Introduction to Perl

Perl is a popular scripting language for biological applications and bioinformatics. It is especially useful for data processing or automating tasks. Today I'm going to introduce you to a few simple examples of Perl scripts. Perl scripts can be written in any text editor. When you save your perl script, give file name should end with ".pl". Examples of perl script names "testrun.pl", "GenomeEditor.pl". To run a perl script in a Mac or Linux environment, just type "perl" and then the program name (e.g., "**`perl testrun.pl`**").

Let's Start …

1) The first line!

The first line in any Perl script is the following

```
#!/usr/bin/perl
```

2) Comments

Comments are just statements that you add that are NOT part of the program. To make a comment, just make a pound sign '#', and Perl ignores everything to the right of the pound sign. Including comments in your script is a very good habit.

Examples of comments:

```
#  Blah, blah, blah
#  In the next line, I am defining a new variable
```

3) Printing

The command "**`print`**" prints the text that follows in quotation marks. If not told to do otherwise, the text will be printed to the screen.

4) My first program

Write the following script:

```
#!/usr/bin/perl

#I'm going to tell Perl to print out a special message

print "I love writing Perl scripts!"
```

5) Special characters:

In your print statement (the stuff between the quotation marks), there are several special characters called backslash escapes that you will use a lot. Here are a few important ones:

\n New line
\n\n Double-space (2 new lines)
\t Tab
\" Print a double-quote

Try the following Perl Script:

```
#!/usr/bin/perl

#this is my second perl script

print "I love writing perl scripts!\n\tI am a
\"programmer\"!\n";
```

Variables

Variables are denoted by a dollar sign '**$**' followed by the variable name. They are defined using an equal sign (see below). A variable can be a **string** (or a sequence of characters), or a **number** (decimal or integer), but in Perl, you don't have to define the type of variable. Try the example below:

```
#!/usr/bin/perl

$myjob = "scientist" ;

#string variables should be surrounded by quotes when they
#are defined

$coffeeprice = 1.75;
$officenum = 218;

print "I am a $myjob\nMy office is $officenum Carr Hall\n";
print "I spent $coffeeprice on coffee\n";
```

Functions

Often we want to either learn some information about a variable or manipulate the variable (in a good way!). To do this, we use **functions**, which are essentially simple commands. Here are a few examples of functions for strings and for numbers:

String Functions (note $myjob is just an example of a string variable name)

```
length($myjob)
     #calculates the length of a string (number of
     #characters)
reverse($myjob)
     #reverses the order of characters in a string
uc($myjob) #converts the string to all uppercase
lc($myjob)  #converts the string to all lowercase
```

Number Functions

```
+  #used to add 2 numbers
```

```
-  #used to subtract 2 numbers
*  #used to multiple 2 numbers
/  #used in division
```

Try this example script:

```
#!/usr/bin/perl

$myjob = "scientist" ;
$joblength = length($myjob);
$revjob = reverse($myjob);
$MYJOB = uc($myjob);

print "What is my job?\n\n";
print "My job title is $joblength letters long.\n";
print "On opposite day I am a $revjob\n";
print "No, I am not a not a $myjob\nI am a $MYJOB \n";
```

The **substitute** and **transliterate** operators are often used to edit strings. These operators act like search and replace commands.

Substitution pattern matching operator:
s/PATTERN/REPLACEMENT/[option]
This replaces the entire **PATTERN** with the ***REPLACEMENT***. In the **option** section, "**g**" means replace globally. That is, it replaces every instance of the **PATTERN**. If there is no "**g**" in the options, only the first instance will be changed. Another commonly used option is "**i**". This makes the match case-insensitive. Here are some examples of how a substitution pattern matching operator can work:

```
$myjob = "scientist";
$myjob =~ s/s/S/;   #Now $myjob equals "Scientist"
```

or

```
$myjob = "scientist";
$myjob =~ s/s/S/g;  #Now $myjob equals "ScientiSt"
```

or

```
$myjob = "scientist";
$myjob =~ s/S/x/;   #$myjob is still "scientist"
```

or

```
$myjob = "scientist";
```

```
$myjob =~ s/S/x/i;  #Now $myjob equals "xcientist"
```

or

```
$myjob = "scientist";
$myjob =~ s/S/x/ig;  #Now $myjob equals "xcientixt"
```

Hint: If you are using the substitution pattern matching operator make sure there is NO space between the **=** and **~**.

The transliterate pattern matching operator.
tr/CHANGETHESE/TOTHESE/
This is similar to substitution pattern matching operator. The differences are demonstrated below:

```
$myjob = "scientist";
$myjob =~ tr/s/S/;   #Now $myjob equals "ScientiSt"
```

vs.

```
$myjob = "scientist";
$myjob =~ s/s/S/;   #Now $myjob equals "Scientist"
```

(In this example, note that the transliterate operator is always "global". In contrast, the substitution operator will only replace the first instance of a pattern unless you specify the global (**g**) option.)

OR

```
$myjob = "scientist";
$myjob =~ tr/st/ST/;   #Now $myjob equals "ScienTiST"
```

vs

```
$myjob = "scientist";
$myjob =~ s/st/ST/;   #Now $myjob equals "scientiST"
```

(In this example, the transliterate operator treats the s and t separately. In other words, it replaces all **s**'s with **S**'s and **t**'s with **T**'s. In contrast, the substitution operator treats **st** as a single unit. In other words, it only replaces **st** when the letters occur together; it does not replace individual **s**'s or **t**'s.)

Extra Credit. Write a perl script that will define a variable representing a DNA sequence "**AGCTATGCCTGGAAAG**" and print out the reverse compliment of the sequence (the compliment of A is T, C is G, G is C, and T is A).

OUTPUT
To open an output file:

open OUT, ">outputfilename";

This opens a file handle called OUT. The greater than sign ("**>**") tells the computer that it is an output file. The file handle doesn't have to be called OUT; you can give it any name you want. You will use this file handle (in our example "OUT") within the script to refer to the output file. The *outputfilename* is the name of the file you are writing. You can also name the output file anything you want. A file with this name will appear on your computer as the script runs.

Try this example script:

```
#usr/bin/perl

open OUT, ">mytruethoughts.txt";

print "Gordon is an OK scientist\n";
#this will be printed to the screen

print OUT "Gordon is the best scientist ever!\n";
#this will be printed to a file called "mytruethoughts.txt"
#it does not appear on the screen
#check your computer for the new file
```

***A tip. When you write the output like this:
open OUT, ">Samplefile.txt";
a new output file called "Samplefile.txt" will appear on your computer in the directory where you are running your Perl script. If your computer already has a file called "Samplefile.txt", it will be replaced by this new file. That is, all the content in the old file will be deleted, and a new file will open up. This is usually not a problem. However, if you want to keep the old file and append the new output to the old file just type:
open OUT, ">>Samplefile.txt";
The only difference is that you've typed two greater-than signs instead of one. The ">>" means that, if the file Samplefile.txt exists, the new output will be appended to the existing file.

Finally, we are going to learn about **for loops**, which help to automate tasks. For loops allow us to perform a single task a number of times. The task that you want to perform repeatedly is specified within curly brackets after the for loop.

Try the following useful script:

```
#!/usr/bin/perl

#the for loop will perform the task times
for (1..10)
{
    #here is the repeated task
    print "I'm sorry!\n";
}
```

When you are running a for loop as just shown, the variable $_ when used within the for loop represents the number of the loop.
For example – try this:

```
#!/usr/bin/perl

#the for loop will perform the task times
for (5..15)
{
    #this will print the number of the loop
    print "I'm running replicate $_!\n";
}
```

So what does this have to do with HPC? One simple use is that you can easily generate many output files. These files can be shell scripts use to submit jobs to the HPC cluster.

For example:

```
#!/usr/bin/perl

for (1..10)
{
    $rep = $_;
    open OUT, ">Myjob.$rep.sh";
    print OUT "\#!/bin/bash\n\n\#PBS –M
myemail\@ufl.edu\n\#PBS -l walltime=1:00:00\n\#PBS -l
pmem=750mb\n\#PBS -l nodes=1:ppn=1\n\n";
    print OUT "cd /scratch/hpc/mydirectory\n";
    print OUT "command line";
}
```

This script will print out 10 shell scripts. You can even tell Perl to submit these jobs to the queue by typing **system "qsub $Myjob.$rep.sh";** inside your for loop.
WARNING: BE CAREFUL- Perl is powerful; you can easily do something like submit a billion jobs to the queue, erase all of your files, or create extremely large output files.