Perl

~/IFT383/mod-4

Objectives

- Manipulate variables, constants and data types in Perl
- Read files from the command line
- Arrays
- Decisions and looping
- Regular Expression parsing in Perl

Introduction

- Perl is a scripting language based on the best parts of Awk, shell scripting and Sed
- Like Awk; Perl is best used for processing and manipulating text
  - PERL = Practical Extraction and Report Language
- Borrows syntax from C/C++ and shell scripting
- Unlike shell scripts and Awk; scripts are not parsed line-by-line
  - They are read entirely into memory before being executed
  - Results in faster, more reliable and more portable scripts

Perl command syntax

- Similar to Awk; Perl can be given one-line commands, or script files
  - The **-e** option is used to provide a one-line script
  - *perl -e ‘print “Hello, World\,”;’*
    - Notice the semicolon “;” at the end of the command
    - a semicolon is used to mark the end of a statement
      - similar to Java, JavaScript, C/C++
- Using a script file
  - provide the name of the script as an argument to Perl
  - *perl myScript.pl*
    - Perl scripts use the **.pl** or **.cgi** suffixes

A simple example (helloWorld.pl)

```
#!/usr/bin/perl
# This is a comment
# Just as we saw in BASH and awk scripts
print "Hello, World!\n";
```

Variables

- There are three types of variables in Perl; each identified using a special “sigil” character
  - Scalar variables
    - identified with “$”, such as $VARIABLE
    - stores a single value
  - Arrays
    - Identified with “@”, such as @MYARRAY
    - stores a collection of values, indexed starting at 0
  - Hash maps
    - Identified with “%”, such as %MYMAP
    - similar to associative arrays in Awk

Scalars and constants

Scalar variables - $

- Contains a single value; either a number, or a string
- You do not need to explicitly declare Perl variables
  - The default value is 0, or an empty string
- Assignment is similar to awk or BASH
  - $MYNUMBER = 10
- Data type is determined automatically based on context
  - $MYVAR = 10
    - Stores a number 10
  - $MYVAR = “10”
    - Stores the characters 1 and 0
    - can cause problems when trying to do arithmetic operations

Operations on numbers

- Perl supports the majority of operators we used in awk
  - $VAR++, --, +=, -=
- There are also many built-in functions
  - int($MYVAR)
    - Converts a floating-point number to an integer
  - rand(NUMBER)
    - generates a random integer from 1 to NUMBER
  - log(NUMBER)
    - Calculates natural log of NUMBER
  - abs(NUMBER)
    - Absolute value of NUMBER
  - many more exist!

Working with scalars (numbers.pl)

```
#!/usr/bin/perl
# Demonstration of assigning, accessing and manipulating integers
$apples=0;  # assign
print "I currently have $apples apples.\n";
$apples++;  # increment
print "I added an apple and now have $apples\n";
$apples+=9; # add and assign
print "I added 9 apples and now have $apples\n";
$apples = $apples / 2; # divide existing value and assign result
print "I gave away half my apples and now have $apples\n";
```

Operations on strings

- Double quotes and single quotes work as they did in BASH
  - $MYVAR = “$WORD”
    - Will substitute the value of $WORD
  - $MYVAR = ‘$WORD’
    - Will use the literal value $WORD without substitution
- You can join strings (concatenate) just as you did in awk and BASH. Perl also gives us a special “dot operator” to join strings
  - $MYVAR=”$THING1$THING2”
  - $MYVAR=$THING1 . $THING2
- Also supports our usual escape characters, and a few to convert case
  - $UPPER=”\U$VAR\E” converts $VAR to upper case
  - $LOWER=”\L$VAR\E” converts $VAR to lower case

String operators

- **x** is used to specify string repetition
  - print “-” x 100;
    - prints 100 dashes
- To get the length of a string; use the **length** operator
  - print length “123456789”;
    - prints 9
- The lc and uc operators can convert to upper and lower case characters
  - print lc “HELLO”;
    - prints hello
  - print uc “hello”;
    - prints HELLO

String functions (index and substr)

- index(string, search, position);
  - Returns an integer containing where in the string the search phrase appears
  - position is an optional parameter indicating where to start the search
    - if not specified, defaults to 0
- rindex(string, search, offset);
  - like index' but searches from the end of the string
  - offset is the number of characters away from the end to begin search
- substring(source, start, length)
  - returns the characters from source, starting at position start and reading as many characters as specified by length

Using strings (strings.pl)

```
#!/usr/bin/perl
# Demonstrates working with strings
$THING1 = "Hello,";
$THING2 = "World!";
$LINE = '-';
print "Using double-quotes: $THING1 $THING2\n"; # Using double quotes
print 'Using single quotes: $THING1 $THING2\n' . "\n"; # using single quptes
print "Excape sequences: \U$THING1\E\t\t\L$THING2\E\n"; # escape sequences
print $LINE x 50 . "\n"; # multiply
print length "$THING1 $THING2"; # length
print uc "\nuppercase: $THING1 $THING2\n"; # uc = uppercase
print lc "lowercase: $THING1 $THING2\n"; # uc = uppercase
print index($THING2,"l",0); # index
```

‘Strict’ mode

- As in other languages, Perl supports *strict mode* which forces tighter restrictions on declaring strings and enables support for variable scoping
- By default all Perl variables are available in global scope which allows the contents of a variable to be accessed anywhere in the script
  - Enabling strict mode allows declaration of variables in local scope which prevents accidental assignment or access in other areas of the script
  - this will become more clear as we start working with loops and conditionals
  - local variables are defined using the my keyword

```
use strict;
my $VARIABLE = “Something secret”;
```

Using strict mode
strict-broken.pl and strict-fixed.pl

strict-broken.pl

```
#!/usr/bin/perl
use strict;
my $VARIABLE;
$VARIABLE = "Hello, World!";
print $VARIABLE . "\n";
{
	my $VARIABLE2="Goodbye!"
}
print $VARIABLE2 . "\n";
```

strict-fixed.pl

```
#!/usr/bin/perl
use strict;
my $VARIABLE;
$VARIABLE = "Hello, World!";
print $VARIABLE . "\n";
{
	my $VARIABLE2="Goodbye!";
	print $VARIABLE2 . "\n";
}
```

Constants (assigning)

- A constant is a special type of scalar variable that can onlybe assigned once
- Useful for values that never change; such as days of the week, PI, feet in a mile, etc.
- are **not** preceded with a $
- Declaring a constant
  - Single constant
    - use constant pi => 3.14159;
  - Multiple constant
    - use constant {
      PI => 3.14159;
      WEEK => 7;
      }
  - “use constant” is a pragma and => in this case, is a digraph

Constants (accessing)

- When accessing a constant; the $ prefix is not used
  - constants may not be used as part of string expansion
  - incorrect
    - print “the constant pi is equal to PI”;
  - correct
    - print “The constant pi is equal to ” . PI;
- Constants are available in the global scope

Getting input

- We can read data from standard input; either data from a pipe, or data entered interactively by the user
- Reading from standard input is done using the <STDIN> object

```
$MYVAR=<STDIN>
print $MYVAR
```

- If no input was provided via a pipe; Perl will wait for input from the keyboard
  - press **CTRL+D** to stop input from keyboard
- Data from standard input is often preceded by a newline “\n” character
  - To remove this, we use the **chomp** function

```
$MYVAR=chomp(<STDIN>);
```

Input example (echo.pl)

```
#!/usr/bin/perl
use constant SEP => "-";
$INPUT=<STDIN>;
print "input without calling chomp:\n";
print SEP . $INPUT . SEP . "\n";
print "input after calling chomp:\n";
chomp($INPUT);
print SEP . $INPUT . SEP . "\n";
```

Arrays

Array variables

- Just as in Awk; holds a list of scalar values
  - indexed by integers starting at 0
  - Names of arrays are preceded by the @ sigil
    - @myArray = (“Apple”, “Blueberry”, “Cantaloupe”)
- Elements of an array are accessed using their index
  - @myArray[1];
- Address all elements in the array by not using an index
  - @myArray;
- Array size (number of elements)
  - $size = scalar @myArray
    - counts the number of scalar elements in the array

Populating an array

- Filling an array with the same value
  - @myArray = 100 x 5;
    - Will create an array with 5 elements; all containing 100
- Creating an array with a series
  - @letters = (“A”..”Z”);
  - @numbers = (1024 .. 65534);
- Using a list
  - @months = (“Jan”, “Feb”, “March”, “April”);
  - @months = qw(Jan Feb Mar April);
    - The **quote words** function is a shortcut for defining lists
- Dynamically add elements
  - push(@months. “Aug”, “Sep”);
    - adds “Aug” and “Sep” to the end of @months

Array - Populating from standard input

- Store standard input in an array; separated by newline “\n”
  - @myArray = <STDIN>
- Elements of the array will include the newline character
  - Use chomp() to remove if needed

Using split() and join()

- Split takes a pattern (RegEx) and uses it to split a single string into an array

  ```
  @myArray = split(/,/, “Purple,Blue,Yellow,Maroon,Gold”);

  print @myArray[0];  # will print Purple
  print @myArray[1];  # will print Blue
  ```

- Join converts an array to a single string, using the provided seperator

  ```
  @myArray = split(/,/, “Purple,Blue,Yellow,Maroon,Gold”);
  print join(“--”, @myArray);    #prints; Purple--Blue--Yellow--Maroon--Gold
  ```

Arrays example (arrays.pl)

```
#!/usr/bin/perl
$INPUT = <STDIN>;
chomp($INPUT);
@things = split(/,/, $INPUT);
print "You entered " . scalar @things . " things\n";
print "The first thing was: " . @things[0] . "\n";
print "The last thing was: " . @things[(scalar @things) - 1] . "\n";
print "If I add one more thing...\n";
push(@things, "another thing!");
print "There are now " . scalar @things . " and they are...\n";
print join("\n", @things) . "\n";
```

Hash maps
(associative arrays)

Hash maps - associative arrays

- Hash maps in Perl are similar to associative arrays in Awk
  - They use keys rather than indexes
  - keys are constructed from strings
- Create an empty hash map
  - %myMap;
- Create a new hashmap with data

```
%myMap = (“key1”, “value1”, “key2”, “value2”);
%myMap = (
    “key1” => “value1”,
    “key2” => “value2”
);
```

Hash Maps - assignment and access

- Assigning a value
  - $myMap{“key”} = “value”;
    - note the use of { } curly brackets
    - Also notice that we assign and access elements of a hash with $ rather than %
- accessing a value
  - print $myMap{“key”}

Hash maps - keys and values

- Get an array of keys from a hashmap
  - @myArray = keys(%myMap);
    - @myArray will contain all possible key values currently in %myMap
- Get a list of values
  - @myArray = values(%myMap);
    - @myArray will contain a list of all values from %myMap
- Order is not guaranteed!
  - keys and values returned from these functions are in no particular order

hash example (map.pl)

```
#!/usr/bin/perl
%myMap = (
        "sleep" => "ZzzZzzZzz...",
        "eat" => "Om nom nom!",
        "read" => "You read a short story...",
        "watch" => "You watch a documentary about Liamas"
);
$myMap{"homework"} = "You get all your homework done! YAY!";
@keys = keys(%myMap);
print "please enter one of the following ways to spend Saturday
morning:\n\n";
print join("\n",@keys) . "\n";
$response=<STDIN>;
chomp($response);
print $myMap{$response} . "\n";
```

If statement

This looks familiar...

```
if (CONDITION) {
    # do something
} elsif (CONDITION) {
    # do something else
} else {
    # something else
}
```

IMPORTANT! Notice that the “else if” is spelt “elsif”

The Perl statement qualifier

- In addition to our standard if statement, Perl has another unique form of if;

  print “Something was true” if (CONDITION);

- Adding if to the end of a line, requires CONDITION to be true in order for the line to be executed
- JavaScript has something similar…

  if (CONDITION) comsole.log(“Something was true”);

```
#!/usr/bin/perl
$VAR1=100;
$VAR2=50;
if ($VAR1 > $VAR2) {
        print $VAR1 . " is greater than " . $VAR2 . "\n";
} elsif ($VAR1 < $VAR2) {
        print $VAR1 . " is less than " . $VAR2 . "\n";
} else {
        print $VAR1 . " is equal to " . $VAR2 . "\n";
}

print "something, something, something...\n" if ($VAR2==50);
print "another something\n" if ($VAR2!=50);
```

if.pl

For loop

This looks familiar too...

```
for ($var=0; $var < 5; $var++) {
    print $var . “\n”
}
```

Output;

```
0
1
2
3
4
```

For loop example (nl.pl)

```
#!/usr/bin/perl
@input = <STDIN>;
for ($line=0; $line < scalar @input; $line++) {
        chomp(@input[$line]);
        print $line+1 . "\t" . @input[$line] . "\n";
}
```

for each loop

foreach through array

Runs once for each element in an array

```
foreach $thing (@myArray) {
    print $thing
}

foreach (@myArray) {
    print $_
}
```

The two loops are equivalent. The first uses a named variable; the other uses a built-in ‘default’ variable

foreach through hashmap

Similar to what we used in Awk, but we need to use the **keys** directive, as Perl will not automatically do this for us as Awk did.

```
foreach $key (keys %myMap) {
    print $myMap{$key}
}
```

or

```
foreach (keys %myMap) {
    print $myMap{$_}
}
```

While loops

Have we met before?

```
while (CONDITION) {
    # do something
}
```

Input files

Input file from command line

- You can specify an input file after the name of your script when calling Perl
  - perl myScript.pl data.csv
- You can also specify the input file as a script argument
  - ./myScript.pl data.csv
    - This requires your script to have the interpreter line #!/usr/bin/perl
- Reading lines from an input file;

```
while (<>) {
    # $_ contains each line. The loop runs until there are no more lines
    print;  # without any arguments, prints contents of $_
}
```

  - Note: this also works when data is passed via standard in!

while loop
(nlWhile.pl)

```
#!/usr/bin/perl
$LINE=1;
while (<>) {
        chomp; # with no argument, chomp acts on $_
        print $LINE . "\t" . $_ . "\n";
        $LINE++;
}
```

Subroutines
(similar to functions)

Perl user-defined subroutine

- To define a subroutine, we use the **sub** keyword

```
sub sandwich {
    # arguments are passed in the array @_
    print “making a sandwich with” . join(“, ”, @_)
}
```

- Calling a subroutine that does not return a value
  - &sandwich(“Lettuce”, “Tomato”, “Onion”);
- Calling a subroutine as a function that returns a value
  - $mySandwich = sandwich(“Lettuce”, “Tomato”, “Onion”);

Subroutine example (sandwich.pl)

```
#!/usr/bin/perl
sub sandwich {
        print "making a sandwitch with:\n";
        print join("\n", @_);
        print "\n";
        return 1.99 * scalar @_;
}

&sandwich("peanut butter","jelly");
print "-" x 25 . "\n";
$mySandwich = sandwich("Ham", "Turkey", "Provalone");
print 'Your total cost is: $' . $mySandwich . "\n";
```