CHAPTER 17

Fun One-Liners

Raku is a expressive language with functional features. This makes it relatively easy to capture user and system input in one end of the pipeline, and spew processed data or any kind of action out the other end. Contests such as the Perl and Raku Challenge also emphasize short programs, which has given rise to a whole lot of simple scripts that are powerful, albeit sometimes illegible in other languages. However, Raku strives for expressivity, so it's much easier to understand them and create them with Raku.

These one-liners also employ very creative programming patterns. We use them as an ingredient in all the recipes in this chapter.

Recipe 17-1. Write a Guessing Game in a Single Code Line

Problem

As a challenge, you need to write a game that provides players with a certain amount of turns to guess a number and gives them hints every turn.

Solution

Raku uses `;` as a statement separator, so you can put as many statements as you want on a single line. Additionally, you can use some techniques to shorten the program and get as few characters as possible: use single-character variables (or no sigil variables), define them when you first use them (or use the implicit variable, no definition needed), use operators instead of control structures (`??!!` instead of `if`), and, in general, sacrifice legibility for length.

J.J. Merelo, *Raku Recipes*, https://doi.org/10.1007/978-1-4842-6258-0_17

How It Works

This guessing game generates a small random number and the player tries to determine the number by guessing. When the players guess a number, the game indicates if the true answer is a bigger or smaller number, thereby allowing the players to narrow their scope of numbers to guess. The game ends when the correct number is guessed. Here's is a non-one-liner version:

```
my $number = 6.rand.Int;
my $prompt = "*";
say $number;
while ( my $guess = prompt("$prompt Your guess>") ) ne "" {
    if $guess == $number { last }
    elsif $guess < $number { $prompt = "<" }
    else { $prompt = ">" }
}
```

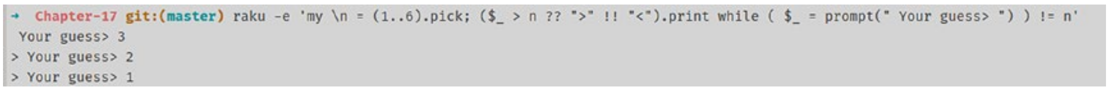
The number is generated and we use part of the prompt to indicate if the guess is larger or smaller than the number entered. Then we run a loop that exits (using `last`) if the guess is correct, and changes the prompt if it is not. The loop runs forever or until we use an empty string in response to it. The command used to write a message and collect what is written is, effectively, `prompt`.

Let's try to use all we've got to shorten this program to a single code line:

```
my \n = (1..6).pick; ($_ > n ?? ">" !! "<").print while ( $_ = prompt("
Your guess> ") ) != n
```

Note Okay, so it won't fit into a single line in this book, due to the book's margins. But you can see in Figure 17-1 that it's on one code line in Raku.

This line is 93 characters long (some of which are spaces), and it has two statements; we need one to declare a variable. We use the default variable $_ to avoid declaring one, as well as sigil-less variable to avoid typing $, a shorter version of the random number generation (and probably faster), but it's essentially the same. One-liners needn't even have their own file—you can run them directly from the command line using `-e`, as shown in Figure 17-1



```
→  Chapter-17 git:(master) raku -e 'my \n = (1..6).pick; ($_ > n ?? ">" !! "<").print while ( $_ = prompt(" Your guess> ") ) != n'
 Your guess> 3
> Your guess> 2
> Your guess> 1
```

Figure 17-1. *Running a one-liner using raku -e*

Note that the script needs to be within quotes. Since we are using double quotes already, we will use single quotes around the entire script.

There are many options if you need to do this in Windows:

1. The latest Windows versions include an Ubuntu subsystem; you can use the shell to install and run Raku in the same way you would do so in Linux. Another option is CygWin, which helps you install many Linux programs. Finally, the GitHub client includes a shell, which is a version of CygWin and where you can install other Linux command-line utilities and run them, should you wish to do so.

2. These last versions also include PowerShell, which has the same quoting conventions as the Linux command line. Use PowerShell to run it in the same way.

3. Finally, you can simply use double quotes outside, and escape all the double quotes inside with \ if you are using older versions of Windows and can't (or don't care to) install the Linux command-line tools.

Remember that Raku is quite liberal in its use of quotes, so if you use single quotes inside the script, you can substitute them for others or simply escape them:

```
 raku -e 'my \n = (1..6).pick; ($_ > n ?? ｢>｣ !! ｢<｣).print while ( $_ =
prompt(｢ Your guess> ｣) ) != n'
```

Can we make this even shorter? Well, we can at least make a single logical line and make it functional as well:

```
{ $^b > $^a
        ?? &?BLOCK($^a, prompt("> "))
        !! $^b < $^a ?? &?BLOCK($^a, prompt("< ")) !! "✓".say
}((1..6).pick,0)
```

This is 153 characters; once again, eliminating spaces would make it shorter, but anyway it would be longer than before. But it's a single logical line. It creates a block and calls it recursively. The block has two arguments, the first one is the number to guess, and the second one is the guess. If the guess is bigger, it shows a "greater than" symbol in the prompt; it will show a "lesser than" otherwise. It does a three-way comparison using `??!!`; if it's not lesser or greater, it must be the solution, so it prints a checkmark.

The trick here is to call the block recursively: no lexical variables are defined, and the number to guess passes along as `$^a` (an implicit variable, which you can call anything you want as long as the order of the arguments is also the alphabetical order of the name of the variables), and the guess passes along as `$^b`. There's no assignment, only the binding of arguments, which makes this purely functional.

The real magic is in the use of the implicit variable `&?BLOCK`. Raku has many ways to do introspection (which might not always be a good thing, but comes in handy here). For instance, the blocks of code have a way of referring to themselves: this one. `&?BLOCK` does not care which block it's in. It will always be *this* block, and you can use it to recurse just the way we do it here. Another option, which might use more lines, is to call the routine by a short name and refer to it by name. That, again, could add a logical newline, since `sub` declarations and code might be thought of as residing on different lines.

So, take your pick. Logically or physically shorter. In either case, Raku has got a way for you to do it.

Recipe 17-2. Compute the *nth* Element in a Sequence Using a Single Line

Problem

You need to compute the *nth* element of a sequence that has been defined recursively, given its first element and a general term to compute *n* given the previous elements.

Solution

This one is easy, since sequences can be defined in a "natural" way in Raku. The position in the sequence can be read from the command line. However, one-liners need to be tested and evaluated, so we'll propose several solutions to do that.

How It Works

Sequences in Raku use a class called `Seq`, which can hold lazily defined and possibly infinite sequences, but the main way it operates is using the `...` (ellipsis) operator. This operator can:

- Generate arithmetic or geometric progressions based on its first terms.
- Define a sequence based on its first terms, and a general term that computes them from the previous ones.

Let's start with an easy one: Compute the factorial of a number, that is, the product of that number and all the preceding ones. (For example, the factorial of 4 is 4x3x2x1=24.)

```
say  [*] 1..( @*ARGS[0]
               // %*ENV<NUMBER>
               // die "Use $*PROGRAM <num> or NUMBER=<num> $*PROGRAM" );
```

Although this solution is shown as several lines for readability, it's a single line logically. The bulk of the script checks for input: it uses the command line or an environment variable, or it dies with a usage message if none of them is present. The `//` is the defined-or operator. It uses its left side if available, and goes on to the right if not. So we can run this by typing the following:

```
./factorial.p6 225
```

Or by writing the following in the Linux or OSX command line:

```
NUMBER=225 ./factorial.p6
```

It will produce a message like this if none of the arguments is available:

```
Use /home/jmerelo/progs/perl6/raku-recipes-apress/Chapter-17/factorial.p6
<num> or NUMBER=<num> /home/jmerelo/progs/perl6/raku-recipes-apress/
Chapter-17/factorial.p6
```

The path to the program is taken from the `$*PROGRAM` dynamic variable. This is in a "die" message to produce an immediate exit (and not an additional error), which is typical of an one-liner hack.

However that number is obtained, it's used to generate a range and multiply all its elements using the `reduce` meta-operator on the * operator. The result is a fast program, but of course there are other ways to do it. For instance, this one:

```
sub MAIN( Int $number = %*ENV<NUMBER>) { say  [*] 1..$number }
```

This is the opposite of the previous approach. It's a single line physically (even shorter than the previous one), but logically it's at least three lines. We use the default value mechanism of signatures to use the environment variable, if available. In this case, the error with no value will be produced by the `sub MAIN` itself, and it might be less informative, although it makes the program logically simpler:

```
Type check failed in binding to parameter '$number'; expected Int but got
Any (Any)
```

It's not really a usage message, and it obviously omits the fact that you can use the environment variable. On the other hand, we get a helpful usage message if we run it with `-help`, as follows:

```
Usage:
  factorial-v2.p6 [<number>]
```

This indicates the name of the parameter and the fact that it's optional (in square brackets). There's even a third alternative.

```
sub MAIN( Int $number =%*ENV<NUMBER> ) { my $c = 1; say  (1,* * $c++...∞)
[$number-1]  }
```

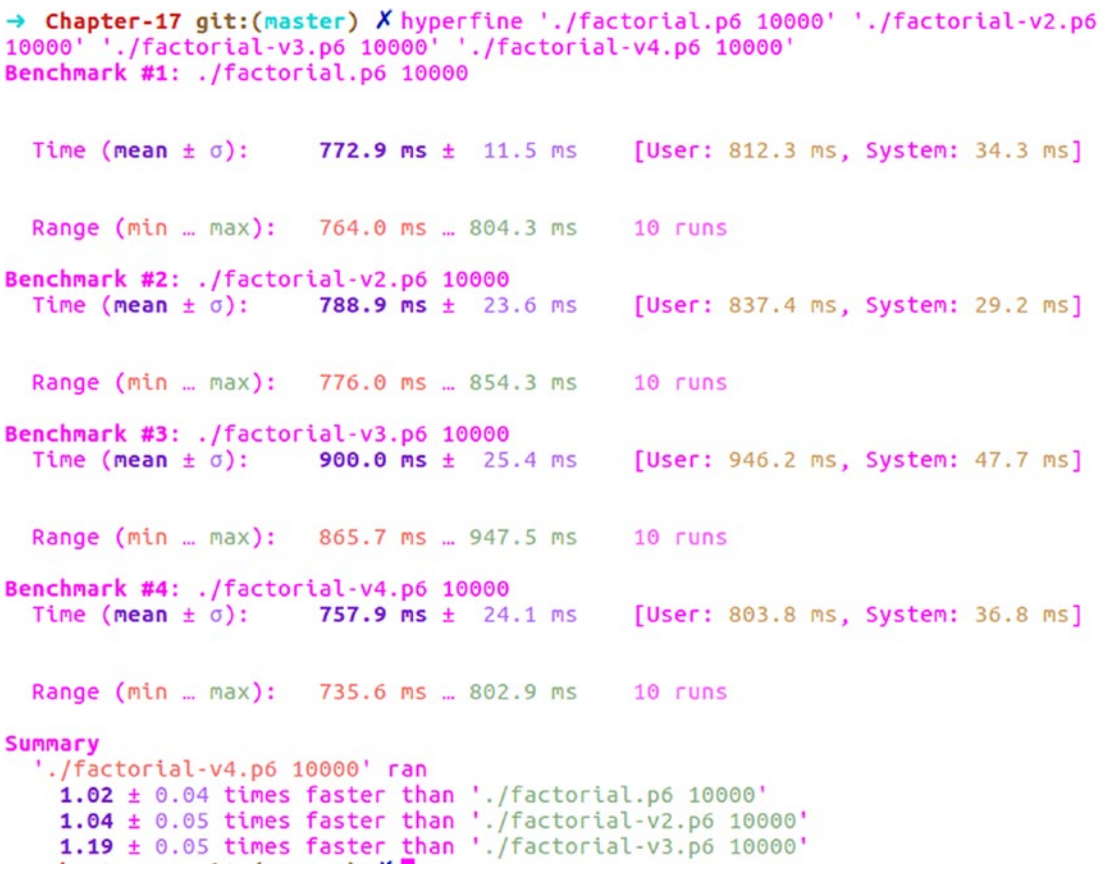
We use an auxiliary variable to take into account the index of the position the operation is in; `$c` will contain that index and will multiply it by the previous value, generating a sequence where the `$c` element is the factorial. But we're actually generating an infinite (but lazy) sequence to do so. We don't need to do that, so let's try a simple loop, as follows:

```
sub MAIN( Int $n =%*ENV<NUMBER> ) { my $ფ = 1; $ფ *= $_ for 1..^$n;
$ფ.say;}
```

We are using the Georgian letter *phar*, since it sounds like *factorial* and also looks like a little nose with glasses on it. We golfed down the program to occupy the minimum number of characters, and could in fact make it shorter if we gave the environment variable another name. But there's another issue here. Which one is the fastest of all the ones that we tried? If we time how the factorial of 10,000 is computed, using time on the command line, the result will be like this.

```
time raku ./factorial-v4.p6 10000
0,80s user 0,06s system 106% cpu 0,806 total
```

As a matter of fact, this happens to be the fastest on my machine. The previous one is the slowest, with 0.95 seconds. It's the one that uses the simplest data structures, as well as a `for` loop, that's quite optimized anyway. It's still using a range, but that is not so slow, and in fact the first version is the second fastest; since `sub MAIN` adds a bit of overhead, this version might, in fact, be quite fast. The result of benchmarking these four versions using the `hyperfine` command tool is shown in Figure 17-2.



```
→ Chapter-17 git:(master) ✗ hyperfine './factorial.p6 10000' './factorial-v2.p6
10000' './factorial-v3.p6 10000' './factorial-v4.p6 10000'
Benchmark #1: ./factorial.p6 10000

  Time (mean ± σ):     772.9 ms ±  11.5 ms    [User: 812.3 ms, System: 34.3 ms]

  Range (min … max):   764.0 ms … 804.3 ms    10 runs

Benchmark #2: ./factorial-v2.p6 10000
  Time (mean ± σ):     788.9 ms ±  23.6 ms    [User: 837.4 ms, System: 29.2 ms]

  Range (min … max):   776.0 ms … 854.3 ms    10 runs

Benchmark #3: ./factorial-v3.p6 10000
  Time (mean ± σ):     900.0 ms ±  25.4 ms    [User: 946.2 ms, System: 47.7 ms]

  Range (min … max):   865.7 ms … 947.5 ms    10 runs

Benchmark #4: ./factorial-v4.p6 10000
  Time (mean ± σ):     757.9 ms ±  24.1 ms    [User: 803.8 ms, System: 36.8 ms]

  Range (min … max):   735.6 ms … 802.9 ms    10 runs

Summary
  './factorial-v4.p6 10000' ran
    1.02 ± 0.04 times faster than './factorial.p6 10000'
    1.04 ± 0.05 times faster than './factorial-v2.p6 10000'
    1.19 ± 0.05 times faster than './factorial-v3.p6 10000'
```

Figure 17-2. *Benchmarking the four versions, with the last one being two percent faster than the first*

Doing things quickly can take you nowhere very speedily, however. So we need to test, and we need to test these one-liners too. Everything is fun and laughter until that one-liner breaks down when we least expect it.

Fortunately, we can do white-box testing with Raku; that is, we can run tests on scripts written in Raku in the same way we would any other function or module. We do so using the `Test::Script` module that was recently released by yours truly to the ecosystem. Let's test these four modules and check that they effectively work:

```
use Test::Script;
use lib <.>;

for <factorial factorial-v2 factorial-v3 factorial-v4> -> $f {
    my $filename = "Chapter-17/$f.p6";
    output-is($filename, "3628800\n",
       "Output well computed for 10",
       args => [10]);
}
```

`Test::Script` provides a series of functions, one of which is `output-is`. It verifies that the output to the script, which is the second argument to the function, is correct. The arguments to the script are given via the `args` named `argument`, as an array. In this case, we will try to compute the factorial of ten, which is the indicated amount. Is everything okay? Well, it is not. The last script was wrong. Here's the correct version:

```
sub MAIN( Int $n =%*ENV<NUMBER> ) { my $൩ = 1; $൩ *= $_ for 1..$n;
$൩.say;}
```

I'm not going to ask you to go back and forth between this and the last version: the ^ before the `$n` was excluding the last element to be considered in the computation, and the result was one zero less than expected. Testing caught the error, and now it's fixed. You see? Testing is important.

Note I reran the benchmarks to see if this error had any influence on the total performance, and it did not. There's one multiplication more, but that's not too important in the context of 10,000 of them.

We can compute more complicated sequences using a single line. For instance, we could compute approximations to the Neperian logarithm base, `e`; this is the summation of the inverse of the factorials up to a number. That is, we compute the factorials of a series of numbers up to a point, we take their inverses, and then we add them up. For instance, if the number is 3, the three factorials would be 1, 2, and 6, and their inverses are 1, ½, ⅙, which adds up to 1⅔. The bigger the number, the bigger the precision. This script will do it:

```
say [+] (1,| [\*] (1...∞))[^@*ARGS[0]].map: 1/*
```

The base of this is the infinite succession of all factorials. `[\*]` is an accumulating reduction-operator: It multiplies every element by the result of all the previous ones, but instead of throwing away the result, it creates a series with all of them. It doesn't use `0`, which is a special case for which the factorial is `1`, but it would break the series, so we put it in front and collate the whole series using the slip operator, `|`.

The result of that will be the infinite sequence of all factorials. But we just need the `n` first to compute an approximation of `e`, so we use the argument given in the command line to cut it where we want. However, to obtain `e`, we need to invert that series. We map every element to its inverse, and then sum them all to obtain an approximation to `e`. For 1,000 elements, we will obtain 2.718281828459045, and it's pretty much stuck there for bigger values. Can we go any further? Well, `20!` is already a pretty big number, and `1/20!` is very small. It goes beyond the capacity to represent floating-point numbers in the system, so why throw all those cycles to waste? Let's just cut the sequence where it can't be improved any further:

```
say [+] (1,| [\*] (1...^max(@*ARGS[0],20))).map: 1.0/*
```

Instead of defining an infinite sequence, which is okay from a theoretical point of view, but not very practical, we cut the sequence of numbers at the argument if it's smaller than 20 or simply 20. Same as above, but we'll get to the result faster no matter what, since 20 terms is all we'll be computing. But then, why are we doing this, if all we've got is crappy precision that we could achieve with a much simpler formula? We need to crank up the precision. And, Raku, again, is helpful in that sense:

```
say [+] (1,| [\*] (1..@*ARGS[0])).map: { FatRat.new(1,$_) }
```

We eliminate the cap for the values, which we don't need any more. But the key here is using `FatRat`, or big rational numbers of arbitrary precision, to compute every new term. While `Rats` (or rationals) become `Nums` (or floating-point numbers) when they exceed a certain size, and are thus limited in precision, `FatRats` do not. With `FatRats`, we can keep computing terms until we get tired. For 1,000 terms, it will be as follows:

```
2.71828182845904523536028747135266249775724709369995957496696762772407663O
[...41 lines here...]
201024950551881694803221002515426494639812873677658927688163598312 5
```

Raku gives you precision when you need it, but the only two data structures with infinite precision are `Ints` and `FatRats`, so if you need to go to the umpteenth position in a number, you need to include one of them in your one-liner (or any other program).

Recipe 17-3. Perform a System Administration Task Repeatedly Using a Single Code Line

Problem

You need to check the logs from time to time, or create an alert, or monitor some variable, if possible with a single line of code. You need to repeat these checks at an established frequency.

Solution

Use `supply` to create periodic tasks; these should be short if you want to fit them in a single line. If you need to use an external module to perform a system task, you can provide it in the command line using `-M`.

How It Works

System interaction is not easy, and we devoted an entire chapter to it, Chapter 2. In general, you will need to use an external library to parse logs properly or use the correct command to interact with the operating system API, and that will usually be in an external module.

For instance, say you need to know the evolution of the filesystem and determine how it's being filled up. There are different commands you can use; `df` and `du` in Linux and OSX, and probably other kind of commands in Windows. If you don't want to work with all the different cases, you could use a module such as `FileSystem::Capacity`, which uses these utilities to return directory and volume capacity.

Now you need to do something periodically. Again, you can use the operating system services to do that. This is, however, highly dependent on the operating system, and needs additional capabilities. Let's just use a script to achieve the same, if possible.

There are several ways to achieve this feat in Raku. Promises can be timed, and you can launch another promise with each fulfilled one. However, the simplest way to achieve this is to use a supply. A supply creates a data stream; a live supply will produce values forever or until it's stopped. There are many ways to create these kinds of supplies, but we are going to use `.interval`, which generates a stream of incrementing integer numbers every number of seconds given as an argument. `Supply.interval(2)` will produce increasing numbers every two seconds. We don't care about the numbers, but about processing something every two seconds, and this is what we will do in this one-liner.

```
react whenever Supply.interval(@*ARGS[0]) {
        with volumes-info()</> {
            say "Free M ", (.<free>/2**20).Int,
            "- Used ", .<used%>
        }
}
```

Again, 167 characters with spacing and everything. It's a long line indeed, but it's a single logical sentence, wrapped in several control structures. The external one, `react`, will activate whenever one message is received. The messages it reacts to are included in a `whenever` clause, which checks when the (periodic) supply emits something. We don't really care about the numbers (which would be in the `$_` variable), but we do care about our stuff: `volumes-info` will return a hash with all the volumes as keys and information on them as hashes and `</>` will be the root volume (in Linux; this is OS specific and will not work in Windows). We could change the direction of the slash (from `/` to `\`) to detect the OS, but that will add to the length and it's probably easier to just change it if you install it onto a different operating system.

Using `with` helps shorten the script, because it places its value in the implicit variable. `.<free>` will return that value from the hash, and ditto for `.<used%>`. We convert the value to megabytes, since by default it's given in bytes (at least in Ubuntu).

We also use the arguments to specify the interval, and we pick it up via `@*ARGS`. We can then run it this way:

```
 raku -M FileSystem::Capacity::VolumesInfo -e 'react whenever Supply.
interval(@*ARGS[0]) {with volumes-info()</> { say "FreeM ", (.<free> / 2
** 20).Int, "- Used ", .<used%> } }' 15
FreeM 193112- Used 57%
...
```

I really wanted this code to fit on one line, which is why I reduced it so much. Here it is in normal type, in several lines:

```
raku -M FileSystem::Capacity::VolumesInfo
    -e 'react whenever Supply.interval(@*ARGS[0]) {
       with volumes-info()</> {
         say "FreeM ", (.<free> / 2 ** 20).Int, "- Used ", .<used%>
       }
    }' 15
```

The `FileSystem::Capacity::VolumesInfo` module contains `volumes-info` and is part of the `FileSystem::Capacity` distribution. Using `-M` in the command line is equivalent to the use statement we placed at the front of the scripts. By offloading it to the command line, we have saved a statement in the program, keeping it as a single line.

To make it useful, you will probably want to run it at startup and redirect its output to some log file in the system logging directory, `/var/log`, for instance. But that's something additional. The single-line script here will perfectly solve your problem.