Perl II

Operators, truth, control structures, functions, and
processing the command line

Dave Messina

Math

```
1 + 2   = 3         # kindergarten
x       = 1 + 2     # algebra
my $x   = 1 + 2;    # Perl
```

What are the differences between the
algebra version and the Perl version?

Math

```
my $x = 5;
my $y = 2;
my $z = $x + $y;
```

Math

```
my $sum        = $x + $y;
my $difference = $x - $y;
my $product    = $x * $y;
my $quotient   = $x / $y;
my $remainder  = $x % $y;
```

Math

```
my $x = 5;
my $y = 2;
my $sum     = $x + $y;
my $product = $x - $y;
```

Variable names are arbitrary. Pick good ones!

What are these called?

```
my $sum        = $x + $y;
my $difference = $x - $y;
my $product    = $x * $y;
my $quotient   = $x / $y;
my $remainder  = $x % $y;
```

Numeric operators

Operator	Meaning
+	add 2 numbers
-	subtract left number from right number
*	multiply 2 numbers
/	divide left number from right number
%	divide left from right and take remainder
**	take left number to the power of the right number

Numeric comparison operators

Operator	Meaning
<	Is left number smaller than right number?
>	Is left number bigger than right number?
<=	Is left number smaller or equal to right?
>=	Is left number bigger or equal to right?
==	Is left number equal to right number?
!=	Is left number not equal to right number?

Why == ?

Comparison operators are yes or no questions

or, put another way, true or false questions

True or false:
> Is left number larger than right number?

```
2 > 1        # true
1 > 3        # false
```

Comparison operators are true or false questions

```
 5 >   3
-1 <=  4
 5 ==  5
 7 !=  4
```

What is truth?

0 the number 0 is false

`"0"`   the string 0 is false

`"" and ''`   an empty string is false

`my $x;`   an undefined variable is false

everything else is true

Examples of truth

```
my $a;        # FALSE (not yet defined)
$x = 1;       # TRUE
$x = 0;       # FALSE
$x = "\n";     # FALSE
$x = 'true';  # TRUE
$x = 'false'; # TRUE (watch out! "false" is a nonempty string)
$x = ' ';     # TRUE (a single space is non-empty)
$x = "\n\n";    # TRUE (a single newline is non-empty)
$x = 0.0;     # FALSE (converts to string "0")
$x = '0.0';   # TRUE (watch out! The string "0.0" is not the
              #       same as "0")
```

Sidebar: `=` vs `==`

1 equals sign to *make* the left side equal the right side.
2 equals signs to *test* if the left side is equal to the right.

```
my $x;            # x is undefined
my $x = 1;        # x is now defined
if ($x == 1)      # is $x equal to 1?
if ($x = 1)       # (wrong)
```

`use warnings` will catch this error.

Logical operators

Use <u>and</u> and <u>or</u> to combine comparisons.

Operator	Meaning
and	TRUE if left side is TRUE and right side is TRUE
or	TRUE if left side is TRUE or right side is TRUE

Logical operator examples

```
if ($i < 100 and $i > 0) {
  print "$i is the right size\n";
}
else {
  print "out of bounds error!\n";
}


if ($age < 10 or $age > 65) {
   print "Your movie ticket is half price!\n";
}
```

Let's test some more

Logical operators

Use `not` to reverse the truth.

```
$ok = ($i < 100 and $i > 0);
print "a is too small\n" if not $ok;

# same as this:
print "a is too small\n" unless $ok;
```

defined **and** undef

defined lets you test whether a variable is defined.

```
if (defined $x) {
    print "$x is defined\n";
}
```

undef lets you empty a variable, making it undefined.

```
undef $x;
print $x if defined $x;
```

if not

Testing for defined-ness:

```
if (defined $x) {
    print "$x is defined\n";
}
```

What if you wanted to test for undefined-ness?

```
if (not defined $x) {
    print "x is undefined\n";
}
```

if not

or you could use unless:

```
unless (defined $x) {
    print "$x is undefined\n";
}
```

Sidebar: operator precedence

Some operators have higher
precedence than others.

```
my $result = 3 + 2 * 5;

# force addition before multiplication
my $result = (3 + 2) * 5 = 25;
```

The universal precedence rule is this:
multiplication comes before addition,
use parentheses for everything else.

String operators

Operator	Meaning
eq	Is the left string same as the right string?
ne	Is the left string not the same as the right string?
lt	Is the left string alphabetically before the right?
gt	Is the left string alphabetically after the right?
.	add the right string to the end of the left string

String operator examples

```perl
my $his_first = 'Barry';
my $his_last  = 'White';
my $her_first = 'Betty';
my $her_last  = 'White';

my $his_full  = $his_first . ' ' . $his_last;
if ($his_last eq $her_last) {
    print "Same\n\n";
}
if ($his_first lt $her_first) {
    print "$his_first before $her_first\n\n";
}
```

Comparing numeric and string operators

Numeric	Meaning	String
==	equal to	eq
!=	not equal to	ne
>	greater than	gt
<	less than	lt
+	addition/concatenation	.

Control structures

Control structures allow you to control if and how a line of code is executed.

You can create alternative branches in which different sets of statements are executed depending on the circumstances.

You can create various types of repetitive loops.

Control structures

So far you've seen a basic program, where every line is executed, in order, and only once.

```
my $x = 1;
my $y = 2;
my $z = $x + $y;
print "$x + $y = $z\n\n";
```

Control structures

Here, the print statement is only executed some of the time.

```
my $x = 1;
my $y = 2;
if ($x == $y) {
    print "$x and $y are equal\n\n";
}
```

Components of a control structure

1. a keyword

2. a statement in parentheses

3. squiggly brackets

```
if ($x == $y) {
    print "$x and $y are equal\n\n";
}
```

The part enclosed by the squiggly brackets is called a block.

Components of a control structure

When you program, build the structure first and then fill in.

1. a keyword

2. a statement in parentheses

3. squiggly brackets

```
if ($x == $y) {
    print "$x and $y are equal\n\n";
}
```

4. now add the print statement

if

```
if ($x == $y) {
    print "$x and $y are equal\n\n";
}
```

If $x is the same as $y, then the print statement will be executed.

or said another way:

If ($x == $y) is true, then the print statement will be executed.

if — a common mistake

```
if ($x = $y) {
    print "$x and $y are equal\n\n";
}
```

What happens if we write it this way?

else

If the `if` statement is false, then the first print statement will be skipped and only the second print statement will be executed.

```perl
if ($x == $y) {
    print "$x and $y are equal\n\n";
}
else {
    print "$x and $y aren't equal\n\n";
}
```

elsif

Sometimes you want to test a series of conditions.

```perl
if ($x == $y) {
    print "$x and $y are equal\n\n";
}
elsif ($x > $y) {
    print "$x is bigger than $y\n\n";
}
elsif ($x < $y) {
    print "$x is smaller than $y\n\n";
}
```

elsif

What if more than one condition is true?

```perl
if (1 == 1) {
    print "$x and $y are equal\n\n";
}
elsif (2 > 0) {
    print "2 is positive\n\n";
}
elsif (2 < 10) {
    print "2 is smaller than 10\n\n";
}
```

given-when

is another way to test a series of conditions (whose full power you'll learn later).

```perl
my $x = 3;
given($x) {
    when ($x % 2 == 0) {
        print '$x is even';
    }
    when ($x < 10) {
        print '$x is less than 10';
    }
    default {
        die q(I don't know what to do with $x);
    }
}
```

unless

It's exactly the opposite of `if (something) *`
These statements are equivalent:

```
if ($x > 0) {
    print "$x is positive\n\n";
}
unless ($x < 0) {
    print "$x is positive\n\n";
}
```

If the statement `($x < 0)` is false, then the print statement will be executed.

*except you can't `unless..else` or `unless..elsif`

while

As long as `($x == $y)` is true, the print statement will be executed over and over again.

```
while ($x == $y) {
    print "$x and $y are equal\n\n";
}
```

Why might you want to execute a block repeatedly?

one line conditionals

An alternative form that sometimes reads better. The conditional comes at the end and parentheses are optional.

```
print "x is less than y\n" if $x < $y;

print "x is less than y\n" unless $x >= $y;
```

However, you can execute only one statement because there's no longer brackets to enclose multiple lines. Only works for `if` and `unless`.

functions

Functions are like operators — they do something with the data you give them. They have a human-readable name, such as `print` and take one or more arguments.

```
print "The rain in Spain falls mainly on the plain.\n\n";
```

functions

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\n";
```

functions

You can enclose the argument list in
parentheses, or leave the parentheses off.

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

function examples

You can pass multiple values
separated by commas to `print`,
and 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\n";


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

functions

A function may return no value, a single
value, or multiple values.

```
# print returns nothing.

print "The rain in Spain falls mainly on the plain.\n\n";


# The length function calculates the length of a string
# and returns the answer.

my $length = length "The rain in Spain falls mainly on the
plain.\n\n";
```

processing the command line

Often when you run a program, you want to pass it some information. For example, some numbers, or a filename.

These are called arguments.

```
$ add 1 2

$ parse_blast.pl mydata.blast
```

What are the command-line
arguments in these examples?

processing the command line

You can give arguments to Perl programs you write, and you can see those arguments inside your script using the <u>shift</u> function.

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

my $arg1 = shift;
my $arg2 = shift;
print "my command-line arguments were $arg1 and $arg2\n";
```