Beginning Perl Scripting

Simple scripts, Expressions, Operators, Statements, Variables

Simon Prochnik & Lincoln Stein

Suggested Reading

Learning Perl (6th ed.): Chap. 2, 3, 12,
Unix & Perl to the Rescue (1st ed.): Chap. 4 Chapters 1, 2 & 5 of *Learning Perl*.

Lecture Notes

Problems

What is Perl?

Perl is a Programming Language

Written by Larry Wall in late 80's to process mail on Unix systems and since extended by a huge cast of characters. The name is said to stand for:

1. Pathologically Eclectic Rubbish Lister
2. Practical Extraction and Report Language

Perl Properties

1. Interpreted Language
2. "Object-Oriented"
3. Cross-platform
4. Forgiving
5. Great for text
6. Extensible, rich set of libraries
7. Popular for web pages
8. Extremely popular for bioinformatics

Other Languages Used in Bioinformatics

C, C++

Compiled languages, hence very fast.
Used for computation (BLAST, FASTA, Phred, Phrap, ClustalW)
Not very forgiving.

Java

Interpreted, fully object-oriented language.
Built into web browsers.
Supposed to be cross-platform, getting better.

Python , Ruby

Interpreted, fully object-oriented language.
Rich set of libraries.
Elegant syntax.
Smaller user community than Java or Perl.

Some Simple Scripts

Here are some simple scripts to illustrate the "look" of a Perl program.

Print a Message to the Terminal

Code:

```
#!/usr/bin/perl
# file: message.pl
use strict;
use warnings;
print "When that Aprill with his shoures soote\n";
print "The droghte of March ath perced to the roote,\n";
print "And bathed every veyne in swich licour\n";
print "Of which vertu engendered is the flour...\n";
```

Output:

```
(~) 50% perl message.pl
When that Aprill with his shoures soote
The droghte of March ath perced to the roote,
And bathed every veyne in swich licour
Of which vertu engendered is the flour...
```

Do Some Math

Code:

```
#!/usr/bin/perl
# file: math.pl
use strict;
use warnings;
print "2 + 2 =", 2+2, "\n";
print "log(1e23)= ", log(1e23), "\n";
print "2 * sin(3.1414)= ", 2 * sin(3.1414), "\n";
```

Output:

```
(~) 51% perl math.pl
2 + 2 =4
log(1e23)= 52.9594571388631
2 * sin(3.1414)= 0.000385307177203065
```

Run a System Command

Code:

```
#!/usr/bin/perl
# file: system.pl
use strict;
use warnings;
system "ls";
```

Output:

```
(~/docs/grad_course/perl) 52% perl system.pl
index.html          math.pl~            problem_set.html~   what_is_perl.h
index.html~         message.pl          simple.html         what_is_perl.h
math.pl             problem_set.html    simple.html~
```

Return the Time of Day

Code:

```
#!/usr/bin/perl
# file: time.pl
use strict;
use warnings;
$time = localtime;
print "The time is now $time\n";
```

Output:

```
(~) 53% perl time.pl
The time is now Thu Sep 16 17:30:02 1999
```

Mechanics of Writing Perl Scripts

Some hints to help you get going.

Creating the Script

A Perl script is just a text file. Use any text (programmer's) editor. *Don't use word processors like Word.*

By convention, Perl script files end with the extension *.pl*.

I suggest Emacs, because it is already installed on almost all Unix machines, but there are many good options: vi, vim, Textwrangler, eclipse

The Emacs text editor has a *Perl mode* that will auto-format your Perl scripts and highlight keywords. Perl mode will be activated automatically if you end the script name with **.pl**.

GUI-based script writing tools (Aquamacs, xemacs, Textwrangler, Eclipse) are easier to use, but you may have to install them yourself.

Let's write a simple perl script. It'll be a simple text file called time.pl and will contain the lines above.

Let's try doing this in emacs

Emacs Essentials

A GUI version is simpler to use e.g. Aquamacs, run it by adding the icon for the application to your Dock then clicking on the icon. You can also run emacs in a Terminal window. Emacs will be installed on almost every Unix system you encounter.

```
(~) 50% emacs
```

The same shortcuts you can use on the command line work in Emacs
e.g.

control-a (^a)
    move cursor to beginning of line etc

The most important Emacs-specific commands

control-x control-f (^x ^f)
    open a file
control-x control-w (^x ^w)
    save as...
control-x control-c (^x ^c)
    quit
control-g (^g)
    cancel command
shift-control-_ (^_)
    Undo typing
control-h ?(^h ?)
    Help!!
option-; (M;)
    Add comment
option-/ (M/)
    Variable/subroutine name auto-completion (cycles through options)

Running the Script

Don't forget to save any changes in your script before running it. The filled red circle at the top left of the emacs GUI window has a dot in it if there are unsaved changes.

Option 1 (quick, not used much)
    Run the **perl** program from the command line, giving it the name of the script file to run.

```
(~) 50% perl time.pl
The time is now Thu Sep 16 18:09:28 1999
```

Option 2 (as shown in examples above)

Put the magic comment

```
#!/usr/bin/perl
```

at the top of your script.

It's really easy to make a mistake with this complicated line and this causes confusing errors (see below). Double check, or copy from a friend who has it working.

And always add

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

to the top of your script like in the example below

```
#!/usr/bin/perl
# file: time.pl
use strict;
use warnings;
$time = localtime;
print "The time is now $time\n";
```

Now make the script executable with *chmod +x time.pl*:

```
(~) 51% chmod +x time.pl
```

Run the script as if it were a command:

```
(~) 52% ./time.pl
The time is now Thu Sep 16 18:12:13 1999
```

Note that you have to type "./time.pl" rather than "time.pl" because, by default, **bash** does not search the current directory for commands to execute. To avoid this, you can add the current directory (".") to your search PATH environment variable. To do this, create a file in your home directory named `.profile` and enter the following line in it:

```
export PATH=$PATH:.
```

The next time you log in, your path will contain the current directory and you can type "time.pl" directly.

Common Errors

Plan out your script before you start coding. Write the code, then run it to see if it works. Every script goes through a few iterations before you get it right. Here are some common errors:

Syntax Errors

Code:

```
#!/usr/bin/perl
# file: time.pl
use strict;
use warnings;
time = localtime;
print "The time is now $time\n";
```

Output:

```
(~) 53% time.pl
Can't modify time in scalar assignment at time.pl line 3, near "localtime;
Execution of time.pl aborted due to compilation errors.
```

Runtime Errors

Code:

```
#!/usr/bin/perl
# file: math.pl
use strict;
use warnings;

$six_of_one = 6;
$half_dozen = 12/2;
$result = $six_of_one/($half_dozen - $six_of_one);
print "The result is $result\n";
```

Output:

```
(~) 54% math.pl
Illegal division by zero at math.pl line 6.
```

Forgetting to Make the Script Executable

```
(~) 55% test.pl
test.pl: Permission denied.
```

Getting the Path to Perl Wrong on the #! line

Code:

```
#!/usr/local/bin/pearl
# file: time.pl
use strict;
use warnings;
$time = localtime;
print "The time is now $time\n";
```

```
(~) 55% time.pl
```

```
    time.pl: Command not found.
```

This gives a very confusing error message because the command that wasn't found is 'pearl' not time.pl

Useful Perl Command-Line Options

You can call Perl with a few command-line options to help catch errors:

-c
: Perform a syntax check, but don't run.

-w
: Turn on verbose warnings. Same as

```
    use warnings;
```

-d
: Turn on the Perl debugger.

Usually you will invoke these from the command-line, as in *perl -cw time.pl* (syntax check *time.pl* with verbose warnings). You can also put them in the top line: *#!/usr/bin/perl -w*.

Perl Statements

A Perl script consists of a series of *statements* and *comments*. Each statement is a command that is recognized by the Perl interpreter and executed. Statements are terminated by the semicolon character (;). They are also usually separated by a newline character to enhance readability.

A *comment* begins with the # sign and can appear anywhere. Everything from the # to the end of the line is ignored by the Perl interpreter. Commonly used for human-readable notes. Use comments plentifully, especially at the beginning of a script to describe what it does, at the beginning of each section of your code and for any complex code.

Some Statements

```
$sum = 2 + 2; # this is a statement

$f = <STDIN>; $g = $f++;  # these are two statements

$g = $f
  /
  $sum;        # this is one statement, spread across 3 lines
```

The Perl interpreter will start at the top of the script and execute all the statements, in order from top to bottom, until it reaches the end of the script. This execution order can be modified by loops and control structures.

Blocks

It is common to group statements into *blocks* using curly braces. You can execute the entire block conditionally, or turn it into a *subroutine* that can be called from many different places.

Example blocks:

```
{  # block starts
```

```
  my $EcoRI = 'GAATTC';
  my $sequence = <STDIN>;
  print "Sequence contains an EcoRI site" if $sequence=~/$EcoRI/;
}  # block ends

my $sequence2 = <STDIN>;
if (length($sequence) < 100) {  # another block starts
  print "Sequence is too small. Throw it back\n";
  exit 0;
} # and ends

foreach $sequence (@sequences) {  # another block
  print "sequence length = ",length($sequence),"\n";
}
```

Literals

A *literal* is a constant value that you embed directly in the program code. You can think of the value as being *literally* in the code. Perl supports both *string literals* and *numeric literals*. A string literal or a numeric literal is a *scalar* i.e. a single value.

Literals cannot be changed. If you want to change the value of some data, it needs to be a *variable*. Much, much more on this coming up, until you're really sick of the whole thing.

String Literals

String literals are enclosed by single quotes (') or double quotes ("):

```
'The quality of mercy is not strained.';  # a single-quoted string
"The quality of mercy is not strained.";  # a double-quoted string
```

The difference between single and double-quoted strings is that variables and certain special escape codes are interpolated into double quoted strings, but not in single-quoted ones. Here are some escape codes:

\n	New line
\t	Tab
\r	Carriage return
\f	Form feed
\a	Ring bell
\040	Octal character (octal 040 is the space character)
\0x2a	Hexadecimal character (hex 2A is the "*" character)
\cA	Control character (This is the ^A character)
\u	Uppercase next character

\l	Lowercase next character
\U	Uppercase everything until \E
\L	Lowercase everything until \E
\Q	Quote non-word characters until \E
\E	End \U, \L or \Q operation

```
"Here goes\n\tnothing!";
  # evaluates to:
  # Here goes
  #     nothing!

'Here goes\n\tnothing!';
  # evaluates to:
  # Here goes\n\tnothing!

"Here goes \unothing!";
  # evaluates to:
  # Here goes Nothing!

"Here \Ugoes nothing\E";
  # evaluates to:
  # Here GOES NOTHING!

"Alert! \a\a\a";
  # evaluates to:
  # Alert! (ding! ding! ding!)
```

Putting backslashes in strings is a problem because they get interpreted as escape sequences. To inclue a literal backslash in a string, double it:

```
"My file is in C:\\Program  Files\\Accessories\\wordpad.exe";

  # evaluates to: C:\Program Files\Accessories\wordpad.exe
```

Put a backslash in front of a quote character in order to make the quote character part of the string:

```
"She cried \"Oh dear! The parakeet has flown the coop!\"";

  # evaluates to: She cried "Oh dear! The parakeet has flown the coop!"
```

Numeric Literals

You can refer to numeric values using integers, floating point numbers, scientific notation, hexadecimal notation, and octal. With some help from the Math::Complex module, you can refer to complex numbers as well:

```
123;       # an integer
1.23;      # a floating point number
-1.23;     # a negative floating point number
1_000_000; # you can use _ to improve readability
1.23E45;   # scientific notation
0x7b;      # hexadecimal notation  (decimal 123)
0173;      # octal notation (decimal 123)
use Math::Complex;  # bring in the Math::Complex module
12+3*i;    # complex number 12 + 3i
```

Backtick Strings

You can also enclose a string in backtics (`). This has the helpful property of executing whatever is inside the string as a Unix system command, and returning its output:

```
`ls -l`;
# evaluates to a string containing the output of running the
# ls -l command
```

Lists

The last type of literal that Perl recognizes is the *list*, which is multiple values strung together using the comma operator (,) and enclosed by parentheses. Lists are closely related to *arrays*, which we talk about later. *Lists* (and *arrays*) are composed from zero, one or more *scalars*, making an empty list, a list containing a single item or a more typical list containing many items, respectively.

```
('one', 'two', 'three', 1, 2, 3, 4.2);
  # this is 7-member list contains a mixure of strings, integers
  # and floats
```

Operators

Perl has numerous *operators* (over 50 of them!) that perform operations on string and numberic values. Some operators will be familiar from algebra (like "+", to add two numbers together), while others are more esoteric (like the "." string concatenation operator).

Numeric & String Operators

The "." operator acts on strings. The "!" operator acts on strings and numbers. The rest act on numbers.

Operator	**Description**	**Example**	**Result**
.	String concatenate	'Teddy' . 'Bear'	TeddyBear
=	Assignment	$a = 'Teddy'	$a variable contains 'Teddy'
+	Addition	3+2	5
-	Subtraction	3-2	1
-	Negation	-2	-2
!	Not	!1	0
*****	Multiplication	3*2	6
/	Division	3/2	1.5
%	Modulus	3%2	1
******	Exponentiation	3**2	9
<FILEHANDLE>	File input	<STDIN>	Read a line of input from standard input
>>	Right bit shift	3>>2	0 (binary 11>>2=00)
<<	Left bit shift	3<<2	12 (binary 11<<2=1100)
|	Bitwise OR	3|2	3 (binary 11|10=11
&	Bitwise AND	3&2	2 (binary 11&10=10
^	Bitwise XOR	3^2	1 (binary 11^10=01

Operator Precedence

When you have an expression that contains several operators, they are evaluated in an order determined by their *precedence*. The precedence of the mathematical operators follows the rules of arithmetic. Others follow a precedence that usually does what you think they should do. If uncertain, use parentheses to force precedence:

```
2+3*4;    # evaluates to 14, multiplication has precedence over addition
(2+3)*4;  # evaluates to 20, parentheses force the precedence
```

Logical Operators

These operators compare strings or numbers, returning TRUE or FALSE:

Numeric Comparison		**String Comparison**	
3 == 2	equal to	**'Teddy' eq 'Bear'**	equal to
3 != 2	not equal to	**'Teddy' ne 'Bear'**	not equal to
3 < 2	less than	**'Teddy' lt 'Bear'**	less than
3 > 2	greater than	**'Teddy' gt 'Bear'**	greater than

3 <= 2	less or equal	**'Teddy' le 'Bear'**	less than or equal
3 >= 2	greater than or equal	**'Teddy' ge 'Bear'**	greater than or equal
3 <=> 2	compare	**'Teddy' cmp 'Bear'**	compare
		'Teddy' =~ /Bear/	pattern match

The **<=>** and **cmp** operators return:

- **-1** if the left side is less than the right side
- **0** if the left side equals the right side
- **+1** if the left side is greater than the right side

File Operators

Perl has special *file operators* that can be used to query the file system. These operators generally return TRUE or FALSE.

Example:

```
print "Is a directory!\n" if -d '/usr/home';
print "File exists!\n" if -e '/usr/home/lstein/test.txt';
print "File is plain text!\n" if -T '/usr/home/lstein/test.txt';
```

There are many of these operators. Here are some of the most useful ones:

-e filename	file exists
-r filename	file is readable
-w filename	file is writable
-x filename	file is executable
-z filename	file has zero size
-s filename	file has nonzero size (returns size)
-d filename	file is a directory
-T filename	file is a text file
-B filename	file is a binary file
-M filename	age of file in days since script launched
-A filename	same for access time

Functions

In addition to its operators, Perl has many *functions*. Functions have a human-readable name, such as **print** and take one or more arguments passed as a list. A function may return no value, a single value (AKA "scalar"), or a list (AKA "array"). You can enclose the argument list in parentheses, or leave the parentheses off.

A few examples:

```
  # The function is print.  Its argument is a string.
  # The effect is to print the string to the terminal.
print "The rain in Spain falls mainly on the plain.\n";

  # Same thing, with parentheses.
print("The rain in Spain falls mainly on the plain.\n");

  # You can pass a list to print.  It will print each argument.
  # This prints out "The rain in Spain falls 6 times in the plain."
print "The rain in Spain falls ",2*4-2," times in the plain.\n";

  # Same thing, but with parentheses.
print ("The rain in Spain falls ",2*4-2," times in the plain.\n");

  # The length function calculates the length of a string,
  # yielding 45.
length "The rain in Spain falls mainly on the plain.\n";

  # The split function splits a string based on a delimiter pattern
  # yielding the list ('The','rain in Spain','falls mainly','on the plain.')
split '/','The/rain in Spain/falls mainly/on the plain.';
```

Creating Your Own Functions

You can define your own functions or redefine the built-in ones using the **sub** function. This is described in more detail in the lesson on creating subroutines, which you'll be seeing soon..

Often Used Functions (alphabetic listing)

For specific information on a function, use **perldoc -f *function_name*** to get a concise summary.

abs	absolute value
chdir	change current directory
chmod	change permissions of file/directory
chomp	remove terminal newline from string variable
chop	remove last character from string variable
chown	change ownership of file/directory
close	close a file handle
closedir	close a directory handle
cos	cosine
defined	test whether variable is defined
delete	delete a key from a hash

die	exit with an error message
each	iterate through keys & values of a hash
eof	test a filehandle for end of file
eval	evaluate a string as a perl expression
exec	quit Perl and execute a system command
exists	test that a hash key exists
exit	exit from the Perl script
glob	expand a directory listing using shell wildcards
gmtime	current time in GMT
grep	filter an array for entries that meet a criterion
index	find location of a substring inside a larger string
int	throw away the fractional part of a floating point number
join	join an array together into a string
keys	return the keys of a hash
kill	send a signal to one or more processes
last	exit enclosing loop
lc	convert string to lowercase
lcfirst	lowercase first character of string
length	find length of string
local	temporarily replace the value of a global variable
localtime	return time in local timezone
log	natural logarithm
m//	pattern match operation
map	perform on operation on each member of array or list
mkdir	make a new directory
my	create a local variable
next	jump to the top of enclosing loop
open	open a file for reading or writing
opendir	open a directory for listing
pack	pack a list into a compact binary representation

package	create a new namespace for a module
pop	pop the last item off the end of an array
print	print to terminal or a file
printf	formatted print to a terminal or file
push	push a value onto the end of an array
q/STRING/	generalized single-quote operation
qq/STRING/	generalized double-quote operation
qx/STRING/	generalized backtick operation
qw/STRING/	turn a space-delimited string of words into a list
rand	random number generator
read	read binary data from a file
readdir	read the contents of a directory
readline	read a line from a text file
readlink	determine the target of a symbolic link
redo	restart a loop from the top
ref	return the type of a variable reference
rename	rename or move a file
require	load functions defined in a library file
return	return a value from a user-defined subroutine
reverse	reverse a string or list
rewinddir	rewind a directory handle to the beginning
rindex	find a substring in a larger string, from right to left
rmdir	remove a directory
s///	pattern substitution operation
scalar	force an expression to be treated as a scalar
seek	reposition a filehandle to an arbitrary point in a file
select	make a filehandle the default for output
shift	shift a value off the beginning of an array
sin	sine
sleep	put the script to sleep for a while

sort	sort an array or list by user-specified criteria
splice	insert/delete array items
split	split a string into pieces according to a pattern
sprintf	formatted string creation
sqrt	square root
stat	get information about a file
sub	define a subroutine
substr	extract a substring from a string
symlink	create a symbolic link
system	execute an operating system command, then return to Perl
tell	return the position of a filehandle within a file
tie	associate a variable with a database
time	return number of seconds since January 1, 1970
tr///	replace characters in a string
truncate	truncate a file (make it smaller)
uc	uppercase a string
ucfirst	uppercase first character of a string
umask	change file creation mask
undef	undefine (remove) a variable
unlink	delete a file
unpack	the reverse of pack
untie	the reverse of tie
unshift	move a value onto the beginning of an array
use	import variables and functions from a library module
values	return the values of a hash variable
wantarray	return true in an array context
warn	print a warning to standard error
write	formatted report generation

Ok, now you know all the perl functions, so we're done. Thanks for coming.

Variables

A variable is a symbolic placeholder for a value, a lot like the variables in algebra. These values can be changed. Compare literals whose values cannot be changed. Perl has several built-in variable types:

Scalars: **$variable_name**
: A single-valued variable, always preceded by a $ sign.

Arrays: **@array_name**
: A multi-valued variable indexed by integer, preceded by an @ sign.

Hashes: **%hash_name**
: A multi-valued variable indexed by string, preceded by a % sign.

Filehandle: **FILEHANDLE_NAME**
: A file to read and/or write from. Filehandles have no special prefix, but are usually written in all uppercase.

We discuss arrays, hashes and filehandles later.

Scalar Variables

Scalar variables have names beginning with $. The name must begin with a letter or underscore, and can contain as many letters, numbers or underscores as you like. These are all valid scalars:

- $foo
- $The_Big_Bad_Wolf
- $R2D2
- $______A23
- $Once_Upon_a_Midnight_Dreary_While_I_Pondered_Weak_and_Weary

You assign values to a scalar variable using the **=** operator (not to be confused with **==**, which is numeric comparison). You read from scalar variables by using them wherever a value would go.

A scalar variable can contain strings, floating point numbers, integers, and more esoteric things. You don't have to predeclare scalars. A scalar that once held a string can be reused to hold a number, and vice-versa:

Code:

```
$p = 'Potato';  # $p now holds the string "potato"
$bushels = 3;   # $bushels holds the value 3
$potatoes_per_bushel = 80;  # $potatoes_per_bushel contains 80;

$total_potatoes = $bushels * $potatoes_per_bushel;  # 240

print "I have $total_potatoes $p\n";
```

Output:

```
I have 240 Potato
```

Scalar Variable String Interpolation

The example above shows one of the interesting features of double-quoted strings. If you place a scalar variable inside a double quoted string, it will be interpolated into the string. With a single-quoted string, no interpolation occurs.

To prevent interpolation, place a backslash in front of the variable:

```
print "I have \$total_potatoes \$p\n";

# prints: I have $total_potatoes $p
```

Operations on Scalar Variables

You can use a scalar in any string or numeric expression like `$hypotenuse = sqrt($x**2 + $y**2)` or `$name = $first_name . ' ' . $last_name`. There are also numerous shortcuts that combine an operation with an assignment:

$a++
: Increment $a by one

$a--
: Decrement $a by one

$a += $b
: Modify $a by adding $b to it.

$a -= $b
: Modify $a by subtracting $b from it.

$a *= $b
: Modify $a by multiplying $b to it.

$a /= $b
: Modify $a by dividing it by $b.

$a .= $b
: Modify the **string** in $a by appending $b to it.

Example Code:

```
$potatoes_per_bushel = 80;  # $potatoes_per_bushel contains 80;

$p = 'one';
$p .= ' ';      # append a space
$p .= 'potato'; # append "potato"

$bushels = 3;
$bushels *= $potatoes_per_bushel; # multiply

print "From $p come $bushels.\n";
```

Output:

```
From one potato come 240.
```

String Functions that Come in Handy for Dealing with Sequences

Reverse the Contents of a String

```
$name          = 'My name is Lincoln';
$reversed_name = reverse $name;
print $reversed_name,"\n";
# prints "nlocniL si eman yM"
```

Translating one set of letters into another set

```
$name = 'My name is Lincoln';
# swap a->g and c->t
$name =~ tr/ac/gt/;
print $name,"\n";
# prints "My ngme is Lintoln"
```

Can you see how a combination of these two operators might be useful for computing the reverse complement?

Processing Command Line Arguments

When a Perl script is run, its command-line arguments (if any) are stored in an automatic array called **@ARGV**. You'll learn how to manipulate this array later. For now, just know that you can call the **shift** function repeatedly from the main part of the script to retrieve the command line arguments one by one.

Printing the Command Line Argument

Code:

```
#!/usr/bin/perl
# file: echo.pl
use strict;
use warnings;
$argument = shift;
print "The first argument was $argument.\n";
```

Output:

```
(~) 50% chmod +x echo.pl
(~) 51% echo.pl tuna
The first argument was tuna.
(~) 52% echo.pl tuna fish
The first argument was tuna.
(~) 53% echo.pl 'tuna fish'
The first argument was tuna fish.
(~) 53% echo.pl
The first argument was.
```

Computing the Hypotenuse of a Right Triangle

Code:

```
#!/usr/bin/perl
# file: hypotense.pl
use strict;
use warnings;
$x = shift;
$y = shift;
$x>0 and $y>0 or die "Must provide two positive numbers";

print "Hypotenuse=",sqrt($x**2+$y**2),"\n";
```

Output:

```
(~) 82% hypotenuse.pl
Must provide two positive numbers at hypotenuse.pl line 6.
(~) 83% hypotenuse.pl 1
Must provide two positive numbers at hypotenuse.pl line 6.
(~) 84% hypotenuse.pl 3 4
Hypotenuse=5
(~) 85% hypotenuse.pl 20 18
Hypotenuse=26.9072480941474
(~) 86% hypotenuse.pl -20 18
Must provide two positive numbers at hypotenuse.pl line 6.
```