Andrew Shitov

Raku One-Liners

Getting the most of Raku's expressive syntax for your daily routines

DeepText — 2019

Raku One-Liners

Getting the most of Raku's expressive syntax for your daily routines

© Andrew Shitov, author, 2019

In this book, you will find a lot of short programs, so short that they can be written in a single line of code. The seven chapters will guide you through Raku's syntax elements that help to create short, expressive, but still useful programs.

It is assumed that the reader knows the basics of the Raku programming language and understands programming in general.

Published on 18 October 2019

Published by DeepText, Amsterdam
www.deeptext.media

ISBN 978-90-821568-9-8

Contents

Chapter 3
Working with Numbers

Chapter 4
Working with Strings

Chapter 5
Working with Dates

Chapter 6
Raku Syntax

Chapter 7
Raku Golf

Appendix on Compiler Internals

Preface

Dear reader,

You are reading a book about the Raku programming language. This language has appeared as a rename of Perl 6 in October 2019.

Like its parent, Perl 5, the Raku language keeps the spirit of being a powerful tool in many areas, from devops programs for configuration management through different command-line applications to concurrent web servers.

In this book, you will find a number of short programs that you may want to use in your daily practice. You will also find a number of one-line snippets that can enter into your bigger programs.

The goal of the book is not to give a copy-and-paste list of coding examples, but to explain the various bits of Raku that help to use the language more efficiently.

To run the program examples from the rest of the book, you need to download and install the most recent Rakudo Star compiler pack from its website, rakudo.org. If you are using the previous, Perl 6-based compiler, create an alias in your `.profile` file so that you can use the `raku` command to run the compiler:

```
alias raku=perl6
```

I wish you a pleasant journey in the magic Raku language.

Andrew Shitov
Amsterdam, 18 October 2019

Chapter 1
Command-Line Options

Using command-line options

Let us talk about the command-line options that the Rakudo[1] compiler offers to us.

-e

The first option to know when working with Raku is `-e`. It takes a string with your Perl 6 one-liner and executes it immediately.

For example, print the name of the current user:

```
$ perl6 -e'$*USER'

ash
```

-n

This option repeats the code for each line of input data. This is quite handy when you want to process a file. For example, here's a one-liner that adds up the values in a row and prints the sum:

```
$ raku -ne'say [+] .split(" ")' data.txt
```

If the `data.txt` file contains the following:

```
10 20 30 40
1 2 3 4
5 6 7 8
```

[1] Rakudo (rakudo.org) is an implementation of Raku. The rest of the book assumes you are using the Rakudo compiler and run it as `raku` from command line. If you have an older version, which has the `perl6` executable, make an alias in your `.profile`: `alias raku=perl6`.

then the result of the one-liner is:

```
100
10
26
```

There's no difference whether you use shell's input redirection or not; the following line also works:

```
$ raku -ne'say [+] .split(" ")' < data.txt
```

Make sure you place the `e` option the last in the list (so, not `raku -en'...'`) or split the options: `raku -n -e'...'`.

-p

This option is similar to `-n` but prints the topic variable after each iteration.

The following one-liner reverses the lines in the file and prints them to the console:

```
$ raku -npe'.=flip' data.txt
```

For the same input file, the result will look like this:

```
04 03 02 01
4 3 2 1
8 7 6 5
```

Notice that you have to update the `$_` variable, so you type `.=flip`. If you only have `.flip`, you only reverses the string, but the result is not used and the *original* line is printed.

An equivalent program with `.flip` and with no `-p` looks like this:

```
$ raku -ne'.flip.say' data.txt
```

Examples of short one-lines

To warm up, let's start with a few simple one-liners for working with files. (There's also the whole Chapter 2 which is about working with files).

Double-space a file

```
$ raku -npe's/$/\n/' text.txt
```

Remove all blank lines

```
$ raku -ne'.say if .chars' text.txt
```

Depending on how you define 'blank', you may want another one-liner that skips the lines containing whitespaces:

```
$ raku -ne'.say if /\S/' text.txt
```

Number all lines in a file

```
$ raku -ne'say ++$ ~ ". " ~ $_' text.txt
```

This code, probably, requires a comment. The `$` variable is a *state* variable and it can be used without declaration.

Convert all text to uppercase

```
$ raku -npe'.=uc' text.txt
```

Strip whitespace from the beginning and end of each line

```
$ raku -npe'.=trim' text.txt
```

Print the first line of a file

```
$ raku -ne'.say ; exit' text.txt
```

Print the first 10 lines of a file

```
$ raku -npe'exit if $++ == 10' text.txt
```

This time, the postfix `++` operator was applied to the `$` variable.

Reading files with $*ARGFILES

`$*ARGFILES` is a built-in dynamic variable that may be handy when working with multiple input files.

How do you read two or more files passed in the command line?

```
$ raku work.pl a.txt b.txt
```

If you need to process all files together as if they are a single data source, you could ask the variable to do the job in a one-liner:

```
.say for $*ARGFILES.lines
```

Inside the program, you don't have to think about looping over the files; `$*ARGFILES` will automatically do that for you.

If there are no files in the command line, the variable will be attached to STDIN:

```
$ cat a.txt b.txt | raku work.pl
```

Handy indeed, isn't it?

$*ARGFILES and MAIN

I also have to warn you if you will want to use the `$*ARGFILES` variable in bigger programs. Consider the following example:

```
sub MAIN(*@files) {
    .say for $*ARGFILES.lines;
}
```

In the recent versions of Raku, `$*ARGFILES` works differently *inside* the `MAIN` subroutine and *outside* of it.

This program will perfectly work with the earlier versions (before and including Rakudo version 2018.10). Starting from Rakudo Star 2018.12, `$*ARGFILES`, if used inside MAIN, is always connected to `$*IN`.

Chapter 2
Working with Files

Renaming files

Let us solve a task to rename all the files passed in the command-line arguments and give the files sequential numbers in the preferred format. Here is an example of the command line:

```
$ raku rename.raku *.jpg img_0000.jpg
```

In this example, all image files in the current directory will be renamed to img_0001.jpg, img_0002.jpg, etc.

And here's the possible solution in Raku (save it in `rename.raku`):

```
@*ARGS[0..*-2].sort.map: *.Str.IO.rename(++@*ARGS[*-1])
```

The pre-defined dynamic variable `@*ARGS` contains the arguments from the command line. In the above example, the shell unrolls the `*.jpg` mask to a list of files, so the array contains them all. The last element is the renaming sample img_0000.jpg.

If you are familiar with C or Perl, notice that the variable is called `ARGS`, not ~~`ARGV`~~.

To loop over all the files (and skipping the last file item with the file mask), we are taking the slice of `@*ARGS`. The `0..*-2` construct creates a range of indices to take all elements except the last one.

Then, the list is sorted (the original `@*ARGS` array stays unchanged), and we iterate over the file names using the `map` method.

The body of `map` contains a `WhateveCode` block (see Chapter 6); it takes the string representation of the current value, makes an `IO::Path` object out of it, and calls the `rename` method. Notice that the `IO` method creates an object of the `IO::Path` class; while a bare `IO` is a *role* in the hierarchy of the Raku object system.

Finally, the increment operator `++` changes the renaming sample (which is held in the last, `*-1`st, element of `@*ARGS`). When the operator is applied to a string, it increments the numeric part of it, so we get img_0001.jpg, img_0002.jpg, etc.

Merging files horizontally

Let us merge a few files into a single file. The task is to take two (or three, or more) files and copy their contents line by line. For example, we want to merge two log files, knowing that all their lines correspond to each other.

File a.txt:

```
2019/12/20 11:16:13
2019/12/20 11:17:58
2019/12/20 11:19:18
2019/12/20 11:24:30
```

File b.txt:

```
"/favicon.ico" failed (No such file)
"/favicon.ico" failed (No such file)
"/robots.txt" failed (No such file)
"/robots.txt" failed (No such file)
```

The first one-liner illustrates the idea:

```
.say for [Z~] @*ARGS.map: *.IO.lines;
```

It is assumed that the program is run as follows:

```
$ raku merge.raku a.txt b.txt
```

For each filename (`@*.ARGS.map`) in the command line, an `IO::Path` object is created (`.IO`), and the lines from the files are read (`.lines`).

In the case of two files, we have two sequences which are concatenated line by line using the zip meta-operator `Z` applied to the concatenation infix `~`.

After that step, we get another sequence which we can print line by line (`.say for`).

```
2019/12/20 11:16:13"/favicon.ico" failed (No such file)
2019/12/20 11:17:58"/favicon.ico" failed (No such file)
2019/12/20 11:19:18"/robots.txt" failed (No such file)
2019/12/20 11:24:30"/robots.txt" failed (No such file)
```

The result is formally correct, but let's add a space between the original lines. Here is an updated version of the one-liner:

```
.trim.say for [Z~] @*ARGS.map: *.IO.lines.map: *~ ' '
```

Here, a space character is appended to the end of each line (`.map: *~ ' '`), and as there will be one extra space at the end of the combined line, it is removed by the `trim` method. Its sibling, `trim-trailing`, could be used instead (or a regex if you care about original trailing spaces happened to be in the second file).

With the above change, the files are perfectly merged now:

```
2019/12/20 11:16:13 "/favicon.ico" failed (No such file)
2019/12/20 11:17:58 "/favicon.ico" failed (No such file)
2019/12/20 11:19:18 "/robots.txt" failed (No such file)
2019/12/20 11:24:30 "/robots.txt" failed (No such file)
```

There's no problem to merge the same file to itself, or to provide more than two files, for example:

```
$ raku merge.raku a.txt a.txt a.txt
```

Reversing a file

In this section, we are creating a one-liner to print the lines of a text file in reversed order (as `tail -r` does it).

The first one-liner does the job with the STDIN stream:

```
.say for $*IN.lines.reverse
```

Run the program as:

```
$ raku reverse.raku < text.txt
```

`$*IN` can be omitted in this case, which makes the one-liner even shorter:

```
.say for lines.reverse
```

If you want to read the files directly from Raku, modify the program a bit to create a file handle out of the command-line argument:

```
.say for @*ARGS[0].IO.open.lines.reverse
```

Now you run it as follows:

```
$ raku reverse.raku text.txt
```

It is important to remember that the default behaviour of the `lines` method is to exclude the newline characters from the final sequence of lines (the method returns a `Seq` object, not an array or a list).

In Raku, the `lines` method splits the lines based on the value stored in the `.nl-in` attribute of the `IO::Handle` object.

You can look at the current value of the line separators with the following tiny script:

```
dd $_ for @*ARGS[0].IO.open.nl-in
```

This is what you find there by default:

```
$["\n", "\r\n"]
```

The interesting thing is that you can control the behaviour of `lines` and tell Raku not to exclude the newline characters:

```
@*ARGS[0].IO.open(chomp => False).lines.reverse.put
```

The `chomp` attribute is set to `True` by default. You can also change the default separator:

```
@*ARGS[0].IO.open(
    nl-in => "\r", chomp => False
).lines.reverse.put
```

Notice that without chomping, you do not need an explicit `for` loop over the lines: in the last two one-liners, the `.put` method is called directly on the sequence object. In the earlier versions, the strings did not contain the newline characters, and thus they would be printed as a single long line.

A small homework for you: *Tell the difference between* `put` *and* `say`.