LECTURE 7: INTRODUCTION TO PERL

OUTLINE

- Introduction to Perl
  - PERL: "Practical Extraction and Report Language"
- Language syntax
  - Overview
  - Data Types

INSTALLATION PROBLEMS?

- Mac-Linux should be fine
- Download from perl.org
- Or use Linux terminals

WHAT IS PERL

- A high-level language–very high level
- A glue language–excellent for uniting different systems
- A scripting language–commands are usually placed in a text file, then interpreted by the perl interpreter
- Optimized for text-processing, I/O, and system tasks
- Incorporates syntax parts from bsh, csh, awk, sed, grep and C
- Open-source and free language-supported by a helpful international community

WHAT IS PERL

- The language compiler/interpreter program
- Compiles and interprets the source code in a single step
- Accepts many useful command-line arguments for simple “one-line” scripts

```
perl -ne 'print' hello.txt
```

```
perl -ne 'print if /first/' lines.txt
perldoc perlrun
```

HELLO WORLD

```
#!/usr/bin/perl
print "Hello World!\n";
```

```
Hello World!
```

```
$ perl hello.pl
```

```
$chmod u+x hello.pl (or chmod 777 hello.pl)
```

```
$./hello.pl
```

LANGUAGE SYNTAX OVERVIEW

- Perl code is written in plain text
  - Use your favorite text editor (emacs, vi/vim, sublime, notepad++, etc.) to enter code
- Running Perl programs
  - Programs may be run by the perl interpreter
  - Programs may be made executable
- perl my_program.pl (Without shebang line)
- ./my_program (With shebang line)

“SHEBANG” LINE IN UNIX

- The very first line of the script
  - Also called the interpreter line. Used by shell scripts, Python, and other interpreted languages
  - Specifies absolute path to the interpreter to use for a program. e.g., #!/usr/bin/perl
- A source file with a shebang line can be made an executable file and run directly

PRAGMA

- Compiler directives that affect program compilation
- Beginners should consider the following mandatory!
- use strict; (and no strict; to disable it)
  - Restricts unsafe constructs, requires declaration of variables, helps prevent common errors, make code more formal and less casual
- use warnings; (and no warnings;)
  - Enable or disable warning. Warnings provide helpful diagnostics. E.g., variables used only once, writing to files that were opened read-only, etc

GENERAL

- White space is ignored, so format for readability
- Comments begin with a # character
- All statements end with semicolons
- Statements may be combined into blocks with curly braces {}

```
#!/usr/bin/perl
# The Perl version of "Hello, world.”

use strict;
use warnings;

print "Just another Perl hacker.\n";
```

DATA TYPES

- Three main data types:
  - Scalars (Single valued variables)
  - Arrays (List of ordered scalars, indexed by number)
  - Hashes (Unordered scalars, indexed by strings)
- All variables are global unless otherwise declared
  - Declared in a local scope with a my qualifier (if not declared in any block, its scope is till the end of file)

SCALARS

- All scalars start with the $ character
  - Have a name preceded by a $ character
- Store any single value
- Scalars are typeless, no differentiation between string, integer, floating point, or character values
  - Dynamically assume whatever value is assigned

```
$a=12; $b=‘hello’;
$c=‘11’;
$d=$c*2;
#$d is 22, * implies an int conversion
$e=“hello”;
```

SCALARS STORE ANY SINGLE VALUE

- number
  - integer(12, 1e+100)
  - real(3.141592)
  - decimal(15), octal(017), hexadecimal(0xF)
- string
  - a single character(“a”)
  - many characters(“A quick brown…”)
  - Unicode ("\x{263A}", UTF-8 format)
- reference

SCALARS

- Name can be up to 251 characters(alpha-numeric, underscore)
- Must not begin with a digit
- Case-sensitive
- Some special names are reserved($_,$1,$/)
- Good style recommends descriptive words separated by underscores for readability

SCALARS IN A LOCAL SCOPE

- my keyword. Required by use strict;
- Limits the use of variable to enclosing block
- Declaring a variable without assigning a value leaves its value undefined
  - May declare a variable and assign a value in the same statement

```
use strict;
my $apple_variety; # Value undefined
$apple_variety = "Red Delicious"; # Defined
$apple_vareity = "Granny Smith"; # Error
my $apple_color = "red"; # Declared and defined
```

DYNAMICALLY ASSUME VALUES

- Scalar is a basic type in Perl
- Any scalar value can be freely assigned to any scalar variable
- No need to "cast” or re-declare values

```
my $quantity = 6;
# Declare&define
$quantity = "half dozen";
# Now a string
$quantity = 0.5 * 12;
# Numeric again
```

ARRAYS

- Store any number of ordered scalars
- Have a name preceded by an @ character
- Declared in a local scope with a my qualifier
- Indexed by number
- May be assigned in several ways
- Dynamically assume whatever values or size needed
- May be sliced
- Easy to iterate

ARRAY DECLARATION

- Store any number of ordered scalars
  - numbers, strings, references
  - any combination of above

```
my @fibonacci = (1, 1, 2, 3, 5, 8, 11);# Numbers
my @fruits = ("apples", "bananas", "cherries");
# Strings
my @grade_synonyms= (100,"A++","Perfect"); #Both
```

INDEXED BY NUMBER

- An index in square brackets refers to an array item
- Each item is a scalar, so use scalar syntax $array[$index]
- First item has index 0, last item has index $#array
- Negative numbers count from end of list
- Can directly assign an array item

```
my @fruits = ("apples", "bananas", "cherries");
print "Fruit flies like $fruits[1].\n"; # bananas
print "Life is like a bowl of $fruits[$#fruits].\n"; # cherries
print "We need more $fruits[-3] to make the pie.\n"; #apples
$fruits[0] = "oranges"; # Replace apples with oranges
```

MAY BE ASSIGNED IN SEVERAL WAYS

- Items bounded by parentheses, separated by comma
- Numeric value ranges denoted by .. operator
- Quoted word lists using qw operator
- Sublists are “flattened” into a single array

```
@prime_numbers = (2, 3, 5, 7, 11, 13); # Comma-separated
@composite_numbers = (4, 6, 8..10, 12, 14..16); # Numeric ranges
@fruits = ("apples", "bananas", "cherries");
@fruits = qw(apples bananas cherries);
@veggies = qw(radishes spinach);
@grocery_list = (@fruits, @veggies, "milk");
print "@grocery_list\n"
```

DYNAMIC VALUES OR SIZE AS NEEDED

- Can dynamically lengthen or shorten arrays
- May be defined, but empty
- No predefined size or "out of bounds" error
- unshift and shift add to and remove from the front
- push and pop add to and remove from the end
- see array_example.pl

ARRAYS MAY BE SLICED

- A slice (or sub-array) is itself an array
- Take an array slice with @array[@indices]
- $array[0] is a scalar, that is, a single value
- @array[0] is an array, containing a single scalar
- Scalars always begin with $
- Arrays always begin with @

ARRAY SLICING EXAMPLE

- `my @fruits = qw(apples bananas cherries oranges);`

-
  ```
  my @yummy = @fruits[1,3];
  print "My favorite fruits are: @yummy\n";
  ```

-
  ```
  my @berries = @fruits[2];
  push @berries, "cranberries";
  print "These fruits are berries: @berries\n";
  ```

ITERATING OVER ARRAYS

- foreach loop iterates over entire array
- Good to localize the scalar to the loop

```
my @fruits = qw(apple orange grape cranberry);
foreach my $fruit (@fruits) {
print "We have $fruit juice in the refrigerator.\n";
}
```

FIN!