everything typed to $rnaSeq
  - `$dnaSeq` includes "\n" generated by Enter key
    - To get rid of newline safely and efficiently
    - `chomp $rnaSeq;`" is used
  - Only "\n" at the end is removed;
    - All other characters are unaffected
    - Do not confuse this with `chop`!
- Always chomp when reading from a file (line by line)
- Thus a common construction is:

```
#!/usr/bin/perl
use strict;
use warnings;
use feature qw(say);
my $rnaSeq = <STDIN>;
chomp $rnaSeq;
say "You wrote " , $rnaSeq;
##don't chop
chop $rnaSeq;
say "You wrote " , $rnaSeq;
```

```
while (my $line = <INFILE>){
        chomp $line;
        print $line , "\n";
}
```

```
I know what to do!
You wrote I know what to do!
You wrote I know what to do
```

chomp.pl

die

- **die** kills your script safely and prints a message

- Used to prevent you doing something regrettable:
  - Running your script on a file that doesn't exist
  - Overwriting an existing file
  - When you forgot to code something
  - If you come across a value not expected or not given

```
if (! $DnaInfileName){
    die; "You did not provide a input file with a DNA sequence";
}

unless ( open(DNAFILE, "<", $DnaInfileName) ) {
    die "Cannot open file " , $DnaInfileName ,  " " , $!;
}
```

The Default Variable, $_

- Many operations that take a scalar argument, such as `length($x)`, are assumed to work on `$_` if the `$x` is omitted:

```
$_ = "Hello";
print;
print length;
```

```
Hello5
```

- So we can also read a whole file like this:
- Mnemonic: underline is understood to be underlying certain undertakings

This line is equivalent to
`while (my $line = <INFILE>)) {`

```
my $DnaInfileName = 'flyUtr.fasta';
unless (open (INFILE, "<", $DnaInfileName ) ){
    die "Cannot open file " , $DnaInfileName , " " , $!;
}
while (<INFILE>) {
    chomp ; #always when reading
}
close INFILE;
```

Very important to understand how the $_ variable is used!!

Embedding Shell Commands

- use backquotes (`) around shell command
- example using EMBOSS to reverse-complement:

```
`revseq mySeq.fasta mySeq_rc.fasta 2>/dev/null`;
```

- Capture stdout from shell command if desired

```
my $revCom = `revseq mySeq.fasta -filter 2>/dev/null`;
```

```
#!/usr/bin/perl
use strict;
use warnings;
use feature qw(say);

`revseq mySeq.fasta mySeq_rc.fasta 2>/dev/null`;


my $date = `date`;
my $revCom = `revseq mySeq.fasta -filter 2>/dev/null`;

say "date: " , $date;
say "Reverse compliment: " , "\n" , $revCom;
```

testScript16.pl

For Thursday

- Be prepared for a quiz
- Go over all all example scripts from today and know how they work
  - If you have questions, ask!
- Why `use warnings` and `use strict` are so important!!
    - http://www.perlmonks.org/?node_id=87628
- You should:
  - Know how to print
  - How scalars, arrays, and hashes are used
  - Go through all the code examples so far:
    - Make sure you understand them
    - Practice coding some generic examples
- **Catch up on your Perl Readings, if you have not finished up with those yet**
- **Then read**
  - **3. Lists and Hashes**
  - **4. Loops and Decisions**