

Introduction to Perl

Practical Extraction and Report Language
or
Pathologically Eclectic Rubbish Lister
or
...

Perl? perl? PERL?

- The name of the language is "Perl".
- Any expansion you may read was made up after the fact. "PERL" is **never** correct.
 - The logo on the course website notwithstanding
- The executable program that interprets Perl code is "perl"
- "Only perl can parse Perl"
- See also: `perldoc -q difference`

Basic Structure

- Bears resemblances to C or Java
- semi-colons separate executable statements
- { } delimit blocks, loops, subroutines
- Comments begin with # and extend to end of line
- No main() function – code executed top-down.
- Analogous to C++ namespaces, packages are virtual containers
 - default package main
- Function arguments are comma-separated
 - parentheses are (usually) optional
- External libraries stored in Perl Modules (.pm files)
- Local variables declared with **my**. Global variables simply exist, belong to a single package.
 - Any local var not declared in a block/loop/subroutine is visible to the entire file, but no external file.

print

- print takes a comma-separated list of arguments to print.
- optional filehandle precedes arguments
 - defaults to `STDOUT` if none given
 - no comma between filehandle & arguments
- ```
print "Hello World\n";
print "Hello", " ", "World", "\n";
print("Five plus three: ", 5+3);
print STDERR "Warning!!!";
```

---

---

---

---

---

---

---

---

## First Complete Program

- ```
print "Hello World\n";
```
- Three different ways to run the code:
- "interactive mode"
 - start the perl interpreter: type `perl` at command line
 - type the code
 - send end-of-file (CTRL-D in Unix)
- Give interpreter a filename
 - Start your favorite unix text editor
 - If you don't have one, I strongly suggest `pico`
 - Type code, save file, exit editor.
 - give perl interpreter the filename: `perl hello.pl`
- Execute file directly (Preferred method!!)
 - shebang + `chmod` . . .

What do I do with this?

- To tell the OS that perl should execute the following code, use a shebang.
- **MUST** be first line of the file
 - no comments, no spaces prior to the shebang
- Standard: `#!/usr/bin/perl`
- Unfortunately, on RCS, `/usr/bin/perl` is an outdated version.
- Instead, use `#!/usr/bin/env perl`
- Notes:
 - no 'e' in 'usr',
 - space after 'env',
 - 'usr' preceded with slash

Making the code executable

- To tell the OS that this is an executable file, you need to change mode:
- **chmod u+x filename.pl**
 - This is a one-time command. No need to re-run chmod after editing filename.pl
- After making executable, simply run the file:
- **filename.pl**
 - if current directory isn't in your PATH, try:
./filename.pl

Before we get started...

- Perl is a very lenient language. It will allow you to do a whole host of things that you probably shouldn't be able to.
 - printing a variable that you never assigned to
 - trying to add 5 + 'foo'
- If you want Perl to warn you about this sort of thing (and you do): **use warnings;**
 - Has to come after shebang, but should be before any executable code
 - You may see legacy code that enables warnings by adding "-w" to the end of the shebang

Variables

- Three major types of Perl variables
- Scalars – single values
- Arrays – contain lists of values
- Hashes – "associative arrays"
- There are others (filehandles, typeglobs, subroutines)... we'll cover them later

Scalars

- A Scalar variable contains a single value.
- All of the standard types from C can be stored in a scalar variable
 - int, float, char, double, etc
 - No declaration of type of variable
- Scalar variable name starts with a \$
- Next character is a letter or underscore
- Remaining characters: letters, numbers, or underscores.
 - name can be up to 255 characters long
 - don't do that.
- All scalars have a default value of **undef** before assigned a value. (Not to be confused with 5-character string 'undef')

Examples

- `$num = 42;`
- `$letter = 'a';`
- `$name = 'Paul';`
 - NOTE! Strings are *single values*!
- `$grade = 99.44;`
- `$Big_String = 'The Quick Brown Fox...';`

Package vs Lexical variables

- Package variables are global.
- Lexical variables are 'local' to innermost enclosing block/file, and cannot be seen anywhere else.
 - In Perl, 'local' means something else entirely
- Package variables belong to a given package, but can be accessed anywhere, by any piece of code.
 - default package is "main".
 - other packages declared with the **package** statement.
- Package variables are not declared. They simply spring into existence.

Lexical variables

- declared with keyword **my**
- exist only from time of declaration until end of innermost enclosing block
 - or end of file, if not declared in a block
- Visible to any subroutines defined in same scope
- ```
my $file = 'text.txt';
my ($x, $y) = (10, 20);
print '$x + $y = ', ($x+$y), "\n";
```

---

---

---

---

---

---

---

---

## Package variables

- ```
#!/usr/bin/env perl  
$main::name = 'Paul';  
$Lalli::type = 'faculty';  
print "$main::name is a member  
of $Lalli::type\n";
```

 - To put variables inside a string, must use "" instead of ''.
 - We'll discuss this more next week.

Gee, that's helpful...

- If you use any variable not declared as a lexical, Perl assumes you meant to use a global, and assumes you meant the current package.
- ```
#!/usr/bin/env perl
my $name = 'John';
my $grade = 97.43;
print "$name has grade: $Grade\n";
```
- prints " has grade: "
- There is no lexical **\$name** nor **\$Grade**.
- Therefore, must mean the global variables **\$main::name** and **\$main::Grade**
- Both variables are previously unused, so contain **undef**, which prints as the empty string

---

---

---

---

---

---

---

---

### Let's be a little more strict

- You can see the kinds of problems this helpful "feature" can create.
- The feature can be disabled with the pragma:  
**use strict;**
- All package variables must now be fully qualified
  - in prior, **\$name** isn't a declared lexical, so it must be a global, but it's not fully qualified (ie, not written **\$main::name**)
- Typo'ing a lexical variable will now result in a compilation error.
  - "Package variable \$name must be fully qualified..."
- Best Practice: Always **use strict;** and use lexical variables unless you have a \*really\* good reason to use package variables.
  - (You won't in this class until we do Object-oriented programming)

---

---

---

---

---

---

---

---

### Lists

- A list is a comma-separated sequence of scalar values.
- Any number, and any types of scalars can be held in a list:
- (42, 'Douglas Adams', 'HHGttG');
- Lists are passed to functions, stored in arrays, and used in assignments
- **my (\$foo, \$bar, \$baz) = (35, 43.4, 'hi');**

---

---

---

---

---

---

---

---

### List assignments

- An assignment of a list of variables need not contain the same number of values on the left and right:
- **my (\$foo, \$bar) = (34, 'hello', 98.3);**
  - 98.3 simply discarded
- **my (\$alpha, \$beta) = (5);**
  - lexical **\$alpha** created, assigned to 5
  - lexical **\$beta** created, given default **undef**
- List assignments can be used to 'swap' variables:
  - **my (\$one, \$two) = ('alpha', 'beta');**  
**(\$one, \$two) = (\$two, \$one);**

---

---

---

---

---

---

---

---

## Arrays

- Arrays are variables that contain a list.
  - Some texts say that "array" is interchangeable with "list". These books are *\*wrong\**.
  - Analogous to difference between a string value 'Paul' and the variable **\$name** that holds it.
  - Any expression that calls for a string can take a variable holding that string instead. The reverse is not necessarily true:
    - `$name = 'Paul';`
    - `$new_name = $name; #OK, can use $name for 'Paul'`
    - `'Joe' = 'Paul'; #WRONG, cannot assign to string.`
- Similarly, any expression that calls for a list can take an array instead. However, not all expressions requiring an array can take a list.
- Arrays are not declared with any size or type. They can hold lists containing any number or type of values.
- Size can grow/shrink dynamically.

---

---

---

---

---

---

---

---

## Array variables

- All array variable names begin with @
- Just like scalars, second char is either a letter or underscore, and remaining are letters, underscores, or numbers.
- `my (@s, @stuff, $size);`
- `@s = ('a', 'list', 'of', 'strings');`
- `@stuff = (32, 'Paul', 54.09);`

---

---

---

---

---

---

---

---

## Accessing array elements

- To get at a specific element of the array, place the index number in [] after the array name, and replace the @ with a \$
  - You're accessing a *\*single\** value
- `my @foo = ("Hello", "World");`  
`my $greeting = $foo[0];`  
`my $location = $foo[1];`
- `my @age = ('I am', 29, 'years old');`  
`$age[1] = 30;`  
`@age now => ('I am', 30, 'years old')`

---

---

---

---

---

---

---

---

## Array flattening

- Remember that lists (and therefore arrays) can only contain scalar values.
- Trying to place one array in another will result in array flattening:
  - `my @in = (25, 50, 75);`
  - `my @out= ('a', 'b', 'c', @in, 'd', 'e');`
- @out contains eight elements:
  - a, b, c, 25, 50, 75, d, e
- Arrays on LHS of assignment will 'eat' remaining elements:
  - `my ($foo, @bar, $baz) = (5, 6, 7, 8);`
  - \$foo = 5, @bar = (6, 7, 8), \$baz undefined
- Creating two dimensional arrays involves references
  - week 6

---

---

---

---

---

---

---

---

## Special array variable

- for any array @foo, there exists a scalar variable \$#foo, which contains the last index of @foo
- `@stuff = (5, 18, 23, 10);`
  - \$#stuff → 3
- This variable can be used to dynamically alter the size of the array:
  - `$#stuff = 10; # @stuff now has 7 undef's`
  - `$#stuff = 1; # @stuff now => (5, 18)`
- This is rarely necessary: Just assign to the positions you want to exist:
- `my @things;`
  - `$things[2] = 'foo';`
  - @things → (undef, undef, 'foo')

---

---

---

---

---

---

---

---

## Arrays and Scalars

- `my $foo = 3;`
- `my @foo = (43.3, 10, 5.12, 'a');`
- \$foo and @foo are \*completely unrelated\*
- In fact, \$foo has nothing to do with \$foo[2];
- "This may seem a bit weird, but that's okay, because it *is* weird."
  - Programming Perl, pg. 54

---

---

---

---

---

---

---

---

## Array Trivia

- Arrays can also accept negative indices
  - `my @lets = ('a', 'b', 'c', 'd', 'e');`
  - `$lets[-1] → 'e';`
  - `$lets[-2] → 'd';`
  - etc
- Array slices – read or write pieces of an array
  - `@lets[2,4] → ('c', 'e');`
  - `@lets[1..3] → ('b', 'c', 'd');`
  - `@lets[2] → ('c');`
    - This is almost *\*never\** what you want
    - Compare with `$lets[2] → 'c'`
  - `@lets[3,5] = ($foo, $bar);`

---

---

---

---

---

---

---

---

## push ARRAY, LIST

- add values of LIST to end of ARRAY
- `push @array, 5;`
  - adds 5 to end of @array
- `push @foo, 4, 3, 2;`
  - adds 4, 3, and 2, to the end of @foo
- `@a = (1, 2, 3); @b = (10, 11, 12);`
- `push @a, @b, 'bar';`
  - @a now → (1, 2, 3, 10, 11, 12, 'bar')
- This is the preferred method of adding values to an array.
- (This is also an example of a difference between a list and an array. You cannot push new values onto a list)

---

---

---

---

---

---

---

---

## pop ARRAY

- remove and return last element of ARRAY
  - if no arg provided, uses @ARGV (week 3)
  - if within a subroutine, uses @\_ (week 6)
- `my @array = (1, 5, 10, 20);`
- `my $last = pop @array;`
  - `$last → 20`
  - `@array → (1, 5, 10)`
- `my @empty;`
- `my $value = pop @empty;`
  - `$value → undef.`

---

---

---

---

---

---

---

---

## unshift ARRAY, LIST

- Add elements of LIST to front of ARRAY
- **unshift @array, 5;**
  - adds 5 to front of @array
- **unshift @foo, 4, 3, 2;**
  - adds 4, 3, and 2, to the front of @foo
- **@a = (1, 2, 3); @b = (10, 11, 12);**
- **unshift @a, @b, 'bar';**
  - @a now → (10, 11, 12, 'bar', 1, 2, 3)

---

---

---

---

---

---

---

---

## shift ARRAY

- remove and return **first** element of ARRAY
  - like pop, uses @ARGV or @\_ if no arg provided
- **my @array = (1, 5, 10, 20);**
- **my \$first = shift @array;**
  - \$first → 1
  - @array → (5, 10, 20);
- **my @empty;**
- **my \$value = shift @empty;**
  - \$value → undef
- In addition to push/pop/shift/unshift, can use **splice** to work with the inner elements of an array.
  - `perldoc -f splice`

---

---

---

---

---

---

---

---

## join GLUE, LIST

- join takes a list of values, and returns a string consisting of the values separated by GLUE
  - **my @vals = (1, 2, 3, 4);**
  - **my \$nums = join '-x-', @vals;**
    - \$nums → '1-x-2-x-3-x-4'

## split SEP, STRING

- split takes string and a separator\*, and returns a list of values that SEP was separating
  - **my \$string = 'Hello World!';**
  - **my @vals = split ' ', \$string;**
    - @vals → ('Hello', 'World!')
- \*(Not really a string. See week 4.)

---

---

---

---

---

---

---

---

## Hashes

- Also known as associative arrays
- a list of key/value pairs.
- Similar to arrays, but 'indices' can be any scalar value, not just integers.
  - both the keys and values can be any scalar value, including multiple types in the same hash.
- Used to maintain a list of corresponding values.
- Hash names start with a %
  - remainder follows same rules as array & scalar

---

---

---

---

---

---

---

---

## Hash Example

- ```
my %legs_on = (  
    human    => 2,  
    dog      => 4,  
    octopus  => 8,  
    centipede => 100,  
    cat      => 4);
```
- Similar to arrays, access specific element by replacing % with \$, and enclosing the key in { }
 - if the key is a single 'word', you can omit the quotes:
`$legs_on{dog}`
- ```
print "Rover has $legs_on{dog} legs\n";
```
- Tip: If you feel the need to keep two lists of values, and access corresponding values in each list – you want a hash.

---

---

---

---

---

---

---

---

## Writing to a hash

- Keys must be distinct. Writing the value of a hash at an existing key **overwrites** the existing value
- Writing the value of a hash at a non-existing key **adds** a new key/value pair
- ```
my %age_of = ('Paul' => 30);
```

 - if key is single word, can omit the quotes:
`my %age_of = (Paul => 30);`
- ```
$age_of{Paul} = 31;
```

**#hash still has only 1 key/value pair**
- ```
$age_of{John} = 35;
```

#hash now has 2 key/value pairs
- There is no push/unshift equivalent, because...

More about hashes

- Hashes are unordered. There is no "first" and no "last". Order elements are added to hash is irrelevant.
- Hashes will "flatten" if assigned to an array:
 - `my %prof_of = (Perl => 'Lalli', CS1 => 'Hardwick', CS2 => 'Ingalls');`
 - `my @profs_and_classes = %prof_of;`
 - `@profs_and_classes` → ('CS1', 'Hardwick', 'CS2', 'Ingalls', 'Perl', 'Lalli')
 - (or possibly any other order – as long as key/value pairs are kept together)
- Hash slice – pieces of a hash:
 - `my @cs1_2 = @prof_of{'CS1', 'CS2'};`

What kind of variable is this?

	no subscript	[]	{ }
\$	scalar variable	array element	hash element
@	array variable	array slice	hash slice
%	hash variable	N/A	N/A

Built-in Variables

- Perl pre-defines some special variables
- See Chapter 28 of Camel for full list
 - or `perldoc perlvar`
- `$!` – last error received by operating system
- `$,` – string used to separate items in a printed list
- `$"` – string to use to separate items in an interpolated array (this makes sense next week)
- `$_` - "default" variable, used by several functions
- `%ENV` – Environment variables
- `@INC` – directories Perl looks for include files
- `$0` – name of currently running script
- `@ARGV` – command line arguments

Reading from the keyboard

- The "diamond" operator: `<>`
- Encloses the filehandle you want to read from. For now, the only filehandle is STDIN:
- **my \$line = <STDIN>;**
 - Reads next line from standard input (ie, the keyboard), and stores it in \$line.

chomp LIST

- When you read a line, the *entire* line is read – including the trailing `"\n"`.
- If you don't want the `"\n"`, make sure you **chomp** your string.
- **chomp** takes a list of strings. If any string passed in contains a trailing newline, that newline is removed.
 - Operates directly on argument
 - Does not return the "chomp"ed string
 - Returns the number of newlines removed (ie, total number of strings passed in that ended with a newline)
 - `my $x = chomp($foo);`
 - either 1 or 0.
 - Does not remove multiple newlines from a single string.

chomp examples

- **my \$input = <STDIN>;**
chomp \$input;
- This is so common that there's a shorthand idiom:
 - `chomp (my $input = <STDIN>);`
- **my @strings =**
("foo\n", "bar", "baz\n\n");
chomp @strings;
 - @strings → ("foo", "bar", "baz\n");
- There exists a similar function: **chop**
 - removes last character of a string, regardless of what it is.

Back to printing

- We said `print` takes a list of arguments to output.
- If no list is passed, prints the `$_` variable:

```
- $_ = "hello world";  
  print;
```
- Arguments printed are separated by the `$,` variable. Default is an empty string.

```
- my @nums = (24, 25, 26);  
  print "The numbers are:\n";  
  print @nums;  
#outputs: 242526
```
- `$_,` = `' , '`;

```
- $_, = ' , ';  
  print @nums;  
#outputs: 24, 25, 26
```

Best Practice

- Whenever you're changing a built-in variable like `$,` "localize" your changes:
- ```
{
 local $_, = ' , ';
 print @nums;
}
```

  
# `$_,` goes back to previous value
- `local` simply assigns a temporary value to a global variable.

---

---

---

---

---

---

---

---

## say (5.10 Exclusive)

- Perl 5.10 introduced several new features that would break old code.
- In order to use any of these new 5.10 features, you must tell Perl you want to use them:
- `use feature ':5.10';`
- You now have access to, among other things, the `say` function
- prints out a list of strings, followed by a newline:
- `say 'Hello World';`
- `say 'The value is', $x;`
- `say $foo` is implemented as

```
{
 local $_ = "\n";
 print $foo;
}
```

---

---

---

---

---

---

---

---