Introducing the Raku programming language

Supported by The Perl Foundation

raku.org
perlfoundation.org

Hi, my name is **Camelia**. I'm the spokesbug for Raku, the plucky little sister of Perl. Like her world-famous big sister, Raku intends to carry forward the high ideals of the Perl community. Raku is developed by a team of dedicated and enthusiastic volunteers.

Notes

Camelia is intended primarily to represent Raku, The Language, not any other aspect of Raku culture, corporate or otherwise. In particular, various implementations and distributions are free to use their own logos and/or mascots.

Certain variants are also permissible; since Camelia knows how to change her wing colors at will, any color scheme (or lack thereof) in the same pattern is fine. She just happens to like bright colors most of the time because they make her happy. But she's willing to blend in where necessary. :)

Going to the other extreme, a textual variant also exists:

»Ö«

Many other variants are possible. Have fun. Good taste and positive connotations are encouraged, but cannot of course be required.

Those of you who think the current design does not reflect good taste are entitled to your opinion. We will certainly allow you to change your mind later as you grow younger. :)

Online resources

* http://www.nntp.perl.org/group/perl.perl6.language/2009/03/msg31263.html

Contents

This document was produced with donations through the Enlightened Perl Organisation and The Perl Foundation

enlightenedperl.org
perlfoundation.org

Why Change from Perl 6 to Raku?

Raku started out as a challenge to modernize Perl and bring it into the 21st century. We started out with a list of changes to the Perl core, and ended up with the language now called Raku. Along the way what started out to be a "simple" version increment became a new language, yet one that was still in the Perl family.

Much like the proverbial axe, Perl 6 replaced Perl 5's back end with a VM, replaced its front end with a new grammar engine, and never changed its name. The decision to rename the language remained in the hands of Larry Wall, the creator of both Perl (1-5) and (now) Raku.

As of 2019 the matter came to a head, Larry allowed the change of name, and a vote was taken to rename the language from Perl 6 to Raku. It is still very much the multi-paradigm language envisioned at the outset, runs on all the major platforms, and lets you write in the style you prefer.

Raku lets you write quick one-liners with 'raku -e', short admin scripts become even shorter and self-documenting with sub MAIN {}, and large web-scale systems become easier to read with Raku's new attribute system and metaprogramming abilities.

Its new numeric core has large banks interested in its capabilities, its Unicode support is second to none, and it supports multi-core CPUs out of the box with data sources, sinks and a built-in Promise library.

Welcome to Raku

After 15 years of design and development Raku was released and is now being used in production. Raku is a supremely flexible language, adapting to your style of programming, whether that be quick one-liners for sysadmins, scripts to manage a database import, or the full stack of modules necessary to realise an entire website.

Raku enhances Perl's long-term appeal with a proper object system including roles, threading and multi-method dispatch. Raku has spent a long time coming to fruition and has learned from other programming languages building on their success and learning from the issues of the past. We believe Raku is a language that will last for decades as it has been conceived to adapt to future trends and is flexible in its usage with other languages.

We have collected here a list of some of the many advantages to using Raku. We have also, thanks to the kindness of the authors, reprinted chapters from some of the books that have been written and recently released on Raku.

You will find a quick guide to using the most popular implementation of Raku which is Rakudo Star and with that how to get started with Raku.

We close with a list of some of the many community resources and sites available for, and about, Raku.

Raku: Quick Features

There are many reasons to learn Raku - here are a few of our favourites.

General

- Raku is a clean, modern, multi-paradigm language; it offers procedural, object-oriented AND functional programming methodologies.
- Easy to use consistent syntax, using invariant sigils for data-structures.
- Raku is a very mutable language (define your own functions, operators, traits and data-types, which modify the parser for you).
- Adding a custom operator or adding a trait is as simple as writing a subroutine.
- Advanced error reporting based on introspection of the compiler/runtime state. This means more useful, more precise error messages.
- Multiple versions of a module can be installed and loaded simultaneously.
- System administration is simplified due to simpler update/upgrade policies.
- Runtime optimization of hot code paths during execution (JIT), by inlining small subroutines and methods.
- Runs on small (Raspberry Pi, Android) and large multi-core hardware.
- Advanced arena-based garbage collection.
- Fewer lines of code allow for more compact programs. Commonly-used operators and names are the shortest for easy typing.

```
# Words from file
for 'dict.txt'.IO.words -> $word {
  say "$word probably rhymes with Perl"
    if $word ~~ /[ea?|u|i] rl $/;

  say "$word is a palindrome"
    if $word eq $word.flip;
}
```

Text Processing

- Full grapheme based Unicode support, including Annex #29, meaning almost unparalleled Unicode support.
- Extensible grammars for parsing data or code (which Raku uses to parse itself).
- Execute code at any time during parsing of a grammar, or when a certain match occurred.
- Regular expressions are cleaned up, made more readable, taken to the next level of usability, with a lot more functionality. Named regular expressions are made possible for ease of use.

Scoping

- Dynamic variables provide a lexically scoped alternative to global variables.
- Emphasis on composability and lexical scoping to prevent “action at a distance”. For example, imports are always lexically scoped.
- Easy to understand consistent scoping rules and closures.
- Phasers (like BEGIN/END) allow code to be executed at scope entry/exit,

Typing

- Multi dispatch on identically named subroutines/methods with different signatures, based on arity, types and optional additional code.
- Compile time error reporting on unknown subroutines/impossible dispatch.
- Optional gradual type-checking at no additional runtime cost. With optional type annotations.
- Easy command-line interface accessible by MAIN subroutine with multiple dispatch and automated usage message generation.

Object Oriented Programming

- Powerful object orientation, with classes and roles (everything can be seen as an object). Inheritance. Subtyping. Code-reuse.
- Introspection into objects and meta-objects (turtles all the way down).
- Meta Object Protocol allowing for meta-programming without needing to generate/parse code.
- Subroutine and method signatures for easy unpacking of positional and named parameters, and data structures.
- Methods can be mixed into any instantiated object at runtime, e.g.to allow adding out-of-band data.

Data Structures

- Junctions allowing easy checking of multiple possibilities, e.g. $a == 1|3|42 (meaning is $a equal to 1 or 3 or 42).
- Lazy evaluation when possible, eager evaluation when wanted or necessary. This means, for example, lazy lists, and even infinite lazy lists, like the Fibonacci sequence, or all prime numbers.
- Lazy lists defined with a simple iterator interface, which any class can supply by minimally supplying a single method.
- Native data types for faster, closer to the metal, processing.
- Floating point math without precision loss because of Rats (rational numbers).
- Large selection of data-types, plus the possibility to create your own types.
- Multi-dimensional shaped and/or native arrays with proper bounds checking.
- Automatic generation of hyper-operators on any operator (system or custom added).

Concurrency, Parallelism, Asynchrony

- High level concurrency model, both for implicit as well as explicit multi-processing, which goes way beyond primitive threads and locks. Raku's concurrency offers a rich set of (composable) tools.
- Multiple-core computers are getting used more and more, and with Raku these can be used thanks to parallelism, both implicit (e.g. with the >>. method) and explicit (start { code }). This is important, because Moore's Law is ending.
- Structured language support is provided to enable programming for asynchronous execution of code.
- Supplies allow code to be executed when something happens (like a timer, or a signal, or a file-system event, or gui events).
- The keywords react/whenever/supply allow easy construction of interactive, event driven applications.

Interoperability

- Interfacing to external libraries in C/C++ is trivially simple with Native \Call.
- Interfacing with Perl 5 (CPAN)/Python modules is trivially simple with Inline::Perl5 and Inline::Python.
- Raku runs on a variety of back-ends. Currently MoarVM & JVM, JavaScript is in development, more may follow.
- The keywords react/whenever/supply allow easy construction of interactive, event driven applications.

Learning to program with Raku

First Steps: Getting into programming without leaving the command line

Kindle Edition

https://www.amazon.com/gp/product/B07221XCVL

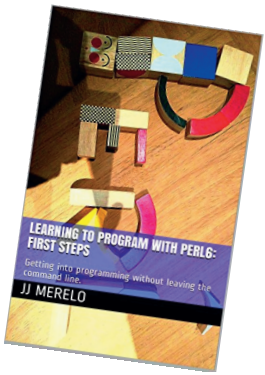
Raku is a modern language that has been designed with the current computers, operating systems and Internet in mind. It is very expressive, allows to transcribe mathematical thought to program very easily, and also has a thriving and helpful online community.

It is good for this community to grow, and the best thing is to try and learn this language as your first, or maybe 0.5th, language.

The basic prerequisite for this book is to want to learn to program, and the second is to be curious and not afraid of testing new things.
It is probably also OK if you already know a bit of programming, but you do not need to know Perl specifically. If you do, this book will try to teach you idiomatic Raku and also help you through some good practices when learning, or teaching, a new language.

This is also a free as in free speech book, with a creative commons license. Check out its GitHub repo (https://github.com/JJ/perl6em) for updates, examples and some other goodies (including a theatrical play).

Book Extract



Chapter 7

The expressions

The first thing you need to know about a language, any language, is how to compute things. Compute in the more general sense: combine things to give other things. This, in general, is called expression, which, also in general, is a bunch of symbols linked by **operators**.

Generally, modern computer languages are able to work with many different kind of symbols, and Perl6 is no exception. Let's start with that.

REPLs and how to use them as glorified calculators

A REPL is a Read-Eval-Print loop. it is a program included with most interpreted languages, that presents a command-line prompt and into which you can type expressions, and, later on, full statements. But, for the time being, let's type

```
perl6
```

And we will be into a perl6 REPL into which you can type Perl6 stuff.

Let's try the simplest thing:

```
To exit type 'exit' or '^D'
> sin( /2)
```

You will have to find a way to type that , by copy/pasting it from some website or Google or right from here. It will return the sine of /2, which, as you know, is 1.

And this is cool, because only some languages are able to handle this kind of expressions, and, even more, to use it correctly in math. But you can do even more:

```
sin( /2) +1
```

You can do that by copying/pasting, or else install `Linenoise`, a command line that allows you to go back to previous command by typing arrow-up. Do it with

```
zef install Linenoise
```

You can use the usual arithmetic operators `+,-,*,/` but Perl6 adds two typographic operators, ÷ and × (these ones are produced in the Spanish keyboard with May+AltGr+ comma or period), like

```
sin( /3) × sin( /3) + cos( /3) × cos( /3)
```

Or, even better

```
sin( /3)² + cos( /3)²
```

One of the objectives of Perl6 is to use the whole range of characters that Unicode, if not modern keyboards, offer. This simplifies expressions and makes them more readable.

You will not find all numbers in superscript mode. If you want to raise a number to the /x/th power, use `**`.

```
3**25
```

Arithmetic only takes you so far in programming. We will learn how to deal, and operate, with all kinds of data in this glorified calculator.

Now that we mention Unicode

Unicode is the way to express all alphabets in the world, and then some things that are not really alphabets. It uses up to around sixty thousand symbols, and allows us to write, using modern operating systems, editors and languages, anything that would have to be expressed in living or dead languages, and even some emoticons. Unicode is evolving constantly, and for the people means that

they will be able to use characters that are usual in their own language, and also some usual in mathematical expressions.

Since not all languages, editors, operating systems or even keyboards are **modern** in that sense, some impedance should be expected. But Perl 6 will not get in your way, allowing you to use them just they way they should, so if you want to raise something to the second power you will not have to, although you can, write x**2 but simply x^2.

It's not only numbers

All the expressions written above are numbers.

```
(sin( /3)² + cos( /3)²).^name
```

is going to return `Num`, indicating that it is simply a Number, actually a real number. This `.^name`, together with surrounding the expression via parentheses meaning grouping, is a way to apply a *property* or to call a *method* on that object. In Perl6, everything is an object, and objects have a class, and you want to call methods that correspond to objects of that class, append a dot and use the method, possibly with some arguments like `WHAT(is, "the", $what)`. Let's not worry about that for the time being, or about classes themselves. Just with the fact that every expression is an object, and those objects belong to a class; every class in Perl6 *descends* from the ur-class called or `Mu`. And among the properties of Mu is that you can call this `^name`. Any other class descends from this one, so *you can call `.^name` on any object of any class.* That is the take-home message, even if you might not know, so far, what is an object, or a class. Second take home message: *different objects also have different classes.*

Since both *objects* at the sides of the `+` are Nums, you can add or subtract them or do any other arithmetic operation, but you cannot do

```
sin( /3)² + cos( /3)² + " is 1"
```

will yield this error

```
> sin(π/3)² + cos(π/3)² + " is 1"
sin(π/3)² + cos(π/3)² + " is 1"
Cannot convert string to number: base-10 number must begin with valid digits or '.' in ' ⏏i
s 1' (indicated by ⏏)
  in block <unit> at <unknown file> line 1

>
```

Figure 7.1: Errored expression in the REPL

And the reason for that can be found out by typing:

```
" is 1".^name
```

Which, whatever it is, is not a `Num`, so it cannot be added. That shows that there are more types of data you can use and work with from the REPL. In fact, there are a lot. In general, you cannot mix and match and, also in general, every one has got its own operators you can work with. You can mix fractions with integer numbers, for instance:

```
+4/3
```

And

```
( +4/3).^name
```

will return `Rat`, a Rational, same as `+4`. In fact, most floating point numbers in Perl 6 will be represented as rational, unless we explicitly tell the interpreter to deal with them as floating point, that is real, numbers, which, BTW, cannot be something else that fractional numbers since they use a finite representation in computers. Mostly.

However, in some cases you can try and mix different things using an operator. Operator "~" concatenates stuff, that is, joins things that look like words and letters, for instance

```
6 ~ "6"
```

will return `66`, and anything you put there will be concatenated. `~` is an operator that is not picky about what it has got in both ends.

You cannot add strings together, because that is what they are, but curiously enough, you can multiply them:

```
"1" ~ "\n" ~ "2" x 2 ~ "\n" ~ "3" x 3 ~ "\n" ~ "2" x 2 ~ "\n" ~ "1"
```

This being a rather nice and utterly useless example on the operator `x`, which *multiplies* or rather *replicates* whatever character of string it is related to. Introduced together with `\n`, the carriage return, so that if forms a nice pile of stuff.

Which is shorter and better in this example

```
for <1 2 3 2 1>  { say $_ x $_ }
```

but that's something we will see later on, when we talk about loops and all that's nice and beautiful about it.

Lists of things are also game.

Numbers and words are simple things. But you can string them together in something more complex. You can have groups of them, or lists of them, or combine them as sets of lists of sets of whatever. Perl 6 is great because you do not need to make all things in a complex structure be of the same type. You can create a list with the less than and more than sign, this way:

```
<a b 7   ²>
```

And with lists, you can do things like sorting:

```
sort <a b 7   ²>
```

or combine lists to create a new one using the `X` operator, called **cross product**

```
<a b 7   ²> X < → ← >
```

You can also combine in some other ways, adding one list to another.

```
<a b 7   ²> , < → ← >
```

The simple *comma* operator is going to create a new list with two elements, each one of which is a list. You can **flatten** it:

```
flat <a b 7   ²> , < → ← >
```

You can already do interesting things with these lists (or arrays, or vectors, stuff in a row, whatever). For instance, you want to pick one element randomly,

```
(flat <a b 7   ²> , < → ← >).pick
```

will return, every time you run it, a different element. You can do that as many times as you want, but it is much easier to use `roll` to do it many times for you.

```
< → ← >.roll(6)
```

will return a whole quiver of arrows.

Maybe you want a single element of the array:

```
< → ← >.roll(6)[3]
```

This will return the 4th element, taking into account that all arrays start with 0. Otherwise known as a random arrow. Or you might want to extract a range

```
(flat <a b 7   ²> , < → ← >)[3..6]
```

uses the *range* operator `..` (that is, two points), which generates a contiguous sequence of elements. Otherwise known as, well, range. But these ranges also behave as arrays:

```
(0..10)[3..6]
```

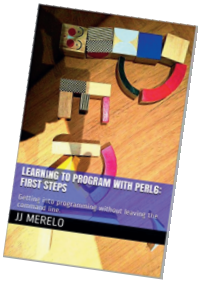
although they are not exactly the same:

```
(0..10).^name
```

will, effectively, return `Range`.

This is just the start of complex structures with Perl. More to come in the next chapters.

Chapter 8



Thinking like computers do

You probably know, or at least have made an educated guess, that computers are unlike humans. But since expressions are entered in pretty much the same way you would use somewhere else, it's no big deal. However, once you want to deconstruct what actually needs to be done to make a computer do whatever you want it to do, you have to start to think like a computer, in what is known as computational thinking. It is kind of a game of "Simon says". You have to precede every instruction you give a computer by something that indicates you want it to do precisely that, and then you have to say very precisely what it needs to do.

But there is more to that, the fact that you also need to indicate the *sequence* of actions you want the computer to do by indicating that in your program. When you work with the REPL, as we have done above, the sequence is quite clear: you say something, press Enter, the computer thinks for a while or what looks like nothing, you have the response. However, when the program gets a bit more complicated, you also have to think, in a particular point in time, where the computer might be and what that implies regarding values or results you need to proceed.

Which is why we talk about computational thinking as a series of techniques for solving problems using computers, including all the steps you need to take to identify the problem, put it in a way that can be understood and processed by a computer, and then express every step as instructions in program that will, eventually, solve the problem.

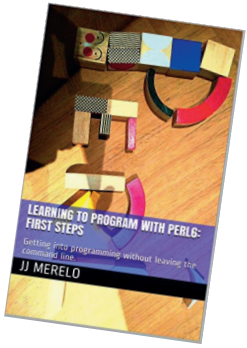
We will get back to this later on, but for the time being there are a couple of techniques you will need to understand. First is *pattern recognition*, which means finding what several different pieces of information have in common and, in turn, what to do to solve a problem in a case given that you know how to solve it in another case which follows the same pattern. When you find something that has a regular increasing or arithmetic pattern, for instance, you

will discover that you can use lists or sequences to solve it, and apply whatever you know already on how to deal with sequences.

The second technique we should focus on right now is *problem decomposition*, how to break down a problem in different parts that can, more easily, solved. For instance, you want to find what would be the number a sequence will reach if left to grow until infinity; first you will have to solve the problem of representing the sequence, which might not be immediate, and then how to apply known techniques of limit finding, such as finding the function that represents the sequence and then applying what is known about function limits to that sequence.

Every part of a problem will eventually become an instruction or group of instructions called usually *subroutines* or *functions*. But for the time being, it is enough to know that breaking down a problem in parts you already know how to solve is the key for solving problems of any size.

Chapter 9



To infinity and beyond

If you know in advance every single term of a list, writing them down as above is the way to handle them. However, you might know a few terms of the lists, or how they are generated, and that is that. Let us see how to deal with them in Perl 6

Working with ranges and sequences

For long lists, you might want to use only the first and last term

```
1...222
```

via the **yada, yada, yada** operator, or, even better,

```
1 … 333
```

But the coolest thing with lists is the stuff you can do to all of them at the same time:

```
[+] 1 … 333
```

will add everything together. Any operator you put inside brackets will be applied to all in turn. Try `[*] 1 … 333`, for instance.

But the coolness factor can be increased:

```
[+] 1,3 … 333
```

and even

```
[+] 1,3,9 … 333
```

The [] is called a *reduce* operation. If you have heard some big data buzz, you have probably heard about an operation called **map/reduce**. Well, this is the *reduce* part. And it is so easy to do with Perl6.

Because Perl6 is able to deal with arithmetic and geometric progressions out of the box. And even infinite ones:

```
1,3,9 … ∞
```

You can obtain the 100th term using

```
(1,3,9 … ∞)[100]
```

or, why not, the terms from 1000 to 1100

```
(1,3,9 … ∞)[1000…1100]
```

which will return a pile of numbers, separated by spaces. It is quite usual to start from 0 and go to a particular number. The *caret* = ^ = is used to indicate *0 to* the number that follows it

```
(0,5,10 … ∞)[^25]
```

will list the 25 first elements of the list of multiples of 5.

Besides, at the same time, we have seen how to deal with a single term in a list, and how to work with a series of terms. You can use infinite syntax to generate also finite lists if you do not want to compute in advance the precise terms of it. For example, above you will be working on the 1000th term and on of an infinite list, without working out if it is exactly 3000 or some such. You can check out this Advent calendar entry for a few samples of Perl6 coolness too.

Operating on lists

Lists are perfectly good subjects for this calculator on steroids we have in the REPL. Whatever combination you think about, it is probably possible to do it on lists; some of them will work also on infinite lists, but most probably not. We have already seen [+] work on a list of numbers. Any sensible operation like [*] will also work. But this will also yield a result:

```
[~] 'a'..'z'
```

collating together all elements in the alphabet. Preceding it with \. which you can think of as an accumulator, will instead create another array whose elements are the accumulation of the operation up to that element. Better if you try it:

```
[\~] 'a'..'z'
```

This can be very useful when working on accumulative series, for instance, what is the sequence of factorials up to 25?

```
[\*] 1..25
```

This accumulator is called a *zip* operator. We will see later on what it actually means, for the time being it just makes operating with series a bit easier.

But single lists only take you so far. Previously we have seen the comma for kind-of joining two lists. But there are multiple ways of creating new lists by combining them. For instance, the *cross* operator `X` will create a list of lists from two of them

```
(1,3...10) X (2,4...10)
```

will combine all even and odd numbers in pairs, combining 1 with 2 to 10, then doing the same with 3… This can be useful if you want to create a combination, but even more so if you turn `X` into a hyper-operator by using it to precede any operation such as `*`

```
(1,3...10) X* (2,4...10)
```

will create a *flat* list with the results of multiplying the pairs we have generated before. This can be useful for complicated arithmetics, but sometimes we only want to pair a couple of lists to create a new one that takes one element from each one, combining them like the tooth of a zipper. This operation is appropriately named Zip and represented by `Z`

```
(5,10...Inf)[^20] Z (4,8...Inf)[^20]
```

This creates a new list that zips together similar terms in the sequence of multiples of 4 and 5. Can we multiply them to create a succession of multiples of 20? Maybe…

```
(5,10...Inf)[^20] Z* (4,8...Inf)[^20]
```

Doing stuff to lists

Well, that is precisely what we have been doing above. But we need to do more. A lot more.

All we have been doing is combining lists with each other. We have also been using lists of exactly the same length. But we might need to do some basic operation to a list, or create one list that is not exactly an arithmetic or geometric progression. For instance, this

```
(1/2,1/3...Inf)[5]
```

will not do what you expect it to do, which would be 1/5. Writing the whole range

```
(1/2,1/3...Inf)[^5]
```

will show that, what it is actually doing is to turn it into an arithmetic sequence that subtracts 0.166667 from the previous one, despite being relatively clear, for a human, that we are trying to create the `1/n`. Succession. We can do that, however, using the hyperoperators `<<>>` and derived. Check this out

```
1 <</<< (1..100)
```

will return precisely what we are looking for, a descending sequence of numbers that ends with 0.01. Please note that we can no longer use an infinite (lazy) sequence: we have to be concrete.

This `<</<<` is known as an *hyperoperator*, because it takes a humble operator like `/` and turns it into a machine that deals with lists. It can also be written `«/«` with the direction of the angular brackets pointing at the *smaller* thing, in this case a single number vs. a list.

What happens if you do

```
<1 2> <</<< (1..100)
```

is kind of funny. It is like applying the cookie cutter in the left hand side to the right hand side: the first element will be divided by 1, the second by 2, and so on... You can even take one wing `<<` from the hyperdrive, and use it to, for instance, negate a sequence:

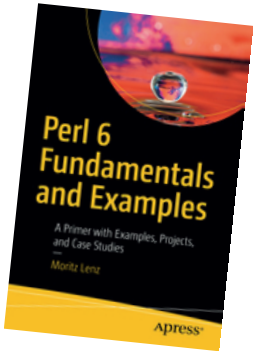
```
-<< (1..100)
```

When the two lists have the same length, the arrows can go in any direction, it will not matter much. Let us create random fixtures for a (subset of) the Premier League

```
( <ARS AST BOU CHE EVE LEI LIV MCI MUN NEW>.pick(10)
  «~»
    ( " - " «~« <ARS AST BOU CHE EVE LEI LIV MCI MUN NEW>.pick(10)))
    »~» "\n"
```

This, which could admittedly be a little shorter, uses these hyperoperators to combine acronyms so that they are separated by a dash, which is what = " - " «~« = does, and then put the whole result in different lines, which is done by the = »~» `\n` = in the last line. A great achievement, with a small amount of coding involved. We are using throughout the ~ string concatenation operator, which is what allows us to create such a compact statement.

Book Extract



12. Plotting Using Inline::Python and Matplotlib

Occasionally I come across git repositories where I want to know how active they are and who the main developers are.

Let's develop a script that plots the commit history, and explore how to use Python modules in Perl 6.

12.1 Extracting the Stats

We want to plot the number of commits by author and date. We can get this information easily by passing the some options to `git log`:

```
my $proc = run :out, <git log --date=short --pretty=format:%ad!%an>;
my (%total, %by-author, %dates);
for $proc.out.lines -> $line {
    my ( $date, $author ) = $line.split: '!', 2;
    %total{$author}++;
    %by-author{$author}{$date}++;
    %dates{$date}++;
}
```

`run` executes an external command and `:out` tells it to capture the command's output, making it available as `$proc.out`. The command is a list, with the first element being the actual executable and the rest of the elements are command line arguments to this executable.

Here `git log` gets the options `--date short --pretty=format:%ad!%an`, which instructs it to produce lines like `2017-03-01!John Doe`. This line can be parsed with a simple call to `$line.split: '!', 2`, which splits on the `!` and limits the result to two elements. Assigning it to a two-element list `( $date, $author )` unpacks it. We then use hashes to count commits by author (in `%total`), by author and date (`%by-author`), and finally by date. In the second case, `%by-author{$author}` isn't even a hash yet and we can still hash-index it. This is due to a feature called *autovivification*, which automatically creates ("vivifies") objects where we need them. The use of `++` creates integers, `{...}` indexing creates hashes, `[...]` indexing, `.push` creates arrays, and so on.

To get from these hashes to the top contributors by commit count, we can sort `%total` by value. Since this sorts in ascending order, sorting by the negative value returns the list in descending order. The list contains Pair[1] objects, where we only want the first five, and only their keys:

[1] https://docs.perl6.org/types/Pair

```
my @top-authors = %total.sort(-*.value).head(5).map(*.key);
```

For each author, we can extract the dates of their activity and their commit counts like this:

```
my @dates  = %by-author{$author}.keys.sort;
my @counts = %by-author{$author}{@dates};
```

The last line uses *slicing*, that is, indexing a hash with a list to return a list of elements.

12.2 Plotting with Python

Matplotlib is a very versatile library for all sorts of plotting and visualization tasks. It is based on NumPy[2], a Python library for scientific and numeric computing.

Matplotlib[3] is written in Python and for Python programs, but that won't stop us from using it in a Perl 6 program.

But first, let's take a look at a basic plotting example that uses dates on the `x` axis:

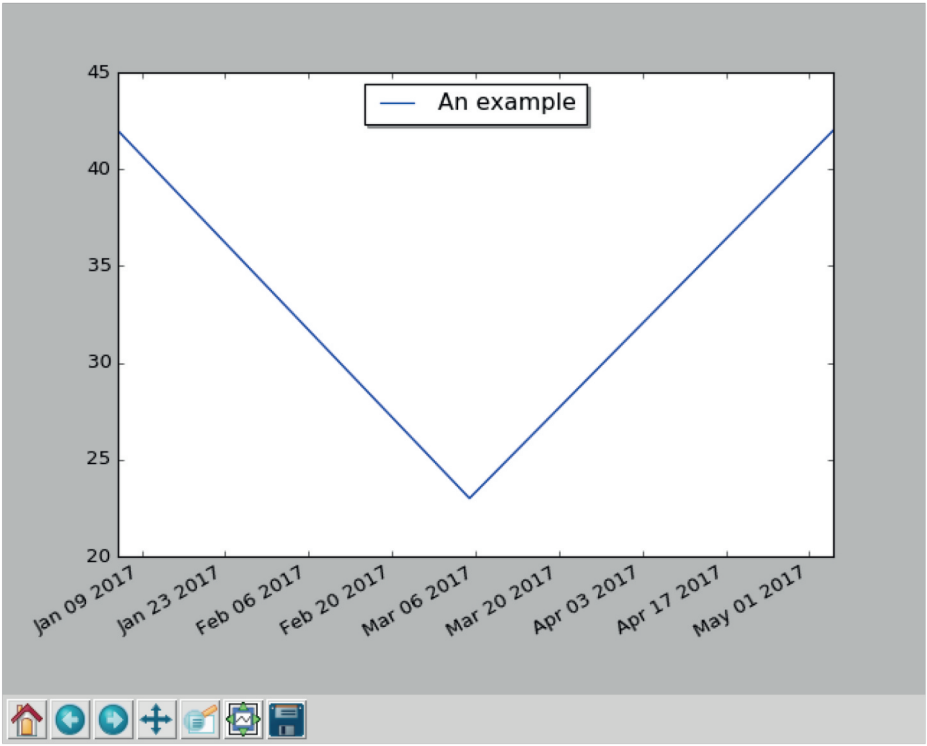
```
import datetime
import matplotlib.pyplot as plt

fig, subplots = plt.subplots()
subplots.plot(
    [datetime.date(2017, 1, 5), datetime.date(2017, 3, 5), datetime.date(2017\
, 5, 5)],
    [ 42, 23, 42 ],
    label='An example',
)
subplots.legend(loc='upper center', shadow=True)
fig.autofmt_xdate()
plt.show()
```

To make this run, you have to install Python 2.7 and matplotlib. You can do this on Debian-based Linux systems with `apt-get install -y python-matplotlib`. The package name is the same on RPM-based distributions such as CentOS or SUSE Linux. MacOS users are advised to install Python 2.7 through homebrew and macports and then use `pip2 install matplotlib` or `pip2.7 install matplotlib` to get the library. Windows installation is probably easiest through the conda[4] package manager, which offers pre-built binaries of both Python and matplotlib.

When you run this script with `python2.7 dates.py`, it opens a GUI window, showing the plot and some controls, which allow you to zoom, scroll, and write the plot graphic to a file:

[2] http://www.numpy.org/
[3] https://matplotlib.org/
[4] https://conda.io/docs/



Basic matplotlib plotting window

12.3 Bridging the Gap

The Rakudo Perl 6 compiler comes with a handy library for calling foreign functions[5] – called 'NativeCall' – which allows you to call functions written in C, or anything with a compatible binary interface.

The Inline::Python[6] library uses the native call functionality to talk to Python's C API and offers interoperability between Perl 6 and Python code. At the time of writing, this interoperability is still fragile in places, but can be worth using for some of the great libraries that Python has to offer.

To install `Inline::Python`, you must have a C compiler available, and then run

```
$ zef install Inline::Python
```

Now you can start to run Python 2 code in your Perl 6 programs:

[5] https://docs.perl6.org/language/nativecall
[6] https://github.com/niner/Inline-Python

```
use Inline::Python;

my $py = Inline::Python.new;
$py.run: 'print("Hello, Perl 6")';
```

Besides the `run` method, which takes a string of Python code and executes it, you can also use `call` to call Python routines by specifying the namespace, the routine to call, and a list of arguments:

```
use Inline::Python;

my $py = Inline::Python.new;
$py.run('import datetime');
my $date = $py.call('datetime', 'date', 2017, 1, 31);
$py.call('__builtin__', 'print', $date);    # 2017-01-31
```

The arguments that you pass to `call` are Perl 6 objects, such as the three `Int` objects in this example. `Inline::Python` automatically translates them into the corresponding Python built-in data structure. It translates numbers, strings, arrays, and hashes. Return values are also translated in the opposite direction, though since Python 2 does not distinguish properly between byte and Unicode strings, Python strings end up as buffers in Perl 6.

Objects that `Inline::Python` cannot translate are handled as opaque objects on the Perl 6 side. You can pass them back into Python routines (as shown with the `print` call above) and you can call methods on them:

```
say $date.isoformat().decode;               # 2017-01-31
```

Perl 6 exposes attributes through methods, so Perl 6 has no syntax for accessing attributes from foreign objects directly. For instance, if you try to access the `year` attribute of `datetime.date` through the normal method call syntax, you get an error:

```
say $date.year;
```

dies with

```
'int' object is not callable
```

Instead, you have to use the `getattr` builtin:

```
say $py.call('__builtin__', 'getattr', $date, 'year');
```

12.4 Using the Bridge to Plot

We need access to two namespaces in Python, `datetime` and `matplotlib.pyplot`, so let's start by importing them and writing some short helpers:

```
my $py = Inline::Python.new;
$py.run('import datetime');
$py.run('import matplotlib.pyplot');
sub plot(Str $name, |c) {
    $py.call('matplotlib.pyplot', $name, |c);
}

sub pydate(Str $d) {
    $py.call('datetime', 'date', $d.split('-').map(*.Int));
}
```

We can now call `pydate('2017-03-01')` to create a Python `datetime.date` object from an ISO-formatted string and call the `plot` function to access functionality from matplotlib:

```
my ($figure, $subplots) = plot('subplots');
$figure.autofmt_xdate();

my @dates = %dates.keys.sort;
$subplots.plot:
    $[@dates.map(&pydate)],
    $[ %dates{@dates} ],
    label     => 'Total',
    marker    => '.',
    linestyle => '';
```

The Perl 6 call `plot('subplots')` corresponds to the Python code `fig, subplots = plt.subplots()`. Passing arrays to Python functions needs a bit of extra work, because `Inline::Python` flattens arrays. Using an extra `$` sigil in front of an array puts it into an extra scalar and thus prevents the flattening.

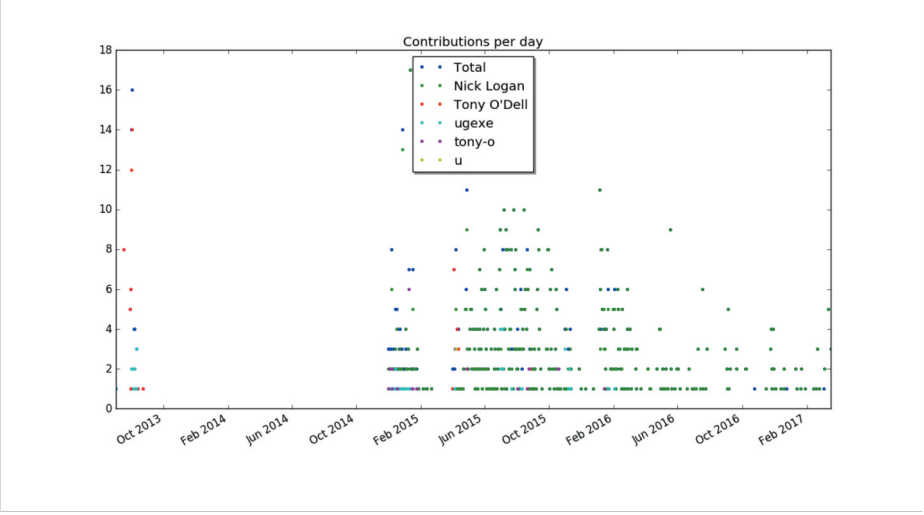
Now we can actually plot the number of commits by author, add a legend, and plot the result:

```
for @top-authors -> $author {
    my @dates = %by-author{$author}.keys.sort;
    my @counts = %by-author{$author}{@dates};
    $subplots.plot:
        $[ @dates.map(&pydate) ],
        $@counts,
        label     => $author,
        marker    =>'.',
        linestyle => '';
}


$subplots.legend(loc=>'upper center', shadow=>True);
```

```
plot('title', 'Contributions per day');
plot('show');
```

When run in the zef git repository[7], it produces this plot:



Contributions to zef, a Perl 6 module installer

12.5 Stacked Plots

I am not yet happy with the plot, so I want to explore using stacked plots for presenting the same information. In a regular plot, the y-coordinate of each plotted value is proportional to its value. In a stacked plot, it is the distance to the previous value that is proportional to its value. This is nice for values that add up to a total that is also interesting.

Matplotlib offers a method called `stackplot`[8] for this task. Contrary to multiple `plot` calls on a subplot object, it requires a shared x-axis for all data series. Hence we must construct one array for each author of git commits, where dates with no value are set to zero.

This time we have to construct an array of arrays where each inner array has the values for one author:

[7] https://github.com/ugexe/zef
[8] http://matplotlib.org/devdocs/api/_as_gen/matplotlib.axes.Axes.stackplot.html

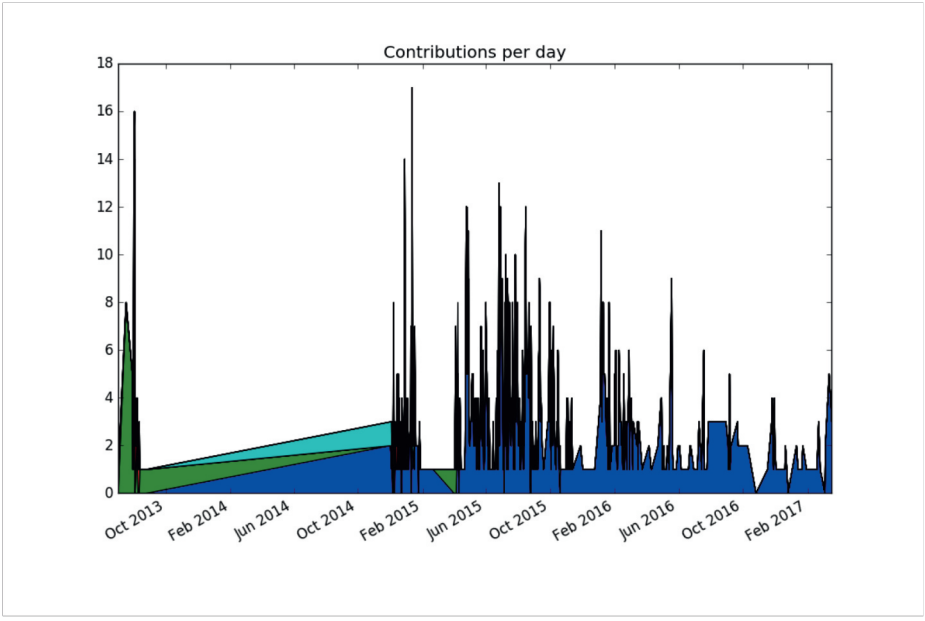
```
my @dates = %dates.keys.sort;
my @stack = $[] xx @top-authors;

for @dates -> $d {
    for @top-authors.kv -> $idx, $author {
        @stack[$idx].push: %by-author{$author}{$d} // 0;
    }
}
```

Now plotting becomes a simple matter of a method call, followed by the usual commands to add a title and show the plot:

```
$subplots.stackplot($[@dates.map(&pydate)], @stack);
plot('title', 'Contributions per day');
plot('show');
```

The result (again run on the zef source repository) is this:



Stacked plot of zef contributions over time

Comparing this to the previous visualization reveals a discrepancy: There were no commits in 2014, and yet the stacked plot makes it appear this way. In fact, the previous plots would have shown the same "alternative facts" if we had chosen lines instead of points. It comes from matplotlib (like nearly all plotting libraries) interpolates linearly between data points. But in our case, a date with no data points means zero commits happened on that date.

To communicate this to matplotlib, we must explicitly insert zero values for missing dates. This can be achieved by replacing

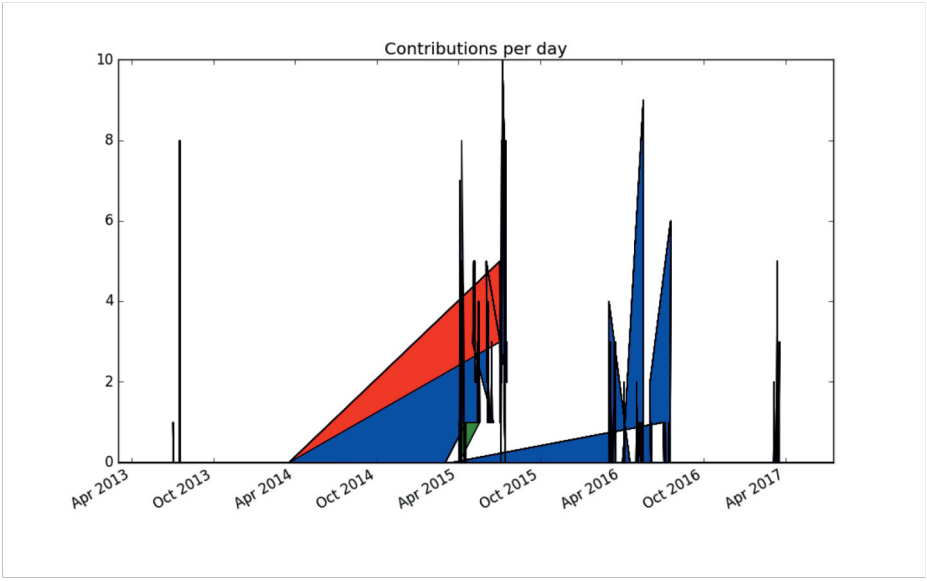
```
my @dates = %dates.keys.sort;
```

with the line

```
my @dates = %dates.keys.minmax;
```

The `minmax` method[9] finds the minimal and maximal values, and returns them in a Range[10]. Assigning the range to an array turns it into an array of all values between the minimal and the maximal value. The logic for assembling the `@stack` variable already maps missing values to zero.

The result looks a bit better, but still far from perfect:



Stacked plot of zef contributions over time, with missing dates mapped to zero

Thinking more about the problem, contributions from separate days should not be joined together, because it produces misleading results. Matplotlib doesn't support adding a legend automatically to stacked plots, so this seems to be to be a dead end.

Since a dot plot didn't work very well, let's try a different kind of plot that represents each data point separately: a bar chart, or more specifically, a stacked bar chart. Matplotlib offers the `bar` plotting method where the named parameter `bottom` can be used to generate the stacking:

[9] https://docs.perl6.org/routine/minmax#class_Any
[10] https://docs.perl6.org/type/Range

```
my @dates = %dates.keys.sort;
my @stack = $[] xx @top-authors;
my @bottom = $[] xx @top-authors;

for @dates -> $d {
    my $bottom = 0;
    for @top-authors.kv -> $idx, $author {
        @bottom[$idx].push: $bottom;
        my $value = %by-author{$author}{$d} // 0;
        @stack[$idx].push: $value;
        $bottom += $value;
    }
}
```

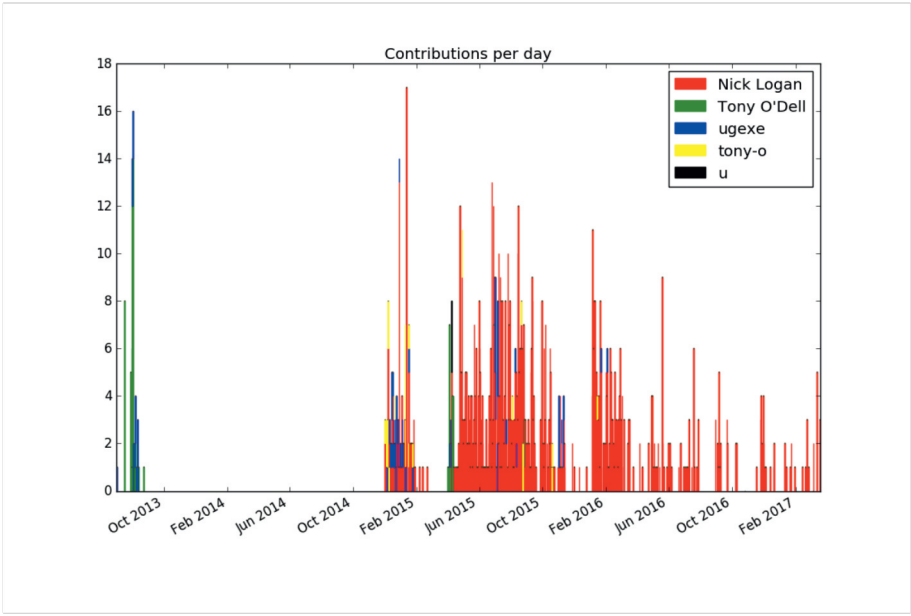
We need to supply color names ourselves and set the edge color of the bars to the same color, otherwise the black edge color dominates the result:

```
my $width = 1.0;
my @colors = <red green blue yellow black>;
my @plots;

for @top-authors.kv -> $idx, $author {
    @plots.push: plot(
        'bar',
        $[@dates.map(&pydate)],
        @stack[$idx],
        $width,
        bottom => @bottom[$idx],
        color => @colors[$idx],
        edgecolor => @colors[$idx],
    );
}
plot('legend', $@plots, $@top-authors);

plot('title', 'Contributions per day');
plot('show');
```

This produces the first plot that's actually informative and not misleading (provided you're not color blind):



Stacked bar plot of zef contributions over time

If you want to improve the result further, you could experiment with limiting the number of bars by lumping together contributions by week or month (or maybe `$n`-day period).

12.6 Idiomatic Use of Inline::Python

Now that the plots look informative and correct, it's time to explore how to better emulate the typical Python APIs through `Inline::Python`.

Types of Python APIs

Python is an object-oriented language, so many APIs involve method calls, which `Inline::Python` helpfully automatically translates for us.

But the objects must come from somewhere and typically this is by calling a function that returns an object, or by instantiating a class. In Python, those two are really the same under the hood, since instantiating a class is the same as calling the class as if it were a function.

An example of this (in Python) would be

```
from matplotlib.pyplot import subplots
result = subplots()
```

But the matplotlib documentation tends to use another, equivalent syntax:

```
import matplotlib.pyplot as plt
result = plt.subplots()
```

This uses the `subplots` symbol (class or function) as a method on the module `matplotlib.pyplot`, which the import statement aliases to `plt`. This is a more object-oriented syntax for the same API.

Mapping the Function API

The previous code examples used this Perl 6 code to call the `subplots` symbol:

```
my $py = Inline::Python.new;
$py.run('import matplotlib.pyplot');
sub plot(Str $name, |c) {
    $py.call('matplotlib.pyplot', $name, |c);
}

my ($figure, $subplots) = plot('subplots');
```

If we want to call `subplots()` instead of `plot('subplots')`, and `bar(args)` instead of 'plot('bar', args), we can use a function to generate wrapper functions:

```
my $py = Inline::Python.new;

sub gen(Str $namespace, *@names) {
    $py.run("import $namespace");

    return @names.map: -> $name {
        sub (|args) {
            $py.call($namespace, $name, |args);
        }
    }
}

my (&subplots, &bar, &legend, &title, &show)
    = gen('matplotlib.pyplot', <subplots bar legend title show>);

my ($figure, $subplots) = subplots();

# more code here

legend($@plots, $@top-authors);
title('Contributions per day');
show();
```

This makes the functions' usage quite nice, but comes at the cost of duplicating their names. One can view this as a feature, because it allows the creation of different aliases, or as a source for bugs when the order is messed up, or a name misspelled.

How could we avoid the duplication should we choose to create wrapper functions?

This is where Perl 6's flexibility and introspection abilities pay off. There are two key components that allow a nicer solution: the fact that declarations are expressions and that you can introspect variables for their names.

The first part means you can write `mysub my ($a, $b)`, which declares the variables `$a` and `$b`, and calls a function with those variables as arguments. The second part means that `$a.VAR.name` returns a string `'$a'`, the name of the variable.

Let's combine this to create a wrapper that initializes subroutines for us:

```
sub pysub(Str $namespace, |args) {
    $py.run("import $namespace");

    for args[0] <-> $sub {
        my $name = $sub.VAR.name.substr(1);
        $sub = sub (|args) {
            $py.call($namespace, $name, |args);
        }
    }
}

pysub 'matplotlib.pyplot',
    my (&subplots, &bar, &legend, &title, &show);
```

This avoids duplicating the name, but forces us to use some lower-level Perl 6 features in sub `pysub`. Using ordinary variables means that accessing their `.VAR.name` results in the name of the variable, not the name of the variable that's used on the caller side. So we can't use slurpy arguments as in

```
sub pysub(Str $namespace, *@subs)
```

Instead we must use `|args` to obtain the rest of the arguments in a Capture[11]. This doesn't flatten the list of variables passed to the function, so when we iterate over them, we must do so by accessing `args[0]`. By default, loop variables are read-only, which we can avoid by using `<->` instead of `->` to introduce the signature. Fortunately, that also preserves the name of the caller side variable.

An Object-Oriented Interface

Instead of exposing the functions, we can also create types that emulate the method calls on Python modules. For that we can implement a class with a method `FALLBACK`, which Perl 6 calls for us when calling a method that is not implemented in the class:

[11] https://docs.perl6.org/type/Capture.html

```
class PyPlot is Mu {
    has $.py;
    submethod TWEAK {
        $!py.run('import matplotlib.pyplot');
    }
    method FALLBACK($name, |args) {
        $!py.call('matplotlib.pyplot', $name, |args);
    }
}

my $pyplot = PyPlot.new(:$py);
my ($figure, $subplots) = $pyplot.subplots;
# plotting code goes here
$pyplot.legend($@plots, $@top-authors);

$pyplot.title('Contributions per day');
$pyplot.show;
```

Class `PyPlot` inherits directly from `Mu`, the root of the Perl 6 type hierarchy, instead of `Any`, the default parent class (which in turn inherits from `Mu`). `Any` introduces a large number of methods that Perl 6 objects get by default and since `FALLBACK` is only invoked when a method is not present, this is something to avoid.

The method `TWEAK` is another method that Perl 6 calls automatically for us, after the object has been fully instantiated. All-caps method names are reserved for such special purposes. It is marked as a `submethod`, which means it is not inherited into subclasses. Since `TWEAK` is called at the level of each class, if it were a regular method, a subclass would call it twice implicitly. Note that `TWEAK` is only supported in Rakudo version 2016.11 and later.

There's nothing specific to the Python package `matplotlib.pyplot` in class `PyPlot`, except the namespace name. We could easily generalize it to any namespace:

```
class PythonModule is Mu {
    has $.py;
    has $.namespace;
    submethod TWEAK {
        $!py.run("import $!namespace");
    }
    method FALLBACK($name, |args) {
        $!py.call($!namespace, $name, |args);
    }
}

my $pyplot = PythonModule.new(:$py, :namespace<matplotlib.pyplot>);
```

This is one Perl 6 type that can represent any Python module. If instead we want a separate Perl 6 type for each Python module, we could use roles, which are optionally parameterized:

```
role PythonModule[Str $namespace] is Mu {
    has $.py;
    submethod TWEAK {
        $!py.run("import $namespace");
    }
    method FALLBACK($name, |args) {
        $!py.call($namespace, $name, |args);
    }
}

my $pyplot = PythonModule['matplotlib.pyplot'].new(:$py);
```

Using this approach, we can create type constraints for Python modules in Perl 6 space:

```
sub plot-histogram(PythonModule['matplotlib.pyplot'], @data) {
    # implementation here
}
```

Passing in any other wrapped Python module than `matplotlib.pyplot` results in a type error.

12.7 Summary

We've explored several ways to represent commit occurrence in plots, and utilized `Inline::Python` to interface with a Python based plotting library.

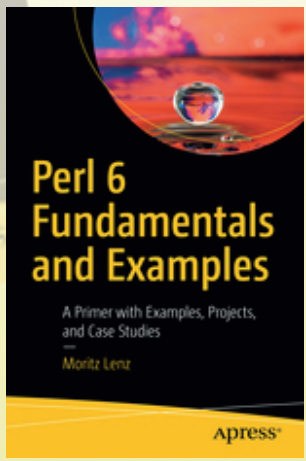
A bit of Perl 6 meta programming allowed us to emulate different kinds of Python APIs pretty directly in Perl 6 code, allowing a pretty direct translation of the original library's documentation into Perl 6 code.

Raku Fundamentals
A Primer with Examples, Projects, and Case Studies
Moritz Lenz

http://www.apress.com/us/book/9781484228982#aboutBook

Gain the skills to begin developing Raku applications from the ground up in this hands-on compact book, which includes a foreword from Larry Wall, creator of Perl. You'll learn enough to get started building with Raku, using Perl 6's gradual typing, handy object orientation features, powerful parsing capabilities, and human-usable concurrency. After a short introduction, each chapter develops a small example project, explaining the Raku features used. When the example is done, you'll explore another aspect, such as optimizing further for readability or testing the code.

Along the way you'll see Raku basics, such as variables and scoping; subroutines; classes and objects; regexes; and code testing. When you've mastered the basics, Raku Fundamentals moves onto more advanced topics to give you a deeper understanding of the language. You'll learn, amongst other things, how to work with persistent storage, how to generate good error messages, and how to write tricky applications such as a file and directory usage graph and a Unicode search tool.



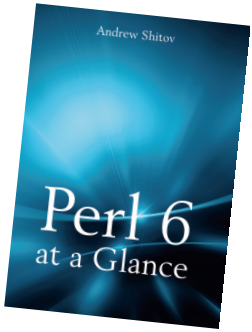
What You'll Learn

- Get coding with Raku
- Work on several hands-on examples and projects
- Integrate Python libraries into your Raku programs
- Parse INI files using regexes and grammars
- Build a date time converter
- Carry out refactoring and other automated tests

Who This Book Is For

If you already know one or more programming languages, and want to learn about Raku, this book is for you.

Moritz Lenz is a Raku core developer. He has contributed significantly to the official test suite, the Raku compiler, and is the initiator of the official Perl 6 documentation project. He has also authored several modules, and runs infrastructure for the Raku community. In his day job, he develops Perl 5 and Python code, and maintains a Continuous Delivery system for his employer.



Regexes and Grammars

Grammars in Raku are the “next level” of the well-known regular expressions. Grammars let you create much more sophisticated text parsers. A new domain-specific language (DSL), language translator, or interpreter can be created without any external help, using only the facilities that Raku offers with grammars.

Regexes

In fact, Raku just calls regular expressions regexes. The basic syntax is a bit different from Perl 5, but most elements (such as quantifiers * or +) still look familiar. The regex keyword is used to build a regex. Let us create a regex for the short names of weekdays.

```
my regex weekday
  {[Mon | Tue | Wed | Thu | Fri | Sat | Sun]};
```

The square brackets are enclosing the list of alternatives. You can use the named regex inside other regexes by referring to its name in a pair of angle brackets. To match the string against a regex, use the smartmatch operator (~~).

```
say 'Thu' ~~ m/<weekday>/;
say 'Thy' ~~ m/<weekday>/;
```

These two matches will print the following.

```
[Thu]
 weekday => [Thu]
False
```

The result of matching is an object of the Match type. When you print it, you will see the matched substring inside small square brackets

```
[...].
```

Regexes are the simplest named constructions. Apart from that, rules and tokens exist (and thus, the keywords rule and token). Tokens are different from rules first regarding how they handle whitespaces. In rules, whitespaces are part of the regexes. In tokens, whitespaces are just visual separators. We will see more about this in the examples below.

```
my token number_token { <[\d]> <[\d]> }
my rule number_rule { <[\d]> <[\d]> }
(Note that there is no need in semicolons after the closing brace.)
```

The <[...]> construction creates a character class. In the example above, the two-character string 42 matches with the number_token token but not with the number_rule rule.

```
say 1 if "42" ~~ /<number_token>/;
say 1 if "4 2" ~~ /<number_rule>/;
```

The $/ object

As we have just seen, the smartmatch operator comparing a string with a regex returns an object of the Match type. This object is stored in the $/ variable. It also contains all the matching substrings. To keep (catch) the substring a pair of parentheses is used. The first match is indexed as 0, and you may access it as an array

element either using the full syntax $/[0] or the shortened one: $0.
Remember that even the separate elements like $0 or $0 still contain objects of the Match type. To cast them to strings or numbers, coercion syntax can be used. For example, ~$0 converts the object to a string, and +$0 converts it to an integer.

```
'Wed 15' ~~ /(\w+) \s (\d+)/;
say ~$0; # Wed
say +$1; # 15
```

Grammars

Grammars are the development of regular expressions. Syntactically, the grammar is defined similar to a class but using the keyword grammar. Inside, it contains tokens and rules. In the next section, we will be exploring the grammar in the examples.
Simple parser
The first example of the grammar application is on grammar for tiny language that defines an assignment operation and contains the printing instruction. Here is an example of a programme in this language.

```
x = 42;
y = x;
print x;
print y;
print 7;
```

Let's start writing the grammar for the language. First, we have to express the fact that a programme is a sequence of statements separated by a semicolon. Thus, at the top level the grammar looks like this:

```
grammar Lang {
    rule TOP {
        ^ <statements> $
    }
    rule statements {
        <statement>+ %% ';'
    }
}
```

Here, Lang is the name of the grammar, and TOP is the initial rule from which the parsing will be started. The rule's content is a regex surrounded by with a pair of symbols, ^ and $, to tie the rule to the beginning and the end of the text. In other words, the whole programme should match the TOP rule. The central part of the rule, <statements>, refers to another rule. Rules will ignore all the spaces between their parts. Thus, you may freely add spaces to the grammar definition to make it easily readable.
The second rule explains the meaning of statements. The statements block is a sequence of separate statements. It should contain at least one statement, as the + quantifier requires, and the delimiter character is a semicolon. The separator is mentioned after the %% symbol. In grammar, this means that you must have the separator character between instructions, but you can omit it after the last one. If there's only one percent character instead of two, the rule will also require the separator after the last statement.
The next step is to describe the statement. At the moment, our language has only two operations: assignment and printing. Each of them accepts either a value or a variable

name.

```
rule statement {
   | <assignment>
   | <printout>
}
```

The vertical bar separates alternative branches like it does in regular expressions in Perl 5. To make the code a bit better-looking and simplify the maintenance, an extra vertical bar may be added before the first subrule. The following two descriptions are identical:

```
rule statement {
     <assignment>
   | <printout>
}
rule statement {
   | <assignment>
   | <printout>
}
```

Then, let us define what do assignment and printout mean.

```
rule assignment {
   <identifier> '=' <expression>
}
rule printout {
   'print' <expression>
}
```

Here, we see literal strings, namely, '=' and 'print'. Again, the spaces around them do not affect the rule.
An expression matches with either an identifier (which is a variable name in our case) or with a constant value. Thus, an expression is either an identifier or a value with no additional strings.

```
rule expression {
   | <identifier>
   | <value>
}
```

At this point, we should write the rules for identifiers and values. It is better to use another method, named token, for that kind of the grammar bit. In tokens, the spaces matter (except for those that are adjacent to the braces).
An identifier is a sequence of letters:

```
token identifier {
   <:alpha>+
}
```

Here, <:alpha> is a predefined character class containing all the alphabetical characters. A value in our example is a sequence of digits, so we limit ourselves to integers only.

```
token value {
   \d+
}
```

Our first grammar is complete. It is now possible to use it to parse a text file.

```
my $parsed = Lang.parsefile('test.lang');
```

If the file content is already in a variable, you may use the Lang.parse($str) method to parse it. (There is more about reading from files in the Appendix.)
If the parsing was successful, that if the file contains a valid programme, the $parse variable will contain an object of the Match type. It is possible to dump it (say $parsed) and see what's there.

```
[x = 42;
y = x;
print x;
print y;
print 7;
]
 statements => [x = 42;
y = x;
print x;
print y;
print 7;
]
  statement => [x = 42]
   assignment => [x = 42]
    identifier => [x]
    expression => [42]
     value => [42]
  statement => [y = x]
   assignment => [y = x]
    identifier => [y]
    expression => [x]
     identifier => [x]
  statement => [print x]
   printout => [print x]
    expression => [x]
     identifier => [x]
  statement => [print y]
   printout => [print y]
    expression => [y]
     identifier => [y]
  statement => [print 7]
   printout => [print 7]
    expression => [7]
     value => [7]
```

This output corresponds to the sample programme from the beginning of this section. It contains the structure of the parsed programme. The captured parts are displayed in the brackets[...]. First, the whole matched text is printed. Indeed, as the TOP rule uses the pair of anchors ^ ... $, and so the whole text should match the rule.
Then, the parse tree is printed. It starts with the <statements>, and then the other parts of the grammar are presented in full accordance with what the programme in the file contains. On the next level, you can see the content of both the identifier and value tokens.
If the programme is grammatically incorrect, the parsing methods will return an empty value (Any). The same will happen if only the initial part of the programme

matches the rules.
Here is the whole grammar for your convenience:

```
grammar Lang {
    rule TOP {
        ^ <statements> $
    }
    rule statements {
        <statement>+ %% ';'
    }
    rule statement {
        | <assignment>
        | <printout>
    }
    rule assignment {
        <identifier> '=' <expression>
    }
    rule printout {
        'print' <expression>
    }
    rule expression {
        | <identifier>
        | <value>
    }
    token identifier {
        <:alpha>+
    }
    token value {
        \d+
    }
}
```

An interpreter

So far, the grammar sees the structure of the programme and can tell if it is grammatically correct, but it does not execute any instructions contained in the programme. In this section, we will extend the parser so that it can actually execute the programme.
Our sample language uses variables and integer values. The values are constants and describe themselves. For the variables, we need to create a storage. In the simplest case, all the variables are global, and a single hash is required: my %var;.
The first action that we will implement now, is an assignment. It will take the value and save it in the variable storage. In the assignment rule in the grammar, an expression is expected on the right side of the equals sign. An expression can be either a variable or a number. To simplify the variable name lookup, let's make the grammar a bit more complicated and split the rules for assignments and printing out into two alternatives.

```
rule assignment {
    | <identifier> '=' <value>
    | <identifier> '=' <identifier>
}
rule printout {
    | 'print' <value>
    | 'print' <identifier>
```

```
}
```

Actions

The grammars in Raku allow actions in response to the rule or token matching. Actions are code blocks that are executed when the corresponding rule or token is found in the parsed text. Actions receive an object $/, where you can see the details of the match. For example, the value of $<identifier> will contain an object of the Match type with the information about the substring that actually was consumed by the grammar.

```
rule assignment {
    | <identifier> '=' <value>
        {say "$<identifier>=$<value>"}
    | <identifier> '=' <identifier>
}
```

If you update the grammar with the action above and run the programme against the same sample file, then you will see the substring x=42 in the output. The Match objects are converted to strings when they are interpolated in double quotes as in the given example: "$<identifier>=$<value>". To use the text value from outside the quoted string, you should make an explicit typecast:

```
rule assignment {
    | <identifier> '=' <value>
        {%var{~$<identifier>} = +$<value>}
    | <identifier> '=' <identifier>
}
```

So far, we've got an action for assigning a value to a variable and can process the first line of the file. The variable storage will contain the pair {x => 42}. In the second alternative of the assignment rule, the <identifier> name is mentioned twice; that is why you can reference it as to an array element of $<identifier>.

```
rule assignment {
    | <identifier> '=' <value>
      {
          %var{~$<identifier>} = +$<value>
      }
    | <identifier> '=' <identifier>
      {
          %var{~$<identifier>[0]} =
          %var{~$<identifier>[1]}
      }
}
```

This addition to the code makes it possible to parse an assignment with two variables: y = x. The %var hash will contain both values: {x => 42, y => 42}. Alternatively, capturing parentheses may be used. In this case, to access the captured substring, use special variables, such as $0:

```
rule assignment {
    | (<identifier>) '=' (<value>)
      {
          %var{$0} = +$1
```

```
    }
  | (<identifier>) '=' (<identifier>)
    {
        %var{$0} = %var{$1}
    }
}
```

Here, the unary ~ is no longer required when the variable is used as a hash key, but the unary + before $1 is still needed to convert the Match object to a number. Similarly, create the actions for printing.

```
rule printout {
  | 'print' <value>
    {
       say +$<value>
    }
  | 'print' <identifier>
    {
       say %var{$<identifier>}
    }
}
```

Now, the grammar is able to do all the actions required by the language design, and it will print the requested values:

```
42
42
7
```

As soon as we used capturing parentheses in the rules, the parse tree will contain entries named as 0 and 1, together with the named strings, such as identifier. You can clearly see it when parsing the y = x string:

```
statement => [y = x]
 assignment => [y = x]
  0 => [y]
   identifier =>[y]
  1 => [x]
   identifier => [x]
```

An updated parser looks like this:

```
my %var;

grammar Lang {
   rule TOP {
      ^ <statements> $
   }

   rule statements {
      <statement>+ %% ';'
   }
   rule statement {
      | <assignment>
      | <printout>
   }

   rule assignment {
      | (<identifier>) '=' (<value>)
        {
           %var{$0} = +$1
        }
```

```
    | (<identifier>) '=' (<identifier>)
      {
        %var{$0} = %var{$1}
      }
  }

  rule printout {
    | 'print' <value>
      {
        say +$<value>
      }
    | 'print' <identifier>
      {
        say %var{$<identifier>}
      }
  }

  token identifier {
    <:alpha>+
  }
  token value {
    \d+
  }
}

Lang.parsefile('test.lang');
```

For convenience, it is possible to put the code of actions in a separate class. This helps a lot when the actions are more complex and contain more than one or two lines of code.

To create an external action, create a class, which will later be referenced via the :actions parameter upon the call of the parse or parsefile methods of the grammar. As with built-in actions, the actions in an external class receive the $/ object of the Match type.

First, we will train on a small isolated example and then return to our custom language parser.

```
grammar G {
  rule TOP {^ \d+ $}
}

class A {
  method TOP($/) {say ~$/}
}

G.parse("42", :actions(A));
```

Both the grammar G and the action class A have a method called TOP. The common name connects the action with the corresponding rule. When the grammar parses the provided test string and consumes the value of 42 by the ^ \d $ rule, the A::TOP action is triggered, and the $/ argument is passed to it, which is immediately printed.

AST and attributes

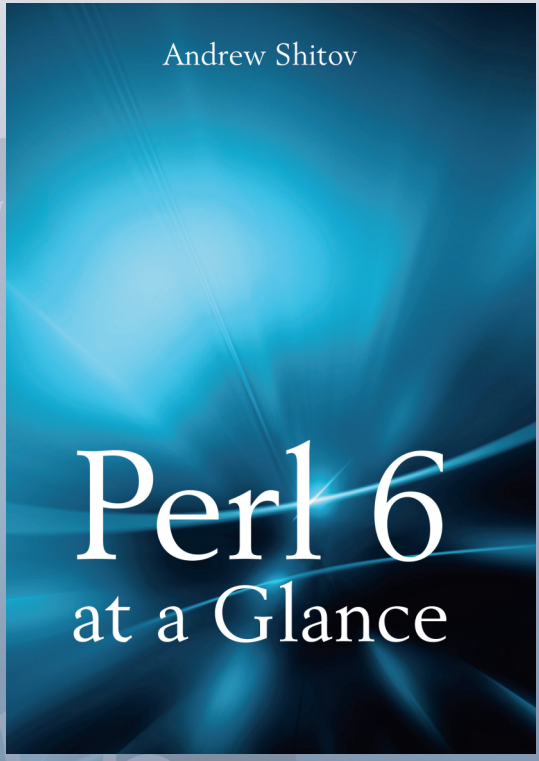
Now, we are ready to simplify the grammar again after we split the assignment and printout rules into two alternatives each. The difficulty was that without the split, it

Raku at a Glance

Andrew Shitov

https://deeptext.media/perl6-at-a-glance/

This book is about Raku, a programming language of the Perl family. It covers many basic and deep topics of the language and provides the initial knowledge you need to start working with Raku. The book does not require any previous knowledge of Perl.



The Raku programming language has a long story. It emerged in 2000 and got its first stable specification in 2015. This book is the first one based on the standard 6.c of the language.

A lot of examples and explanations let you follow the ideas of the language design and will help you to start programming in Raku right away.

was not possible to understand which branch had been triggered. You either needed to read the value from the value token or get the name of the variable from the identifier token and look it up in the variable storage.
Raku's grammars offer a great mechanism that is common in language parsing theory, the abstract syntax tree, shortened as AST. First of all, update the rules and remove the alternatives from some of them. The only rule containing two branches is the expression rule.

```
rule assignment {
    <identifier> '=' <expression>
}
rule printout {
    'print' <expression>
}
rule expression {
    | <identifier>
    | <value>
}
```

The syntax tree that is built during the parse phase can contain the results of the calculations in the previous steps. The Match object has a field ast, dedicated especially to keep the calculated values on each node. It is possible to simply read the value to get the result of the previously completed actions. The tree is called abstract because how the value is calculated is not very important. What is important is that when the action is triggered, you have a single place with the result you need to complete an action.
The action can save its own result (and thus pass it further on the tree) by calling the $/.make method. The data you save there are accessible via the made field, which has the synonym ast.
Let's fill the attribute of the syntax tree for the identifier and value tokens. The match with an identifier produces the variable name; when the value is found, the action generates a number. Here are the methods of the actions' class.

```
method identifier($/) {
    $/.make(~$0);
}
method value($/) {
    $/.make(+$0);
}
```

Move one step higher, where we build the value of the expression. It can be either a variable value or an integer. As the expression rule has two alternatives, the first task will be to understand which one matches. For that, check the presence of the corresponding fields in the $/ object. (If you use the recommended variable name $/ in the signature of the action method, you may access its fields differently. The full syntax is $/<identifier>, but there is an alternative version $<identifier>.)
The two branches of the expression method behave differently. For a number, it extracts the value directly from the captured substring. For a variable, it gets the value from the %var hash. In both cases, the result is stored in the AST using the make method.

```
method expression($/) {
    if $<identifier> {
```

```
            $/.make(%var{$<identifier>});
        }
        else {
            $/.make(+$<value>);
        }
    }
```

To use the variables that are not yet defined, we can add the defined-or operator to initialise the variable with the zero value.
$/.make(%var{$<identifier>} // 0); Now, the expression will have a value attributed to it, but the source of the value is not known anymore. It can be a variable value or a constant from the file. This makes the assignment and printout actions simpler:

```
method printout($/) {
    say $<expression>.ast;
}
```

All you need for printing the value is to get it from the ast field. For the assignment, it is a bit more complex but can still be written in a single line.

```
method assignment($/) {
    %var{$<identifier>} = $<expression>.made;
}
```

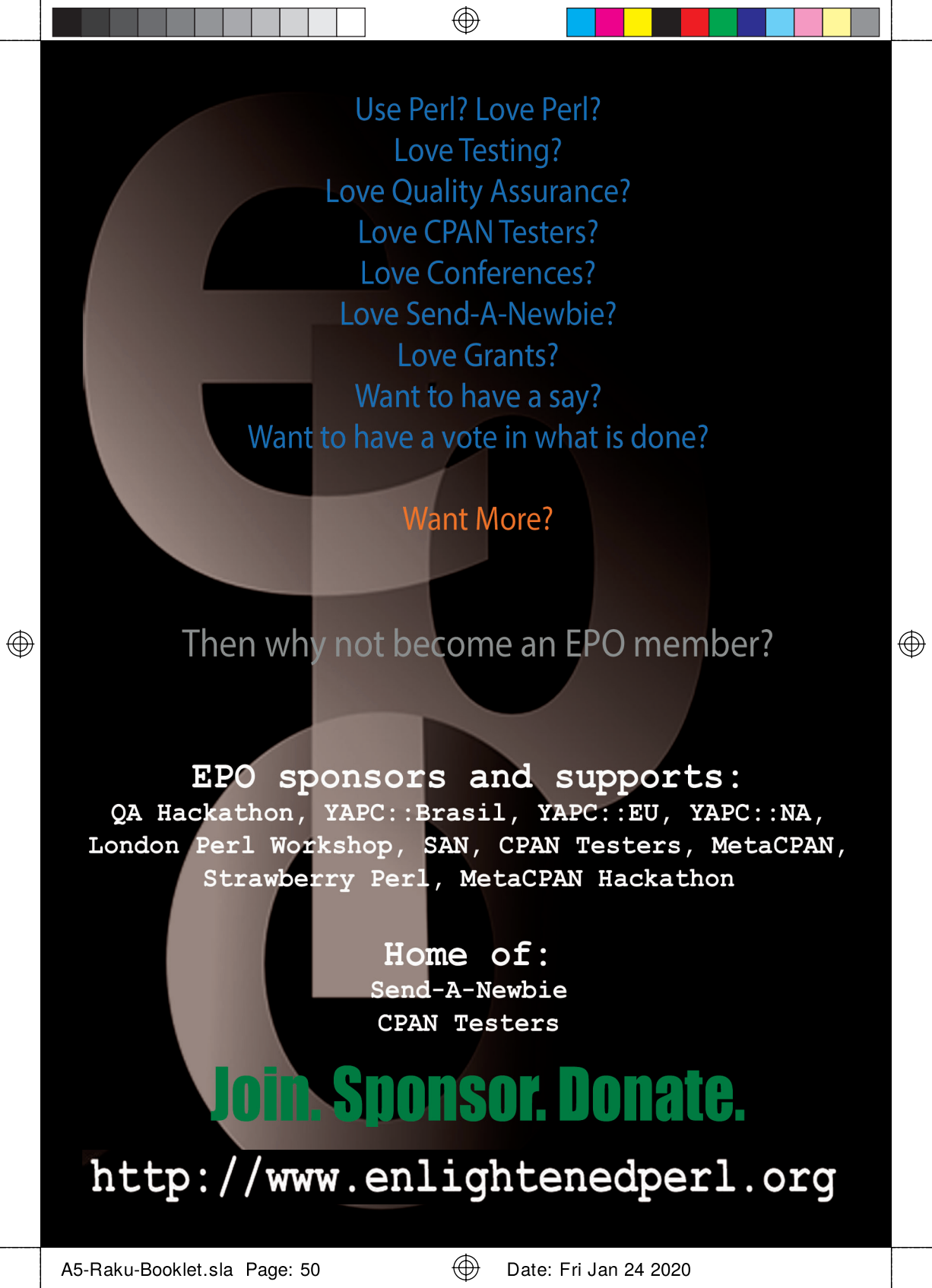
The method gets the $/ object and uses the values of its identifier and expression elements. The first one is converted to the string and becomes the key of the %var hash. From the second one, we get the value by fetching the made attribute.
Finally, let us stop using the global variable storage and move the hash into the action class (we don't need it in the grammar itself). It thus will be declared as has %!var; and used as a private key variable in the body of the actions: %!var{...}. After this change, it is important to create an instance of the actions class before paring it with a grammar:

```
Lang.parsefile(
    'test.lang',
    :actions(LangActions.new())
);
```

Here is the complete code of the parser with actions.

```
grammar Lang {
    rule TOP {
        ^ <statements> $
    }
    rule statements {
        <statement>+ %% ';'
    }
    rule statement {
        | <assignment>
        | <printout>
    }
    rule assignment {
        <identifier> '=' <expression>
    }
    rule printout {

        'print' <expression>
```


Use Perl? Love Perl?
Love Testing?
Love Quality Assurance?
Love CPAN Testers?
Love Conferences?
Love Send-A-Newbie?
Love Grants?
Want to have a say?
Want to have a vote in what is done?
Want More?
Then why not become an EPO member?
EPO sponsors and supports:
QA Hackathon, YAPC::Brasil, YAPC::EU, YAPC::NA,
London Perl Workshop, SAN, CPAN Testers, MetaCPAN,
Strawberry Perl, MetaCPAN Hackathon
Home of:
Send-A-Newbie
CPAN Testers
Join. Sponsor. Donate.
http://www.enlightenedperl.org

```
    }
    rule expression {
        | <identifier>
        | <value>
    }
    token identifier {
        (<:alpha>+)
    }
    token value {
        (\d+)
    }
}

class LangActions {
    has %var;

    method assignment($/) {
        %!var{$<identifier>} = $<expression>.made;
    }
    method printout($/) {
        say $<expression>.ast;
    }
    method expression($/) {
        if $<identifier> {
            $/.make(%!var{$<identifier>} // 0);
        }
        else {
            $/.make(+$<value>);
        }
    }
    method identifier($/) {
        $/.make(~$0);
    }
    method value($/) {
        $/.make(+$0);
    }
}

Lang.parsefile(
    'test.lang',
    :actions(LangActions.new())
);
```

Calculator

When considering language parsers, implementing a calculator is like writing a “Hello, World!” programme. In this section, we will create a grammar for the calculator that can handle the four arithmetical operations and parentheses. The hidden advantage of the calculator example is that you have to teach it to follow the operations priority and nested expressions.

Our calculator grammar will expect the single expression at a top level. The priority of operations will be automatically achieved by the traditional approach to grammar construction, in which the expression consists of terms and factors. The terms are parts separated by pluses and minuses:

```
<term>+ %% ['+'|'-']
```

Here, Raku's %% symbol is used. You may also rewrite the rule using more traditional quantifiers:

```
<term> [['+'|'-'] <term>]*
```

Each term is, in turn, a list of factors separated by the symbols for multiplication or division:

```
<factor>+  %% ['*'|'/']
```

Both terms and factors can contain either a value or a group in parentheses. The group is basically another expression.

```
rule group {
    '(' <expression> ')'
}
```

This rule refers to the expression rule and thus can start another recursion loop. It's time to introduce the enhancement of the value token so that it accepts the floating point values. This task is easy; it only requires creating a regex that matches the number in as many formats as possible. I will skip the negative numbers and the numbers in scientific notation.

```
token value {
    | \d+['.' \d+]*
    | '.' \d+
}
```

Here is the complete grammar of the calculator:

```
grammar Calc {
    rule TOP {
        ^ <expression> $
    }
    rule expression {
        | <term>+ %% $<op>=(['+'|'-'])
        | <group>
    }
    rule term {
        <factor>+  %% $<op>=(['*'|'/'])
    }
    rule factor {
        | <value>
        | <group>
    }
    rule group {
        '(' <expression> ')'
    }
    token value {
        | \d+['.' \d+]*
        | '.' \d+
    }
}
```

Note the $<op>=(...) construction in some of the rules. This is the named capture. The name simplifies the access to a value via the $/ variable. In this case, you can reach the

value as $<op>, and you don't have to worry about the possible change of the variable name after you update a rule as it happens with the numbered variables $0, $1, etc. Now, create the actions for the compiler. At the TOP level, the rule returns the calculated value, which it takes from the ast field of the expression.

```
class CalcActions {
    method TOP($/) {
        $/.make: $<expression>.ast
    }
    . . .
}
```

The actions of the underlying rules groups and value are as simple as we've just seen.

```
method group($/) {
    $/.make: $<expression>.ast
}

method value($/) {
    $/.make: +$/
}
```

The rest of the actions are a bit more complicated. The factor action contains two alternative branches, just as the factor rule does.

```
method factor($/) {
    if $<value> {
        $/.make: +$<value>
    }
    else {
        $/.make: $<group>.ast
    }
}
```

Move on to the term action. Here, we have to take care of the list with its variable length. The rule's regex has the + quantifier, which means that it can capture one or more elements. Also, as the rule handles both the multiplication and the division operators, the action must distinguish between the two cases. The $<op> variable contains either the * or the / character. This is how the syntax tree looks like for the string with three terms, 3*4*5:

```
expression => [3*4*5]
 term => [3*4*5]
  factor => [3]
   value => [3]
  op => [*]
  factor => [4]
   value => [4]
  op => [*]
 factor => [5]
  value => [5]
```

As you can see, there are factor and op entries on the top levels. You will see the values as $<factor> and $<op> inside the actions. At least one $<factor> will always be available. The values of the nodes will already be known and available in the ast property. Thus, all you need to do is to traverse over the elements of those two arrays and perform either multiplication or division.

```
method term($/) {
    my $result = $<factor>[0].ast;

    if $<op> {
        my @ops = $<op>.map(~*);
        my @vals = $<factor>[1..*].map(*.ast);

        for 0..@ops.elems - 1 -> $c {
            if @ops[$c] eq '*' {
                $result *= @vals[$c];
            }
            else {
                $result /= @vals[$c];
            }
        }
    }

    $/.make: $result;
}
```

In this code fragment, the star character appears in the new role of a placeholder that tells Raku that it should process the data that it can get at this moment. It sounds weird, but it works perfectly and intuitively. The @ops array with a list of the operation symbols consists of the elements that we got after stringifying the $<op>'s value:

```
my @ops = $<op>.map(~*);
```

The values themselves will land in the @vals array. To ensure that the values of the two arrays, @vals and @ops, correspond to each other, the slice of $<factor> is taken, which starts at the second element:

```
my @vals = $<factor>[1..*].map(*.ast);
```

Finally, the expression action is either to take the calculated value of group or to perform the sequence of additions and subtractions. The algorithm is close to the one of the term's action.

```
method expression($/) {
    if $<group> {
        $/.make: $<group>.ast
    }
    else {
        my $result = $<term>[0].ast;

        if $<op> {
            my @ops = $<op>.map(~*);
            my @vals = $<term>[1..*].map(*.ast);
            for 0..@ops.elems - 1 -> $c {
                if @ops[$c] eq '+' {
                    $result += @vals[$c];
                }
                else {
                    $result -= @vals[$c];
                }
            }
        }
        $/.make: $result;
```

```
    }
}
```

The majority of the code for the calculator is ready. Now, we need to read the string from the user, pass it to the parser, and print the result.

```
my $calc = Calc.parse(
        @*ARGS[0],
        :actions(CalcActions)
    );
say $calc.ast;
```

Let's see if it works.

```
$ perl6 calc.pl '39 + 3.14 * (7 - 18 / (505 - 502)) - .14'
42
```

It does. On github.com/ash/lang, you can find the continuation of the code demonstrated in this chapter, which combines both the language translator and the calculator to allow the user write the arithmetical expressions in the variable assignments and the print instructions. Here is an example of what that interpreter can process:

```
x = 40 + 2;
print x;
y = x - (5/2);
print y;
z = 1 + y * x;
print z;
print 14 - 16/3 + x;
```

Rakudo Star

Download Rakudo

Rakudo Star is a useful and usable production distribution of Raku which supports the latest Christmas Perl 6 (6.c language version).

Installing from binaries

Windows users can directly install the most recent 64 bit or 32 bit MSI of Rakudo Star from the download directory. Mac users can directly install using the most recent .dmg from the same location.

https://rakudo.org/files

Docker users can directly install with docker pull rakudo-star

Installing from source

People on UNIX-like systems can try out Rakudo Star from an extracted source tarball. You will need recent versions of perl, git, make and gcc. Also a basic familiarity with UNIX.

For full instructions go to http://rakudo.org/files and follow the installation instructions for your particular OS.

```
% wget https://rakudo.perl6.org/downloads/star/rakudo-star-2017.04.tar.gz
% tar xfz rakudo-star-2017.04.tar.gz
% cd rakudo-star-2017.04
# perl Configure.pl --gen-moar --prefix /opt/rakudo-star-2017.04
# make install
```

Run Raku

To run a Raku program, include the installation directory in your system PATH variable and issue a command like:

$ raku hello.rk

If you invoke Raku without an explicit script to run, it enters a small interactive mode that allows the execution of Raku statements from the command line.

```
# Custom operators
sub postfix:<!> (Int $n) {
    fail "Not a Natural Number in Factorial"
        if $n < 0;
    [*] 2..$n
}

use Test;
isa-ok (-1)!, Failure, "Factorial for -1 fails";
ok 0! == 1, "Factorial for 0";
ok 1! == 1, "Factorial for 1";
ok 5! == 120, "Factorial for a larger integer";
```

Raku and its VMs

Raku on MoarVM (see Rakudo Star, rakudo.org) is currently the most advanced implementation of Perl 6. Any implementation that passes the official test suite can call itself “Raku”.

Raku now runs on three virtual machines: MoarVM, JVM and JavaScript. MoarVM offers a wider support for specified features than Raku on JVM or JavaScript. MoarVM is solely developed for Raku and not (yet) used for other programming languages.

Modules for Raku

For a current list of Raku modules, look at https://modules.raku.org where you'll find modules from App::Mi6 to zef, and in between you'll find JSON, async IO support, database abstraction layers, web frameworks and a wealth of other modules created by Raku users.

You can get all of these modules with the 'zef' module installer, built in to your Raku distribution. Simply type 'zef install Module::Name' and it will download, test and install the module. Documentation is available online, or in the module's source, look in the module's source directory for more info.

Web developers should check out the fully asynchronous web framework Cro, and its website https://cro.services. Python users should try Inline::Python, and Perl users that want CPAN available should check out Inline::Perl5.

```
# Infinite Lazy Lists
my @fib = 0, 1, *+* ... *;
say "Fibonacci number #8 is @fib[7]";
```

https://rakudo.org/

Getting Involved

If you are inspired now and want to contribute to the Raku community, there are some resources available to you. A full list of these are on the inside cover of this booklet but we reccomend you start with the raku.org homepage.

IRC: the channel #raku on irc.freenode.net discusses all things Raku. The people are very friendly and very busy developing Raku. Keep an eye on this to stay up-to-date. The channel is logged, and you can read back to see what has been discussed: irclog.perlgeek.de/raku/today

Mailing lists

Send an email with subject 'subscribe' to:

perl6-announce-subscribe@perl.org	Announcements and news. Low traffic.
perl6-users-subscribe@perl.org	User questions and discussions regarding the Raku language and compilers.
perl6-language-subscribe@perl.org	For issues regarding the Raku language specification.
perl6-compiler-subscribe@perl.org	For issues regarding various Raku compilers.

Due to the recent rename, these lists will change to raku- in place of perl6

Download & Use

raku.org/getting-started	Getting started
rakudo.org	Download binaries, get the latest installation instructions.
moarvm.org	To learn more about the MoarVM virtual machine

Useful places to start learning Raku

perl6intro.com	Raku Introduction
learnxinyminutes.com/docs/perl6	Raku Learn x in y minutes
rosettacode.org/wiki/Category:Perl_6	Raku section of solving problems with different programming languages
stackoverflow.com/questions/tagged/raku	Raku questions and answers
rakuadventcalendar.wordpress.com	Raku Advent calendar: every day leading up to Advent, a new interesting Raku example
rakudoweekly.blog	Raku Weekly, with the latest developments in Raku curated by a core developer
6guts.wordpress.com	Perl 6 Guts, the newest innards of Perl 6 explained by Jonathan Worthington
theperlfisher.com	A blog on developing with Raku
strangelyconsistent.org	Strangely Consistent, an insightful blog by Carl Masak
pl6anet.org	Perl 6 Planet, a collection of blogs and articles
szabgab.com/perl6.html	Screencasts about Raku
raku-musings.com	Perl 6 Musings by Arne Sommer
blogs.perl.org/users/damian_conway	If you ever have the chance to attend a presentation by Damian, enjoy!
outreachy.org	J.J. Merelo is working with Outreachy to support internships in Raku and Open Source Software

Come and Join the Party!

The official full first stable version, Raku version 6.c, was released on Christmas of 2015. The whole world can use it now and be a part of how awesome programming can be!

It enhances Raku's long-term appeal with a proper object system including roles and multi-method dispatch.

Raku learns from other languages. For instance: taking threading from Java (simplified to a handful of methods); Foreign-function interfaces from Lisp make accessing libraries as simple as one line of code.

Regular expressions are now turned up to 11 with the introduction of Parser Expression Grammars, which let you tackle huge parsing tasks. Strictures and warnings are now automatic, cutting out huge swathes of potential errors.

Raku's mottos remain the same as Perl: “Perl is different. In a nutshell, Perl is designed to make the easy jobs easy, without making the hard jobs impossible.” and “There Is More Than One Way To Do It”. Now with even more -Ofun added.

Look inside for a list of some of Raku’s many features, an introduction to the community, book extracts and useful sites.

This Introduction booklet is free to you but not free to make!
We welcome donations to both the Enlightened Perl Organisation and The Perl Foundation to support the creation of more materials, support conferences, programmers and attendees and to enhance the community.

Updated 24/01/2020 SJM TPF
Document source: https://github.com/tpf/marketing-materials

The Enlightened Perl Organisation
http://www.enlightenedperl.org