Learning Perl 6

brian d foy, <brian@stonehenge.com>

Version 0.6, Nordic Perl Workshop 2007

for the purposes of this tutorial

Perl 5 never existed

Don't really do this

```
$ ln -s /usr/local/bin/pugs /usr/bin/perl
```

Introduction

* It's a completely new language
* That other one never existed
* Llama 6 is a long way off
* This is the basics of the language
* Next week it might be different

Basis for this talk

* Apocalypses
* Exegeses
* Synopses
* Perl6-Pugs-N-NN
  * docs/quickref/data
  * examples/
* Actual Pugs behavior

Writing programs you can actually run

In 30 minutes, I can cover

* Data
* Variables
* Control structures
* Input / Output

If I had more time

* Subroutines
* Regular expressions
* Using modules
* Creating classes, objects, &c.

Getting Pugs

- http://www.pugscode.org
- Needs Glasgow Haskell Compiler (GHC)
- Get the **binary** builds
  - compilation can take a long, long time
  - and it eats up your CPU

Making a P6 program

* Programs are just **text files**
* Syntax is **C like**, mostly
  * whitespace is not significant, mostly
  * statements separated by semicolons
  * comments are # to end of line
* **Use pugs** on the shebang line

```
#!/usr/local/bin/pugs
say "Hello World";
```

Objects & Methods

* Data are **objects** or **nouns**
* Methods are **verbs** (actions)
* Object.Method

```
#!/usr/local/bin/pugs

"Hello World".say;
```

Run from command line

```
$ pugs hello.p6
Hello World

$ ./hello.p6
Hello World

$ pugs -e 'say "Hello World"'
Hello World

$ pugs
pugs> say "Hello World"
Hello World
bool::true
```

Scalars

Scalars are single values

Numbers

Strings

Boolean

Literal numbers

```
3 3.14 3.14e7 -4.56 0

123_456

0b0110

0o377 0o644

0xAB 0xDEAD_BEEF
```

say

* say can be used a method
* Outputs the value and tacks on a **newline**
* More output stuff later

```
"Hello World".say;

say "Hello World";
```

Arithmetic

`2 + 3`	5
`2 - 3`	-1
`2 * 3`	6
`2 / 3`	0.66666666...
`2 ** 3`	8
`2 % 3`	2

Method call forms

```
# indirect form
say 3;

# direct form
3.say;

# parens to group
( 10 / 3 ).say;

# / really just a method
(10./(3)).say;
```

Strings

- Sequence of zero or more characters
- Perl inter-converts automatically with numbers

Single quoted strings

```
'Hello World'.say;

'I said \'Hello World!\''.say;

'I need a literal \\'.say;

q/I don't need to escape/.say;
```

Double quoted strings

```
"Hello\tWorld".say;

"I said \"Hello World!\"".say;

"Hello World".print; # no newline

"Hello World\n".print;

qq/Hello World\n/.print;
```

String concatenation

- ~ stitches strings together

```
( "Hello" ~ "World" ).say;
HelloWorld

( "Hello" ~ " " ~ "World" ).say;
Hello World
```

String replication

x repeats and joins string

```
( "Hi" x 3 ).say;       HiHiHi

( "Hi" x 2.5 ).say;     floor - HiHi

( "Hi" x -1 ).say;      error!
```

Booleans

* True or False, Yes or No, On or Off, 1 or nothing
* Often the result of a comparison

Numeric comparisons

`5 < 6`	`True`
`5 > 6`	`False`
`5 == 6`	`False`
`5 <= 6`	`True`
`5 >= 6`	`False`
`5 != 6`	`True`

String comparisons

`'fred' lt 'barney'`	*False*
`'fred' gt 'barney'`	*True*
`'fred' eq 'barney'`	*False*
`'fred' le 'barney'`	*False*
`'fred' ge 'barney'`	*True*
`'fred' ne 'barney'`	*True*

Scalar variables

* Stores a single value
* Name starts with a letter or underscore, followed by letters, underscores, or digits
* Has a special symbol (sigil) prepended, `$`
* Starts off undefined (absence of value)
* We have to assign it a value
* Declare with `my` on first use

Scalar Assignment

```
my $num = 5;
"The number is $num".say;

my $str = "Pugs";
"Just another $str hacker, ".say;
```

Scalar value type

* The `ref` method gives the type of scalar

```
my $s1 = 5 < 6;
my $s2 = "Perl";
my $s3 = 6 - 5;
my $s4 = 3.14;

$s1.WHAT;               Bool
$s2.WHAT;               Str
$s3.WHAT;               Int
$s4.WHAT;               Rat
```

Standard input

```
"Enter a name> ".print;
my $input = (=$*IN).chomp;

"Enter another name> ".print;
$input = (=<>).chomp;
```

Control Structures

if-elsif-else

```
if 5 < 6  { "5 less than 6".say }


if 5 > 6  { "5 more than 6".say }
else      { "5 not more than 6".say }


if 5 < 4    { "5 less than 4".say }
elsif 5 > 4 { "5 more than 4".say }
else        { "5 not more than 4".say }
```

Complex comparisons

```
if( 5 < $x < 10 )
    {
    "$x is between 5 and 10".say
    }
else
    {
    "$x is not between 5 and 10".say
    }
```

Junctions

```
my $num  = 5;

if( $num == any( <5 6 7> ) )
    {
    "$num is in the set".say
    }
else
    {
    "$num is not in the set".say
    }
```

Expression modifiers

* Apply a condition to a single expression

```
"5 is greater".say if 5 > 6;

"5 is less".say if 5 < 6;
```

loop

```
loop ( init; test; increment ){ }
```

```
loop ( $i = 1; $i < 10; $i++ ) {
  "I can count to $i".say;
  }
```

```
I can count to 1
I can count to 2
I can count to 3
I can count to 4
...
```

next

* skips the rest of the block
* goes to next iteration

```
loop ( $i = 1; $i < 10; $i++ ) {
  next if $i % 2;
  "I can count to $i".say;
  }
```

```
I can count to 2
I can count to 4
I can count to 6
I can count to 8
```

last

* skips the rest of the iterations
* continues after the loop

```
loop ( $i = 1; $i < 10; $i++ ) {
  last if $i == 5;
  "I can count to $i".say;
  }
```

```
   I can count to 2
   I can count to 4
   I can count to 6
   I can count to 8
```

redo

* starts the current iteration again
* uses the same element (if any)

```
loop {
  "Do you like pugs?> ".print;
  my $answer = (=$*IN).chomp;

  redo if $answer ne 'yes';
  last;
  }
```

Number guesser

```
"Guess secret number from 1 to 10".say;
my $secret = rand(10+1).int;

loop {
  "Enter your guess> ".print;
    my $guess = (=$*IN).chomp;

  if $guess < $secret
    { "Too low!".say;   redo }
  elsif $guess > $secret
    { "Too high!".say;  redo }
  else
    { "That's it!".say; last }
  }
```

Lists & Arrays

Literal Lists

```
( 1, 2, 3, 4 )

<a b c d>

my $x = 'baz'
<<foo bar $x>>
«foo bar $x»

( 1 .. 3 )
( 'a' .. 'z' )
```

List replication

```
'f'  xx 4               <f f f f>

<g> xx 6                <g g g g g g>

< a b c > xx 2          < a b c a b c >
```

Joining elements

```
<1 2 3 4>.join(' ')    1 2 3 4

<1 3 5 7>.join(':')    1:3:5:7
```

Ranges

```
( 4 .. 7 )                < 4 5 6 7 >

( 'a' .. 'e' )            < a b c d e >

reverse 1 .. 3            < 3 2 1 >

( 1 .. 3 ).reverse        < 3 2 1 >
```

Arrays

* Array variables **store** multiple scalars
* Indexes list with **integers**, starting at **0**
* Same variable naming rules as a scalar
* Special character is **@** (think @rray)
* Name comes from a separate **namespace**
* Nothing to do with scalar of same name

Array assignment

```
my @a = < a b c >;

my @a = << a b $c >>

my @a = 1 .. 6;
```

Bounds

```
my @r = 37..42;
say "Minimum is " ~ @r.min;
say "Maximum is " ~ @r.max;

my @a = < 3 5 9 2 5 0 1 8 4 >;
say "Minimum is " ~ @a.min;
say "Maximum is " ~ @a.max;
```

Array elements

```
my @a = <a b c d e f g>;

my $first = @a[0];               a

my $last  = @a[-1];              g

my $count = @a.elems;            7

my @slice = @a[0,-1];            < a g >
```

Unique elements

```
my @a = <a b c a b d b d>;

my @b = @a.uniq;        < a b c d >
```

Hyperoperators

* Apply operator pairwise

```
my @nums    =   1  ..  10;
my @alphas  =  'a' .. 'j';

my @stitch = @nums >>~<< @alphas;
< 1a 2b 3c 4d 5e 6f 7g 8h 9i 10j >

my @square = @nums >>*<< @nums;
< 1 4 9 16 25 36 49 64 81 100 >
```

for

```
for 1 .. 5 -> $elem {
  "I saw $elem".say;
  }
```

```
I saw 1
I saw 2
I saw 3
I saw 4
I saw 5
```

for

```
for @ARGS -> $arg {
  "I saw $arg on the command line".say;
  }
```

```
  I saw fred on the command line
  I saw barney on the command line
  I saw betty on the command line
```

Hashes

Hash variables

* Hash variables stores unordered pairs
* Index is the "key", a unique string
* Makes a map from one thing to another
* Same naming rules as scalar and array
* Special character is % (think %hash)
* Name comes from a separate namespace
* Nothing to do with scalar, array of same name

Hash elements

```
my %h = <a 5 b 7 c 3>;

my $a_value = %h{'a'};          5

my $b_value = %h<b>;            7

my $count = %h.elems;           3

my @values = %h{ <b c> };   < 7 3 >

my @values = %h<b c>;       < 7 3 >
my %hash = (
  'fred'    => 'flintstone',
  'barney' => 'rubble',
  );

%hash.say;
  barney rubblefred flintstone
%hash.join("\n").say;
  barney rubble
  fred flintstone
```

Hash keys

```
my %hash = (
  'fred'   => 'flintstone',
  'barney' => 'rubble',
  );

for %hash.keys -> $key {
  "$key: %hash{$key}".say;
  }
```

```
barney: rubble
fred: flintstone
```

Hash values

```
my %hash = (
  'fred'   => 'flintstone',
  'barney' => 'rubble',
  );

for %hash.values -> $value {
  "One value is $value".say;
  }
```

```
One value is rubble
One value is flintstone
```

By pairs

```
my %hash = (
  'fred'   => 'flintstone',
  'barney' => 'rubble',
  );

for %hash.kv -> $key, $value {
  "$key ---> $value".say;
  }
```

```
barney ---> rubble
fred ---> flintstone
```

Counting words

```
my %words;

for =<> -> $line {
  for $line.chomp.split -> $word {
    %words{$word}++;
    }
  }

for %words.kv -> $k, $v {
  "$k: $v".say
  }
```

exists

* True if the key is in the hash
* Does not create the key

```
my @chars = <fred wilma barney betty>;

my %hash = (
    'fred'   => 'flintstone',
    'barney' => 'rubble',
    );

for @chars -> $char {
    "$char exists".say if %hash.exists($char);
    }
```

delete

* Removes pair from hash

```
my %hash = (
   'fred'    => 'flintstone',
   'barney'  => 'rubble',
   'dino'    => undef,
   );

%hash.delete('dino');
%hash.join("\n").say;
```

```
   barney rubble
   fred flintstone
```

Input
Output

Standard input

```
"Enter a name> ".print;
my $input = (=$*IN).chomp;

"Enter another name> ".print;
$input = (=<>).chomp;
```

File input operator

* The =<> reads from files from the command line arguments

```
for =<> -> $line {
  "Got $line".print;
  }
```

Opening files to read

```
my $fh = open( $file, :r );

for =$fh -> $line {
   "Got $line".print;
   }
```

Die-ing

```
my $file = "not_there";

my $fh = open( "not_there", :r )
  err die "Couldn’t open $file: $!";

for =$fh -> $line {
  "Got $line".print;
  }
```

try

* Catches exceptions

```
try {
  die "I'm dying" if time.int % 2;
  "I made it".say;
  };

"Error was $!".say if $!;
```

Standard filehandles

* Default filehandles `$*OUT` and `$*ERR`

```
$*ERR.say( "This goes to stderr" );

$*OUT.say( "This goes to stdout" );
```

Writing to files

```
my $file = "not_there";

my $fh = open( "not_there", :w )
  err die "Couldn't open $file: $!";

print $fh: @stuff;
# $fh.print( @stuff );
```

try

* Catches exceptions

```
try {
  die "I'm dying" if time.int % 2;
  "I made it".say;
  };

"Error was $!".say if $!;
```

Files and Directories

File tests

```
my $file = "file_tests.p6";

"Found file".say if $file ~~ :e;
"Readable file".say if $file ~~ :r;

my $file_size = $file ~~ :s;
# $file_size = stat($file).size

"File size is $file_size".say;
```

Other topics

* given is like C's switch (but better)
* variable value types
* complex data structures
* regular expressions - PCRE and new stuff
* sorting, string manipulation etc.
* subroutines have better calling conventions

Summary

* Perl 6 is a new language
* It borrows from Perl (and ancestors)
* It's not done yet, but it's almost usable