CS3157: Advanced Programming

Lecture #1

May 22

Shlomo Hershkop
shlomo@cs.columbia.edu

1

First!

- I am not Shlomo!
- Who am I (I am the TA)
- Shlomo's excuse: Giving a talk in California, will be back on Wednesday!

2

It summer Session!

- Welcome (Greeting from Prof Shlomo)
- Ask yourself, is it better to spend the summer outside or inside on this stuff ?!
- Hope to be very informal
- I hope to convince you this is more fun

3

Today

- Basics (of the course)
  - Overview of the course and objectives
  - Administrative issues
- Basic Perl
  - Absolute minimum
  - Syntactic details
- Setting up environment (during the break?)

4

Today

- More Perl
  - Subroutine
  - File I/O
  - Regular Expression
  - Debugging
- Basic Shell Programming (if we have time)

5

Basic - overview of the class:

6

What?

- CS3157: Third course for CS majors.
- Prerequisites:
  - Intermediate knowledge in Programming
  - Object Oriented Programming:
    - What, why, how, and when.
  - Program Designs.
    - Not enough to know how to write the program, need to know how to do it correctly.
    - Need to learn tool of the trade

7

Goal (of the course)

- Thing are much easier if everyone knows why they are here, and what we are trying to accomplish.
- I will not stand here an lecture (although there will be some of that). This is going to be a very interactive course.
- **We will learn about programming ideas (and some practical skills) while trying to have fun**.

8

So what are we going to be doing?

- Cover some practical languages:
  - Perl
  - C
  - C++
- Cover practical skills:
  - Debugging
  - Environmental setup
  - CGI/Web based programming
  - Regular expressions
  - Web scripting
  - Plus more!

9

Point

- Programming is not Java !
- Computer science != programming

- Give you a feel of the real world:
  1. Problem description blurry
  2. Many choices for programmer
  3. Will learn best tools to stay lazy ☺

10

Basic – admin Issues

11

Basics

- Instructor: Professor Shlomo Hershkop (shlomo@cs.columbia.edu)
- Class website:
  - **cs.columbia.edu/~sh553/teaching/su06-3157/**
  - Check it regularly (at least twice a week).
    - See announcement sections for update info.
  - Will give you my background on Wednesday when we meet
- Meet twice a week: 627 Mudd

12

Resources

- TA's:
  - Tae Yano (ty2142@columbia.edu)
  - I will have office hour 2 to 3 hours a week, will post the time
  - Place: Mudd 122a (TA room)
  - else available by appointment
- Since we are a small class:
  - We will setup a mail list so that an email will go to everyone:
    - Good:
      How do I check what version of java is running?
      Hints for problem X
    - What does Error ?@?@!?@ mean ?
  - Bad: What is wrong with the following code:
    public class foo()
  - These kind of questions email privately to TA or Instructor

13

Requirements

- Interest to learn about Computer Science
- Interest to Learn to use cool tools
- Interest to Learn to make your own tools

14

Textbook

- Textbook can be acquired online or at the Columbia Bookstore.
  - Else: borrow, threaten, or ‘acquire’ a book
- Perl
  - Programming Perl (get the last edition)
  - Wall, Christiansen, and Orwant
    O'Reilly
- C
  - C how to program
  - Dietel and Dietel

15

Reading

- I will be posting reading on the website and in class notes
- Please try to keep up with the reading
  - I will try to make up examples for class, but there are random stuff which the book covers which is good to see in print
  - Feel free to ask questions from the book

16

Course Structure

- 6 Labs – 120 points
  - Out Wednesday, Due by class time
- 2 Homeworks – 60 points
  - Will have about 2 weeks per homework
- Final (60 points)
  - open book
- **Homework/Lab is very important**:
  - Firm believer in hands on learning
  - Start early
  - Come to office hours, and ask questions
    - We are here for YOU!

17

Class participation and Attendance

- Attendance and participation is **expected**
  - Very interactive lectures & Labs
  - Small class, means more help
  - Class anonymous feedback system
- If you have to miss class, I expect you to catch up.
  - **It's a short semester**, (better coming to class than catching up later...)
  - There will be class notes posted to the website
  - There will be many examples in class only, so make sure to get someone's notes.

18

Homework & Projects

- Written:
  - Will be collected at first class after HW deadline.
- Programming:
  - Online submission
  - **Must be able to run on cunix system** (this is important).
- Late policy:
  - You have late days that can be used during the semester.
  - I can only review the homework and approaches if everyone submits on time, so try to ask for help earlier rather than later

19

Labs

- Generally will create a few programs
- If everyone has a laptop we can work in class (end of class)
- Online submissions – will create script in first lab!
- Will be around to answer questions hints
- Can **NOT** ask for code from other students
  - Can ask input/output
  - General ideas
  - Use your best judgment

20

Cheating

•Don’t

21

Cheating Policy

- Plagiarism and cheating:
  - I’m all against it. It is unacceptable.
- You’re expected to do homeworks by yourself
  - This is a learning experience.
  - You will only cheat yourself.
  - My job is to help you learn, not catch you cheating, but….
- Automated tools to catch plagiarizers
  - http://www.cs.berkeley.edu/~aiken/moss.html
  - Moving stuff around, renaming, etc. doesn’t help
- Results: instant zero on assignment, referral to academic committee
  - **Columbia takes dishonesty very seriously**
  - I’d much rather you come to me or the TAs for help

22

Feedback System

- Last minute of class will be set aside for feedback:
  - Please bring some sort of scrap paper to class to provide feedback.
  - Feel free to leave it anonymous.
  - Content: Questions, comments, ideas, random thoughts.
- I will address any relevant comments at the beginning of each class
- Summer is short, so provide feedback !
- Please feel free to show up to office hours or make an appointment at any time

23

Shopping List

- You need either a cunix or CS account
  - CS: https://www.cs.columbia.edu/~crf/accounts
  - Try to log into the account asap
- Make textbook plans
  - Recommend : Programming Perl
  - You can choose any perl reference
- Check out the class page

24

Survey 1

- Programming background ?
- Do you have access to a laptop?
- Any cool technologies you would like to see covered ?

(collect during the break)

25

Any Questions ?

26

Last plug

- One of the points of computer science is to teach you how to think, learn, and analyze computational related information.
- Example:
  - Task: Create a program to run a web based game, which will be marketed to both desktop and phone users.
  - Any ideas on how to design the programming backend?
  - Ideas on how to measure requirements.
  - What else is important?

27

Basic Perl - preliminary

28

Perl

- *What is it?*
  - *Perl* was originally designed as a logging tool, released by Larry Wall in 1987.
  - Open source and cross platform. Current version 5.8.7.
  - Referred to as “duct-tape” of the internet
  - Will quickly learn why ☺

29

Perl

- What is it?
  - Scripting language
  - Aims to be a USEFUL language
  - Base + tons of libraries
  - Both a compiler and byte code executable
- Where to get it?
  - cpan.org
  - www.activestate.com/Products/ActivePerl/

30

Difference: Java and PERL?

- Java
  - High Level Language (or low, depends on which way you see it from)
  - Source code is compiled to byte code
  - Byte code = java execution instructions
  - Byte code executed by java
- Perl
  - Scripting language - Very very non-rigid structure. **Many way to say the same thing.**
  - code can be interpreted line by line in real time - i.e. compiles and executes each time invoked
  - A lot of functionality built into base language. **String (text) handling second to none (python?)**

31

- How would you write a program to extract all email addresses from a text file ?
  - With Java??
  - Perl would be much better at such problem

32

(Before start) Environmental Hazards

- Depending on the local system will behave differently:
- Cunix
  - Anyone know what operating system they run ?
- CS students - CS department has both of these main os's :
  - Linux
  - SunOS
- Windows
  - Active perl
  - Cygwin
    - Perl
- VNC
  - Allow you to remote connect to CS if you have an account

33

Pre-programming

- What do you name a perl script ?
  - Something.pl (e.g., "test.pl")
- Make sure there is a Perl
  - Many times its important to check which version of Perl is being used. Perl evolved over time. Might require a minimum version to work.
    - perl –v
  - On unix/linux/sun can see where the Perl compiler is located (this gives you correct path to the Perl).
    - which perl

34

Running (or executing) the code

- From the command line.
- On unix/linux, you need to tell system to execute your perl script

```
chmod +x test.pl
./test.pl
```

- The other way is to call perl directly

```
perl test.pl
```

35

test.pl

```
#!/usr/bin/perl
#test.pl – should be called
#hello_world.pl

#your first perl program
print “hello everyone\n”;
```

36

Line1: Compiler/interpreter

- Perl is interpreted
- The script needs to tell the system where the interpreter is sitting
- Accomplished by special command on the first line of your program:

```
#!/usr/bin/perl
   or
#!c:\perl\bin
```

37

Line2: Comments

- Comments start the line with a hash, will continue to end of line mark, S.A

```
#your first perl program
```

38

Line3: Built in functions

- Can call tons of built in functions to do stuff in Perl
- Can define your own (more later today)
- One is the print command (such as this)

- `print “something\n”;`

39

(end of Basic Perl - preliminary)
(next – syntactic detail)

40

Basic Perl – syntactic detail

41

Technical details

- By default the start of your code is the equivalent of “main”
- Will run each line in turn, execute and then next line
- Will end when reach end of code

42

- Whitespace
  - only needed to separate terms
  - all whitespace (spaces, tabs, newlines) are treated the same
  - Use them to make the code look nice, easier to look over
- Semicolons
  - every simple statement must end with one
  - except compound statements enclosed in braces (i.e., no semicolon needed after the brace)
  - except final statements within braces
  - Advice: ALWAYS use semicolons for commands

43

Strings

- Double quoted strings are *interpreted*, so you can have a scalar in it and it will be translated

```
"hello: $name\n";
```

- Use a period to combine strings

```
"hello" . $name . "\n";
```

- Single quote strings are not interpreted, they are read *literally*

44

Something New

- Most languages you know already, variables need to be declared ahead of time, what type they will deal with.
- By default **Perl, will try to figure out what you mean.**
- Which means as soon as you use a variable for the first time, Perl will assign it a type then
- So initialization and declaration/assignment happen at once

45

Variables

- Variables
  - **Data dependent** (examples follow)
  - No space in name
  - names consist of letters, digits, underscores; up to 255 chars
  - CASE SENSITIVE
  - Should start with letter or underscore
  - Initialized variables have the value of **`undef`**
    - Can use it later to test if a variable has been used/assigned

46

Data types

- The basic data types are as follows, we will go through each in turn

- scalars ($)
- arrays (@)
- hashes (%)
- subroutine(&)
- typeglob(*)

47

Scalars

- This type of variable starts with a '$'
  - `$first`
  - `$course`
- Can hold: int, real, string
  - 234
  - -89
  - 36.34
  - “hello world”

48

- Context dependent (can take any type, so to speak)

```
$name = "shlomo";
#perl sees this is a string
$n = 123;
#perl sees this as a number
```

49

Arrays

- Starts with @
- Order list of scalars

```
@class3157 =
 ("shlomo","weijen","edward");
```

- The scaler type in the array can be anything a scalar can hold
- So can mix numbers and strings etc

50

- To reference elements, use the variable name with a dollar in front and subscript

```
$class3157[0]; #is shlomo
$class3157[i]; #in general
```

- Since perl tries to be useful what do you think this should be? `$class3157[-1];`

```
$class3157[14];
```

51

- Can get the length easily by :

```
$a = @class3157; or
print @class3157
```

- Referencing an array through a scaler

```
$ref = \@class3157;
```

- De-refrencing an array

```
$$ref[0]
```

  - This can be done with any perl type
  - Will print **ARRAY(0x18328cc)** when printing a referenced array

52

When to use reference?

- Why would you want to use a reference to an array ?
- Say you want to pass in 2 arrays into functions, you will need to reference the arrays…..
- More of this later

53

Anonymous Arrays

- Really cool Perl feature:
  - Can create the equivalent of an anonymous array by simply putting parenthesis around scalars, s.a;

```
return($max,$average);
```

- Another example;

```
($first,$second)= @number
```

  - This will grab the first and second scalars from the number array

54

Hashes

- Very useful built in type.
- A set of name/value pairs.
- Think it as a data type like array, but with string for index instead of int.
- Start with %. Curly braces to access elements.

55

- name/values pairs can be defined in one of two ways:

```
%phonelist = {adam=>718, barry=>345};
```

or

```
%phonelist = {“adam”,718,”barry”,345};
```

- Use the name to find the value

```
$phonelist{“adam”}  #is 718
```

56

recall email problem

- So how can we use hashes to keep track of email user counts ?

57

test2.pl

```
#!/usr/bin/perl
#this is the snip of the code.

$name = getnextemail($sometext);
#What does this do?
if ($emailcount{$name})  {
      $emailcount{$name}++;
  }else{
      $emailcount{$name} = 1;
  }

#but, we should cover operator and logic structure.
```

58

Reserved variables

- there's a (long) list of global special variables... a few important ones:
  - $_ = default input and pattern-searching string, its usually the last scalar you touched
- example:

```
#!/usr/bin/perl
@b = (2,4,6,8);
foreach (@b){
      print $_,"\n";
} #hold on to the question on foreach
```

59

- $/
  - input record separator (default is newline)
- $$
  - process id of the perl process running the script
- $<
  - real user id of the process running the script
- $0
  - (0=zero) name of the perl script

60

- @ARGV
  - list of command-line arguments
- %ENV
  - hash containing current environment

(below are filehandles - we will learn about them soon)

- STDIN
  - standard input
- STDOUT
  - standard output
- STDERR
  - standard error

61

Operators

- unary:
  1. ! : logical negation
  2. - : arithmetic negation
  3. ~ : bitwise negation
- arithmetic
  1. +,-,*,/,% : as you would expect
  2. ** : exponentiation
- relational
  1. >, <=, <=, <= : as you would expect

62

- equality
  1. ==, != : as you would expect
  2. <=> : comparison, with signed result:
  3. returns -1 if the left operand is less than the right;
  4. returns 0 if they are equal;
  5. returns +1 if the left operand is greater than the right
- assignment, increment, decrement
  1. =
  2. +=, ++
  3. -=, --
  4. *=, **=, /=, %=
  5. &&=, ||=

  ...Just Like C/C++/Java

63

Next

- Now that we covered the basic types of Perl, lets start to get to the logic/rules of the code

64

Statements

- simple statements are expressions that get evaluated
- they end with a semicolon (;)
- a sequence of statements can be contained in a block, delimited by braces ({ and })
- the last statement in a block does not need a semicolon
- blocks can be given labels:

```
myblock: {
   print "hello class\n";
}
```

65

Conditional Statements

- simple if

  if (expression) {block} else {block}
- unless

  unless (expression) {block} else {block}
- compound if

  if (expression1) {block}

  elsif (expression2) {block}

  ...

  elsif (expressionN) {block}

  else {block}

66

Loops

- while

  while (expression) {block}

- for

  for (expression1; expression2; expression3) {block}

- **foreach**

  foreach var (list) {block}

67

while

Syntax:

while (expression) {block}

Example:

```
#!/usr/bin/perl
@b = (2,4,6,8);
$a = @b;
$i=0;
while ( $i < $a ) {
   print "i=",$i," b[i]=",$b[$i],"\n";
   $i++;
}
```

68

for

Syntax:

for (expression1; expression2; expression3) {block}

Example:

```
#!/usr/bin/perl
@b = (2,4,6,8);
$a = @b;
for ( $i=0; $i<$a; $i++ ) {
   print "i=",$i," b[i]=",$b[$i],"\n";
}
```

69

foreach

The 'foreach' statement allows you to quickly cycle through array values

Syntax:

foreach var (list) {block}

Example:

```
#!/usr/bin/perl
@b = (2,4,6,8);
$a = @b;
foreach $e (@b) {
   print "e=",$e,"\n";
}
```

70

Question

- So if foreach allows you to cycle through arrays...
- How would you cycle through hash, since its composed of key->value pairs ?

71

test3.pl

```
#!/usr/bin/perl
# this is a snip of test3.pl

foreach $k (keys $emailcount) {
  print “$k = “
  print $emailcount{$k};
  print “\n”;
}
```

72

keys

- Built in command (or function)
- Allows you to fetch all the keys of the hash type
- Use each one to access the individual value pair

73

Side note

- Side note - To look up Perl command can use the perldoc command, do:

```
perldoc –f keys
```

- which would output (something like):

```
keys HASH
Returns a list consisting of all the keys of the
named hash. (In scalar context, returns the number
of keys.) The keys are returned in an apparently
random order. The actual random order is subject
to change in future versions of perl, but it is
guaranteed to be the same order as either the
"values" or "each“ function produces (given that
the hash has not been modified). ...
```

74

Modifiers

- Nifty grammar, but should be careful about assuming a line will execute, read it carefully. (There are many way to say same thing...)
- You can follow a simple statement by an `if, unless, while,` or `until` modifier:

  statement ***if*** expression;

  statement ***unless*** expression;

  statement ***while*** expression;

  statement ***until*** expression;

75

Examples

- Examples:

```
#!/usr/bin/perl
@b = (2,4,6,8);
$a = @b;

print "hello world!\n" if ($a < 10);
print "hello world!\n" unless ($a < 10);
#print "hello world!\n" while ($a < 10);
#print "hello world!\n" until ($a < 10);
```

76

Controlling loops

- `next`

  within a loop allows you to skip the current loop iteration
- `last`

  allows you to end the loop
- Just like continue/break statement from C/C++/Java

77

(end of Basic Perl – syntactical details)

(next – More Perl – Subroutine, File I/O, RE)

(take a break, if have not. collect survey)

78

More Perl – Subroutine, File I/O, RE

79

Ok so far

- We have variable types, arithmetic operators, and some logic
- More interesting parts. Starting with writing your own functions, which Perl calls ***subroutines***
- Talk of scope before going into subroutine...

80

Scope

- What is scope?
- Default scope is main
- $name can also be referred to as

  $main::name
- package NAMESPACE

  Within any block of code, can declare that the rest of the code will belong to a specific namespace

81

Variables and Modifiers

- Modifiers allow you to differentiate between variable types.
- Local (current) - variable exists within the local (current) section (scope), either main or a subroutine or code block (between braces):

```
my $name
```

- Global – can be seen from other section of the code:

```
our $name
local $name
```

82

Scope: usage

- my $time, $out;
  - here, only time is a local variables
- my ($time,$out)
  - correct way to do it.

- Remember to place more than one variable in parenthesis!!

83

Subroutine

- syntax for defining:

  sub name {block}

  sub name (proto) {block}

- where proto is like a prototype, where you put in sample arguments

- syntax for calling from your code:

  name(arg list);

  name arglist;

(the & sign used to be required when calling functions, this has been changed in the latest perl : it is optional if you use parenthesis in the method call)

84

- Usually the sub definitions are placed at the bottom of the file
- No reason, but makes code easier to read

85

sub1.pl

```
print “welcome to the program”;
testsub();

sub testsub(){
  print “hi everyone\n”;
}
```

86

Passing arguments

- All arguments to a subroutine come in on the @_ array
- That is argument list is packaged as an array and can be accessed in the sub

87

- 3 different ways to grab arguments:

```
$a = @_[0];
($a) = @_;
$a = shift;
```

- Shift moves out the top of an array, if we don't specify it (and its first) we are talking about the default array (nice and confusing).

88

Passing by values

- By default **pass by value**
- Which means a copy is sent in
- Any changed will be only local to subroutine

89

So…what will be printed here?

```
$n = 45;
print "n is now $n\n";
testsub($n);
print "n is now $n\n";

sub testsub{
   $a = shift;
   print "in testsub 1 = $a\n";
   $a++;
   print "in testsub 2 = $a\n";
}
```

90

Pass by reference

- To have variables changed, pass the reference to the scalar to the sub…i.e. you are really manipulating the original one
- Use a backslash to say it's a scalar
- Use extra dollar to say dereference

91

Pass by reference

```
$n = 45;

print "n is now $n\n";
testsub(\$n);
print "n is now $n\n";

sub testsub{
    $a = shift;
    print "in testsub $a\n";
    $$a++;
}
```

92

Different subs

- Perl will make a best effort to both figure out what the variables mean and which functions you are trying to call

93

Example

```
#!c:\perl\bin
#this is sub4.pl snip

($first,$last) = &getname();
print "First is $first";

#return the fill name as a string
sub getname(){
   return "shlomo hershkop";
}
#return name split
sub getname(){
    return ("shlomo","hershkop");
}
```

94

(end Subroutine)
(next – File I/O)

95

Working with files

- Many things which are complicated in other language become super easy in Perl. **File I/O with Perl is nice and easy.**
- To go through a file 3 step process:
  - Open
  - Read / Write
  - Close

96

Open files

- When you open, you need to say what type of operations you will be doing:

```
open( FILEHANDLE, filename );  # to open a file for read in
open( FILEHANDLE, >filename ); # to open a file for writing
open( FILEHANDLE, >>filename );  # to open a file for
   appending
```

97

Print to files

- Once you've opened the file, can send something to it by putting it after the print command:

  ```
  print FILEHANDLE, "….";
  ```

- The filehandle is the variable you specified with the open command
- Don't forget to close the file when done writing

98

Example:

```
#!/usr/bin/perl
#this is the snip from fh1.pl
open( MYFILE,">a.dat" );
#for example of s.c, see the next
print MYFILE "hi there!\n";
print MYFILE "bye-bye\n";
close( MYFILE );
```

99

Read form file

- Once you open a file handle, can get stuff from it using the *pointy brackets*, like this:

  <MYFILE>;

- This basically reads a line of text from the file
- Since destination isn't specified its read to the $_
- You can also read in to a variable.

  $line = <MYFILE>;

100

Errors handling

- So what if the open command fails?
- A trick - use double pipe to do something like this;

  ... || warn print "message";
- Or if you want to fail:

  ... || die print "message";

101

Example II

```
#!/usr/bin/perl
open( MYFILE1,“a.dat" ) || warn "file 1
  not found!";
open( MYFILE2,“>b.dat" ) || die "file 2
  problems!";
while ( <MYFILE1> ) {
     print MYFILE2 "$_\n"
   }

   close( MYFILE2 );
```

102

Take a second to look this over

```
#!/usr/bin/perl

open( TEST,“test.txt" ) || die “can not
  open file!\n”;
$linecount =0;
while ($line = <TEST>){
  $linecount++;
}
close( MYFILE );
print “number of lines in the file:
  $linecount\n”;
```

103

Not only read line

- Can also read individual bytes from files (useful in low level graphic manipulation)
- *getc* FILEHANDLE

reads next byte from filehandle

104

chomp

- Not lunch! Another built-in function
- Removes \n very useful when processing logs
- V. useful when you are reading line from file

105

Other Useful built-in functions (there are much much more)

- chomp $var
- chomp @list

  removes any line-ending characters
- chop $var
- chop @list

  removes last character
- chr number

  returns the character represented by the ASCII value number
- eof filehandle

  returns true if next read on filehandle will return end-of-file
- exists $hash{$key}

  returns true if specified hash key exists, even if its value is undefined
- exit

  exits the perl process immediately

106

(end File I/O)
(next – Regular Expression)

107

Ok lets switch gears

- You now have enough knowledge to code some simple but powerful programs.
- Lets do something that Perl is really good at.

108

Patterns

- Many times when you process text, you need to find ***patterns***
  - Email addresses
  - Phone numbers
  - Credit cards
- How would you look for all phone numbers in a text file using Java ? (not really pretty)

109

Regular Expressions

- Regular Expressions is an elegant way of Expressing patterns
- Use of simple building blocks to express even the most complex patterns
- Perl has the extensive support for Regular Expression

110

Match function

- Matching function in Perl

  m/ /;

- Pattern goes between slashes, m is optional
- Use the =~ operator to match against text
- Returns true if match occurs, false otherwise

```
if( $sometext =~ m/computer/){print
“matched”;}
```

this will look for any occurrence of the word 'computer' in the scalar string $sometext.

111

Simplest pattern

- Is a string
- e.g., “computer”
- ...Is the *pattern* “c followed by o followed by m ...”
- Will not match to “cosputer”

112

Example (simple)

What will this do ?

```
#!c:\perl\bin
# this is snip of re1.pl
$name = "shlomo hershkop";

if($name =~ /lom/){
  print "have found match\n";
}
else{
  print "no match found\n";
}
```

113

Example 2

```
#!/usr/bin/perl

$s = "hello world";
print '$s=[',$s,"]\n";
$t = ($s =~ s/l/x/g);
print '$t=[',$t,"]\n";
print '$s=[',$s,"]\n";

# this outputs:
# $s=[hello world]
# $t=[3]
# $s=[hexxo worxd]
```

114

Flip patterns

- Can flip the match by saying:
  !~/ /
- Example:
```
$line !~ /great/
```
- Will match any line with the word 'great'

115

Regular Expression attributes

- The matching operator can be modified by following the last slash with specific characters.
  - g = match globally (all instances)
  - i = do case insensitive matching
  - e = evaluate right side as an expression
  - s = let . match newlines
  - m = $ and ^ can refer to inside newlines
  - c = compliment

116

Example 1

```
#!/usr/bin/perl
$s = "hello world";
print '$s=[',$s,"]\n";
if ($s =~ m/x/)
  { print "there's an x in ",$s,"\n" }
else
  { print "there isn't\n" }
if ($s =~ m/L/i)
  { print "there's an l in ",$s,"\n" }
else
  { print "there isn't\n" }

# this outputs:
# $s=[hello world]
# there isn't
# there's an l in hello world
```

117

Substitute function

- Instead of matching operation, can use substitute

  s/ / /;

- First space is pattern to find, second space is pattern to replace it with
- Returns number of times substituted

118

Anchors

- You can force ***where*** on the line the pattern is match by:
- ^
  - Caret. Matches at the start of the line
- $
  - Dollar sign. Mathes at the end of the line
- Example:

```
$line =~ /^credit/
     #line begins with word credit
$line =~ /bye$/
     #line ends with word bye
$line !~ /^great/i
```

119

Character choices

- [...]
- We can specify character ranges by using the square brackets:

- Examples:

```
if( $string =~ /[AEIOUY]/i )
{ print "contains a vowel!\n“; }
#Can also specify ranges
if( $string =~  /[^a-e]/I ) {
  something }
```

120

Metacharacters

- Complex regular expressions use ***metacharacters*** to describe various options in building a pattern.

- ****
  - Backslash. A character prefixed by backslash is called escape characters and have special meanings (e.g., “\n”).
- **.**
  - Period. Match any single character
  - e.g., /compu.er/ can match 'computer' and 'compuser'

121

- *
  - Asterisk. Match zero or more of the preceding character
  - /comp*uter/ will match 'computer', 'compppppputer', and 'comuter'

- +
  - Plus sign. Match 1 or more of the preceding character
  - /ab+a/ will match 'aba', 'abba', 'abbbbba', but not 'aa'

- ?
  - Question mark. Match 1 or 0 of the preceding character

122

Quantifiers

- { n1, n2 }
  - Match n1 to n2 of the preceding character
- { n, }
  - Match n or more of the preceding character
- { n }
  - Match exactly n of the preceding character

123

Examples:

```
/ba*b/;       #b,zero or more a, b
/ba{3,5}b/;   #b, 3 and 5 a's, b
/ba{2}b/;     #b, exactly 2 a's, b
/(ab){4,}/;   #4 or more ab's
/[a-h]{1,4}/; #1 to 4 character a~h
```

124

Escape shortcuts

- \w
  - Match "word" character (alphanumeric plus "_")
- \W
  - Match non-word character
- \s
  - Match whitespace character
- \S
  - Match non-whitespace character
- \d
  - Match digit character
- \D
  - Match non-digit character

125

More escape codes

- \t
  - Match tab
- \n
  - Match newline
- \r
  - Match return
- \f
  - Match formfeed
- \a
  - Match alarm (bell, beep, etc)
- \e
  - Match escape

126

Groups

- (...)
  - Things inside parenthesis are taken as "groups"
- To allow groups of alternative choices use pipe

Examples:

```
if($string =~ /(A|E|I|O|U|Y)/i)
   { print "String contains a vowel!\n"; }

if($string =~ /(Clinton|Bush)/)
  { print "President sir!\n"; }
```

127

Groups

- Perl has shortcuts to allow us to reference for selection and subsitution
- Each group can be referred to by scalar $1, $2, $3 ….
- Example:

```
$line = "From s@aol.com Wed Jun 3
  12:12:12 2005"
if($line =~ /^From (.*)(…)(…)(.*)$/)
#each match can be refer back with $1, 2,
  3, and so on.
```

128

(end of RE -followed by practice)

129

Quick question

- How to indicate the period since period matches any character?

130

Usage Example

```
if ( $line =~ /^\s.*\S$/ ) {...}

if ( not $line =~ /cs3157/ ) {...}

if( $line !~ /cs3157/ ) {...}

while ( $line =~ /^\w \w$/) {...}
```

131

What is?

```
open MAIL, “mail.txt” or die “cant open
  file\n”;

while(<MAIL>) {
  print if m/^From: /;
}
```

132

```
open MAIL, “Mail.txt” or die “can’t open
  mail file\n”;

while (<MAIL>) {
  if (/^([^:]+): ?(.+)$/ ) {
     print “Header $1 has val $2\n”;
  }
```

133

```
if($string =~
  m/^\S+\s+(Hershkop|Stolfo|Aho)/i){
   print "$string\n"
   };
```

134

- Should use pattern matches as a security check on input!

```
unless ( $year =~ /^\d\d$/) {
  die (“problem with year input!”);
}
```

135

What is?

```
$name = “advanced programming class”
if($name =~ /programming/){
   print $`  ;
   print $&  ;
   print $’  ;
}
```

136

Useful command

- *Split*

```
split /PATTERN/,EXPR,LIMIT
split /PATTERN/,EXPR
split /PATTERN/
```

split Splits a string into a list of strings and returns that list. By default, empty leading fields are preserved, and empty trailing ones are deleted. …….

137

Short check

- Can you take a second to write a short pattern on what an email looks like ?
- How would we look for a phone number using regular expressions ??
- What about a street address?

138

(end of RE practice)
(next - debugging)

139

Debugging w Perl

- **Use 'strict' pragma**
- Use 'warning' pragma
- Use 'perl -w' option
  - This output warning as it compile
- Use 'perl -c' option
  - This does syntax check
- Use debugger '-d' option
  - bring you down (up?) to debug mode.

140

Pragmas

- Perl allows you to control the interpreter
- Pragmas are compiler hints to allow you to operate in some special mode

- Two examples:
- use warning
  - This will give you ideas of what Perl thinks it is doing
- use strict
  - This will be very strict about variables, so you will need to declare each variable (my,our) before being able to use it in your program

141

Strict mode

- This isn't about the midterm
- Tells Perl to only allow variable you explicitly create in your programs
  - Prevents typos
  - Easier to maintain
  - Less work for interpreter
  - Will clearly state what it thinks you need to be doing to get things correct

142

Use strict

- Goes at top of your program:

```
use strict;
```

- For example, for loop below:

```
for( $i =0; $i < 100; $i++)  #won't work if first
  use of $i
for (my $i; $i < 100; $i++)  #will work for strict
```

- USE it for homework/lab assignment !

143

Debug mode

- 'perl -d your_code.pl'
- Essential command to know:
  - 'q' for exit from debug more
  - 'h' for help (give you the list of commands)
  - 'v' view where in the code you are
  - 'v [line]' for viewing around [line]
  - 's' for step over. execute code line by line
  - 'n' for step over (skip subroutines)
  - 'c [line]' continue to the [line]

144

Good advice

- Learn to read errors and warnings
- It will tell you what the problem is and what line it thinks its on
- Do not ignore, ask for help if you need it

145

Useful Unix commands

- ls -la
- chmod
- man
- uname –a
- pwd
- who
- finger
- cd
- mkdir
- locate
- which

146

Perl References

- there are lots and lots of advanced and funky things you can do in perl; this is just a start!

here's a quick start reference:
- http://www.comp.leeds.ac.uk/Perl/
- http://www.perl.com

function reference list is here:
- http://www.perldoc.com/perl5.6/pod/perlfunc.html

147

Perldoc

- Use 'perldoc', or 'man'
- Below are some of the helpfulones:
  - perlre (1) - about Regular Expression
  - perlvar (1) - predefined variables explained here
  - perlrun (1) – about command line option
  - perlop (1) – about operators
  - perlfunc (1) – about built in functions
  - perldebtut (1) – debugger help

148