Introduction to Perl

Le Yan

User Service @ HPC

Adapted from Dave Cross's "Introduction to Perl"

Outline

- Variables
- Control Flow
- Operators and built-in functions
- References
- Functions
- Regex

What is Perl

- **P**ractical **E**xtraction and **R**eporting **L**anguage
- “Perl” is the language
- “perl” is the command

Typical Uses of Perl

- Text processing
- System administration tasks
  - Parsing log files etc.
- Web programming
- Database

Not-So-Typical Use of Perl

- Solving complex PDEs

Philosophy of Perl

- TIMTOWTDI – There Is More Than One Way To Do It

Perl on HPC Systems

System	Version	Softenv key
Super Mike 2	5.16.3	+perl-5.16.3
Philip	5.16.2	+perl-5.16.2
LONI systems	5.8.4	

Running Perl Programs

- Run a Perl program with `perl <program name>`

```
[lyan1@mike1 tutorial]$ cat hello.pl
print "Hello, world!\n";
[lyan1@mike1 tutorial]$ perl hello.pl
Hello, world!
```


Running Perl Programs (cont)

- Or add the shebang line and make it executable

```
[lyan1@mike1 tutorial]$ cat hello.pl
#!/usr/bin/perl

print "Hello, world!\n";
[lyan1@mike1 tutorial]$ chmod 755 hello.pl
[lyan1@mike1 tutorial]$ ./hello.pl
Hello, world!
```


Comment

- Single-line comments start with a “#”
- Multiple-line comments start with “=<word>” and end with “=cut”

Strict And Warnings

- Perl can be a very loose programming language by default
- Use these two statements at the beginning of your Perl program as a safety net (below the shebang line though)
  - Think them as boiler plate

```
use strict;
use warnings;
```


Data::Dumper

- Writes out varialbe contents in perl syntax
  - Very useful in debugging and learning

```
use Data::Dumper;
my $contacts = { 'Frank' => {'email' => 'frank@lsu.edu',
                             'phone' => '578-5655'},
                 'Amy'   => {'email' => 'amy@lsu.edu',
                             'phone' => '578-1420'}
               };
print Dumper($contacts);

#Output:
$VAR1 = {
          'Amy' => {
                     'email' => 'amy@lsu.edu',
                     'phone' => '578-1420'
                   },
          'Frank' => {
                       'email' => 'frank@lsu.edu',
                       'phone' => '578-5655'
                     }
        };
```


Getting Help

- `perldoc <topic>`
  - Lots of useful pages
- Also available at `perldoc.perl.org`

Outline

- Variables
- Control Flow
- Operators and built-in functions
- Regex
- References
- Functions
- Regex

Perl Variables

- Variable name
  - Consist of numbers and letters
  - May not start with numbers (for user variable)
- Variable type (indicated by a preceding sigil)
  - Scalar ($)
  - Array (@)
  - Hash (%)

Declaring Variables

- Use the "my" function
- Perl do not require variables to be declared, but it is a good habit to avoid scoping and typo issues
  - `use strict` will enforce variable declaration

```
my $scalar = 5;
my $scalar = "Some string";

my @array = (1, 2, 3, 4);

my %hash = ('one', 'red'
            'two', 'blue'
            'three', 'green');
```


Variable Names

- Perl allows variables of different types to have identical names
  - Confusing for programmers, so don't do it

```
# This is perfectly legal in Perl
my $var = 5;
my @var = (1, 2, 3, 4);
my %var = ('one', 'red'
           'two', 'blue'
           'three', 'green');
```


Scalar Variables

- Store a single item
- Does not distinguish between numbers and text

```
my $integer = 5;
my $string = “Some string”;
my $meaning_of_life = “42”;
my $decimal = 0.45;
my $scientific = 6.23e23;
```


Coercion

- Perl converts between strings and numbers when necessary (more on this later)

```
my $scalar = 5;
my $life = '42';

say $scalar + $life; # 47
say $scalar.’ ’.$string; 5 42
```


Quotes

- Single quotes – treat characters literally
- Double quotes – expand variable and escape sequence

```
my $item = 'meal';
my $single = 'I paid $4.99 for this $item';
my $double = "I paid $4.99 for this $item";

say $single; # print the literal string
say $double; # will print an error message
```


Quotes (cont)

- Use \ to escape special characters, including ‘ and “

```
my $item = 'meal';
my $single = ‘He said “I paid $4.99 for this
$item”';
my $double = “He said \”I paid \$4.99 for this
$item\”";

say $single; # again, the literal string
say $double; # He said “I paid $4.99 for this meal”
```


Quotes (cont)

- Use `q( )` and `qq( )` in place of single quotes and double quotes, respectively

```
my $item = 'meal';
my $double = qq(He said ”I paid \$4.99 for this
$item“);

say $double;
```


Special Characters

- Special character that can be used with double quotes or `qq( )`

`/n`	New line
`/t`	Tab
`/r`	Carriage
`/f`	Form feed
`/b`	Backspace

"Undef" Value

- A scalar has a "undef" value when initiated
  - Can use `defined()` to test

```
my $scalar;
print $scalar;
print Dumper($scalar);
```


Array Variables

- Store an ordered list of scalars
  - Again, Perl does not distinguish numerical and textual data

```
my @pets = ('cat','dog','hamster','salamander');
my @numbers = (4, 15, 26, 5, 77);
my @mixture = (5, 'bottle', 3.5, $scalar);
```


Array Elements

- Access individual elements using square brackets [] and indices
  - Index starts from 0
  - Note that "$" is used to indicate a scalar is needed

```
my @pets = ('cat','dog','hamster','salamander');

print $pets[0]; # Will print "cat"

#The size of an array is not fixed and can be
expanded by adding new values
$pets[4] = 'guinea pig';
```


Array Slices

- Access multiple elements in an array
  - Note that "@" is used to indicate an array is needed

```
my @pets = ('cat','dog','hamster','salamander');

# Will print "cat" and "hamster"
print @pets[0,2];
# Will print "cat", "dog" and "hamster"
print @pets[0 .. 2];

@pets[4,5] = ('guinea pig', 'parrot')
# Not necessarily an array
($x, $y) = ($a, $b)
```


Array size

- The size of an array is given by evaluating the array in a scalar context

```
my @pets = ('cat','dog','hamster','salamander');

my $size = @pets;
print $size; # will print 4

# will print 3, the index of last element of
@pets
Print my $last_element = $#pets;
```


Try It Yourself

```
use strict;
use warnings;
use feature ‘say’;

my @numbers = (1 .. 10);

#What will you get and why?
say my $size = @numbers;
say $numbers;
say $#numbers;
say @numbers;
```


Context

- Just as in human languages, the meaning of an operation in Perl often depends on the context
  - Amount context – how many items are expected from an operation
  - Value context – how data is interpreted

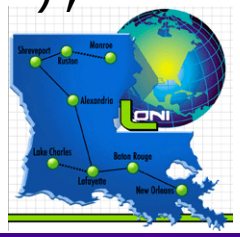
Context (cont)

```
# Amount context: scalar or list
$size = @array; # scalar context
@array_copy = @array; # array context

# Value context: string, scalar or boolean
$five = 5;
print 8 + $five; # numerical context
print ‘His jersey number is ‘.$five; # string context

# Perl relies on the operator to provide context, so
# programmers need to be specific
print ‘five’ + ‘four’; # 0
```


Hash Variables

- Store a number of “key”=>”value” pairs
  - Not ordered
  - Called “associated arrays” sometimes

```
# Create a hash with a flat list
my %phone_book =
('Frank','3465178','Anne','4561098');

# Or use the fat comma (=>), which is clearer
my %lsu_id=
('John' => '892342784', 'Olivia' => '894326756');
```

Hash Values

- Access hash values using {} and the name of the key
  - Again, note the use of "$"

```
my %phone_book =
('Frank','3465178','Anne','4561098');
my $number = $phone_book{'Frank'};

# Like arrays, hashs can be expanded by assigned
new values
$phone_book{'Amy'} = '2985387';
```


Hash Slices

- Similar to arrays, one can access slices of a hash

```
my %phone_book =
('Frank','3465178','Anne','4561098');

my @numbers = $phone_book{'Frank','Anne'};

@phone_book{'Amy','Ted'} = ('2985387', '3362992');
```


Hash Size

- Unlike arrays, it does not help to evaluate hashes in a scalar context
  - "%#" does not work either

```
my %phone_book =
('Frank','3465178','Anne','4561098');

print my $size = %phone_book;
print %#phone_book;

#This is one way of doing it
print my $size = keys %phone_book;
```


Predefined Variables

- Perl has many special predefined variables
  - Some have the form of $/@/% + other punctuations
  - Some have the form of ALL_CAPS
  - Explained in `perldoc perlvar`

Predefined Variables (cont)

- Something that makes Perl interesting
  - For example

```
[lyan1@mike1 tutorial]$ cat cat.pl
while (<>) {
  print;
}
[lyan1@mike1 tutorial]$ perl cat.pl datafile
>Rosalind_1
GATTACA
>Rosalind_2
TAGACCA
```


$_

- The default input and pattern-searching space
  - "the default variable"
    - Many Perl operations set or use the value of $_ if no others are provided

```
$_ = 'I am the default variable';
say;
```


@ARGV

- The array that stores command lines arguments
  - Many array operations in the main program use its value if none is provided

```
my $num_args = @ARGV;
say "$num_args arguments from command line";
say "The first argument is”, shift;
```


@_

- The default array
  - Within a function, @_ contains the parameters passed to that subroutine
  - Many array operations within the function use its value if none is provided
  - More on this later

%ENV

- A hash that stores the shell environment variables
  - Keys are the shell variable names (PATH etc.)
  - Values and the shell variable values

Try It Yourself

```
# This mimic the “cat” shell command
# The readline operator <> takes @ARGV as
# its operand and reads the provide file list
# line by line, which is passed to print as
# the value of $_

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


Outline

- Variables
- Control Flow
- Operators and built-in functions
- References
- Functions
- Regex

Control Flow

- Controls the order of program execution
  - Conditional
  - Loop

Boolean Context

- Perl does not offer a boolean type variable
  - Evaluates different types variable when presented in a Boolean context
  - A expression is false in Perl if it is a
    - Number 0
    - Empty string
    - Undef
    - Empty list
  - Everything else is true

Boolean Context (cont)

```
# Scalars with a value of 0 or undef evaluate
# to “false”
my $zero = 0;
my $another_zero = 100 * $zero;

# But a string like this evaluates to “true”
my $zero_but_true = “0.0”;

# Empty arrays and hashes are “false”
my @array;
my %hash;

# Non-empty arrays or hashes are true, even
# if the values are 0 or undef
$array[0] = undef;
```


Conditional Construct - if

- If…elsif…else construct

```
if (EXPRESSION 1) {
        Block of code
}
elsif (EXPRESSION 2) {
        Block of code
}
else {
        Block of code
}
```


Postfix Form of if

- Simplify the code sometimes

```
say “Hello, Bob” if ($name eq ‘Bob’)
```


Comparison Operators

Equal	==	eq
Not equal	!=	ne
Greater than	>	gt
Greater than or equal	>=	ge
Less than	<	lt
Less than or equal	<=	le

The first column enforces numerical context

Comparison Examples

```
62 > 42                         #True
‘0’ == (3*2)-6                  #True
‘apple’ gt ‘banana’             #false
‘apple’ == ‘banana’             #true
‘apple’ eq ‘banana’             #false
1 + 2 == ‘3 bears’              #true
1 + 3 == ‘three’                #false

if ('apple' == 'banana') {say 'Not good';}
if (! ('apple' eq 'banana')) {say 'This is better';}
```


Boolean Operators

Operator	Choice 1	Choice 2
Not	!	
And	and	&&
Or	or	||

Unless

- Equivalent to “if not”
- Useful for parameter validation

```
sub validate_identity {
  return unless ($id{$name})
}
```


Ternary Conditional Operator

- `EXPR ? EXPR 1 : EXPR 2`
  - Execute `EXPR 1` if EXPR is true, otherwise execute `EXPR 2`

```
my $salutation = male($name) ? 'Mr.' : 'Ms.'
```


Short-Circuit Operators

- Use `or` to write clearer code

```
# Only needs to evaluate the second expression when
the first is false

open FH, ‘datafile’ or die “Can open file $!”;

@ARGV == 3 or print $usage_msg;
```


Loops - for

- C-style for loops
  - Rarely used

```
my @square;
for (my $i = 0; $i <= $#square ; $i++) {
    $square[i] = $i ** 2;
}
```


Loops - foreach

- Loop over a list – preferred over the C-styled for loop

```
foreach my $i (1 .. 10) {
    say “$i * $i =“,$i*$i;
}

#Or use the default variable
foreach (1 .. 10) {
    say “$_ * $_ = “,$_*$_;
}

#Postfix form
say “$_ * $_ = “,$_*$_ foreach (1 .. 10)
```


Loops – foreach (cont)

```
# Loop over a hash
foreach (keys %hash) {
    say $hash{$_};
}

# Perl actually treats for and foreach
interchagably, so the following code is
perfectly fine:
for (keys %hash) {
    say $hash{$_};
}
```


Loops – while

- While (EXPR) { code block}
  - Repeat the same block of code while EXPR is true

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

# This loop will do the same thing as the
one on previous slide
while (@array) {
    my $i = shift @array;
    say “$i * $i = “,$i*$i;
}
```


Loops - until

- Until (EXPR) { code block }
  - Repeat the same block of code while EXPR is false
  - The opposite of while

Loop Control

- `next` – jump to the next iteration
- `redo` – jump to the start of the same iteration
- `last` – jump out of the loop

Outline

- Variables
- Control Flow
- Operators and built-in functions
- References
- Functions
- Regex

Arithmetic Operators

- The usual suspect

+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus
**	Exponentiation

Shortcut Operators

- `$x <op>= $y` performs operations `$x = $x <op> $y`

+=	Add
-=	Subtract
*=	Multiply
/=	Divide
%=	Modulus
**=	Exponentiation

Shortcut Operators (cont)

- $x++ is equivalent to $x -= 1
- $x-- is equivalent to $x -= 1
- ++$x and --$x have the same meanings, but there is subtle difference

```
my @array = (0 .. 10)
my $i = 0;
say $array[$i++]; #will print 0
say $array[++$i]; #will print 2
```


Numeric Functions

- Again the usual suspects

`abs`	Absolute value
`cos, sin, tan`	Trigonometric
`exp`	Exponentiation
`log`	Logarithm
`rand`	Random number generator
`sqrt`	Square root

String Operators

```
my $first_name = ‘Le’;
my $last_name = ‘Yan’;
# Concatenation (.)
my $full_name = $first_name.“ “.$last_name; # “Le Yan”
# Repetition (x)
my $line = ‘-’ x 80;
# Shortcut available
$full_name .= ‘ had his lunch.’;
```


String Functions

```
# length: returns the length of a string
my $full_name = “Le Yan”;
say length $full_name; #will print 6
# uc: returns all uppercase version of string
# lc: returns all lowercase version of string
# ucfirst: returns the string with first letter in
uppercase
# lcfirst:returns the string with first letter in
lowercase
# chomp: returns the string with the trailing new line
character removed (if there is any)
```


Substr

```
# substr: returns a substring
# Syntax: substr EXPR,OFFSET,LENGTH,REPLACEMENT
# Extracts a substring out of EXPR and returns it.

my $s = “The black cat climbed the green tree”;
my $color = substr $s, 4, 5; # black
#If LENGTH is negative, leave that many characters off the end.
my $middle = substr $s, 4, -11; # black cat climbed the
#If LENGTH is omitted, returns everything through the end.
my $end = substr $s, 14; #climbed the green tree
#If OFFSET is negative, start from the end of the string.
my $tail = substr $s, -4;
#If REPLACEMENT is provided, substitute the substring with it (will
happen in place).
substr $s, 14, 7, “jumped from”;
```


Split

```
# split: splits a string into a list of strings and
returns the list
# syntax: split /PATTERN/,STRING

my $s = “The black cat climbed the green tree”;
my @array = split “ “,$s;
say $array[1]; # black

# if PATTERN is an empty string, will split into a list
where each character is an element
my @another_array = split “”,$array[1];
say $another_array[1]; # l
```


Array Functions

- pop, push, shift, unshift

```
my @names = (‘Frank’, ‘John’);

#pop: remove and return last element
say pop @names; # will print ‘John’
#push: add a new element to the end
push @names,’Amy’; # @names is now (‘Frank’,’Amy’)
#shift: remove and return first element
say shift @names; # will print ‘Frank’
#unshift: add a new element to the start
unshift @names,’Fred’; @names is now (‘Fred’,’Amy’)
```


Join

- Opposite of split

```
my @names = (‘Frank’, ‘John’, ‘Amy’, ‘Olivia’);

say join ‘:’,@names; # Frank:John:Amy:Olivia
```


Reverse

- Reverses an given list

```
@array = (2, 5, 3, 1);
@reversed = reverse @array; #(1, 3, 5, 2)
```


Sort

- Returns a sorted list (does NOT sort in place)

```
@array = (2, 5, 3, 1);
@sorted = sort @array; #(1, 2, 3, 5)

# The default order is ascending in ASCII
# Sometimes the result is surprsing
@array = (‘a’,’c’,’D’,’b’,’E’);
@sorted = sort @array; (‘D’,’E’,’a’,’b’,’c’)

@array = (1 .. 10);
@sorted = sort @array; (1,10,2,3,…)
```


Sort (cont)

- `sort` can be customized by adding a sorting block as an argument

```
# Syntax: sort {$a cmp $b} LIST
# $a and $b are two elements being compared.
# $a and $b can be replaced by two expressions
# involing $a and $b, respectively.
# The order is decided by the result of #comparison.

# cmp is for strings
sort {$a cmp $b} @list;
# cmp
sort ($a <=> $b} @list;
```


Sort (cont)

```
# Sort in numerically descending order
@sorted = sort {$b <=> $a} @array;

# Case insensitive sort
@array = (‘a’,’c’,’D’,’b’,’E’);
@sorted = sort {fc($a) cmp fc($b)} @array; # (’a’,’b’,’c’,‘D’,’E’)

# Sort hash keys by their associated values
@sorted = sort {$hash{$a} <=> $hash{$b}} keys %hash;
```


Map

- Evaluate an expression or a block of code for each element of a list
  - Like an implied loop
  - Setting each element as $_

```
# The number of returned elements can be different from the
# input.
# Use an empty list () to skip an element

my @squares = map { $_ > 5 ? ($_ * $_) : () } @numbers;
```


Default Variables

- Many of the functions mentioned above operate on $_, @ARGV or @_

Hash Functions

- `keys`: returns a list of keys
- `values`: returns a list of values
- `delete`: deletes a pair from the hash
- `exists`: tells if a element exists (key or value)

File I/O

- `open` opens a file and associates with a file handle
- `close` closes a file

```
# ‘<‘ is for reading, ‘>’ is for overwriting
# ‘>>’ is for appending
open (my $file, ‘<‘, ‘datafile’);

# Read one line
my $line = <$file>;

# Read all lines
my @lines = <$file>;

close($file);
```


File I/O (cont)

- Use `print` or `say` to write to a file

```
open (my $file, '>', 'datafile');

print $file $somedata;

close($file);
```


File Test Operators

- Use with `if` to check file attributes

`-e $file`	File exists?
`-r $file`	File readable?
`-w $file`	File writable?
`-d $file`	File is a directory?
`-f $file`	File is a normal file?
`-T $file`	File is a text file?
`-B $file`	File is a binary file?

Outline

- Variables
- Control Flow
- Operators and built-in functions
- References
- Functions
- Regex

References

- Perl references are like pointers in C
- It is a mechanism to refer to a value without making a copy
  - Any change made through the reference is made in place
- A reference always fits in a scalar

Reference Operator

- Putting \ in front of a variable creates a reference to it

```
my $scalar_ref = \$scalar;
my $array_ref = \@array;
my $hash_ref = \%hash;
```


Dereferencing

- An extra sigil is needed to deference a reference to access the value it refers to

```
my $scalar_ref = \$scalar;
${$scalar_ref} = ‘some value’;

my $array_ref = \@array;
my $size = @{$array_ref};
${$array_ref}[0] = ‘some other value’;

my $hash_ref = \%hash;
foreach (keys %{$hash_ref}) {
  say ${$hash_ref}{$_};
}
```


Deferencing (cont)

- The curl brackets can be omitted
  - They improve readability though

```
my $scalar_ref = \$scalar;
$$scalar_ref = ‘some value’;

my $array_ref = \@array;
my $size = @$array_ref;
$$array_ref[0] = ‘some other value’;

my $hash_ref = \%hash;
foreach (keys %$hash_ref) {
  say $$hash_ref{$_};
}
```


Deferencing (cont)

- Can also use arrow (->) to access elements of arrays and hashes
  - `${$array_ref}[0]` is equivalent to `$array_ref->[0]`
  - `${$hash_ref}{$key}` is equivalent to `$hash_ref->{$key}`

Anonymous Variables

- It is possible to create anonymous variables using references
  - Only accessible by using references

```
# Use [] for anonymous arrays
my $array_ref = [1 .. 10];
say $array_ref->[2];

# What will happen?
$array_ref = (1 .. 10);

# Use {} for anonymous hashs
my $contacts_ref = ( ‘Frank’ => ‘frank@lsu.edu’,
                     ‘Amy’   => ‘amy@lsu.edu’};
```


Why Use References?

- Complex data structure
- Passing parameters (more on this later)

2-D Arrays (Array of Arrays)

```
# This will result in an array (1,2, …, 9)
# due to list flattening
my @numbers = ((1 .. 3),
               (4 .. 6),
               (7 .. 9));

# [ LIST ] creates an anonymous array and
# returns the reference
# So we are creating an array of references
my @array_of_arrays =
        ([1 .. 3],[4 .. 6],[7 .. 9]);
say $array_of_arrays[0]; # ARRAY(0x1c45a98)
say $array_of_arrays[0]->[1]; # 2
```


Complex Data Structure

- Hash of hashes

```
my %contacts = ( ‘Frank’ => {‘email’ => ‘frank@lsu.edu’,
                             ‘phone’ => ‘578-5655’},
                 ‘Amy’   => {‘email’ => ‘amy@lsu.edu’,
                             ‘phone’ => ‘578-1420’}
               );
say $contacts{‘Frank’}->{‘email’};
```

- You can also define array of hashes, hash of arrays etc.

Outline

- Variables
- Control Flow
- Operators and built-in functions
- References
- Functions
- Regex

Write Your Own Functions

- Declare a function

```
sub function_name {
        block of code
}
```

- No list of input parameters in function definition

Input Parameters

- Instead, all input parameters are put in the array @_
  - The function needs to either unpack it or operate on it directly
  - List flattening applies

```
my %contacts = (‘Frank’ => ‘frank@lsu.edu’,
                ‘Amy’ => ‘amy@lsu.edu’);

flattened_contacts(%contacts);
contacts_as_hash(%contacts);

sub flattened_contacts {
  say “The key is “,shift;
  say “The value is ”,shift;
}

sub contacts_as_hash {
  my %hash = @_;
}
```


Parameter Passing

```
# What will happen?
my @array1 = (1 .. 3);
my @array2 = (4 .. 6);
check_size (@array1, @array2);

sub check_size {


  my (@a1,@a2) = @_;
  say @a1 == @a2 ? ‘Yes’ : ‘No’;

}
```


Parameter Passing (cont)

```
# What will happen?
my @array1 = (1 .. 3);
my @array2 = (4 .. 6);
check_size (@array1, @array2);

sub check_size {
# @_ is flattened as (1,2,3,4,5,6)
# @a1 gets all while @a2 gets none
  my (@a1,@a2) = @_;
  say @a1 == @a2 ? 'Yes' : 'No';

}
```


Parameter Passing (cont)

```
# Should use reference
my @array1 = (1 .. 3);
my @array2 = (4 .. 6);
check_size (\@array1, \@array2);

sub check_size {


  my ($a1,$a2) = @_;
  say @$a1 == @$a2 ? ‘Yes’ : ‘No’;

}
```


Returning Values

- Functions can return none, one or more values

```
my @array1 = (1 .. 3);
my @array2 = (4 .. 6);
say ‘Yes’ if check_size (\@array1, \@array2);

sub check_size {

  my ($a1,$a2) = @_;
  my $result = @$a1 == @$a2 ? ‘Yes’ : undef;
  return $result;

}
```


Outline

- Variables
- Control Flow
- Operators and built-in functions
- References
- Functions
- Regex

Regex

- Regex stand for **REG**ular **Ex**pression
  - Powerful tool for text processing
  - Allows users to define a pattern to describe characteristics of a text segment
  - Can be used to match or modify text
  - Perl regex documentation: `perlretut`, `perlreref`, `perlre`

Matching

- Regex can be used to match literals

```
# Match operator: // or m// or m with any delimiter
# Binding operator: =~

# The match operator returns true/false in scalar context, a list of
captured text in list context
my $s = “GATATATGCATATACTT”;
say ‘Found a match’ if $s =~ /TATA/;

# Operate on $_ if operand is omitted
$_ = “GATATATGCATATACTT”;
say ‘Found a match’ if /TATA/;

# The function index can be used for this purpose as well
# It returns the position of first match or -1 if none
say index $s,’TATA’; # will print 2
```


Substituting

```
# Substitution operator: s/PATTERN/REPLACEMENT/
# Can choose any delimiter
# Also operate on $_ if operand is omitted;

# Returns true if successful in scalar context, the number of
replacements in list context
# The substitution is done in place

$_ = “GA TATA TGCA TATA CTT”;
s/TATA/GCGC/g; # The value is true
say; # will print GA GCGC TGCA GCGC CTT

$_ = “GA TATA TGCA TATA CTT”;
s/TATA/GCGC/; # Without “g”, will return after the first success
say; # will print GA GCGC TGCA TATA CTT
```


Metacharacters

.	Any character except \n		
^	Start of string		
$	End of string		
\s	Any whitespace	\S	Any non-whitespace
\d	Any digit	\D	Any non-digit
\w	Any word	\W	Any non-word
\b	Any word boundary	\B	Anything except a word boundary

Special characters such as ., ^, $ need to be escaped by using "\" if the literals are to be matched

Metacharacters (cont)

```
While (<FILE>) {
    say if m|^http://|; # Match any line that starts with http://
    say if /\bperl\b/; # Match any line with the word “perl”
    say if /\S/; # Match any line with content;
    say if /\$\d\.\d\d/; # Match any line containing $x.xx
}
```


Define Patterns

- The qr// operator define patterns that can be used and reused for later match

```
my $http = qr/^http:\/\//;
my $www = qr/www/;
while (<FILE>) {
    say if /$http$www/;
}
```


Quantifiers

- Allow users to specify the number of occurrence

*	Zero or more
+	One or more
?	Zero or one
{n}	Exactly n times
{n,}	At least n times
{n,m}	At least n, but at most m times

Quantifiers (cont)

```
While (<FILE>) {
    say if /ca?t/; # Match “ct” or “cat”
    say if /ca*t/; # Match “ct”, “cat”, “caat”, …
    say if /ca+t/; # Match “cat”, “caat”, …
    say if /ca{3}t/; # Match “caaat”
    say if /\d{3}-?\d{3}-?\d{4}/; # Match a phone number with
area code
}
```


Greediness

```
# + and * are greedy – they will grab any many character as possible
$gene = “GATATATGCATATACTT”;
$gene =~ s/A.*T/GC/; # GGC

# Appending a “?” will make them “reluctant”
$gene = “GATATATGCATATACTT”;
$gene =~ s/A.*?T/GC/; # GGCATATGCATATACTT
```


Character Classes

- Define a class of characters

/[aeiou]/	Match all vowels
/[^aeiou]/	Match any character except ‘a’, ’e’ ,’i’, ’o’, ’u’
/[a-zA-Z]/	Match any letter
/[-!?]/	Match ‘-’, ‘!’, ‘?’ (hyphen at start)
/!\-?/	Same as above (hyphen escaped)

Modifiers

- Modifiers change the behavior of regex

```
$gene = “GATATATGCATATACTT”;

# /i cause regex to ignore case distinctions
say “Match found” if $gene =~ /tata/i;

# /g cause regex to match globally throughout a string
$gene =~ s/TATA/GCGC/g;
say $gene; # will print GAGCGCTGCAGCGCCTT

# /r cause regex to ignore whitespace in the regex pattern
# Modifiers can be combined
say “Match found” if $gene =~ /g c g c/ri
```


Alternation

- Use “|” to match either one thing or another

```
my $pork = qr/pork/;
my $chicken = qr/chicken/;
say “Food okay” if $food =~ /$pork|$chicken/i;
```


Grouping And Capturing Matches

- Use parentheses to
  - Group atoms into larger unit
  - Capture matches for later use

```
# Will match “por and beans”
my $por_and_beans = qr/pork?.*?beans/
# Will not match “por and beans”
my $pork_and_beans = qr/(pork)?.*?beans/

# () also capture the matched text and store
# them in special variables $1, $2 etc.
while (<FILE>) {
  if (/^(\w+)\s+(\w+)/) {
    say “The first word is: “,$1;
    say “The second word is: “,$2;
  }
}
```


Named Captures

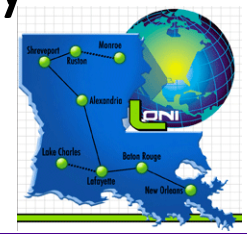
```
# Use ?< name >  in a group to name a capture
# It must appear immediately after the left parenthesis
# Can access it later with $+{name}

my $phone_number = qr/\d{3}-?\d{3}-?\d{4}/;
while (<FILE>) {
  push @numbers, $+{phone} if /(?<phone>$phone_number)/;
}
```


Perl Modules

- Think Perl modules as libraries – reusable codes
- CPAN – tons of modlues developed by the Perl community (www.cpan.org)
  - Before setting out to write something serious, check CPAN first

Installing Modules

- Option 1: manual installation
  - Download the tarball and extract the content
  - Create a Makefile
  - Make, make test and make install
- Option 2: use the “cpan” module
  - Provides a console to search, download and install modules and their dependencies automatically



Installing Modules on HPC Systems

- You don't have root process, so you need to install them in your own user space
  - Most likely somewhere in your home directory
  - Then point the environment variable $PERL5LIB to the location

Using Modules

- Use the `use` function
  - We've seen a few examples

```
use strict;
use warnings;
use Data::Dumper;

#use optional components
use feature qw(say switch fc);
```


Perl One-Liners

- Perl can perform lots of seemingly complex task with one line of code, aka one-liners
  - There are other similar tools (e.g. awk)
- For example:

Number and print only non-empty lines in a file (drop empty lines)

```
perl -ne 'print ++$a." $_" if /.‘ file
```

Remove all consecutive blank lines (leave only one)

```
perl -00 -pe ''
```


Perl Command Line Options

- `-e` allows to enter a program directly on the command line-l, -n, -a
- `-n` assumes the follow loop around the provided program:

```
while (<>) {
  # your program goes here
}
```


More Command Line Options

- `-p` assumes the follow loop around the provided program:

```
while (<>) {
  # your program goes here
} continue {
  print or die "-p failed: $!\n";
}
```

- `-l` automatically chomps the input when used with `-n` or `-p`
- `-a` turns on autosplit mode when used with a `-n` or `-p`.

Double Space A File

```
[lyan1@mike1 tutorial]$ cat data
Apple
Pear
Mango
[lyan1@mike1 tutorial]$ perl -pe '$\="\n"' data
Apple

Pear

Mango
```

```
# The one-liner is equivalent to:
while (<>) {
# if defined, the content of $\ is appended
# to each print command
  $\ = “\n”;
} continue {
  print or die "-p failed: $!\n";
}
```


Printing All Lines between Line A and Line B

```
[lyan1@mike1 tutorial]$ perl -ne 'print if $. >= 2 && $. <= 3’ data
Pear
Mango
```

```
# The one-liner is equivalent to:
while (<>) {
# $. is the current line number
  print if $. >=2 && $. <= 3;
}
```


Not Covered

- Scope
- Advanced function features
- OOP features
- …

Beyond This Tutorial

- http://perldoc.perl.org
- http://learn.perl.org
- http://www.catonmat.net/series/perl-one-liners-explained

Exercise

- Data files can be found under

  `/home/lyan1/traininglab/perl/data`

- Solutions can be found under

  `/home/lyan1/traininglab/perl/solution`

- Each of the solutions is just a solution, not the solution (Remember, 'Tim Toady')

Exercise 1: Hamming Distance

Background

Given two strings with same length, the Hamming distance is the number of corresponding characters that differ. For example, the Hamming distance between `GAGCCTACTAACGGGAT` and `CATCGTAATGACGGCCT` is 7.

Problem

Given: Two strings read from a file

Return: The Hamming distance

Note

There are two data files, one large and one small. The answer should be 7 for `HAMM_data_small` and 489 for `HAMM_data_large`

Exercise 2: Replace and Print

Problem

For a given file, replace all occurrence of “GTAG” with “CATC”, then print each line that contains “GACTA”

Note

There are two data files, `replace_data_large` and `replace_data_small`.

Should be able to do this with a one-liner.

Exercise 3: FASTA file reader

Background

FASTA is a file format used to store genetic strings. Every string in a FASTA file begins with a single-line that contains the symbol '>' along with some labeling information about the string. All subsequent lines contain the string itself. A .fasta file would look like this:

```
>Taxon1
CCTGCGGAAGATCGGCACTAGAATAGCCAGAACCGTTTCTCTGAGGCTTCCGGCCTT
CCC TCCCACTAATAATTCTGAGG
 >Taxon2
CCATCGGTAGCGCATCCTTAGTCCAATTAAGTCCCTATCCAGGCGCTCCGCCGAAGG
TCT ATATCCATTTGTCAGCAGACACGC
```

Problem

Given: A .fasta file

Return: A hash with the labels as the keys and the strings as values

Note

One string can and often span across multiple lines, so you need to take it into consideration.

Exercise 4: Computing GC Content

Background

DNA strings are composed of “A”, “C”, “G” and “T”. The GC content of a given DNA string is the percentage of symbol “G” and “C”.

Problem

Given: DNA strings in FASTA format

Return: The label of the string that has highest GC content, followed by its GC-content.

Note

You might need to use the fasta file reader in Exercise 3.

Exercise 5: Overlap Graphs

Background

For a group of strings, the adjacency list corresponding to O_n is all pairs of strings (***s,t***) such that the last n characters of ***s*** is the same with the first n characters of ***t***.

Problem

Given: DNA strings in FASTA format

Return: The adjacency list corresponding to O_3.

Note

You might need to use the fasta file reader in Exercise 3.

If the start of string ***s*** overlaps with the end of string ***t***, the returned pair should be (***t,s***), not (***s,t***).

A string that overlaps with itself, such as (***s,s***), is not allowed and should not be counted.

Exercise 5: Overlap Graphs

Example

If the given strings are:

```
>Rosalind_0498
AAATAAA
>Rosalind_2391
AAATTTT
>Rosalind_2323
TTTTCCC
>Rosalind_0442
AAATCCC
>Rosalind_5013
GGGTGGG
```

Then the answer should be:

```
Rosalind_0498 Rosalind_2391
Rosalind_0498 Rosalind_0442
Rosalind_2391 Rosalind_2323
```