Perl Basics

Getting started

To work with Perl on your own computer, you need to download and to install ActivePerl.[1] The ActivePerl website offers version for Windows, MAC OSX, and Linux. Note that Perl frequently comes preinstalled with Mac OS X.

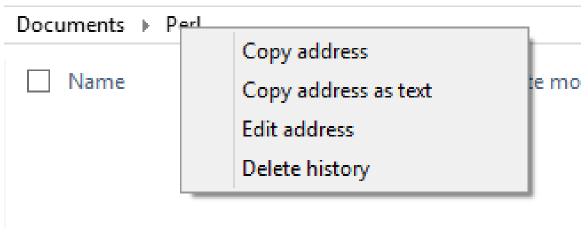
Secondly, you also need an editor in which you can write Perl programs. Under Windows, you can make use of NotePad++.[2] BBedit, TextMate[3] or Padre[4] are good editors for the Mac.

Running Perl programs in Windows

Perl programs can be run in the Windows command prompt. The prompt can be opened conveniently by making use of a Windows Batch ("bat") file. Firstly, copy the code below and paste it into NotePad++.

```
@ECHO OFF
cmd.exe /K "cd [Working Directory]"
```

The text [Working Directory] should be replaced with the full path to the directory that contains all your Perl files. This full path can be found by navigating to this directory in Windows Explorer, and, next, by right-clicking on the name of this directory. In the menu that appears, choose "Copy address as text".



Save the edited batfile as "openPrompt.bat". When you double click on the bat-file, the command prompt should opened immediately in the directory that you have specified.

Running Perl programs in Mac OS X

In Mac OS X, Perl programs can be run in the Terminal application. Open the Terminal and type in "cd " following by the full path to the directory that contains your Perl files.

---

[1] <http://www.activestate.com/activeperl/downloads>
[2] <http://notepad-plus-plus.org/download/v6.7.4.html>
[3] < http://macromates.com/download>

[4] < http://padre.perlide.org/>

To test whether or not Perl is installed correctly on your computer type in the following:

```
> perl -v
```

If Perl is indeed installed correctly, you should see a brief message containing some details about the installation.

Running Perl programs

All perl files should be saved with the .pl extension. To execute your programs, type in the keyword perl, followed by the name of your program. For example:

```
> perl program1.pl
```

Without any further measures, all the output of your program will be shown on the command line. It is also possible to write all the output to a text file. To do this, use the “>” character, followed by a filename, as follows:

```
> perl program1.pl >output.txt
```

Variables

Variables can be thought of as containers in which information can be placed temporarily. Variables in Perl always begin with a dollar sign. `$keyword` and `$fullName` are both examples of valid variable names.

Variables can be given a value. The action of giving a value to a certain variable is called assignment. In the next example, the variable `$age` is assigned the value 27 ;

```
$age = 27 ;
```

The value of a variable is always of a certain data type. The two most common data types are strings and numbers. In the example above, the value of `$age` is clearly a number. Numbers can be integers, as in the example above. In Perl, it is also possible to work with floating point numbers, such as, for instance `3.141`.

Strings are essentially sequences of characters. Strings can be created either with single quotes or with double quotes. When you use double quotes, this will have the effect that the contents of the string will be also interpreted. In the case of strings with double quotes, you can use escape characters to format the text (up to a limited degree). “\n” will create a new line in the string, and “\t” will create a tab. In addition, when you print a variable within a string which is created with double quotes, the output will contain the value that was assigned to this variable, instead of the name of the variable (as would be the case with strings with single quotes). This process is known as extrapolation. Listing 1 illustrates the differences between these two types of strings.

```
$author = "E.M. Forster" ;
$title = "A Room with a View" ;

print 'The novel $title was written
           by $author\n' ;

print "Novel:\t$title\n" ;
print "Author:\t$Author\n" ;
```

Listing 1.

```
The novel $title was written by $author \n

Novel:     A Room with a View
Author:    E.M. Forster
```

Output of listing 1.

Statements

Computer programmes basically consist of sequences of instructions to a machine. In Perl, a single instruction is called a statement. All statements must end in a semi-colon. The semi-colon is a cue which triggers the Perl interpreter to interpret and to execute the statement. The hard return at the end of the command does NOT have his effect. If the semi-colon at the end of an instruction has been omitted, the Perl interpreter will continue to read the code on the next line. This generally results in an error message. In theory, you could use hard returns in the middle of a statement. Evidently, this does not improve the readability of your code.

```
print "This is a statement" ;


print

"This will also work"

;
```

Listing 2.

Operators

Variables are normally created to make certain calculations or data manipulations possible. One of the ways in which you can manipulate the value of variables is by making use of operators. Operators are symbols that represent particular actions that can be applied to values.

In the case of numerical values, you can work with mathematical operators for adding, subtracting, multiplying, etc. Listing 3. illustrates the function of a number of mathematical operators.

```
1.      # Firstly, we create two values $a and $b
2.      # and give them numerical values
3.
4.      $a = 4 ;
5.      $b = 3 ;
6.
7.      $c = $a + b ;
8.      # $c now has value 7
9.      $c = $a * b ;
10.     # $c now has value 12
11.     $c = $a - $b ;
12.     # $c has value 1
13.
14.     $c++ ;
15.     # $c now has value 2
16.     $b-- ;
17.     # $b now has value 2 as well
```

Listing 3.

Incidentally, listing 3 also illustrates another feature of the Perl language. Lines in the code may be preceded with the hash (‘#’) symbol. When you do this, these lines will be ignored by the Perl interpreter. Lines that start with a hash are referred to as comments. Such comments are often very useful when you want to document your code. As can be seen in lines 1-2 in listing 3, comments can be used to explain what happens exactly in the code that follows.

Line 14 demonstrates the so-called increment operator. This is an operator which increases the existing value of a variable with exactly one. In other words, the function of command

```
$c = $c + 1 ;
```

is exactly the same as

```
$c++ ;
```

The decrement operator, which consists of two consecutive minus signs, decreases the existing value with exactly one. An example can be seen on line 19 of listing 3.

One of the operators that you can use in combination with strings is the concatenation operator. Its symbol is the dot (‘.’). You can use this operator to combine two existing strings into a longer string. Listing 4 is a simple example.

```
1.      $first = "Jane" ;
2.      $second = "Austen" ;
3.
4.      $full = $first . " " . $second ;
```

Listing 4.

Note that, in this example, three strings are in fact concatenated. Firstly, there are the two string that were created as `$first` and `$second`. Line 4 glues these two strings together and also places a third string in between. This third string only contains a space.

Functions

Functions, like operators, represent specific actions that can be performed on a given set of values. Functions mostly require so-called arguments. These are the values that the function operates on. In Perl, arguments must be supplied in parentheses. If there are two or more arguments, these are divided by commas.

There are a number of functions that you can use in combination with strings. The function `length()`, for instance, returns the number of characters contained within a string (including its spaces). A second function that you can use with strings is `substr()`, which can create a substring. This function demands three arguments. Firstly, you need to provide the name of the variable that you want to shorten. As a second argument, you need to give the position in the string where the substring should begin. Bear in mind that this function starts counting at 0. The third argument represents the position on which the substring should end. Listing 5 is a demonstration of these two functions.

```
$name = "Percy Busshe Shelley" ;

print "The name $name contains " . length( $name ) .
           " characters. \n";

print "The last name of the poet is " ;
print substr( $name , 13, 19 ) ;
```

Listing 5

```
The name 'Percy Busshe Shelley' contains 20 characters.
The last name of the poet is Shelley
```

Output of listing 5

The function `sprintf()` can be useful when you work with floating point numbers. This function can be used, amongst other things, to specify the number of digits after the decimal point. The first argument is a pattern which indicates how the number should be formatted. The pattern "`$.2f`", for instance, represents a number with two digits following the decimal point. The code below contains an example of the use of `sprint()`.

```
$pi = 3.1415926535 ;
$rounded = sprintf("%.2f", $pi);

# variable $rounded now has value 3.14.
```

Flow Control

By default, the Perl interpreter moves through the code in a linear way. It simply executes all the statements in the order in which they are found in the programme. In some cases, however, a statement or a set of statements only needs to be executed under a certain condition. Alternatively, a statement may also need to be repeated, as long as a certain condition applies. There are various keywords in the Perl language that you can use to specify if, and how many times, a statement needs to be executed. These keywords are referred to collectively as flow control structures.

These flow control structures generally make use of certain tests or conditions. Such a condition is in most cases an assertion which, in a given situation, is either true or false.

In the case of numerical values, you can work with the following comparison operators:

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equals
!=	Not equals

The two following operators can be used in combination with strings:

eq	Equals
ne	Not equals

If a certain section of your program must only be executed if a certain condition is true, you can use the keywords `if`, `elsif` and `else`. The basic syntax is as follows:

```
   if ( condition )
   {
       [code block 1]
   }
```

```
elsif ( condition )
{
      [code block 2]
}
else
{
      [code block 2]
}
```

If the condition that follows the `if` keyword can be evaluated as "true", code block 1 will be executed. If not, the condition that is given after elsif will be evaluated. If that second condition is found to be “true”, the Perl interpreter will execute code block 2. Code block 3 will be executed only if the two earlier conditions are both evaluated to “false”.

Listing 6 further clarifies the if/else structure on the basis of a concrete example. In a programme, it may be the case that want to print a string only if that string actually contains a value. You can test this using the `length()` function which was explained earlier. If the outcome of this function is zero for the string that is provided, you know that the string does not contain any characters. The following code assumes that a variable named `$line` has already been declared at an earlier stage in the programme.

```
if ( length($line) > 0 ) {
        print $line . "\n" ;
}
else {
        print "This string does not contain any
            characters." ;
}
```

Listing 6

Listing 7 is an example of an if/else structure which contains the `eq` operator.

```
$author = "William Shakespeare" ;

print $author . " is the author of " ;

        if ( $author eq "Jane Austen" ) {
                print "Pride and Prejudice" ;
        } elsif ( $author eq "William Shakespeare") {
                print "Hamlet" ;
        } elsif ( $author eq "Charkes Dickens") {
                print "Oliver Twist" ;
        } else {
                print "[unable to select book title]" ;
      }
```

Listing 7

Note that only the keywords `if` and `elsif` can be followed by a condition. The keyword `else` always appears WITHOUT such a condition in parentheses. The code block given after `else` contains the actions that must be performed if all the earlier conditions are false.

There are also flow control structures that can be used for repetitions of statements. One of the keywords that can be used for repetitions is `while`. As is the case for `if`, the parentheses after while should contain a test. The following code can be used to print a multiplication table of 7.

```
1.        while ( $i <= 10 ) {
2.                print "$i times 7 is " . 7 * $i . "\n"  ;
3.                $i++ ;
4.        }
```

Listing 8

Line 3 in listing 8 uses the increment value. It is absolutely necessary that the value of $i is incremented in this programme, since, when the value of $i remains unaltered, the repetition will continue endlessly.

Ensuring the correctness of your programme

As you write your code, it is easy to make typing errors or other types of mistakes. There are a number of measures that you can take to reduce the risk of errors.

In any program that you write, it is advisable to start with the following two statements:

```
use warnings;
use strict ;
```

There can basically be two types of errors in your file. Firstly, there are the so-called 'fatal' errors. These errors cause the Perl interpreter to terminate the programme when it is run. There can also be errors which do not result directly in an interruption of the execution of the program, but which may nevertheless be considered bugs in the code. When the `use warnings` option has been selected, the Perl interpreter will notify the programmer of such potential problems during the execution of the programme.

`use strict` will have the effect that you need to declare all of your variables before you can use them. Declaring variables, in this context, can be done with the keyword `my`, directly before the variable name. With this measure in place, you can only use variables in your program which have actually been declared. If the interpreter comes across an unknown variable name, the execution of the programme will be terminated, and an error message will appear. Such formal declarations can help you to ensure that you do not make any typing errors in the names of variables. Without `use strict`, a typing error in the name of a variable leads to the creation of a new variable, and the programme may consequently not function as it should.

In short, `use strict` and `use warnings` both help you to avoid mistakes, and to debug your programme if there are mistakes.

Reading text

When you have a text file on your computer with the name “shelley.txt”, the full contents of this file can be read using the following code.

```
1.        open( IN , "shelley.txt" ) or die("Can’t open file");
2.
3.        while ( <IN> ) {
4.
5.               print $_   ;
6.
7.        }
8.        close ( IN ) ;
```

Listing 9

In listing 9, the text file is firstly accessed using the open() function. This function requires two values. The second of these is the name of the file that you want to open. The first argument that you must provide is the name that you want to assign to the so-called file handle. A file handle can be seen as a specific type of variable. It essentially represents the file which is accessed using the open() function. When you use the open() function, this will firstly prompt the Perl interpreter to access the file that you have mentioned. If the access is successful, Perl will create a file handle under the name that you have supplied. When you use the die() function, directly after the open() function, this will have the effect that an error message will be printed when the program, for some reason, has difficulties accessing the file. The error message will be the string that you have supplied as an argument to the die function.

After you have successfully created a file handle, the contents of the file can be obtained by using the file handle, surrounded by angular brackets, in combination with the keyword `while`, as can be seen on line 3 of listing 9. By default, the file is retrieved line by line. Each line which is retrieved in this way will initially be assigned to Perl’s so-called default variable, which is `$_`.

It is generally advisable to close a file handler when it is no longer needed. You do this with the close() function.

Regular expressions

A regular expression is basically a text pattern, consisting of actual characters or of generic representations of specific types of characters. Such regular expression can be used to test if specific text fragments that you are working with have certain formal characteristics.

Generic representations of characters are referred to as character classes. The following character classes may be used:

.	Any character, except the newline.
\w	Any alphanumerical character: all 26 alphabetical characters, both in upper case and in lower case, all numbers and the underscore.

\d	Any digit.
\s	White space: the space, a tab or a newline character.
[..]	If classes such as \w or \d are too broad, and if only a limited number of characters are allowed on a specific position in a string, this set of possible characters may be supplied in square brackets.

You can also use quantifiers to specify the number of times a character or a pattern should occur.

{n,m}	Pattern must occur a least n times, at most m times.
{n,}	At least n times
{n}	Exactly n times
?	Is the same as {0,1}
+	Is the same as {1,}
*	Is the same as {0,}

Thirdly, you can also use anchors in regular expression. Anchors do not represent actual characters, but locations within strings.

\b	A word boundary
^	The beginning of a pattern
$	The end of a pattern

Different regular expressions can also be combined using the vertical bar ('|').

As was explained above, characters such as the question mark and the asterisk have a special function within regular expressions. If you actually want to search for these characters themselves, these need to be escaped, using the backslash. An alternative is to place these characters within square brackets, as, in this context, these special characters do not have their regular function.

Using regular expressions in Perl

In Perl, you can use the =~ operator to test a given string against a regular expression. Note that regular expressions are always surrounded by two forward slashes.

```
$text = "The sun shone, having no alternative, on the
nothing new." ;

```

```
if ( $text =~ /\bsun\b/i ) {
        print "There is a match!" ;
}
```

Listing 10.

If you write the character ‘i’ directly after the second forward slash, the test will take place in a case-insensitive manner.

In some cases, it may be necessary not only to test if a string contains a pattern, but also to retrieve the characters in the string that are actually matched by the regular expression. This can be accomplished by working with parenthesis in the regular expression. These parentheses will have the effect that the characters in the string which match the regular expression will be saved as the value of a variable with the name `$1`. Using parentheses can be very useful for creating substrings. If the regular expression also contains a second set of parentheses, the characters that match this second sub-pattern will become available as `$2`, etc.

Listing 11 illustrates this principle. In short, it is a programme which can extract the direct speech from a longer sentence.

```
$text = "\"Oh, good gracious me!\" said Lucy, suddenly
collapsing and again seeing the whole of life in a new
perspective." ;

if ( $text =~ /"(.+)"/) {
        print $1 ;
        # The value of $1 is :
        # Oh, good gracious me!
}
```

Listing 11.

In Perl, regular expressions can also be applied usefully in ‘find and replace’ operations. The syntax for such operations is as follows:

```
s/[regular expression]/[new text]/g;
```

The ‘s’ at the beginning stands for ‘substitution’. This command will have the effect that the regular expression that is given in between the first and the second forward slashes will be replaced by the text that is given in between the second and the third forward slashes. The character ‘g’ will ensure that all the characters that match the regular expression will be replaced. If the ‘g’ is omitted, only the first match will be substituted. The substitution command can be applied to an existing string through the `=~` operator. Listing 12 contains an illustration.

```
$sentence = "The novel Ullyses was first published in
1922." ;
$sentence =~ s/[a-zA-Z .]*//g ;
# the value of $sentence is now 1922
```

Listing 12

Arrays

The variables that have been discussed so far all contained a single value only. In the Perl language, variables with a single value are referred to as scalar variables. It is also possible to create variables which can contain multiple values. One example of this is the array.

Arrays are always preceded by the at-character (‘@’). They can be created as follows (line 1 and 2):

```
1.     my @colours = ("blue", "yellow", "green", "red",
2.     "purple");
3.
4.     print $colours[1] . "\n" ;
5.     print $colours[3]  . "\n" ;
6.
7.     print $colours[-1] . "\n" ;
8.     print $colours[-2] . "\n" ;
9.     push ( @colours, "orange" ) ;
10.    print $colours[5] . "\n" ;
11.
```

Listing 13

```
yellow
red
purple
red
orange
```

Output of listing 13

When arrays are created, Perl numbers the items in the array automatically, according to the order in which they are given. These numbers will function as keys that can be used to access the individual items in the array. It is important to bear in mind that Perl starts counting at 0. In the example above, the item “blue” will have key “0”, and item “red” will have the value “3”. Items in the array can be accessed by using the name of the array in combination with the key of the item that you want to see in square brackets. For examples, see lines 4-10 in listing 13. If you want to add items to an existing array, you can use the `push()` function (as is illustrated on listing 13, line 9).

Importantly, when you refer to an individual item in an array, Perl views this as a scalar variable. This is why there is a dollar sign in front of a variable such as `$colours[0]`, rather than the ‘@’ character.

You can also use negative numbers as keys to the array. When you do this, Perl will start accessing items at the end of the array.

Next to the method which is shown on line 1 of listing 13, arrays can also be created with the `split()` function. This function takes two arguments. The second argument is the string that needs to be split. The first argument specifies the character or characters on which the string needs to be divided. You can provide either a string (in quotes) or a regular expression (in between forward slashes). Listing 14 further clarifies how the functioning of `split()`.

```
$line = "If music be the food of love, play on" ;
```

```
2.
3.          @words = split( /\s/ , $line ) ;
4.
5.          print $words[0] ; # "If"
6.          print $words[7] ; # "play"
7.
8.          @words = sort( @words)  ;
9.
10.         foreach my $w ( @words) {
11.                 print $w . " " ;
12.         }
13.
14.         # output: If be food love, music of on play the
```

Listing 14

The items in an array can also be sorted, using the `sort()` function. See line 8 of listing 14. By default, this function will sort the items alphabetically. The `sort()` function returns a sorted array. This whole array also needs to be saved. In the code above, the sorted array overwrites the existing array `@words`. You could also have assigned the result to a new array, such as, for instance, `@sorted`. Bear in mind that simply using the `sort()` function, without saving the result, will not sort the array.

You can use the keyword `foreach` to navigate through all the items in an array, as shown on lines 10-12. The `foreach` keyword will create a number of iterations, as many as there are items in the array. Directly after `foreach`, you need to supply the name of a variable. During each cycle, the variable that has been supplied will be assigned one of the values that are contained in the array.

Hashes

Hashes are very similar to arrays. Like arrays, hashes are variables that can hold multiple values. In the case of arrays, Perl creates the keys automatically. In case of hashes, however, the keys explicitly need to be associated with values by the programmer. Importantly, each key in the hash must have a unique value. Hashes can be applied usefully in word frequency lists. The key, in this case, can be the word, and the value can be the number of times this particular word occurs in a text.

The names of hashes always begin with a percentage sign, ('%').

The individual items can be accessed by using the name of the hash, directly followed by the key of the item in curly brackets. Note that references to individual items begin with a dollar sign, and not with a percentage sign.

Listing 15 gives you an impression of how hashes function.

```
my %frequency ;

$frequency{"the"} = 5790 ;
$frequency{"of"} = 5634 ;

print $frequency{"the"} ;
## This will print 5790
```

Listing 15

In realistic text mining programmes, frequency values will usually not be typed in manually. In most cases, such programmes works with a function such as `split()` which can divide the text into its separate words. During this process, as more words are found, the frequency values associated with the various keys in the hash (i.e. the words) will be updated using the increment value.

Listing 16 is an example of a simple tokenisation programme based on this principle.

```
1.      use strict ;
2.      use warnings ;
3.
4.      my ( @words, $w , %freq ) ;
5.
6.      open ( FILE , "ARoomWithAView.txt") or die "Can't open
7.      file!" ;
8.
9.      while( <FILE> ) {
10.
11.           @words = split( /\s+/ , $_ ) ;
12.           foreach my $w ( @words ) {
13.
14.                if( $w =~ /(\w+)/ ) {
15.                     $freq{ $1 }++ ;
16.                }
17.           }
18.     }
19.
20.     close (FILE) ;
21.
```

Listing 16

The programme roughly works as follows.

After the text file has been opened using `open()`, on line 6, the file will be read line by line with `while(<FILE>)` on line 9. Note that you are free to name the file handle as you like. While a previous example used `IN`, this code uses `FILE`.

On line 11, the function `split()` is applied to the default variable `$_` which, during each cycle of the while loop, represents a different line from the text file. The first argument of the split() function is the regular expression `/\s+/`. This pattern represents an occurrence of one or more space characters. On the basis of this pattern, the line is divided into separate chunks.

Each item that is found in this way will firstly be tested against another regular expression. It is assumed in this code that a word must consist of at least one alphanumerical character. If this is indeed the case, the word that is found will be used as a key in the hash $freq. The value associated with this key will be increased using the increment operator (listing 16, line 15).

At the end of this programme, the keys in the hash `$freq` represent all the unique words that occur in the text. The values associated with these keys are the number of times these words occur in the text.

You can also view all the items in a hash. As is the case for arrays, you can do this using the `foreach` keyword. The only difference is that you need to provide the word “keys” in front of the name of the hash. The items in the hash can also be sorted, either by key or by value. To sort by key, you can simply use the word “sort” directly before keys. Sort by value is slightly more complicated.

Basically, you need to use the construction { $hash{$a} <=> $hash{b} } directly after “sort”. See listing 17 for an illustration.

```
use strict ;
use warnings ;

my %words ;

$words{"the"} = 5061;
$words{"and"} = 4956 ;
$words{"of"} = 4320 ;

print "\n\nFrequency list, sorted by word.\n" ;

foreach my $c ( sort keys %words ) {

     print $c . "\t" . $words{$c} . "\n" ;

}

print "\n\nFrequency list, sorted by frequency.\n" ;

foreach my $c ( sort { $words{$a} <=> $words{$b} }
keys %words ) {

     print $c . "\t" . $words{$c} . "\n" ;

}
```

Listing 17