Perl

# Introduction to Perl

* Use Basic Perl Syntax
* Use While/For loops in Perl
* Use Basic IO

Perl was created in the 1980, it’s compiled now but it’s really seen as an interpreted language.

It’s VERY good at string processing. It is a full-featured language, and can do some object-oriented constructions (but we won’t do it in this class). Bash might still be better for basic job control.

|  |
| --- |
| Perl was written in part as an awk-killer (awk is another scripting language) as well as an sed-killer. Two of the programs provided with it are **a2p** and **s2p** which are used for converting **awk** scripts and **sed** scripts into Perl.  Perl is one of the earliest of the next generation of scripting languages. (Tcl/Tk can probably claim primacy).  It has powerful integrated regular expression handling with a vastly more powerful language.  It can provide access to almost all system calls, and has extensibility (as opposed to awk and sed)  One of Perl’s mottos is: “TMTOWTDI” (Pronounced: TIM-TOADY)  “There’s more than one way to do it”.  Perl has “objects” but it is more of an add-on than a fundamental part of the language. |

Before we begin let’s create a “Perl” folder in our home directory: **mkdir Perl**

Let’s verify that we have Perl installed: **perl -v**

Our first Perl Script:

Open an Linux editor (VI, nano, doesn’t really matter). Similar to Bash, we had to place a directive at the top of the script file to tell the OS what interpreter we want to use for the code.

In Bash, we added #! /bin/bash

In Perl, we will add the following directive: #! /usr/bin/perl

**vi myFirst.pl**, In the editor, type the following:

Text

Description automatically generated

Again, the “#! /usr/bin/perl” tells our Linux kernel to use the Perl interpreter for this file.

The “print” command is essentially the same as “echo” in Bash. It prints things to STDOUT.

Text

Description automatically generated

To run our Perl script, we would use the command “perl” followed by the filename of the perl script we want to run. In this case, we’ve created a file called “myFirst.pl” and used that. The “.pl” is our extension for perl files (more of an industry standard, no required for execution, as seen below)

Text

Description automatically generated

We’ll notice that **Hello world** is printed to STDOUT (our monitor, or terminal). Once it’s done executing, we’ll see our bash prompt immediately follow it. Adding the new line character to our print statement makes the bash prompt appear on a new line after script execution.

Text

Description automatically generated

**Exercise:** Spend the next couple of minutes creating your first Perl script like above

Next, let’s learn about declaring variables and some other directives we can use. We’ll expand on our first script to include a variable and some directives for detecting errors and warnings.

**vi mySecond.pl**

Text

Description automatically generated

Notes:

* **use strict**
  + this directive will generate compile-time errors if you try do the following:
    - Access variables without declaring them.
    - Use a bareword (sequence of characters suitable for an identifier) identifier in an improper way.
  + This directive will generate run-time errors for symbolic references.
  + It’s a good idea to always use this one – it will catch a lot of typo errors
  + If you don’t use this, you don’t need to declare variables with “my”
* **use warnings**
  + This helps you find typing mistakes, it warns you whenever it sees something wrong with your program.
  + Helps you find mistakes in your program faster.
* Aside:
  + The most important point to note here is that “use strict” would abort the execution the execution of a program if it finds errors. “use warnings”, however, would only provide warnings, it won’t abort execution.
* **my $variableName**
  + Declares the variable to be local to the enclosing block in which it is defined.
  + The purpose of **my** is to define static scoping.
  + This can be used to use the same variable name multiple times but with different values under different contexts.
  + “our” declares global variables

Text

Description automatically generated

Let’s work with errors next. **vi myThird.pl**

Text

Description automatically generated

Note: The nano editor can be run with the -c option, so that line numbers are displayed: (vi does this by default).

We see that Perl has outputted an error on line 6. Taking a look, I can see quickly that I’ve forgotten a semicolon on line 5. Let’s fix that and re-run it.

Text

Description automatically generated

**Lists vs scalars.**

**vi myFourth.pl**

Text

Description automatically generated

Using the “@” symbol in front of a variable declaration will create a **list variable.** The “$” will create a **scalar variable**. Think of lists as arrays and scalars as regular variables (in this example, a string).

Notes:

* **localtime()** is a built-in Perl function for returning the current timestamp. Day, Month, Date, time in 24 hour format, and year.
* Assigning the output of **localtime()** to a list will create a list element for each component of the localtime
* Assigning the output of **localtime()** to a scalar will just treat it as a string.
* When you reference a single element of a list, that is a scalar so you use $ instead of @

Perl comes with a tool called perldoc that is like man pages for parts of Perl. To install it, run **sudo apt install perl-doc**. Then try **perldoc -f localtime** to learn about the **localtime** function

## Variable Concatenation

The main symbols that we’ll use to interact with print and variables will be “.”:

**vi myFifth.pl**

Text

Description automatically generated

Notes:

* **$number1 + $number2** will treat the values in $number1 and $number2 as integers and attempt to perform addition on them.
  + The result is 12
* **$number1 . $number2** will treat the values in $number1 and $number2 as strings and attempt to concatenate them.
  + The result is 57

## Interactive Mode

In Perl, we can accept input from the user (via standard in, or stdin) by using the reserved keyword **<STDIN>**. Let’s try it out. We’ll create a script that displays a prompt to the user via **print** and assign whatever is typed by the user to a variable.

**vi mySixth.pl**

Text

Description automatically generated

Notes:

* **my $name;**
  + This declares a scalar variable called “name” in the local context
* **$name = <STDIN>**;
  + This accepts input from **STDIN** (in this case, the keyboard)
  + Whatever is typed is assigned to the scalar variable **name**
* In the final print statement, notice that there’s a variable ($name) embedded right in the string. This is allowed in Perl (with any string that uses double quotes, just like BASH), and is called variable interpolation.

Output:

* Notice that when we print out $name, for some reason it’s adding a \n to the variable as well. We can see this because the “’” character is on a new line.
* This is because when I entered my name “Joe” in the keyboard, I also pressed ENTER at the end. STDIN captured that and also assigned the \n character to $name variable.

We can trim character, but there is a special function (chomp) that we can use to trim all these types of characters.

Let’s alter our script to include “chomp ($name=<STDIN>), which will essentially clean up our input so only text appears:

Text

Description automatically generated

Text

Description automatically generated

We can see that the new line character from when I pressed the ENTER key has been removed (chomped)

**Exercise1: Write a Perl script to accept input from the user. The script should display a message asking the user for their first name and then their last name. Print out the full name, ensuring that the new line character \n has been trimmed (via chomp).**

# While and For Loops

Perl can use looping structures, which have similar syntax to other languages that you are familiar with.

## The While Loop.

We can use the keyword **while** to create a while loop that accepts input from STDIN. We can use if statements within that loop to terminate the loop given the input is a certain string of characters. For example, if we rewrite the above to make use of while, we can have an if statement that terminates the loop if we type in the string “q”.

**vi stdinwhile.pl**

Text

Description automatically generated

Notes:

* while($name = <STDIN>)
  + This while loop starts with accepting input from stdin and assigning it to the variable $name.
* chomp $name
  + This executes the chomp directive on the variable
  + Removes \n or any other undesirable characters.
* last if ($name eq “q”);
  + An if statement – this is a ***special shorthand syntax*** that can be used when the action is a single statement.
  + General syntax: <do something> **if** (some conditions are met)
  + In this case, “last” instructs the interpreter that this current iteration is the last iteration of the loop **IF** the condition ($name eq “q”) is **true**
* push @names, $name
  + This pushes whatever we have in the $name variable onto the end of the array @names
* $names[0]
  + We can reference individual elements of an array by using $ instead of @ as well as the index position within **[]**.

We can rewrite the IF statement to something we are more familiar with in terms of syntax. We can also utilize Dumper (a Perl library to include) to print out the entire array in a single call.

Let’s alter our script a little:

Text

Description automatically generatedText

Description automatically generated

Notes:

* use Data::Dumper;
  + Directive to include the Data library for perl. We can reference it with our friendly name “Dumper”
* if ($name eq “q”)
  + Re-written if statement with different syntax.
  + Equivalent to the previous one
* @names + 0
  + This strange expression causes Perl to convert @names to an integer so that it can add it to 0. The integer form of a list is actually the number of elements in that list.
    - Perl programmers love weird tricks like this.
* print Dumper @names;
  + Dumper is a library we can use to pass in our array, which then iterates through it and prints it.

## The For Loop

We can use for loops (or foreach in certain scenarios) to iterate through something with a known endpoint.

Example, instead of using Dumper, let’s use a for loop to print out the array:

**vi stdinfor.pl**

Text

Description automatically generatedText

Description automatically generated

Notes:

* for(my $i=0; $i < @names; $i++)
  + a for loop as you are already familiar.
    - for (*initializer*; *condition*; *updater*)
  + Similar to other languages, such as C and JavaScript
  + my $i=0
    - This defines a for loop iteration variable called $i
  + $i < @names
    - Condition to test for **true** or **false**
    - You could also use $i <= $#names
* $names[$i]
  + Grabs the “*i*th” position in the array @names

## Foreach

We can use the foreach loop to iterate through the array itself without any Boolean conditions, or temporary variables:

Text

Description automatically generatedText

Description automatically generated

Notes:

* foreach(@names)
  + a for loop that iterates for **each** of the elements in the array @names
  + Usual syntax: foreach my *$variable* (*@list*)
  + If you don’t specify a variable, it uses the default variable…
* $\_
  + This is called the **default variable**
  + Generally, we either want to make its own variable, or leave it out entirely.
  + $\_ signifies the current value of our iteration.
    - For example, $\_ would be equal to “Joe Herbert” on the first iteration; “Alex Wang” for the second, etc.
  + More info: <https://perlmaven.com/the-default-variable-of-perl>