

# Perl Notes

---

The long Perl handout (Handout #6) by Nick Parlante has the full explanations - these are just the notes for the Perl section. See <http://cslibrary.stanford.edu/108/>.

While you are learning Perl, try running it in interactive mode by running `/usr/class/cs193i/bin/interactive.pl` and cutting and pasting snippets of Perl onto the resulting command prompt. To run a statement that you have run before, type ESC then enter to retrieve a previous statement. If this was not the statement you were looking for, keep typing ESC then Enter until you find it. Then, press Enter to evaluate the statement. Currently, you cannot edit a previous statement. `interactive.pl` is experimental, and is not guaranteed to work all the time. However, Perl normally has no interactive mode, so this tool is good to learn with.

## What is Perl?

Language Invented by Larry Wall, very useful for text processing

Free - open source

Portable - no vendor lock-in

Quick-n-dirty - lack of compile-time type checking

Low-road language (Perl) vs. High-road language (Java)

[www.cpan.org](http://www.cpan.org)

Later on, once you become more familiar with Perl, you may want to learn and compare it with Python (<http://www.python.org/>), which is a similar programming language that has many other benefits. But, for simplicity (and historical reasons) we use Perl in this class.

## Perl Program

Interpreted - program is a text file

Portable - Unix, Mac OS X, Windows...

```
#!/usr/bin/perl -w
print "Hello world!\n";
```

## Scalar Variables

Scalar: a number or string

Syntax: start with a \$

Do not need to pre-declare variables

```
$x = 42;
```

```
$y = $x + 13;
```

```
$z = "hello";
```

**undef** and **defined()**

Variables are the value **undef** at first, use the **defined()** test  
if (defined(\$z)) { ...

## Strings

Double quotes (") or single quotes (')

```
"Hello\nThere\n"    ## string value, with newlines in it
```

'Hello' - Or use single quotes, but then \n's and interpolation are suppressed.

Concat with period (.)

```
$str = $z . " twice " . $z;    ## concat
```

Interpolation

Variables are evaluated and pasted into a quoted string...

```
$x = 42;
```

```
$str = "The answer is $x, so there!\n";    ## the "42" gets pasted in
```

String comparison

```
("hello" eq $str) ## case sensitive string equals test
```

```
("hello" ne $str) ## case sensitive string not equals test
```

Do not use the == comparison with strings, == is for numbers

```
lc(), uc()    ## upper/lower case fns
```

## if / while/ for

()'s required for test

{ }'s required for statements -- note: not like C

if

```
if (test) { ... }
```

```
if (test) {...} else {...}
```

```
if (test) {...} elsif (test2) { ...}
```

while

```
while(test) {...}
```

last - break out of the loop (like break in C)

next - jump to the next iteration (like continue in C)

for loop - like C

## Arrays

Features

All purpose array/list type container built in to language

Arrays know their length

Can change size dynamically

Can contain different types of elements (strings, ints, ... at the same time)

```
@a = (1, 2);
```

```
$x = $a[1];    ## look at the [1] element
```

```
$a[0] = 13;    ## change the [0] element
```

shift / unshift - operate on "front" or [0] of the array

```
shift(@a);    ## return and remove [0] element
```

```
unshift(@a, "hello");    ## insert element at front
push / pop - operate on "end or [len-1] of the array
pop(@a);
push(@a, "hello");
```

@ARGV

Special global array, contains the command line arguments

foreach

```
## iterate $elem over the array
```

```
foreach $elem (@a) {...< code that uses $elem> }
```

sort (@a)

Return a sorted copy of the array

## Hash Arrays

Like a hash table - %dict

Use { } to access by key

Assign

```
$dict{"a"} = "aaa";
```

```
$dict{"b"} = "bbb";
```

Lookup

```
$dict{"a"}          ## lookup value for key "a"
```

```
$dict{"b"}
```

```
defined($dict{"z"}) ==> false, since no value for "z"
```

Array Conversion

A hash can be converted to an array of even length with keys in random order

```
@a = %dict;
```

```
Now, @a is ("b", "bbb", "a", "bbb")
```

%ENV

Special global hash array, contains the environment vars of our runtime

## Subroutines

```
sub foo {
  my($x, $y) = @_;    ## parameters
  my($z);    ## local var
  return($x + $y);
}
```

```
foo(6, 7) ==> 13
```

@\_

@\_ is an array of the passed scalar values. Use '=' to make a parallel assignment into local variables (values must be scalars).

Flattening

Cannot pass an array as one of the args - it all gets "flattened" into @\_

my

```
my() declares something local
```

Use strict - \$::global

Refer to outer global vars as `$$:global` - needed if the "use strict;" or "use strict 'vars';" option is on. Use strict is a good idea for a program more than a page or two long.

## Perl Global Variables

Perl programs will use subroutine parameters where it makes sense, and will leave some things as global variables.

In C or other languages, global variables are regarded as questionable style, but in Perl they are more natural.

This is perhaps more evidence that Perl is not appropriate for large programs.

## File Processing

File Handle

Standards: STDIN, STDOUT, STDERR

Custom file handles are by convention all upper-case: FILE, FILE\_IN, FILE\_OUT

<FILE>

read one line from an input file handle, including the `\n`

Print to file handle

```
print FILE $x, "hello", $y;    ## write to previously opened FILE
print "Hello there\n";        ## writes to STDOUT by default
```

Write to an output file handle. The things to write are separated by commas, but the file-handle is not. File handle defaults to STDOUT

Open for reading

```
open(IN_FILE, "poem.txt");
```

Open for writing

```
open(OUT_FILE, ">poem2.txt");
```

Standard file processing loop...

```
while ($line = <IN_FILE>) {
    chomp($line);    ## remove EOLN
    print OUT_FILE $line, "\n";
}
```

die - error detection - open() returns undef on error

```
open(F, $fname) || die "Cannot open $fname";
```

## Regular Expressions

`$str =~ /pattern/`

`("binky" =~ /ink/) ==> TRUE`

`("binky" =~ /onk/) ==> FALSE`

Match anywhere in string

Do not need to "use up" the whole string

Must use up all of the pattern

`$str` interpolation works in regular expressions, so can say `($a =~ /$b/)`

## Character Codes

Plain characters like 'a', 'X', '9' – just match exactly

. (a period) – matches any single character except "\n"

\w -- matches a "word" character: a letter or digit [a-zA-Z0-9]

\W -- (upper-case W) any non word character

\s -- matches a single whitespace character – space, newline, return, tab, form [  
\n\r\t\f]

\S -- (upper-case S) any non whitespace character

\t, \n, \d -- tab, newline, decimal digit [0-9]

\ -- inhibit the "specialness" of a character. So, for example, use \. to match a period or \\ to match a slash

/ Variants

"i" after the last / makes it case-insensitive

"m" before the first slash lets you use a delimiter such as "

Examples

"piiig" =~ /p...g/ ==> TRUE . = any char

"piiig" =~ /ii/ ==> TRUE you do not need to use up the whole string

"piiig" =~ /p...g/ ==> FALSE the whole pattern must match

"piiig" =~ /p\w\w\wg/ ==> TRUE \w = any letter or digit

"p123g" =~ /p\d\d\dg/ ==> TRUE \d = 0..9 digit

"piiig" =~ /pIiG/i ==> TRUE, "i" at end makes it case insensitive

"piiig" =~ m"piiig" ==> TRUE "m" allows a delimiter other than /

## Control Codes

\* -- 0 or more occurrences of the pattern to its left

+ -- 1 or more occurrences of the pattern to its left

| -- logical or -- matches either the pattern on its left or right

parenthesis () -- group sequences of patterns

^ -- matches the start of the string

\$ -- matches the end of the string

Leftmost & Largest

First, Perl tries to find the leftmost match for the pattern, and second it tries to use up as much of the string as possible – i.e. let + and \* use up as many characters as possible.

## Regular Expression Examples

These are just lifted from the Perl handout.

These examples gradually demonstrate each of the above control codes. Study them carefully – small details in regular expressions make a big difference. That's what makes them powerful, but it makes them tricky as well...

```
#### Search for the pattern 'iiig' in the string 'piiiig'
"piiiig" =~ m/iiig/ ==> TRUE

#### The pattern may be anywhere inside the string
"piiiig" =~ m/iii/ ==> TRUE

#### All of the pattern must match
"piiiig" =~ m/iiii/ ==> FALSE

#### . = any char but \n
"piiiig" =~ m/...ig/ ==> TRUE

"piiiig" =~ m/p.i../ ==> TRUE

#### The last . in the pattern is not matched
"piiiig" =~ m/p.i.../ ==> FALSE

#### \d = digit [0-9]
"p123g" =~ m/p\d\d\dg/ ==> TRUE

"p123g" =~ m/p\d\d\d\d/ ==> FALSE

#### \w = letter or digit
"p123g" =~ m/\w\w\w\w\w/ ==> TRUE

#### i+ = one or more i's
"piiiig" =~ m/pi+g/ ==> TRUE

#### matches iii
"piiiig" =~ m/i+/ ==> TRUE

"piiiig" =~ m/p+i+g+/ ==> TRUE

"piiiig" =~ m/p+g+/ ==> FALSE

#### i* = zero or more i's
"piiiig" =~ m/pi*g/ ==> TRUE

"piiiig" =~ m/p*i*g*/ ==> TRUE

#### X* can match zero X's
"piiiig" =~ m/pi*X*g/ ==> TRUE

#### ^ = start, $ = end
"piiiig" =~ m/^pi+g$/ ==> TRUE

#### i is not at the start
"piiiig" =~ m/^i+g$/ ==> FALSE

#### i is not at the end
"piiiig" =~ m/^pi+$/ ==> FALSE

"piiiig" =~ m/^p.+g$/ ==> TRUE

"piiiig" =~ m/^p.+$/ ==> TRUE

"piiiig" =~ m/^..+$/ ==> TRUE

#### g is not at the start
"piiiig" =~ m/^g.+$/ ==> FALSE

#### Needs at least one char after the g
"piiiig" =~ m/g.+/ ==> FALSE
```

```
#### Needs at least zero chars after the g
"piiiig" =~ m/g.* / ==> TRUE

#### | = left or right expression
"cat" =~ m/^(cat|hat)$/ ==> TRUE

"hat" =~ m/^(cat|hat)$/ ==> TRUE

"cathatcatcat" =~ m/^(cat|hat)+$/ ==> TRUE

"cathatcatcat" =~ m/^(c|a|t|h)+$/ ==> TRUE

"cathatcatcat" =~ m/^(c|a|t)+$/ ==> FALSE

#### Matches and stops at first 'cat'; does not get to 'catcat' on the right
"cathatcatcat" =~ m/(c|a|t)+/ ==> TRUE

#### ? = optional
"12121x2121x2" =~ m/^(1x?2)+$/ ==> TRUE

"aaaxbbbabaxbb" =~ m/^(a+x?b)+$/ ==> TRUE

"aaaxxbbb" =~ m/^(a+x?b)+$/ ==> FALSE

#### Three words separated by spaces
"Easy      does it" =~ m/^\w+\s+\w+\s+\w+$/ ==> TRUE

#### Just matches "gates@microsoft" -- \w does not match the "."
"bill.gates@microsoft.com" =~ m/\w+@\w+/ ==> TRUE

#### Add the .'s to get the whole thing
"bill.gates@microsoft.com" =~ m/^(\\w|\\.)+@(\\w|\\.)+$/ ==> TRUE

#### words separated by commas and possibly spaces
"Klaatu,  barada,nikto" =~ m/^\w+(,\\s*\w+)*$/ ==> TRUE
```

## Character Classes

[aeiou] -- a vowel  
[a-zA-Z0-9] -- a word char  
[\w] -- same as above  
[^aeiou] -- anything but a vowel  
/([\w\_\.]+)@([\w\_\.]+)/ -- match an email addr

## Special \$X Match Variables

if (\$str =~ m/(\w+)\s+(\w+)/) {...}

If the if statement is true...

\$1 = chars from first ( )

\$2 = chars from second ( )

Use to extract things from the string

\$` (back-quote) - before the match

\$& the match

\$' - after the match

## Special Example

Extract email addr

```
while ($str =~ /(([\w._-]+\)[\w._-]+))/ { ## look for an email addr
    print "user:$2 host:$3 all:$1\n";    ## parts of the addr
    $str = $';    ## set the str to be the "rest" of the string
}
```

## ? Trick

Flawed way

```
m/{(.*)}/ -- pick up all the characters between {}'s -- goes too far
```

Old, doofy ^ way

```
m/{([^\^]*)}/ ## the inner [^\^] matches any char except }
```

New, hip ? way

```
m/{(.?*)}/ ## pick up all the characters between {}'s, but stop
## at the first }
```

## Substitution

Change all "is" strings to "is not" -- a sure way to improve any document

```
$str =~ s/is/is not/ig;
```

"i" = not case sensitive

"g" = do repeatedly