INGM master BFG - Practical 1

Perl (1)

Introduction

- This practical is dedicated to Perl Programming. While you work through the exercises below compare your results with those of your fellow students and ask for help and comments if required.
- This document can be found at

  https://homes.di.unimi.it/re/Corsi/INGM_master_BFG/PERLCOMPBIO_practical1.pdf

  and you might proceed more quickly if you cut-and-paste code from that PDF file. Note that a cut-and-paste operation may introduce extra spaces into your code. It is important that those are removed and that your code exactly matches that shown in this worksheet.
- The exercises and instructions in this worksheet assume that you use the Department's Linux systems to experiment with Perl. Use MobaXterm to connect to one of the Linux systems and log in using your MWS username and password.

  If you want to use the Department's Windows systems and our Perl installation on Windows instead, then you can do so. However, note that being able to work using a variety of operating systems is also a skill that employers value, in particular given that, arguably, the majority of computing devices use Unix-based operating systems.
- To keep things simple, we will just use a text editor and a terminal. You can use whatever text editor you are most familiar or comfortable with.
- If you do not manage to get through all the exercises during this practical session, please complete them in your own time before the next practical takes place.

Exercises

1. Let us start with the introductory example you have seen in the lectures.

   a. Open a text editor and enter the following Perl code:

   ```
   #!/usr/bin/perl
   # Author: <fill in your name>
   # Practical 1: Perl Scalars [perl01A]

   print "Hello World!\n";
   ```

   Replace `<your name>` with your own name.

   b. Save the code to a file named `perl01A` in some appropriate directory.

   c. In a terminal, go to the directory in which the file has been stored.

   d. Make sure that the file `perl01A` is readable, writable and executable by yourself but by nobody else by using the command

   ```
   chmod u+x,og-rwx perl01A
   ```

   Remember that you only have to do so once. When you later save the file again after making modifications, the file permissions will be preserved and the file remains executable.

e. Now execute the Perl script using the command
   `./perl01A`
   and check that the output is:

```
Hello World!
```

2. We also looked at single quoted and double quoted strings and the different interpretation of the backslash character in these.

   a. Add the following code at the end of your Perl script, save the file and then execute it.

```
$text = "stop!";
print 'Single-quotes: ','don\'t \'don\'t\' "don\'t" \U$text',"\n";
print "Double-quotes: ","don't 'don't' \"don\'t\" \U$text","\n\n";
print 'Single-quotes: ','glass\\table glass\table glass\ntable',"\n";
print "Double-quotes: ","glass\\table glass\table glass\ntable","\n\n";
```

   Note the character `'` is the one two keys to the right of the character `L` on a standard UK keyboard.

   b. Check that the output is as you would expect.

   c. Add two print statements to your Perl script that each produce the following output:

```
'There's no fun in Java.\\"
```

   One statement should use single quoted strings the other double quoted strings.

3. In the lectures you have learned that Perl converts between strings and numbers depending on the context.

   a. Add the following code at the end of your Perl script, save the file and then execute it.

```
$student_id = "200846369";
$staff_id  = "E00481370" ;
print "student_id = $student_id; staff_id = $staff_id\n";
$student_id++;
$staff_id++;
print "student_id = $student_id; staff_id = $staff_id\n";
$student_id += 1;
$staff_id += 1;
print "student_id = $student_id; staff_id = $staff_id\n\n";
```

   b. Try to figure out what is going on.
   Hint: Consult
   http://perldoc.perl.org/perlop.html#Auto-increment-and-Auto-decrement

4. In the lectures you have learned that Perl distinguishes between numeric comparison operators and string comparison operators.

   a. Add the following code at the end of your Perl script

```
if (35 == 35.0) {
  print("35 is numerically equal to 35.0\n")
} else {
  print("35 is not numerically equal to 35.0\n")
}
if ('35' eq '35.0') {
  print("'35' is string equal to '35.0'\n")
} else {
  print("'35' is not string equal to '35.0'\n")
}
if ('35' == '35.0') {
  print("'35' is numerically equal to '35.0'\n")
} else {
  print("'35' is not numerically equal to '35.0'\n")
}
if (35 < 35.0) {
  print("35 is numerically less than 35.0\n")
} else {
  print("35 is not numerically less than 35.0\n")
}
if ('35' lt '35.0') {
  print("'35' is string less than '35.0'\n\n")
} else {
  print("'35' is not string less than '35.0'\n\n")
}
```

b. Save the file again and then execute it.

c. Check that the output is as you would expect.

5. The code we have seen in Exercise 4 is much too verbose to appeal to a Perl programmer. A much more concise way to code the first conditional statement of Exercise 4 is as follows:

```
print("35 is ",(35 == 35.0) ? "" : "not ",
      "numerically equal to 35.0\n");
```

a. Add the code above at the end of your Perl script, save the file and then execute it.

b. Check that the last line of the output is the same as the output produced by the first conditional statement.

c. In analogy to the code above construct equivalent statements for the remaining four conditional statement of Exercise 4. Add those to your Perl script and check that you constructed those statements correctly by comparing your output to that of the original conditional statements.

6. *Switch statements* are useful in a lot of contexts where otherwise a large number of conditional statements would be required.

a. Open a text editor and enter the following Perl code:

```
#!/usr/bin/perl
# Author: <fill in your name>
# Practical 1: Perl Control Structures [perl01B]

use feature "switch";
$month = 3;
given ($month) {
  when ([1, 3, 5, 7, 8, 10, 12]) { $days = 31 }
  when (2) { $days = 28 }
  default { $days = 0 }
}
print ("Month $month has $days days\n");
```

Replace `<your name>` with your own name.

b. Save the code to a file named `perl01B` in some appropriate directory, adjust its access rights as shown in Exercise 1c, then execute the file.

c. Check that the output is correct.

d. Modify the code so that the remaining months of the year are also covered.

e. Add code so that instead of printing out a number for the month you get the name of the month, for example, the output for `$month = 3` should be

```
print ("March has 31 days\n");
```

Note: The code shown above should not be modified, except for the print statement at the end! The required functionality should be achieved solely by adding code that associates the names of months to numbers.
Hint: An array or a hash might come in handy.

7. Let us put the code you have developed in Exercise 6 to a test.

a. Modify the code of your Perl script so that the code of the switch statement plus the print statement that follows it is contained in the following *for-loop*:

```
for ($month=1; $month<=12; $month++) {
#  <your code here>
}
```

Save the file and execute it.

b. Is the output correct?

c. What happens if you change the for-loop to

```
for ($month=0; $month<=12; $month++) {
#  <your code here>
}
```

Does your code still work? If not, can you fix it?

8. You have seen in the lectures that *list literals* are a convenient way to initialise *arrays*.

a. Open a text editor and enter the following Perl code:

```
#!/usr/bin/perl
# Author: <fill in your name>
# Practical 1: Perl Arrays and Hashes [perl01C]

@array1 = (1..10);
@array2 = (10.0..20.0);
@array3 = (20.5..30.5);
@array4 = ("a".."z");
print "\@array1 = ",join(" ",@array1),"\n";
print "\@array2 = ",join(" ",@array2),"\n";
print "\@array3 = ",join(" ",@array3),"\n";
print "\@array4 = ",join(" ",@array4),"\n\n";
```

Replace `<your name>` with your own name.

b. Save the code to a file named `perl01C` in some appropriate directory, adjust its access rights as shown in Exercise 1c, then execute the file.

c. Try to make sense of the output.

d. Add four print statements to your Perl script that print out the length of each of the four arrays.

e. We have seen that Perl allows both *positive array indices* and *negative array indices*. What happens if those get out of bounds?

   To test this, add the following code at the end of your Perl script, save the file and execute it.

```
for ($index=0;$index<=11;$index++) {
       print "\$array1[$index] = ",$array1[$index],".\n";
}
for ($index=-1;$index>=-11;$index--) {
       print "\$array1[$index] = ",$array1[$index],".\n";
}
print "\n";
```

   Observe what happens if the index gets out of bounds.

f. Add code to your Perl script that determines what value is actually returned once the index gets out of bounds.

9. So far all the array indices that we have used were integers. But we know that Perl does not have an integer datatype. Instead there is the datatype *scalar* that includes integers, floating-point numbers and strings. So, it is natural to suspect that any scalar can be used as an array index.

   a. Let us see whether this is indeed the case. Add the following code at the end of your Perl script, save the file and execute it.

```
print "\$array1[0] = ",$array1[0],"\n";
print "\$array1[1] = ",$array1[1],"\n";
print "\$array1[\"1\"] = ",$array1["1"],"\n";
print "\$array1[\"hello\"] = ",$array1["hello"],"\n";
print "\$array1[12e-1] = ",$array1[12e-1],"\n";
print "\$array1[0.8] = ",$array1[0.8],"\n\n";
```

   b. Try to make sense of the output.

10. In the lectures we have seen that is possible to delete array elements. In the following we want to see in more detail what the effect of that is.

   a. The following code deletes the penultimate element of `@array1`. Add the code at the end of your Perl script, save the file and execute it.

```
print "Length of \@array1: ",scalar(@array1),"\n";
delete($array1[8]);
print "\$array[8] defined: ",
     defined($array1[8]) ? "TRUE" : "FALSE","\n";
print "\$array[8] exists: ",
     exists($array1[8]) ? "TRUE" : "FALSE","\n";
print "Length of \@array1: ",scalar(@array1),"\n\n";
```

   Observe that the length of the array does not change.

b. The following code now deletes the last element of `@array1`. Add the code at the end of your Perl script, save the file and execute it.

```
delete($array1[9]);
print "Length of \@array1: ",scalar(@array1),"\n\n";
```

Observe how the length of the array is now reduces by two.

c. We have seen in Exercise 10a that accessing a deleted array element returns the value `undef`. So, is deleting an array element the same as assigning `undef` to it? To check this, add the following code at the end of your Perl script, save the file and execute it.

```
$array[7] = "undef";
print "Length of \@array1: ",scalar(@array1),"\n\n";
```

Based on the output, what is your conclusion?

11. Let us now use *stack* and *queue* operations on arrays.

a. Add the following code at the end of your Perl script, save the file and execute it.

```
@planets = ("earth");
unshift(@planets,"mercury","venus");
push(@planets,"mars","jupiter","saturn");
print "Array\@1: ", join(" ",@planets),"\n";
$last = pop(@planets);
print "Array\@2: ", join(" ",@planets),"\n";
$first = shift(@planets);
print "Array\@3: ", join(" ",@planets),"\n";
print "    \@4: ",$first, " ",$last, "\n\n";
```

b. Check that the output is as described in the lecture notes.

c. There are still four elements left in the array. What happens if we continue to use pop elements of the array, in particular, if we apply the pop operation more often than there are elements left in the array?
Add the following code at the end of your Perl script, save the file, then execute it.

```
for ($index=1;$index<7;$index++) {
  $next = pop(@planets);
  print("The next planet removed is $next.\n");
}
```

d. Try to find out how you check whether an array is empty.
Modify the code given in 11c in such a way that the pop operation is only executed if the array is not empty.
Hint: Refer to the lecture notes on condition statements and on determining the length of an array.

12. Finally, let us turn to *hashes*.

a. Add the following code at the end of your Perl script, save the file, then execute it.

```
@list = ('a', 'c', 'd', 'b', 'e', 'd', 'c', 'c', 'e', 'b');
%hash1 = ('a' => 1, 'b' => 2);
%hash2 = %hash1;
$hash1{'b'} = 4;
print "\$hash1{'b'} = $hash1{'b'}\n";
print "\$hash2{'b'} = $hash2{'b'}\n\n";
```

Check that the output is as shown in the lectures.

b. Add code for a loop that prints out all the key-value pairs in `%hash1`.

c. A possible application of a hash is to count the number of occurrences of particular strings. So, in our example, `%hash1` may store the information that the character ’a’ has one occurrence and the character ’b’ four occurrences in some text or in a list such as `@list`. Add code to your script that uses a hash `%hash3` to count the number of occurrences of each character in `@list` and then prints out the characters and their counts in order of count.