Spring 5-2019

Designing Computational Biology Workflows with Perl - Part 1

Esma Yildirim
CUNY Queensborough Community College

Title: Designing Computational Biology Workflows with Perl – Part1
Author/Affiliation: Esma Yildirim / Queensborough Community College
Date: 05/15/2019
Material Type: Lecture
CS + Computational Biology
Software/Equipment Dependencies: An Amazon Web Services (AWS) account, a web browser and a command line interpreter program (e.g. Putty on Windows, Terminal on Linux/MacOSX)
Prior Knowledge Needed (if any): None
Keywords: Linux file system, Perl, gene-sequencing file formats
Approximate time needed: 2 hours
Description: This material introduces Linux File System structures and demonstrates how to use commands to communicate with the operating system through a Terminal program. Basic program structures and system() function of Perl are discussed. A brief introduction to gene-sequencing terminology and file formats are given.

Designing Computational Biology Workflows with Perl – Part 1

DNA sequencing speed dramatically increased in 2005 when the “short read”, the massively parallel sequencing technique, revolutionized sequencing capabilities and launched the “next generation” in genomic science. Since then, the data output more than doubled every year. Beyond the massive increase in data output, it also transformed the way scientists think about genetic information, establishing the foundation for personalized genomic medicine as part of standard medical care. Researchers can now analyze thousands to tens of thousands of samples in a single year [1].

A wide plethora of tools (e.g. *GATK, Samtools, Picard*) have been developed by different parties to clean, prepare and analyze the raw data that came out of sequencers (e.g. *Illumina HiSeq*). Different programming languages and platforms were used in the development of these tools. Therefore, best practice workflows were generated to identify the steps for the analysis of gene sequencing data. In each step, the scientists had a range of options to pick a suitable tool for the execution of a step to get the results they needed.

Scripting languages belong to a category of programming languages that support small programs written to automate the execution of tasks that could alternatively be executed one-by-one by a human operator. Among these languages, *Perl* is very popular with computational biology applications. It allows rapid prototyping, can easily combine parts of many other tools and call other programs. It is cross-platform and provides a rich set of regular expression matching capabilities, which is very important for applications that deal with text data. In the following subsections, Perl will be used as a scripting language, and basic syntactic rules to define literals, variables, control structures, and use of the *system()* function to call other programs will be presented.

The gene sequencing data analysis tools are programs developed by many different parties, accepting many different input parameters. Therefore, a scripting language to automate the execution of these tools will allow generating best practice workflows and an immense amount of flexibility.

Scripting languages are designed to run from a command line interpreter program such as the *Terminal* program in *Linux/MacOS* systems or *Putty/Command Prompt* in *Windows* systems. In this course, the students will run their scripts using one of these interactive programs.

The installation of *Perl* and gene sequence analysis tools is a cumbersome process and requires many software dependencies. To make the application of lab assignments easier, a relatively new technology called *Cloud Computing* will be integrated into the course. After the students learn about the basic commands to interact with a *UNIX-based file system* (The file system used in *MacOS* and *Linux* operating systems), they will use the Linux Virtual Machine provided as a material with this course, uploaded into a cloud computing environment (e.g. *Amazon Web Services*) and shared by the instructor. Then the students will launch their own machine instances in the cloud and run their scripts on them remotely.

The following subsections will cover Linux file system commands, Perl basics and introduce gene-sequencing data formats.

1. Linux/Unix File System Structure

The majority of the tools developed for computational biology applications (e.g. gene-sequence analysis tools) are designed for Linux/Unix operating systems. The tools introduced in this course, as part of the gene-sequence analysis workflows can be installed on a Linux platform and the input data for these analysis tools will reside on a Linux File system. Therefore, it is important to understand the structure of such systems and how to interact with one.

When a Linux/Unix operating system is installed on a computer, an automatic path tree is constructed for files and directories. The root of this tree is called **root** and it is represented with the **/** (slash) symbol. Figure 1 presents the directories created as branches under the root in a typical Linux system.

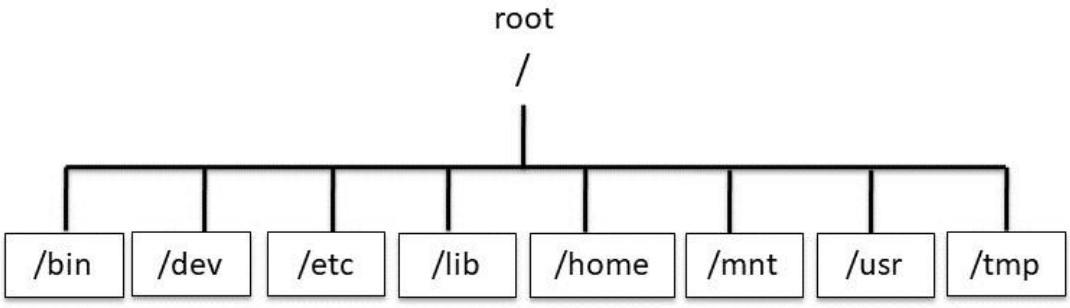

Figure 1. Linux file system tree

The user accounts are created under **/home** directory. For instance, the user account created for this course is **ubuntu** and all the input data is placed in a directory named **input** under ubuntu. The entire path to the input data directory is in this case:

```
/home/ubuntu/input/
```

Before Windows operating systems were so popular, users interacted with a computer through programs called **shells** or **command line interpreters**. These programs still exist in modern operating systems. One such program on a Linux or MacOS system is the “Terminal” program. For Windows systems, it is called “Command Prompt”. For the rest of the materials we will be using the Terminal program to communicate with a Linux File System.

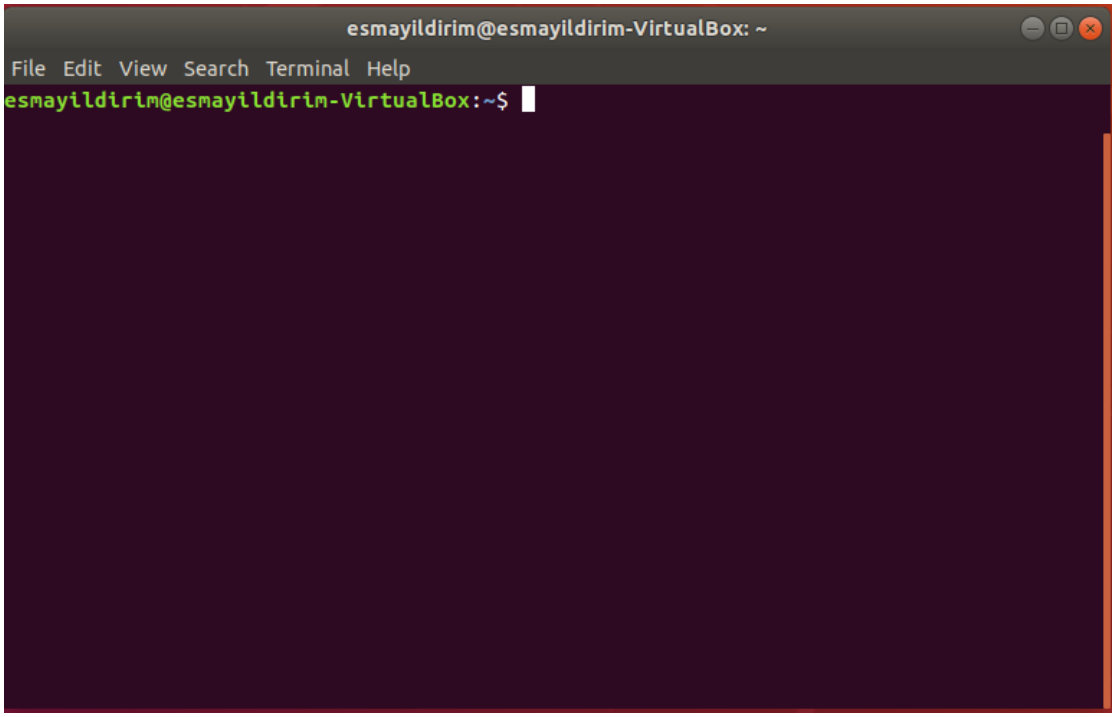

Figure 2. Terminal shell program

Shell programs allow us to type in certain set of commands to interact with the operating system and tell it what to do. The command line usually starts with a $ sign (Figure 2), but % sign is also very common.

A command is actually a program. It consists of the name of the command, its options and parameters:

```
<command name> [options] [parameters]
```

1.1. A list of useful commands

In this section, we will cover a subset of commands that will help communicate with the Linux File System when writing Perl scripts to design computational biology workflows.

1.1.1. List files and directories in a Linux File System: **ls**

The **ls** (list) command displays the list of files/directories under a certain path. The following command is an example where all the files and directories are listed under /home/ubuntu/input/ directory.

```
$ ls  -l /home/ubuntu/input/
```

ls : command name
-l : an option to display detailed properties of the files/directories
/home/ubuntu/input/ : path given as a parameter.

Wildcard character *:

* character can be used in a path to indicate all the names.

Example:

$ls –l /home/ubuntu/input/*
$ls –l /home/ubuntu/input/*.fasta

In the above example, the first command lists all the files and directories under /home/ubuntu/input/ directory, while the second command lists all the files that ends with ".fasta" extension.

1.1.2. Go to an existing directory: **cd**

The **cd** (change directory) command goes to an existing target directory given in the form of a path.

```
cd [path]
```

The following example describes how to go to /home/ubuntu and open the directory:

```
$ cd /home/ubuntu
```

1.1.3. Find out which directory you are currently in: **pwd**

The **pwd** (print working directory) command works without options and parameters and prints the current directory the user is in.

Sample run:

```
$cd /home/ubuntu
$pwd
/home/ubuntu
```

The first command (cd) goes to /home/ubuntu directory and opens it. The second command (pwd) displays in which directory the user is.

1.1.4. Create a directory: **mkdir**

To create a new directory under a certain path, **mkdir** command is used with a parameter for the name of the directory. This name could also include the entire path to where the directory will be created. If such a path is absent, the directory is created under the current working directory.

```
mkdir [directory name]
```

Sample run:

```
$mkdir code
$cd code
$pwd
/home/ubuntu/code
```

The above set of commands are run under the /home/ubuntu directory. It is always a good idea to run the pwd command to see which directory you are currently in. The first command (mkdir) creates a new directory named “code” under /home/ubuntu, the second command (cd) goes to the newly created directory and opens it. The third command (pwd) confirms that we are currently inside the newly created directory.

Important note: If we would like to do operations on files and directories that are not in the current working directory, then we need to supply the entire path of the directory or the file that we are referring to, starting from the root directory /.

There are certain shortcut paths to the current working directory and the parent of the current directory:

. : current working directory
.. : parent directory of the current working directory.

Sample run:

```
$pwd
/home/ubuntu/code
$cd ..
$pwd
/home/ubuntu
```

In the above example, the current working directory is /home/ubuntu/code. When the user wants to go to the parent directory, cd command is used by giving the shortcut name of the parent directory “**..**” as a parameter. A slash after **..** is necessary if you will add other files or directories to the parent directory path.

1.1.5. Create a text file and display its contents: **cat**

To create a text file under current working directory, the **cat** command is used with a **>** operator and a file name is given as a parameter:

```
cat >[name of the file]
```

Once the command runs, the cursor goes into the next line and you can start writing into the file. When finished, use the Ctrl+d key combinations to end. Later, if you would like to see the contents of the file, simply use:

```
cat [name of the file]
```

Sample run:

```
$ cat > myfile.txt
Hello World!
This is my file ^D
$ cat myfile.txt
Hello World!
This is my file
```

In addition, if the files are too big, as in the case of gene sequence data files and you would like to display only a few lines of the file, you may use the head command with an option for the number of lines. In the example below, only one line is displayed from myfile.txt.

Sample run:

```
$ head -n 1 myfile.txt
Hello World!
```

1.1.6. Update contents of a file with a text editor: **vim**

vim is a text editor program that allows editing and updating the contents of a file. The file name is given as a parameter. If the file does not exist, it is created anew.

```
vim [name of the file]
```

Sample run:

```
$vim myfile.txt
```

When the above command runs, the contents of the file named myfile.txt, is displayed. To edit, type **i** and the editor goes into **insert** mode. With the arrow keys, you may go in between lines and make changes. After you are done with editing, you need to exit the insert mode by entering the **esc** key. If you would like to make the updates permanent, type:

```
:wq!
```

If you don't want to keep the changes you made, type:

```
:q!
```

The text editor window will close and you will go back to the terminal view.

The vim editor will be used to write and edit Perl scripts on a remote virtual machine in the subsequent sections.

1.1.7. Remove your files: **rm**

The **rm** command permanently removes the files given as a parameter. There is no Recycle or Trash Bin. Therefore, you should be very careful when using this command. Once the file is deleted with rm, there is no way you can retrieve it back.

Sample run:

```
$ rm myfile.txt
$ ls myfile.txt
ls: myfile.txt: No such file or directory
```

In the above example, the ls command cannot find it after the deletion.

1.1.8. Copy your files: **cp**

The cp command copies one file from a source directory to a destination directory.

```
cp <source file path> <destination directory>
```

Sample run:

```
cp ./myfile.txt ./data/
$ cd ./data
$ ls -l myfile.txt
-rw-r--r--  1 esmayildirim  staff  13 Mar 21 14:15 myfile.txt
```

The above chain of commands copies the file named *myfile.txt* under current working directory to another directory named *data,* which is also under current working directory.

Now that a set of basic commands are covered, it is time to learn how to write Perl scripts and run them through the Terminal program.

2. Perl Basics

In this section, a list of basic Perl structures and program constructs necessary to design and write a gene sequencing workflow script is given.

2.1. Your first program: Print text on the screen

In the following script, the **print** function is used with a text argument to be displayed on the screen:

```
print "Hello, World!\n";
print "My name is Esma Yildirim\n";
```

The argument may include simple text, variable names or escape sequence characters inside the double quotes. The '\n' character is an escape sequence that inserts a new line on the screen so that whatever is printed after, will be printed on the next line. Statements in Perl mostly end with a semicolon.

Perl scripts are written in simple text files with a **.pl** extension**,** then, are run with the **perl** command. After writing the above script in a text file, let's say my_script.pl, the following sample output shows the execution of the script on a terminal program:

```
$perl ./my_script.pl
Hello, World!
My name is Esma Yildirim.
```

2.2. Variables

Variables are names for memory locations that hold values such as integers, floating-point numbers, characters or strings. A variable name in Perl starts with a **$** sign. Variables may or may not be declared before they are used, based on the settings of the Perl interpreter. The keyword **my** is used to declare a variable before it is used. However, its type will only be clear after it is assigned a value.

```
my $var1;
$var1 = 542;
my $var2;
$var2 = 4.5;
$var3 = "This is a string.\n";
```

In the above script, two variables are declared: $var1, $var2. However, the third variable $var3 is used without declaration. The variable names may contain letters [A-Z,a,z], underscore _ character or digits[0-9]. A variable name may start with a letter or an underscore.

In the second line, $var1 is assigned an integer value; $var2 is assigned a floating-point value and finally $var3 is assigned a string of characters.

2.3. Getting Input from the keyboard

The characters <> are used to input text from the keyboard and then the entered value must be assigned to a variable.

```
print "Enter a text: ";
my $my_text = <> ;
print "The entered text is: $my_text";
```

The above script prompts the user to enter a text, gets the input and assigns it to the declared variable $my_text. A declaration and assignment operation can happen in the same line. Then, the print function displays on the screen "The entered text is: " followed by the value of $my_text.

Sample run:

```
$ perl ./my_script.pl
Enter a text: Perl is awesome!
The entered text is: Perl is awesome!
```

Every text entered from the keyboard is followed by the enter key, therefore inserts a '\n' character at the end of the text. The chomp() function can be used to remove the '\n' character:

```
print "Enter text: ";
$text = <>;
print "$text has a new line character\n";
chomp($text);
print "$text does not have a new line character\n";
```

Sample output:

```
Enter text: Hello!
Hello!
has a new line character
Hello! does not have a new line character.
```

2.4. The system() function

The system() function allows us to run other programs from inside a Perl script. This is a very important functionality, since it is going to be used to run gene sequencing analysis tools as part of a workflow.

The function takes a string argument that includes the command line execution of the program from the terminal.

For example, if we would like to list all the files inside the directory /home/ubuntu/data/ , we may write the following Perl script.

```
system("ls /home/ubuntu/data/");
```

Sample run:

```
$perl my_script.pl
myfile1.txt  myfile2.txt  myfile3.txt
```

The above execution of the script will result in calling the ls command from inside the script and see the output of the "ls /home/ubuntu/data/" on the screen. According to that output, /home/ubuntu/data directory has three files: myfile1.txt, myfile2.txt and myfile3.txt

Another use of a Perl script is to create a directory under the current working directory:

```
print "Enter the directory name to be created: ";
```

```
my $folder_name = <>;
chomp($folder_name);
system("mkdir ./$folder_name");
system("ls –l ./");
```

Sample run:

```
$ pwd
/home/esmayildirim/dir
$ perl script.pl
Enter the directory name to be created: newfolder
total 8
drwxr-xr-x 2 esmayildirim esmayildirim 4096 Mar 28 12:46 newfolder
-rw-r--r-- 1 esmayildirim esmayildirim  131 Mar 28 12:42 script.pl
```

The above output indicates that a new directory is created via mkdir command under the current working directory and it is listed with the ls command from inside the Perl script.

2.5. Get output of commands with back quotes ``

Another way to run programs from a Perl script is to enclose the command in back quotes. Then, we can assign the returned output of the executed command and assign it to a variable.

```
$date = `date`;
print "The date is $date";
```

In the above example, the "date" command is run with back quotes and the returned output of the command is assigned to the $date variable. The value is then printed on the screen.

Sample run:

```
$ perl datescript.pl
The date is Thu Mar 28 12:57:38 EDT 2019
```

Another example that gets the output of the ls command:

```
$ls_output = `ls –l  ./`
print "The output of ls: $ls_output";
```

Sample run:

```
$ perl script.pl
The output of ls :  total 12
-rw-r--r-- 1 esmayildirim esmayildirim  111 Mar 28 12:57 datescript.pl
```

```
drwxr-xr-x 2 esmayildirim esmayildirim 4096 Mar 28 12:46 newfolder
-rw-r--r-- 1 esmayildirim esmayildirim  131 Mar 28 12:42 script.pl
```

2.6. Control structures

The control structures of Perl allow the programmer to execute different portions of the script based on a condition.

2.6.1. **if** selection structure

This structure is used with a condition that evaluates to **true** or **false** value. If the condition is true, the body of the **if** selection statement is executed:

```
if (condition)
{
  #body of the if statement
}
```

The statements inside the body are skipped if the condition evaluates to false.

Conditional operators:

- Equal: ==
- Not equal: !=
- Less than: <
- Greater than: >
- Less than or equal to: <=
- Greater than or equal to: >=

Example conditions:

- $x == $y
- $x != 5
- $var >= $var2
- 6 <=$var3
- $y == 0

The following script checks the value of a variable and if it is positive, prints a message on the screen, indicating the variable is positive.

```
print "Enter an integer: ";
my $var = <>;
chomp($var);
if ($var > 0)
{
      print "$var is positive.\n";
}
```

Sample output1:

```
$ perl ifscript.pl
Enter an integer: 6
6 is positive
```

Sampleoutput2:

```
$ perl ifscript.pl
Enter an integer: -5
$
```

In the above outputs, when the user enters 6, the condition in the if statement evaluates to true and the print statement inside the if body is executed. When the user enters -5, the condition evaluates to false; therefore the body of the if statement is skipped and we do not see any output.

2.6.2. **if**-**else** selection structure

This selection structure allows the programmer to execute the statements inside the else body, if the condition in the if statement evaluates to false. Every else must have a matching if structure.

```
if(condition)
{
   #if body

}
else
{
  #else body
}
```

We can alter the previous example to print on the screen when the integer is not positive.

```
print "Enter an integer: ";
my $var = <>;
chomp($var);
if ($var > 0)
{
      print "$var is positive.\n";
}
else
{
      print "$var is not positive.\n";
}
```

Sample outputs:

```
$ perl ifscript.pl
Enter an integer: 6
6 is positive

$ perl ifscript.pl
Enter an integer: -5
-5 is not positive
```

2.6.3. **if**-**elsif**-**else** selection structure

Multiple conditions can be checked with an if-elsif-else selection structure, however only one of the conditions can be evaluated as true. The following program checks for 0 values as well:

```
print "Enter an integer: ";
my $var = <>;
chomp($var);
if ($var > 0)
{
	print "$var is positive.\n";
}
elsif($var == 0)
{
	print "$var is zero.\n";
}
else
{
	print "$var is negative.\n";
}
```

Sample outputs:

```
$ perl ifscript.pl
Enter an integer:6
6 is positive.

$ perl ifscript.pl
Enter an integer:0
0 is zero.

$ perl ifscript.pl
Enter an integer:-5
-5 is negative.
```

2.6.4. **while** iteration structure

The while iteration structure is one of the structures that allows a block of statements to be executed multiple times as long as the while condition remains true.

```
while(condition)
{
 # while body statements

}
```

Eventually the condition should become false during the execution of the program. Therefore, one of the statements inside the while body must make sure the condition will evaluate to false at some point.

```
$counter = 1;
while($counter <=10)
{
    print "The current value of the counter is $counter\n";
    $counter = $counter +1;
}
```

In the above example, the second statement in the while body will increment the $counter variable's current value by 1. After eleven iterations, the $counter will become 11 and the condition will evaluate to false since 11 is not less than or equal to 10. In each iteration of the while, also the value of the $counter will be printed on the screen.

Sample run:

```
$ perl whilescript.pl
The current value of the counter is 1
The current value of the counter is 2
The current value of the counter is 3
The current value of the counter is 4
The current value of the counter is 5
The current value of the counter is 6
The current value of the counter is 7
The current value of the counter is 8
The current value of the counter is 9
The current value of the counter is 10
```

2.6.5. Arithmetic operations

The arithmetic operators are listed as +, -, *, /, %.
+: addition
-: subtraction

*: multiplication
/: division
%: modulus (calculating remainder after division)

Multiplication, division and modulus operations have the same precedence level but have higher precedence when compared to addition or subtraction operations. Therefore, parenthesis must be used to give precedence to addition/subtraction over multiplication/division/modulus. For example:

5 + 6 * 3 => 5 + 18 => 23
(5 + 6) * 3 => 11 * 3 => 33

Examples:

```
$var = 5 % 3 + 1;       #assigns 3 to $var
$var = $var + 1;        # adds 1 to the current value of $var
$var += 3;              #adds 3 to the current value of $var
$var++;                 # adds 1 to the current value of $var
```

Compound assignment statements like += also exist for subtraction, division, modulus and multiplication operations: -=, %/, *=, /=. In addition, there is a corresponding decrement operator (--) similar to the increment operator (++). The following three statements do the same operation: add 1 to the current value of the variable $a.

```
$a = $a +1;
$a +=1;
$a ++;
```

More examples:

```
$a = $a -1 ;
$a -=1;
$a--;

$a = $a / 3;
$a /= 3;

$a = $a %3;
$a %= 3;
```

2.6.6. Arrays

Arrays are variable names that can hold a list of elements each of which can be referred with an index. While a normal variable can hold a single value, arrays can hold multiple values. Array indices start with “0” and goes up until “the number of elements in the array – 1”. For example, to refer to the first element of a 10-element integer array we use index 0, while to refer to the last element of the array we use index 9. Array names are distinguished from

normal variable names with a preceding @ sign (e.g. @array_name), but when we refer to a single element of the array we use the regular $ with the index value in brackets (e.g. $array_name[0]). The following examples show different types of arrays with initialized set of values:

```
@my_int_array = (1, 4, 6, 2, 5);
@my_string_array = ("hello", "my" , "name", "is", "Esma", ".");
@myscalararray = ($a, $b, $c);
```

The first array is an integer array with 5 elements. The second array is a string array with 6 elements. The third array is a scalar array with 3 elements. The following script prints the elements of an integer array on the screen:

```
@array = (12, 45, 56);
# get the number of elements of the array an assigns it to $size
$size = @array;
$i = 0;

while ($i < $size)
{
      print "$array[$i] \n";
     $i++;
}
```

In the above example, the line that starts with # symbol is a comment and it is ignored by the Perl interpreter and is only there for documentation purposes.

Output:

```
$ perl array.pl
12
45
56
```

Another example assigns the output of ls command into an array. Each element of the array is a line in the ls output, which lists all the files under the current working directory:

```
$path = "./";
@array = `ls $path/*`;
$len = @array;
$counter = 0;
while($counter < $len)
{
      print "$array[$counter]";
      $counter = $counter + 1;
}
```

Output:

```
$ perl whilescript.pl
./array.pl
./datescript.pl
./ifscript.pl
./script.pl
./whilescript.pl
```

2.6.7. Passing arguments to a script from the command line

Some programs can accept arguments from the command line. For example, the **ls** command expects an argument in the form of a path. To pass an argument to a Perl script, a predefined array called @ARGV is used. The first parameter given as an argument is stored in $ARGV[0], the second parameter is stored in $ARGV[1] and so on. Example argument passing to a script:

```
$perl script.pl 73 sss "Hello!" myfile.txt
```

script.pl:

```
$first = $ARGV[0];                # assigns 73
$second = $ARGV[1];              # assigns string sss
$third = $ARGV[2];                # assigns string "Hello!"
$fourth = $ARGV[3];              #assigns string "myfile.txt"

print "$first   $ second   $third   $fourth \n";
```

Sample run:

```
$ perl argv.pl 65 fgg "HELLO" myfile.txt
65 fgg HELLO myfile.txt
```

3. Gene-sequencing Data Formats

A DNA sequence has a double helix structure that looks like a staircase consisting of base pairs. There are four types of bases in a DNA molecule: Adenine (A), Thymine (T), Guanine (G) and Cytosine (C). The bases are paired on a sugar-phosphate backbone structure. Adenine base is always paired with Thymine and Guanine base is always paired with Cytosine (Figure 3).

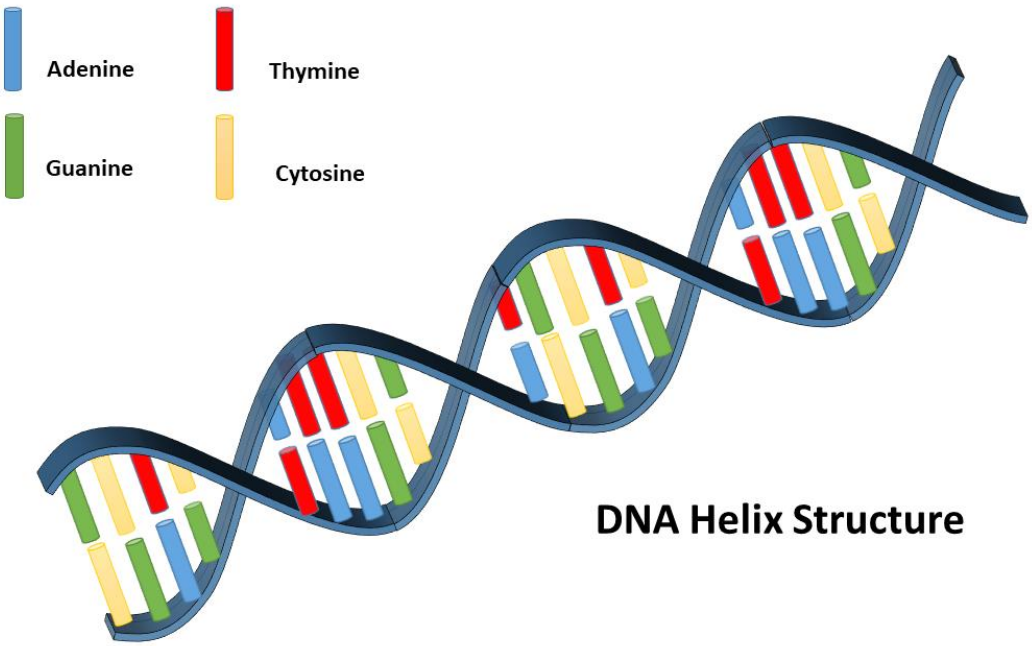

Figure 3. DNA Double Helix Structure

These base pairs (A-T and G-C) cause the two strands of the DNA to have complementary gene sequences. A sample base pair sequence:

Strand 1: ATGGTCGTTAG
Strand 2: TACCTGCAATC

The nitrogen bases can connect through hydrogen bonds (Figure 2.). Adenine and Thymine have two hydrogen bonds in between, while Guanine and Cytosine have three. On the other ends, each base is connected to the other in line via sugar-phosphate molecules.

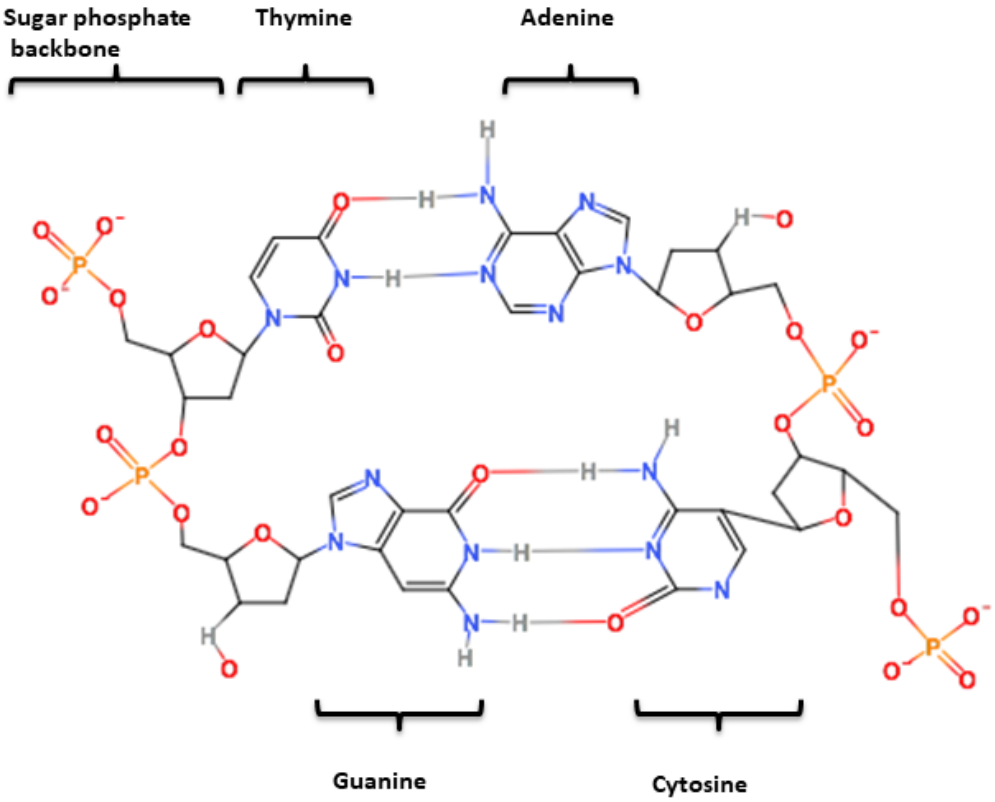

Figure 4. DNA base pair molecules

The revolutionary short read method [1] divides DNA into small samples, replicates them and reads those strands in a massively parallel high-throughput fashion. However these short reads needs to be matched into a reference genome to find certain variations in the gene structure. A reference genome is a completely sequenced genome. Figure 5 presents a very commonly used file format (FASTA) to store the reference genome. The results obtained from a sequencer machine are short reads that needs to be matched into the reference genome. Another common file format for short-read DNA is FASTQ file format. Through a set of tools that perform alignment, calibration and variant calling operations, the mutations in the gene sequence are detected and stored in a file format called VCF. In the second part of this course, we are going to learn how to use the technologies presented so far (Perl, Linux commands, cloud computing) to generate a VCF file given a FASTA reference genome file and a short-read sample of FASTQ file.

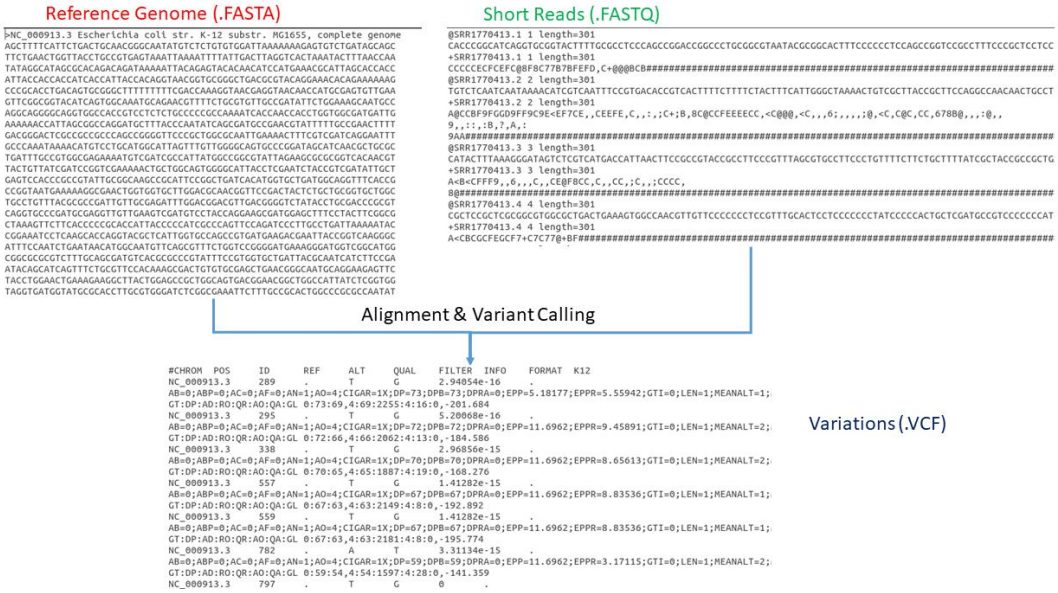

Figure 5. Gene sequencing data formats

References:

[1] An introduction to Next Generation Sequencing Technology

This OER material was produced as a result of the CS04ALL CUNY OER project.