# Introduction to Perl for Programmers (Exercises)

**Damian Conway**

## 1. *Installing a modern Perl*

Install Perl v5.32 on your machine, preferably by first installing perlbrew or berrybrew.

## 2. *Reading the fine manual*

Spend a little time exploring the Perl documentation, either locally (via `perldoc perl`) or on-line (via http://perldoc.perl.org/perl.html).

## 3. *Messing about with some Perl code*

Open up the *"Hello Worlds"* example code in your editor and play around with the code:

a. Add the (sub)planet Eris *(which is just as big as Pluto, so just as worthy to be listed)*
b. What happens if you remove the call to `sort` within the `for` loop?
c. What happens if you `sort` the list of planets being assigned to `%worlds`?
d. What happens if you remove the `use v5.32;` statement?
e. What happens if you remove the `my` in front of the `%worlds` variable?
f. What happens if you change all the double-quotes (`"`) to single-quotes (`'`)?
g. What happens if you remove the `my` in front of the `$planet` variable in the `for` loop?
h. What happens if you remove the entire `my $planet` in the `for` loop?
i. What happens if you change the call to `keys` with a call to `values`?

## 4. *Declaring variables*

Set up a new Perl source file and declare some variables in it.
Initialize the variables to different kinds of values (*e.g.* various types of strings and numbers).
What happens if you assign a string to a variable that was initialized with a number (and vice versa)?
Try declaring both `my` and `our` variables inside and outside different code blocks (`{...}`), and then try accessing them in various places (*e.g.* print out their values using `say`).

## 5. *Generating statistics*

Write a Perl program that reads in a sequence of numbers (entered one per line) and, when the input is complete, then prints out:

a. the range of values (minimum to maximum),

b. the *average* (arithmetic mean),

c. the *median value* (the value with an equal number of values above it and below it),

d. the *modes* (the most frequently occurring values).

## 6. *Processing queues of data*

Write a Perl program that reads in a planet's name and prints out the corresponding description.

Multiple planet names (or none at all) may be specified on each input line, but only a single description (*i.e.* for the next planet in the queue) should be printed after each input.

If a planet name is entered, but is currently already in the queue, the repeated name should be discarded.

## 7. *Reporting the filesystem*

a. Write a Perl program that reads in the name of a file in the current directory and then prints out all other files in the current directory that are larger than the first file.

b. Now modify the program to print out files which are either not readable, or else both larger *and* more recently modified than the first file. Each file should be marked as readable or not.

c. Now modify the program to print out not just the name of the matching files, but also the first line of their contents.

d. Now modify the program to save the information into a new file, as well printing it on STDOUT.

## 8. *Implementing a shell tool*

a. Write a program that re-implements the basic functionality of the standard UNIX `uniq` tool. That is: a utility that reads lines from STDIN and outputs only non-repeated consecutive lines to STDOUT.
See: http://man.he.net/?topic=uniq&section=all

b. Now improve on the original, by creating a variation that removes repeated lines even if they occur *non-consecutively* at some later point in the input.

c. Now add a `-i` option to look for repeated lines case-insensitively.

d. Now add a `-u` option to output only lines that were not repeated in the input.
*(Extra points for outputting these lines in the same order they were input.)*

e. Now add a `-d` option to output only lines that *were* repeated in the input.
*(Once again, extra points for outputting these lines in the same order they were input.)*

**Clarification:**

| | |
|---|---|
| `uniq` | prints only the first instance of only those input line that occur *once or more* |
| `uniq -u` | prints only the first instance of only those input line that occur *exactly once* |
| `uniq -d` | prints only the first instance of only those input line that occur *more than once* |

## 9. *Improving a Perl tool*

The `perldoc` utility doesn't always make it easy to find the Perl documentation you're looking for. Often you need to specify a special option (and occasionally even a different tool) in order to find information about different types of constructs:

| | |
|---|---|
| The `-f` option for builtin functions: | `perldoc -f push` |
| The `-v` option for variables: | `perldoc -v %ENV` |
| No option for Perl manpages: | `perldoc    perlre` |
| No option for module names: | `perldoc    Data::Dumper` |
| The `-q` option for Perl FAQs: | `perldoc -q epoch` |
| Another tool entirely for Perl utilities: | `man        a2p` |

And then, in many cases, the results will instantly scroll away upscreen, because you forget to run it through a pager. Again.

Write a Perl program that accepts an single argument (*e.g.* `push` or `%ENV` or `perlre` or `Data::Dumper` or `epoch` or `a2p`, *etc.*) without any special options, and then tries all of the above alternatives, in the order listed, returning the output of the first alternative that produces a useful result.

Bonus points if your program pipes the result through the user's nominated pager.

Extra bonus points if your program invokes the user's pager only when the output exceeds a full screen of text.

## *10. Exploring references*

Write some Perl code that uses references in simple ways.

Declare scalar, array, and hash variables, and then take references to each of them, assigning each reference to another scalar variable.

Print out the contents of the original variables, via the references you have have taken.

When accessing arrays and hashes via a reference, try out both the old circumfix syntax (*e.g.* `${$array_ref}[0]`, `@{$array_ref}[1,2,3]`, `keys %{$hash_ref}`, *etc.*) and the new postfix syntax (*e.g.* `$array_ref->[0]`, `$array_ref->@[1,2,3]`, `keys $hash_ref->%*`, *etc.*)

## *11. Refactoring with subroutines*

Choose one (or more) of the previous exercises you've completed and refactor the code into an appropriate number of subroutines.

## *12. Managing a tree*

Implement a simple *n*-ary tree with (at least) the following API:

```
sub new_node     ($name, $data)       # allocates ID, returns node
sub find_by_ID   ($root_node, $ID)    # returns one node
sub find_by_name ($root_node, $name)  # returns list of nodes
sub print_tree   ($root_node)
sub insert_child ($root_node, $parent_ID, $child_node)
```

(*Hint: anonymous hashes make good tree nodes; anonymous arrays make good child lists.)*

## 13. Renumbering lists

Write a program that takes an input text, detects numbered bullet points within that text, and renumbers those bullet points sequentially from 1.

For example, given:

```
2. Learn Perl
7. Implement deep-learning
   text editor
4. ????
11. Profit!
```

your program should produce:

```
1. Learn Perl
2. Implement deep-learning
   text editor
3. ????
4. Profit!
```

## 14. Parsing with regular expressions

Write a Perl program that reads in calendar dates (one per line) in as many formats as possible (*e.g. 2005-05-02*; *2/3/07*; *Feb 2, 1996*; *etc.*) and then outputs each date in standard ISO 8601 extended format (*e.g. 2005-05-02*; *2007-03-02*; *1996-02-02*; *etc.*).

Where a date is internationally ambiguous (*e.g. 6.7.2003*), report this fact then list both possibilities (*e.g.* both *2003-06-07* and *2003-07-06*).

## 15. Formatting columns

Write a program that reads in lines of text from STDIN and formats them side-by-side into as many columns as will fit in a total width of 80 characters. For example:

```
> grep '^[a-z].*\(k\)\1' < /usr/share/dict/words | columns
bekko        darabukka  halukkah        knickknackery  steekkan
bookkeeper   dikkop     jackknife       knickknackish  stockkeeper
bookkeeping  dukker     kakkak          knickknacky    stockkeeping
```

Modify the program to allow the width of the gap between the columns to be specified as a command-line option.

Now modify the program so that the total width used is the current width of the terminal window, rather than being fixed at 80 characters.

## 16. Cheating at Scrabble™

Write a program that takes a sequence of up to seven characters as its argument and prints out all the valid Scrabble™ words that can be made from those characters. Allow up to two underscore characters, which stand for blanks that can be replaced by any character. Replacement characters for blanks should be shown in lowercase. For example:

```
> scrabwords bkj_xys
SKYBoX
BYKeS
BoSKY
BuSKY
KYBoS
BYKe
BYeS
BaSK
BaYS
BiSK
BoSK
BoXY
BoYS
BuSK
BuSY
BuYS
JYnX
...etc.
```

Now modify the program so that it sorts the words according to the total value of their letters in Scrabble™ . For example:

```
> scrabwords bkj_xys
21  SKYBoX
20  JYnX
20  JaXY
17  JoKY
15  BoXY
14  JaKS
13  BYKeS
13  BoSKY
13  BuSKY
13  KYBoS
13  JaSY
13  JaYS
13  JoYS
...etc.
```

## 17. Exploring Raku

Either install Raku on your own machine (start at: https://raku.org/downloads/)

or just use the Raku Online Compiler, IDE, Editor, Interpreter and REPL
at https://repl.it/languages/raku

Having set up a Raku environment, write yourself some simple programs in Raku.

You might like to start with the traditional “Hello World”, or you may prefer to try to reimplement one of the preceding exercises in Raku.

Alternatively, you may prefer to load up one or more of the sample Raku programs provided and experiment with modifying or extending them.