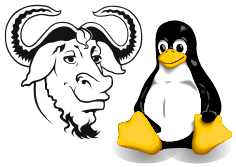

An Introduction to Perl

Log into your Linux account. If you forgot your password, come over to me to reset it to something of your choice.

Open the editor of your choice; available options are:
`emacs`, `xemacs`, `vi`, `gvim`, `gnp`, `pico`, `gedit`, and many others.

1 Background

1.1 Reading Perl documentation

There is a huge amount of documentation about Perl installed on your computer; if you printed it all out, it would fill many books. There is a cross-platform program called `perldoc` that allows you to examine and search the documentation.

Reading about basic Perl syntax: `perldoc perlsyn`

Figuring out which documentation to read: `perldoc perl`

How to search the FAQ: This searches the headings of the questions for text that matches ⟨*string*⟩.

`perldoc -q` ⟨*string*⟩

As you will see later, the string could be a regular expression.

How to read about built in Perl functions: The Perl built-in functions are all described in detail in the `perlfunc` man page. But there is a handy tool for reading about a single built-in function:

`perldoc -f` ⟨*function*⟩

This works in Windows as well as Linux.

Finding out about operators: The man page for Perl operators is called `perlop`.

Finding out about statements: The man page for Perl syntax is called `perlsyn`. To read about *backwards* statements, see the section there called *simple statements*.

Finding out about regular expressions: `perlrequick` and `perlretut` are tutorials for Perl regular expressions. The reference for regular expressions is called `perlre`.

Nick's Handy Perl summary A good overview is at `http://nicku.org/snm/lectures/perl/perl.pdf`

1.2 Reading from Standard Input

To read a line from standard input, simply do something like this:

```
my $input = <STDIN>;
chomp $input; # to remove the newline that will be at the end of the line
```

1.3 The special variable $_

In Perl, most built in functions, statements and operators work with a special variable called `$_`. For example, the following `while` loop reads standard input one line at a time, and prints that line:

```
while ( <STDIN> ) {
    print;
}
```

Notice that the `while` loop reads one line into `$_` at each iteration. The built in `print` statement prints the value of `$_` if you do not tell it to print anything else.

1.4 About the angle operator: <>

Many of the programs you will write will use the *angle operator*, '`<>`'. Since we need it here, I'd better explain what it does.

First, let us understand the terms *command line* and *command-line arguments*. Suppose you have a program called `angle-brackets.pl`. If you execute it like this:

```
angle-brackets.pl
```

then you have no *command line arguments* passed to the program. However, if you execute it like this:

```
angle-brackets.pl file_1 file_2 file_3
```

then the *command line* has three *arguments*, which here, happen to be the names of files.

Now lets try to understand what happens when you do something like this in a program:

```
@array = <>;
```

Here I've written in pseudocode what happens:

```
if there are no command line arguments,
    while there are lines to read from standard input
        read each line from standard input
        put it into the array as the next element
else
    for each command line argument
        open the file
        while there are lines to read
            read each line from the file
            put it into the array as the next element
        close the file
```

So if `file_1` contains:

```
line 1 of file_1
line 2 of file_1
```

and `file_2` contains:

```
first line of file_2
second line of file_2
```

and the program `angle-brackets.pl` contains:

```
#! /usr/bin/perl
@array = <>;
print @array;
```

If you run the program like this:

```
angle-brackets.pl file_1 file_2
```

then the output will look like this:

```
line 1 of file_1
line 2 of file_1
first line of file_2
second line of file_2
```

Specifically, the first element of the array is `$array[0]`, and has the scalar value "`line 1 of file_1\n`"; the second element of the array is `$array[1]`, and has the scalar value "`line 2 of file_1\n`", while the last element of the array is `$array[3]`, which contains the scalar value "`second line of file_2\n`".

The `<>` operator is most commonly used in a `while` loop, rather like this:

```
while ( <> ) {
    ⟨commands to be executed for each line of input⟩
}
```

1.5 Finding matching lines

Backwards `if` statement: With Perl, there is more than one way to do it (TIMTOWTDI™). In particular, as well as normal `while`, `if` statements, you can put them *backwards* if you want to. The backwards `if` statement works like this:

⟨*expr1*⟩ `if` ⟨*condition*⟩`;`

Note that braces and parentheses are not used. Backwards statements are just for when you want to do *one* thing if an expression is true.

Matching Lines: Use regular expressions to find matching lines. This little program will print all lines on the input that contain the string "`string`":

```
while ( <> ) {
    print if /string/;
}
```

2 What to do

1. Write a program that calculates the circumference of a circle with a radius of 12.5. The circumference is $C = 2\pi r$, and $\pi \approx 3.1415927$.

2. Modify the program you just wrote to prompt for and read the radius from the person who runs the program. See section 1.2 on page 2.

3. Write a program that prompts for and reads two numbers, and prints out the result of multiplying the two numbers together.

4. Write a program that prompts for and reads a string and a number, and prints the string the number of times indicated by the number, on separate lines. Hint: read about the "`x`" operator.

5. Write a program that works like `cat`, but reverses the order of the lines. It should read the lines from files given on the command line, or from standard input if no files are given on the command line. Hints: read about the built-in `reverse` function. You will want to use an array. Read sections 1.3 and 1.4 on page 2.

2.1 Regular Expresssions

You will need to do a little research to answer these questions. Read section 1.5 on the preceding page, then read the Perl documentation about regular expressions I pointed to in section 1.1 on page 1.

1. Write a program that will read one or more files on the command line, and print all lines from these files that have at least one '`z`' followed by any number of '`y`'s. So it would match lines containing any one of these words:

   `z breezy buzzy cozy crazy dizzy enzyme frenzy fuzzy hazy jazzy lazy lazybones Lizzy snazzy`

2. Write a regular expression that will recognise a floating point number that Perl will recognise. Here are a few examples of literal floating point numbers that you can use in a Perl program:

`+1`	A leading plus or minus is optional, so is the decimal point
`1.25`	
`7.25e45`	7.25×10^{45}
`-6.5e24`	-6.5×10^{24}
`-12e-24`	-12×10^{-24}
`-1.2E-23`	-1.2×10^{-23} which equals -12×10^{-24}

3. Write a short program that reads files and prints all valid floating point numbers contained in them, one to a line, but does not print any other output. Read about using parentheses for *capturing* matched values.