Beginning Raku

A Two Day Course

Arne Sommer

Beginning Raku

Arne Sommer

Version 1.02, 1.1.2020

Table of Contents

Introduction

This book is written as a companion textbook and reference for my 2 day course «Beginning Raku». See http://course.perl6.eu/beginning for details.

This book and the overhead presentation is in English, but the course can also be held in Norwegian.

I have chosen to cover as much of Raku as possible, and the result is a book that may use too little time on important features. (And I have certainly missed some features, as Raku is a large language.) I have tried to be as detailed as necessary to fully explain the feature. I have added links to further reading when I haven't covered everything. (I haven't always covered every corner case of a feature, so if you want to master a certain feature, do look it up in the online documentation.)

> You need a laptop (Linux, Windows, Mac) that you can install Raku (or Docker) on.
>
> You need some prior programming knowledge, in any language.
>
> This icon is used to show *important information.*

The only way to *really* learn a programming language is by using it. Write programs!

This book has several sample programs. They are shown like this, often with a file name:

File: echo-all

```
.say for lines;
```

Feel free to use the code samples, either as they are or as inspiration for your own work. Attribution would be nice, but isn't required.

> Course participants will get the sample programs as a Docker Image bundled with Raku, and as a zip file.
>
> This icon is used to show *tips.*

We'll show where you can look for more information in the More Information section in the first chapter.

> If you find an error, send an email to course@perl6.eu [mailto:course@perl6.eu] with «Error» as subject so that I can fix it.

The Little Print

I am giving away this first version of the book for free. I do reserve the right to print the book and sell it. You are free to distribute the pdf file or print it. You are also free to distribute printed copies, but you may not get paid for it.

The book is written i Asciidoctor.

⚠ This icon is used to show *warnings*.

Reading Tips

If you read the book from cover to cover, ignore forward references. They are meant as a help when the book is used as a reference.

Look up backward references if you have forgotten the content.

Happy Reading. And Coding.

Arne Sommer, Oslo. December 2019

Content

Chapter 1. About Raku

Raku was formerly known as Perl 6, so you will see that name for quite some time before the transition is completely finished.

1.1. Rakudo

Rakudo is a production ready implementation of Raku, written in NQP («Not Quite Perl»), running on the dedicated MoarVM («Metamodel On A Runtime») virtual machine.

MoarVM is available for Windows, MacOS and Linux (and some other operating systems).

Rakudo has monthly releases. (Or at least, intends to have them.)

Implementations of Rakudo running on jvm and javascript (node.js) are not as complete.

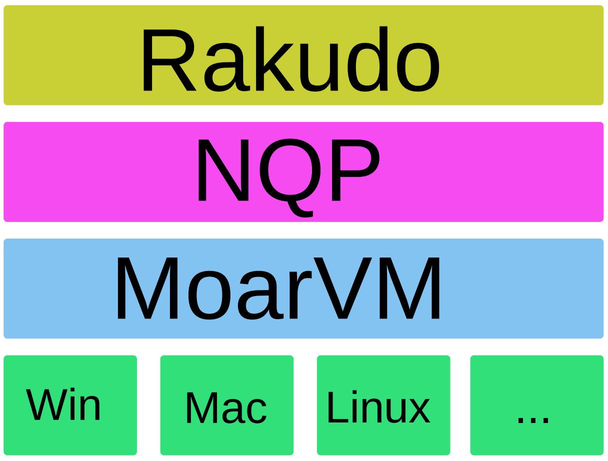


Figure 1. The Rakudo implementation of Raku

1.1.1. Rakudo Star

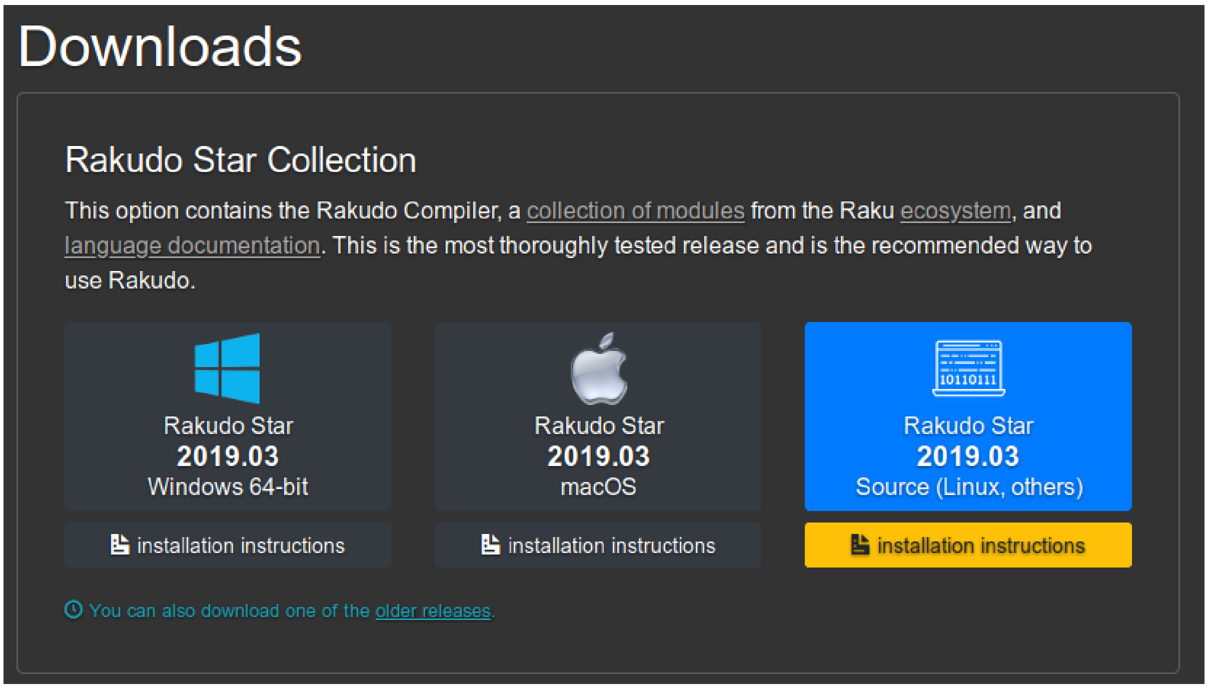
Rakudo Star is released every third month. This is Rakudo bundled with documentation (the `p6doc` command, see section 1.8, "Documentation") and a selection of useful modules (especially the module installer `zef`; see section 12.2, "Module Administration with zef").

This is the easiest way to install Rakudo.

If you are running Linux or MacOS, using Docker and a Rakudo image is an option. See *Appendix 1. Docker* before deciding.

1.1.2. Installing Rakudo Star

Rakudo Star: Go to https://rakudo.org/files


Downloads

Rakudo Star Collection

This option contains the Rakudo Compiler, a collection of modules from the Raku ecosystem, and language documentation. This is the most thoroughly tested release and is the recommended way to use Rakudo.

You can also download one of the older releases.

Figure 2. Rakudo Star Download Page

- Windows & Mac: Use the installation binary. See https://rakudo.org/files/
- Linux: Use the normal package system (Debian, Centos, Fedora, openSUSE, Ubuntu and Alpine). See https://nxadm.github.io/rakudo-pkg/

More information: https://raku.org/downloads/

1.2. Running Raku in the browser

It is possible to run programs in the browser, on these websites:

- https://glot.io/new/perl6
- https://tio.run/#perl6

Note that they do not support modules, and anything changing the state of the server (as file operations) is bound to fail. REPL mode (see the next section) may be a better choice, but that does require installation of Raku.

1.3. REPL

Run `raku` without any arguments to start it in interactive, or `REPL` (Read-Eval-Print Loop), mode.

```
$ raku
To exit type 'exit' or '^D'
> say 12; my $a = False;
12
> my $a = False;
False
```

Note that `REPL` mode always displays a value. If your code prints a value, that is fine. But if not, `REPL` will print whatever the last expression evaluated to.

To save space I sometimes show the output on the same line as the code:

```
> my $a = False;  # -> False
```

1.3.1. Command Line Completion

You can type a partial command, and use <TAB> to expand it. If there are more than one possible commands, use <TAB> again to cycle through them.

Use the up and down arrow key to cycle trough the command history.

> If the arrow keys don't work as expected, you must install `Linenoise` (available for all platforms, and included with Rakudo Star) manually.
>
> Linenoise is *not* included in the native packages (Linux). But the good news is that you don't need «sudo» to install modules.

Exercise 1.1

Install Rakudo Star (*or* Docker and one (or both) of the Docker Images) if you haven't done so already.

Start REPL, and check the version with `$*PERL.compiler` (or `raku -v` in the shell).

Linenoise

The raku module **Linenoise** is included with Rakudo Star. It gives REPL command line history, and the possibility to edit commands, just as a normal shell.

The history is saved to a file, and loaded the next time you start REPL.

If you don't have it (as shown when using the arrow keys in REPL), install it manually.

```
$ zef install Linenoise
```

> Note that since this module has binary components, you'll need a working C compiler.
>
> Everything needed for building Linenoise comes in the «build-essential» package on Debian-based system (such as Ubuntu).

If you don't have «zef», start with installing it. See https://github.com/ugexe/zef for instructions. Or consider installing Rakudo Star, or using Docker.

rlwrap

On a Unix-like system the «rlwrap» library can also be used. It can be installed like this, on a Debian based system:

```
sudo apt install rlwrap
```

zef is not required.

1.4. One Liners

We can use the command line to give Raku a line of code to execute with the «-e» option:

```
$ raku -e 'say e';
2.718281828459045

$ raku -e 'say "hello"'
hello
```

I use `say` to display data on the screen. It adds a newline at the end.

I could have used `print`, but then I have to specify the newline manually.

See section 6.3, "Output" for the details.

On Windows you have to swap the quotes:

```
$ raku -e "say 'hello'";
hello
```

You can get away with Windows quoting on Unix like systems, sometimes…

```
$ raku -e 'say "Hello, World!"'
Hello, World!
```

But shell escape characters will we parsed by the shell:

```
$ raku -e "say 'Hello, World!'"
bash: !'': event not found
```

1.4.1. Command Line Options

The Raku interpreter supports several command line options. The previous section presented `-e`.

Run `raku --help` for a complete list.

1.5. Running Programs

In all the examples in this book I type this to run a program:

```
$ raku program
```

The first line of the code file is:

```
#! /usr/bin/env raku
```

On a Unix like system we can execute the file without specifying `raku`, as the system will read the first line and start Raku for us automatically - *as long as the file has the executable flag set*:

```
$ ./program
```

Exercise 1.2

Make a *text file* with a text editor (like `emacs`, `vi` or `Notepad`) called «hello-world» (or «hello-world.p6» on Windows) with the following content:

File: hello-world

```
#! /usr/bin/env raku

say "Hello, World!"
```

And execute it.

Note that if you run the program by double clicking on it (in Windows), it will open a terminal window for the output, print to it, and close it again on program termination. Well before anybody can read the text. So open a terminal window, and type the command there.

1.6. Error messages

Raku tries very hard to give useful error messages.

- Compile Time Errors are given with the polite «===SORRY!===».
- Run Time Errors are given without it.

The distinction may not be that important for an average user, but errors caught at compile time are preferable as no code has actually been executed yet.

1.7. use v6

If you try to run a Raku program with Perl 5, you will get an error message complaining about the syntax.

```
$ perl hello-world
String found where operator expected at content/code/hello-world line 3, near "say
"Hello, World!""
    (Do you need to predeclare say?)
```

If you add the `use v6;` line in the files, Perl 5 will give a much nicer error message.

Using this directive tells Perl 5 to require a version that it doesn't have, so it will fail - without trying to parse and execute the program.

```
$ perl -e"use v6"
Perl v6.0.0 required--this is only v5.26.2, stopped at -e line 1.
BEGIN failed--compilation aborted at -e line 1.
```

I have not done this in the programs presented in this course.

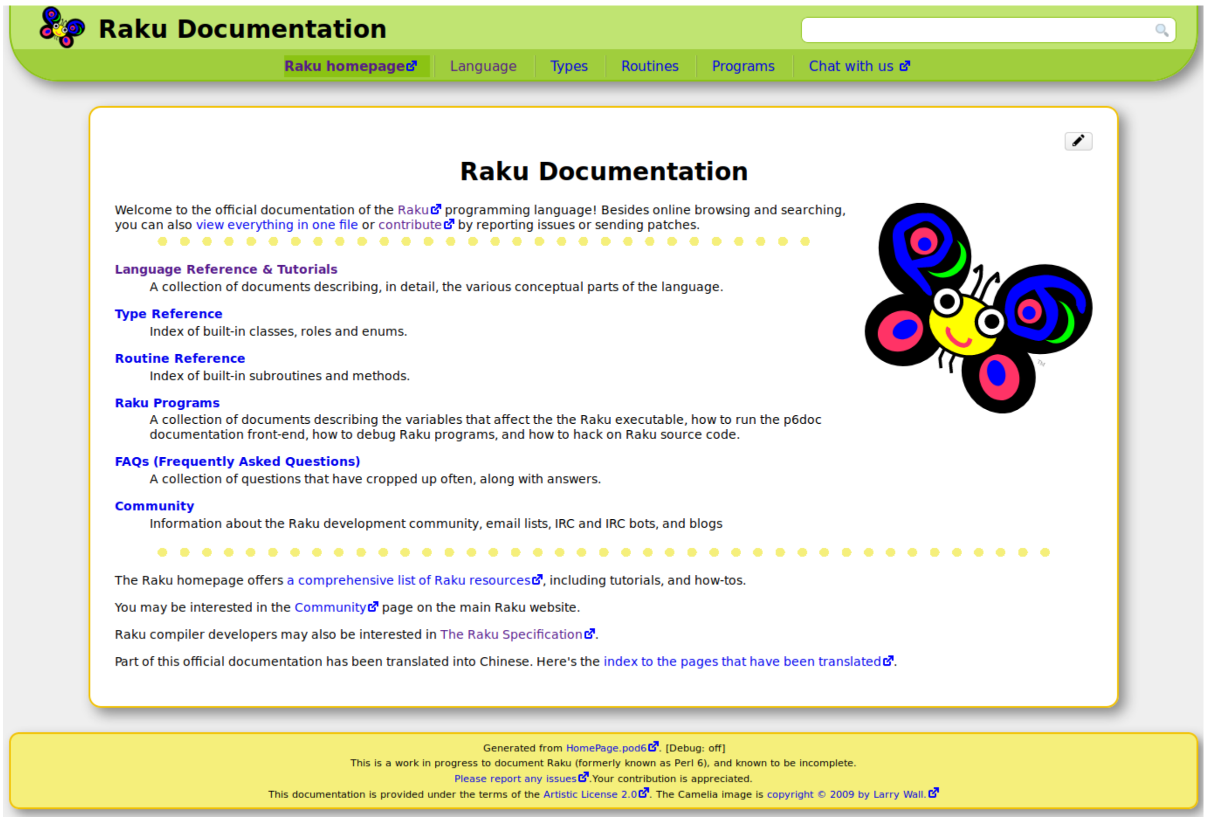
It is possible to specify a *specific* version of Raku as well. E.g. `use v6.c` as I have done in some cases in the code. I did that to pressure users of earlier (pre-release versions; 6.a and 6.b) to upgrade. The downside is that the code will still use 6.c even when newer versions are available. (6.d was released in november 2018.)

See *Appendix 4. Raku Background and History* for more information.

1.8. Documentation

The documentation for Raku is pretty good, and is extended all the time. It is available online (on the web) and offline (local).

1.8.1. Online documentation



Raku Documentation

Welcome to the official documentation of the Raku programming language! Besides online browsing and searching, you can also view everything in one file or contribute by reporting issues or sending patches.

Language Reference & Tutorials
A collection of documents describing, in detail, the various conceptual parts of the language.

Type Reference
Index of built-in classes, roles and enums.

Routine Reference
Index of built-in subroutines and methods.

Raku Programs
A collection of documents describing the variables that affect the the Raku executable, how to run the p6doc documentation front-end, how to debug Raku programs, and how to hack on Raku source code.

FAQs (Frequently Asked Questions)
A collection of questions that have cropped up often, along with answers.

Community
Information about the Raku development community, email lists, IRC and IRC bots, and blogs

The Raku homepage offers a comprehensive list of Raku resources, including tutorials, and how-tos.

You may be interested in the Community page on the main Raku website.

Raku compiler developers may also be interested in The Raku Specification.

Part of this official documentation has been translated into Chinese. Here's the index to the pages that have been translated.

Generated from HomePage.pod6. [Debug: off]
This is a work in progress to document Raku (formerly known as Perl 6), and known to be incomplete.
Please report any issues.Your contribution is appreciated.
This documentation is provided under the terms of the Artistic License 2.0. The Camelia image is copyright © 2009 by Larry Wall.

Figure 3. Raku Documentation: https://docs.raku.org

1.8.2. Local documentation

The documentation is available locally through the `p6doc` command. The «p6doc» package is bundled with Rakudo Star, but can be installed manually with «zef» otherwise.

There are some issues with `p6doc` affecting the usability, so you may find the online documentation easier to use.

Let us try to run it, without arguments:

```
$ p6doc
You want to maintain the index?
To build an index for 'p6doc -f'
          p6doc build

To list the index keys
          p6doc list

To display module name(s) containing key
          p6doc lookup

To show where the index file lives
          p6doc path-to-index

What documentation do you want to read?
Examples: p6doc Str
          p6doc Str.split
          p6doc faq
          p6doc path/to/file

Set the POD_TO_TEXT_ANSI if you want to use ANSI escape sequences to enhance text

You can list some top level documents:
          p6doc -l

You can also look up specific method/routine/sub definitions:
          p6doc -f hyper
          p6doc -f Array.push

You can bypass the pager and print straight to stdout:
          p6doc -n Str
```

Note that the local documentation is a snapshot from when it was installed, and the online documentation may have changed quite a lot since then.

It is possible to run a local copy of the documentation web site, and the easiest way is with Docker.

Run it like this, and go to http://localhost:31415 to use it:

```
docker run --rm -it -p 31415:3000 jjmerelo/perl6-doc
```

the «--rm» part tells Docker to remove the container automatically after it has been stopped. You will still have the image, but running it again can take some time as it will need to set it up again.

See https://github.com/Raku/doc for more information.

Exercise 1.3

Run the command `p6doc list` and notice the sheer size of entries (or keywords).

First a lot of lines starting with `method`, then `routine` and finally `sub`. The meaning of them:

`method`	A method. Something invoked on an object.
`routine`	Can be used both as a method or a sub(routine).
`sub`	A subroutine, function or procedure.

We'll get back to the documentation later on, when we have some knowledge to build on.

1.9. More Information

- The excellent «Weekly Rakudo News» blog gives weekly summaries of (almost) everything related to Raku: https://rakudoweekly.blog/. It has been going on since 2003.
- Ask questions on Raku issues on the #raku IRC channel on irc.freenode.net. See https://raku.org/community/irc for more information
- Books about Raku (and Perl 6): https://perl6book.com/ Note that there are older books in print about Perl 6 not mentioned on this website. Avoid books published before 2016, as they are horrible outdated.

1.10. Speed

Rakudo is generally slower than Perl 5, but much faster than just a year ago.

The developer focus has been: «Make it right, then make it fast».

Raku is fully Unicode compliant, making it slower than it could have been.

Chapter 2. Variables, Operators, Values and Procedures.

2.1. Output with say and print

Before doing anything else, we'll discuss how to display text on the screen:

The normal way (in other languages) is the `print` command. We must add a newline (`\n`) at the end if we want a newline:

```
> print "a"; print "b";
ab>
```

The `>` is the `REPL` prompt, on the same line.

In Raku we have `say` which will attach a newline at the end automatically:

```
> say "a"; say "b";
a
b
```

Use both, depending on what you want to achieve:

```
> print "a"; say "b";
ab
```

We'll cover the details about Newlines in section 6.1, "Newlines".

2.2. Variables

A variable is a named container (also called a bucket) that can hold a value, and the value can be changed at any time.

2.2.1. Sigils

The variable type is decided by the first character, called sigil, that comes before the name. The four types are:

Sigil	**Type**	**Description**
`$`	Anything	anything
`@`	Array	several values
`%`	Hash	several key-value pairs

&	Code	callable code

The sigil is part of the variable name, so «$a» can coexist with «@a», «%a» and «&a», and they are separate variables.

Anything ($)

This can hold a single value, as we'll do initially. But it can also be used to hold *anything*.

In Perl 5 this type is called *Scalar*, and you will probably also see that name used for Raku.

Arrays (@)

An array is a sorted list with one or more values.

See Chapter 8, *Arrays and Lists* for details.

Hashes (%)

A hash is an unsorted collection of pairs of keys and values. Use the key to look up a value.

See Chapter 9, *Pair and Hashes* for details, and the «Advanced Raku» course for even more details.

Code (&)

The & means that we get a reference to the code (usually a procedure name), instead of executing it right away.

The first three sigils should be familiar if you have prior experience with Perl 5, and is the limit of the built-in types for that language.

Raku has several other types, and we'll present them later. If we assign content of one of those types to a hash or array, we change the type. Assign it to a scalar to retain the type, as a scalar can contain *anything*.

The first example we'll encounter (see Chapter 8, *Arrays and Lists*) is the difference between arrays and lists.

We can actually drop the other sigils, and use a the scalar $ for everything. But it doesn't exactly help with readability, so it isn't recommended.

2.2.2. Twigils

The sigil may be followed by a twigil (something that «tweaks a sigil»). The most used are:

Twigil	Description
!	Attribute (class member)
*	Dynamic
.	Method (not really a variable)

`:`	Self-declared formal named parameter
`?`	Compile-time variable
`^`	Self-declared formal positional parameter, also called a Placeholder Variable. See 8.12.1, “Placeholder Variables” for details.

See https://docs.raku.org/language/variables#Twigils for the complete list.

We will look into them in due course.

2.2.3. my

Variables must be declared (with `my`), before they can be used. Or we will get a compile time error:

```
> $s
===SORRY!=== Error while compiling:
Variable '$s' is not declared
------> <BOL>⏏$s
```

The eject symbol (⏏) shows where the compiler thinks the problem is. The actual output depends on the capabilities of your terminal.

```
> my $r
(Any)
```

`(Any)` means that the variable has no value. (Or rather, that it can hold *any* value.) We’ll explain that in Chapter 3, *The Type System*.

`my` defines a lexically scoped variable, meaning that it is available in the current block only, from where it is defined to the first closing bracket (`}`):

File: my

```
{
  my $a = 12;
  say $a;  # -> 12
}

say $a; # -> Variable '$a' is not declared
```

We can declare several variables at the same time:

```
> my ($a, @b, %c);
```

The space between `my` and the opening paren is essential. If you forget it, it is taken as a call to a procedure «my» (that hopefully doesn’t exist) with three arguments.

2.3. Comments

Start a comment with the # sign. The rest of the line is ignored by the compiler.

```
> say "12"; # This is a comment
12
```

2.3.1. Multi-line Comments

Multi-line and embedded comments start with a hash character (#), followed by a backtick (`), and an opening bracketing character (e.g. (, { and [) or group of characters (e.g. ((or {[). It goes one until a matching closing bracketing character(s).

```
say "Hello";
#`[ This is a comment.
The compiler will ignore it.
But you, the reader, cannot ignore it
]
say "Good bye";
```

Recursive brackets (comments inside comments) are possible, but probably not very useful:

```
> say 14 #`{ a { b } c }, 12;
1412
```

The comment goes until the very end of the string.

2.3.2. Embedded Comments

We can use the Multi-line syntax for inline comment (on a single line) as well:

```
> say #`( Yeah, right. Why bother commenting the code? ) "Whatever...";
Whatever...

> say 14; #`({ hidden-comment })
14
```

2.4. Non-destructive operators

Almost every operator and function will return the new modified value, leaving the original intact/unchanged.

```
my $a = 10; my $b = 20;
my $a = $a + $b;
my $string = "abcabc" ~ "ABC";
```

We can assign the new value at the same time:

```
> $a += $b;
> $string ~= "ABC";
```

Why isn’t this the default? Well. The expression 2+2 is valid, but you cannot assign the sum (4) back to the first value (2).

This works for methods as well, see section 3.3, “Everything is an Object”.

2.5. Numerical Operators

An operator does something with one or more values or variables.

We have the usual mathematical ones, of course:

Numerical	Integer	Description
+		Addition
-		Subtraction
++		Increment the variable by 1
--		Decrement the variable by 1
*		Multiplication
/	div	Division

2.5.1. + (Addition)

Use + (the addition operator) to add two numbers (and/or variables):

```
> 2 + 2;  # -> 4
```

2.5.2. - (Subtraction)

Use - (the subtraction operator) to subtract one number (and/or variable) from another:

```
> my $a = 3; my $b = 2;
> $a - $b;  # -> 1
```

2.5.3. ++ (Increment)

Use `++` to increment a variable by 1. The value is coerced to numeric if it isn't numeric already.

It can be used as a prefix and postfix operator:

File: plusplus

```
my $i = 10; say "{ $i++ } $i";  # -> 10 11
   $i = 10; say "{ ++$i } $i";  # -> 11 11
```

Note that it is illegal to combine the prefix and postfix versions. So `--$a++` (and variants) will fail.

2.5.4. -- (Decrement)

Use `--` to decrement a variable by 1. The value is coerced to numeric if it isn't numeric already.

It can be used as a prefix and postfix operator.

File: minusminus

```
my $i = 10; say "{ $i-- } $i";  # -> 10 9
   $i = 10; say "{ --$i } $i";  # -> 9 9
```

Note that it is illegal to combine the prefix and postfix versions. So `++$a--` (and variants) will fail.

2.5.5. * (Multiplication)

Use `*` (the multiplication operator) to multiply two numbers (and/or variables):

```
> 2 * 7;  # -> 14
```

2.5.6. / (Division)

Use `/` (the division operator) to divide one number (and/or variable) with another:

```
> 8 / 4;  # -> 2
```

2.5.7. div

The Integer division Operator `div` is a variant of `/` that can be used if both values are integers:

```
> 8 div 4;  # -> 2
```

It fails if one or both values are anything else.

2.6. Operator Precedence

From highest (also called tightest) precedence to lowest:

Operators	Description
`()`	Parens
`**`	Exponentiation
`*` and `/`	Multiplication and division
`+` and `-`	Addition and subtraction

When operators with the same precedence are encountered in the code, they are executed left to right.

See https://docs.raku.org/language/operators for a complete list of operator precedence rules.

If in doubt, use parens. (Yes, I mean it.)

Exercise 2.1

What is the result of this expression?

```
> say 12 + 10 * 4;
```

2.6.1. = (Assignment)

Use `=` (the assignment operator) to assign a value to a variable.

```
> my $s = 5;  # -> 5
> $s = 10;    # -> 10
```

This is actually an operator in Raku, though not in a mathematical sense.

2.6.2. := (Binding)

Use the `:=` operator to set the variable to point to the thing on the right side.

Normal assignment:

```
my $a = 42;  # (1)
my $b = $a;  # (1)
$a += 10;    # (2)
say "$a $b";
```

This will output `52 42`, as expected.

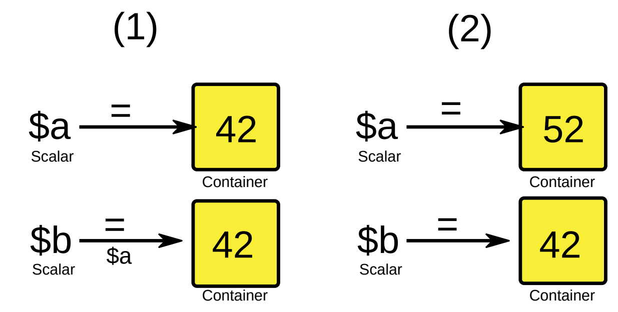


Figure 4. Normal assignment

If the thing on the right is a **variable**, we have *an alias* to the same container:

```
my $a = 42;  # (1)
my $b := $a; # (1)
$a += 10;    # (2)
say "$a $b";
```

This will output `52 52`.

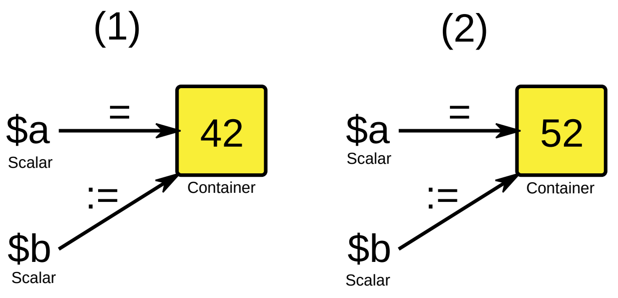


Figure 5. Binding to an existing variable

If the thing on the right is an **expression** (and not a variable), it is evaluated and the value is used:

```
my $a = 42;       # (1)
my $b := $a + 0; # (1)
$a += 10;         # (2)
say "$a $b";
```

This will output `42 52`.

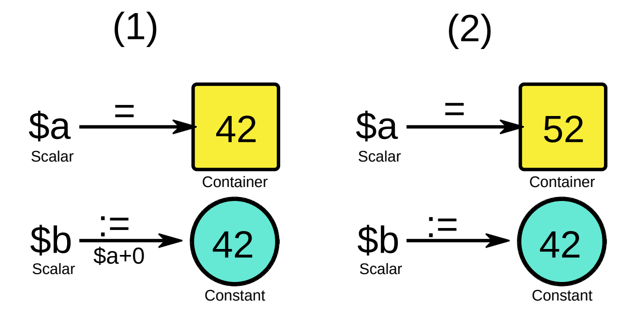


Figure 6. Binding to a value

The expression `my $b := $a + 0;` binds the variable to the value `42` (without a container). Any such (containerless) value is constant, and impossible to change:

```
> $b++; # -> Error
> 12++  # -> Just as this is an error
```

The only way to change the value is to bind it to another one:

```
$b := 4;
```

Normal assignment will try to change the *value*, in this case `4`, and that will not work:

```
>  $b = 14;
Cannot assign to an immutable value
```

Binding works with arrays and hashes as well.

2.7. Values

A value is either a string, a number, or something more complicated as an object as we'll see in Chapter 17, *Classes*.

2.7.1. Strings

String are specified in quotes; single, double or whatever else Unicode has to offer:

```
> my $name = "Arne";            # -> Arne
> my $hello = "Hello, $name";  # -> Hello, Arne
> my $hello = 'Hello, $name';  # -> Hello, $name
```

Variables are interpolated, unless single quotes are used.

The so-called «french quotes» (« and ») interpolate variables, but the string is broken down as an array with the spaces as delimiter character.

```
> «Hello, $name».raku;  # -> ("Hello", "Arne")
```

This applies to the ASCII equivalent << and >> as well.

```
> <<Hello, $name>>.raku;  # -> ("Hello,", "Arne")
```

The single brackets version (with < and >) acts as single quotes, as variables are not interpolated.

```
> <Hello, $name>.raku;    # -> ("Hello,", "\$name")
```

The raku method shows how Raku stores the value(s) internally, and can be helpful at showing what is going on. See section 6.2.3, "raku (perl)" for more information about it.

Note that arrays and hashes inside strings are *not* interpolated if used without an index. We can add an empty index to make it work:

```
> my @a = 1,2,3; say "@a";      # -> @a
> my @a = 1,2,3; say "@a[]";    # -> 1 2 3

> my %a; say "%a{}";              # -> %a
> my %a; %a<a> = 12; say "%a{}";  # -> a   12
```

Or we can place the variable in curlies to ensure interpolation:

```
> my $hello = "Hello, { $name }";
```

This is useful with expressions:

```
> say "I am almost { $age + 1 } years old.";
```

See Chapter 7, *Strings* for more information, and section 7.12, "Quoting" for other quoting constructs.

~ (String Concatenation)

Use ~ to glue two strings together:

```
> my $t = "abc" ~ "def";
abcdef
```

2.7.2. Numbers

If it is without quotes, and looks like a number, it is either a number:

```
12        # Integer
12.8      # A number
1.12e+20 # Floating point
2+4i      # A Complex number
```

Or an error:

```
12A
1.12e
2+4j
```

Complex Numbers will be covered in the «Advanced Raku» course.

2.8. Variable Names

The first character (after the sigil and optional twigil) in a variable name (and any other name; e.g. procedures, classes) must be a letter (as in whatever Unicode has decided is a letter) or underscore (_).

The rest can be letters, underscore (_), minus (-), a single quote (') and digits.

A minus (-) or single quote (') must be followed by a letter or underscore (_), and the last character must be a letter, underscore or a digit.

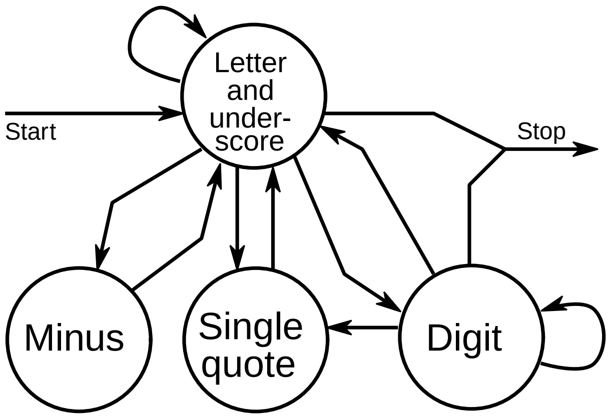


Figure 7. Variable Name Syntax Rule

Some examples:

```
my $r1234;    # OK
my $r1234-56; # ERROR - parsed as "$r1234 - 56"
my $r1234_56; # OK
my $r1234-5A; # ERROR - as "5A" is not a number
my $r1234'5A; # ERROR - as "5A" is not a number
my $Große;    # OK
my $ßßßßßß;   # OK
my $______;   # OK
my $    ;     # OK (doesn't work in print; two chinese letters)
my $_;        # ERROR, as we cannot redeclare this one.
```

We can even use characters that are difficult to print.

```
my $侮僪;        # OK - two chinese letters
```

Common sense is advisable, especially before venturing into the Unicode jungle.

Exercise 2.2

Which of the following variable names are legal?

```
my $don't-do-it;
my $dog;
my $dog2;
my $dog-3;
```

Use `REPL` to check if you are right.

2.9. constant

Do not use variables for values that should stay constant:

```
> constant $pi = 3.14;
```

You cannot change a constant value:

```
> constant $pi = 3.14;
> $pi = 3;
Cannot assign to an immutable value in block <unit> at <unknown file> line 1
```

For now, just think of «immutable» as a fancy word for «read only».

We'll come back to it in the «Advanced Raku» course.

2.10. Sigilless variables

You can drop the sigil, if that makes you feel better:

```
> constant pi = 3.14;
> say pi * 5;
```

We can use binding, but it doesn't add anything compared with assignment (except perhaps clarity):

```
> constant pi := 3.14; # Use assignment instead
```

2.10.1. pi

Raku has a built in `pi` value:

```
> say pi;  # -> 3.141592653589793
```

You can redefine it - without any warnings. But please don't.

2.10.2. Still Constant?

If you think `constant` is too much typing, use a backslash instead when you declare it:

```
> my \z = 2;     # -> 2
> say z + 100;  # -> 102
> z = 2
Cannot modify an immutable Int (1) in block <unit> at <unknown file> line 1
```

The **variable** z is actually constant, so sigiless variables are not variables at all.

2.11. True and False

The Boolean values `True` and `False` are built in:

```
> my $a = False;  # -> False
> my $b = True;   # -> True
```

In numerical context `False` has the value `0`, and `True` has the value `1`. This means that we can do the following, though unwise:

```
> True + False;  # ->  1
> True + True;   # ->  2
> True * 12;     # -> 12
```

2.11.1. so / ? / Bool

If we evaluate a non-Boolean value in Boolean context, we will get `False` if it is undefined, an empty string or the number 0. Everything else will get `True`.

The `so` keyword (or `?` prefix) forces the expression to be evaluated in Boolean context:

```
> "False".Bool;  # -> True
> so "False";    # -> True
> ? "False"      # -> True  ## The space is optional
> "False".so     # -> True
```

2.11.2. Boolean Operators

The Boolean operators come in pairs, one with high precedence and one with low.

High precedence	Low precedence	Description
`!`	`not`	Negation
`&&`	`and`	Both
`||`	`or`	One or both
`^^`	`xor`	Only one (Exclusive Or)

! / not

The `!` and `not` operators can be used to negate a Boolean value:

```
> ! True;   # -> False
> ! False;  # -> True
```

When used on a non-Boolean value, the value is converted to Boolean before the negation:

```
> ! 10;     # -> False
> ! 0;      # -> True
> ! "ABC";  # -> False
> ! "";     # -> True
```

Instead of:

```
> if ! $value == 15
```

use the negated operator != (which we'll explain in section 3.7, "Comparison Operators"):

```
> if $value != 15
```

or negate the test:

```
> unless $value == 15
```

Be careful with the precedence:

```
> not 1 - 1; # -> not (1 - 1) -> not 0
True

> ! 1 - 1; # -> (!1) - 1 ->  False - 1 -> 0 - 1
-1
```

&& / and

Returns a True-ish value if all the arguments evaluate to True, and a False-ish value otherwise.

```
> 1 and 6;          # -> 6
> True and 0;       # -> 0
> True and False;   # -> False
```

The return value is the first argument that evaluates to False or the very last one (that evaluates to True).

This operation short-circuits, so the compiler skips any expressions given after it encounter the first False value:

```
> my $a = 1; $a++ and $a++ and $a++ and $a++;   # -> 4
> my $b = 0; $b++ and $b++ and $b++ and $b++;   # -> 0
```

|| / or

Returns True if at least one of the arguments evaluates to True:

```
> True || False;   # -> True
> True || True;    # -> True
> False || False;  # -> False
```

xor / ^^

Returns True if exactly one of the arguments evaluates to True:

```
> True xor False;   # -> True
> False xor False;  # -> False
```

Nil is returned if more than one of them evaluates to True:

```
> True xor True;                       # -> Nil
> True xor True xor False;             # -> Nil
> False xor False xor False xor True;  # -> True
> False ^^ False ^^ False ^^ True;     # -> True
```

Exercise 2.3

Explain why we get different results when we use ! and not:

```
> my $value = 1;
> ! $value == 15;    # -> False
> not $value == 15;  # -> True
```

2.12. //

The problem with the or and || operators is that they don't differentiate between the value zero and an undefined value:

```
> my $age = 0; # Age in years
> say $age || "unknown";  # -> unknown
```

We can fix this by using the «Defined-or» operator // instead. It returns the first **defined** operand, or the last one if they are all undefined:

```
> my $age = 0; # Age in years
> say $age // "unknown";  # -> 0
> my $price-pound;
> my $price-dollar = 5;
> my $price-yen    = 2;

> say "The price is: { $price-pound // $price-dollar // $price-yen // "unknown" }.";
The price is: 5.
```

(We should have told which currency we gave the price in, but never mind.)

2.12.1. () (Grouping Operator)

Use parens to group expressions together. They have higher precedence than anything else:

```
> 1 + 2 * 3 + 4; # -> 1 + ( 2 * 3 ) + 4;  # -> 11
> (1 + 2) * (3 + 4) # -> 3 * 7;           # -> 21
```

Note that the Grouping Operator does not make a list. We can use a comma (called the List Operator; see section 8.1, “, (List Operator)”) to make a list.

Chapter 3. The Type System

Raku has a complex (as in complicated) type system, and we can choose to use it actively (called «strong typing») - or ignore it.

The types are still there, even if we ignore them, and can cause surprises. Using the type system is generally recommended.

Without using types:

```
> my $a = 12;
> $a = "hello, world!";
```

3.1. Strong Typing

With strong typing:

```
> my Int $a = 12;  # -> 12

> $a = "hello, world!";
Type check failed in assignment to $a; expected Int but got Str ("Hello, world!")
  in block <unit> at <unknown file> line 1
```

Strong typing can prevent programming errors.

When a variable does not contain a value, `REPL` will report the type instead. `Any` is the most general type, as it can represent anything.

We can do the same with arrays and hashes:

```
> my Int @a;
> my Str %h;
```

You can use the type system as and when you want:

```
> my Int $a = 12; # -> 12
> my $b = $a + 1; # -> 13
> $b = "Hi!";     # -> Hi!
```

3.1.1. of (as keyword)

We can use `of` as well to add a type constraint to a variable:

```
> my $i of Int = 42;
> my Int $i = 42;    # The same

> my Int @a;
> my @a of Int;

> my Int %h;
> my %h of Int;
```

3.1.2. WHAT

The `WHAT` method (and procedure) can be used to tell us the type of a value or variable:

```
> 12.WHAT;                # -> (Int)
> WHAT 12;                # -> (Int)
> "12".WHAT;              # -> (Str)
> my $i = 12; say $i.WHAT; # -> (Int)
> $i = "AB";  say $i.WHAT; # -> (Str)
> True.WHAT;              # -> (Bool)
```

3.1.3. ^name

`WHAT` and `^name` give similar results, but are implemented quite differently.

```
> my Numeric $a = 1;
1

> $a.WHAT;  # A one-element list
(Int)

> $a.^name  # A scalar value
Int
```

The `WHAT` (and `^name`) methods show implementation details, and you should not write code depending on them.

One obvious reason is classes and inheritance (which will be discussed in Chapter 17, *Classes*), that can change the class name.

Another (not so obvious) reason is optimizers, which may choose to change the types.

3.1.4. of (as method)

We can use `of` as a method to show the type of the value or variable for arrays and hashes:

```
> my Int %$hash; say %hash.of;  # -> (Int)
> my Str %$hash; say %hash.of;  # -> (Str)
> my @a;         say @a.of;     # -> (Mu)
> my Int @a;     say @a.of;     # -> (Int)
```

The of method doesn't work with scalars, where we'll have to use .VAR to get the scalar object:

```
> my Numeric $a;  # -> (Numeric)
> $a = 3;         # -> 3
> $a.WHAT;        # -> (Int)
> $a.VAR;         # -> 3
> $a.VAR.of;      # -> (Numeric)
```

3.2. ^mro (Method Resolution Order)

The ^mro («method resolution order») method gives a list of types (or classes) an object or value belongs to, in inherited (and priority) order:

```
> say 12.^mro;  # -> ((Int) (Cool) (Any) (Mu))
```

This tells us that the number 12 is of type Int, that Int inherits from Cool, that Cool inherits from Any, and finally that Any inherits from Mu.

What this means it that if we apply say on an Int, the dispatcher (or method resolver) starts with Int and checks if there is a say method there. If not, it continues along the inheritance graph until it finds it, or gives up.

This is useful from an implementation perspective. We inherit methods from base (or parent) classes, and supply custom versions only when needed.

The initial caret in ^mro is there to tell you that this method gives **implementation specific** details. The information from such calls may change in later versions of Raku, without prior warnings.

See https://docs.raku.org/type.html for more information about the types system.

3.2.1. Int Inheritance Tree

^mro can be used on type objects as well. Her we have Int:

```
> say Int.^mro;  # -> ((Int) (Cool) (Any) (Mu))
```

Type	Description
`Mu`	The root of the type system
`Any`	Thing/object
`Cool`	Object that can be treated as both a string and number (short for «Convenient OO Loop»)
`Int`	Integer

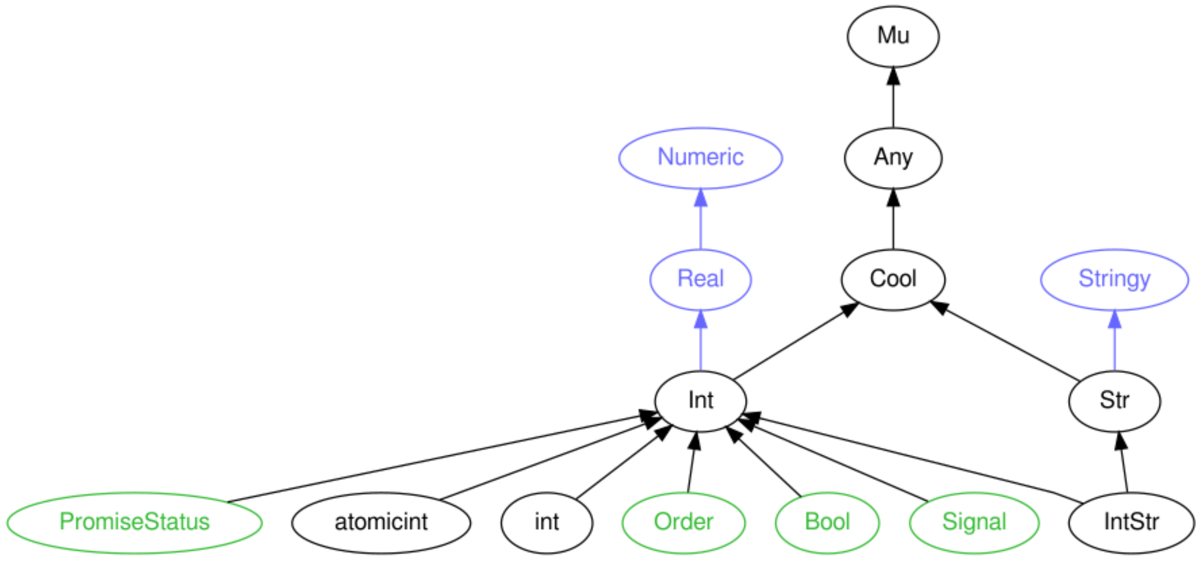


Figure 8. Inheritance Tree for `Int`

The colours used are: Black (types), Green (enums; we'll come back to it in the «Advanced Raku» course), Blue (roles, see 17.14, "Roles"). A type inherits from something that it has a pointer to.

The Graph Online: https://docs.raku.org/type/Int#Type_Graph

See section 6.6.2, "Str Inheritance Tree" for the `Str` Inheritance Tree.

3.2.2. Other Types

Note that we cannot use `^mro` on roles:

```
> Real.^mro
No such method 'mro' for invocant of type 'Perl6::Metamodel::ParametricRoleGroupHOW'
  in block <unit> at <unknown file> line 1

> Numeric.^mro
No such method 'mro' for ...
```

3.3. Everything is an Object

If you want it to be.

Most built-in functions have a corresponding method:

```
> say $a;
> $a.say;

> say "Hello";
> "Hello".say;
```

Remember the following table from chapter 1:

`method`	A method. Something invoked on an object.
`routine`	Can be used both as a method or a sub(routine).
`sub`	A subroutine, function or procedure.

If you want to know if a certain keyword can be used as a method, subroutine or both, look it up with `p6doc list`. For instance `say`:

```
$ p6doc list | grep say
method say
sub say
```

This shows that the documentation isn't 100% consistent. We should have gotten one hit, on «routine say« here.

Most methods (as well as operators, see section 2.4, "Non-destructive operators") leaves the value it is invoked on intact, and returns a modified version.

We can do the assignment back with this short form:

```
$val = $val.something; # This method doesn't exist.
$val     .= something;
```

3.4. Special Values

In this section we will look at the special values `Nil`, `Any`, `Inf` and `NaN`.

3.4.1. Nil & Any

`Nil` is the null value (the absence of a value).

Assign it to a variable to reset it to its default (undefined) value:

```
> my $b = "b"; $b = Nil;    # -> (Any)
> my Int $i = 4; $i = Nil;  # -> (Int)
```

It is possible to use the type instead of `Nil`:

```
> my Int $i = 4; $i = Int;  # -> (Int)
```

Any doesn't work if you use types (strong typing):

```
> my $a = Any;      # -> (Any)

> my Int $i = Any;
Type check failed in assignment to $i; expected Int but got Any (Any)

> my Int $i = Nil;  # -> (Int)
> my Int $i = Int;  # -> (Int)
```

Be careful when trying to output an undefined value:

```
> my $a = Any; say $a;  # -> (Any)

> my $a = Any; say ": $a";
Use of uninitialized value of type Any in string context.
```

See section 6.3.1, "say" for details.

3.4.2. Infinity

Inf is infinity. We can use the unicode infinity symbol ∞ as well.

Infinity is a value larger than any we can express, and will always be out of reach:

```
> say 1000000000000000000000000000000000000000000000000000000 > Inf
False
```

We can negate it:

```
> say -1000000000000000000000000000000000000000000000000000000 < -Inf
False
```

Do not treat Inf as a number. It is useful for comparisons, but doing arithmetic on it is mostly useless:

```
> Inf + 1;    # -> Inf
> -Inf - 1;   # -> -Inf
> -Inf + Inf; # -> NaN
> Inf * 0;    # -> NaN
```

The first one proves that `1 == 0`. Except it doesn't, as Inf isn't a number and cannot be used in expressions.

3.4.3. NaN (Not a Number)

See section 5.10, "NaN (Not a Number)".

Exercise 3.1

What is the *largest* number we can store in an Int?

Use REPL.

3.5. :D (Defined Adverb)

A typed variable accepts values of the specified type, obviously. But it will also accept the default value of Nil. That is normally not a good idea for procedure arguments (see Chapter 10, *Procedures* for details), which we haven't discussed yet.

We can remedy this with a :D (for «Defined») adverb on the type:

```
> my Int:D $a;
===SORRY!=== Error while compiling:
Variable definition of type Int:D requires an initializer at line 2
```

```
> my Int:D $i = Nil
Type check failed in assignment to $i; expected type Int:D cannot be itself...

> my Int:D $i = Int
Type check failed in assignment to $i; expected Int:D but got Int (Int) ...

> my Int:D $i = Any
Type check failed in assignment to $i; expected Int:D but got Any (Any) ...
```

Without the :D adverb, the code above would work.

3.5.1. :U (Undefined Adverb)

It is also possible to prohibit values in a variable with the `:U` (for «Undefined») adverb on the type:

```
my Int:U $i;

> $a = Nil
(Int:U)

> $a = Int
(Int)

$a = 1
Type check failed in assignment to $a; expected Int:U but got Int (1)
  in block <unit> at <unknown file> line 1
```

This is rather pointless, but at a stretch can be used on arguments to a procedure, where we use the type instead of an actual value. But I really don’t recommend it.

3.5.2. defined

Use `defined` to check if a value is defined (has a value):

```
> say Int.defined;  # -> False
> say 12.defined;   # -> True

> my $a;     say $a.defined;  # -> False
> my $a = 1; say $a.defined;  # -> True
```

Note that there also is a `DEFINITE` method that gives *almost* the same result. We’ll come back to it in the «Advanced Raku» course.

3.6. Type Coercion

Raku has automatic and manual type coercion (also called conversion).

3.6.1. Automatic Type Coercion

Raku will automatically convert the values to the required type, if possible. But only if we haven’t used strong typing:

```
> my $string1 = "12"; my $string2 = "13";
> my $sum1 = $string1 + $string2; # Addition
25

> $sum1.WHAT;
(Int)

> my $sum2 = $string1 ~ $string2; # String concatenation
1213

> $sum2.WHAT;
(Str)

> my Int $a = 12; my Int $b = 13;
> my $c = $a ~ $b;
1213

> my Int $d = $a ~ $b;
Type check failed in assignment to $d; expected Int but got Str ("1213")
  in block <unit> at <unknown file> line 1
```

3.6.2. Manual Type Coercion

Manual type coercion is useful in combination with strong typing, when we want (and have) full control of the types:

To	Method	Prefix	Keyword	Function
Numeric	`.Numeric`	`+`		`Numeric()`
String	`.Str`	`~`		`Str()`
Boolean	`.so` or `.Bool`	`?`	`so`	`Bool()`

There is also a `?^` prefix operator, that coerces to Boolean and negates the result, and the `?^` (the same), `?|` and `?&` infix operators that perform logical XOR, OR and AND operations.

The values must be convertible for this to work.

Numeric / +

`Numeric` or the `+` prefix will convert to the *best numeric type* for the given value:

```
> "12".Numeric.WHAT    # -> (Int) # Integer
> "12.1".Numeric.WHAT  # -> (Rat) # Rational number
> "5e+10".Numeric.WHAT # -> (Num) # Floating point

> +("12").WHAT   # -> (Int)
> +("12.1").WHAT # -> (Rat)

> "abc".Numeric
Cannot convert string to number: ...
```

We can specify the actual type, if we are certain of what we want:

```
> "12".Int.WHAT # ->  (Int) # Integer
> "12".Rat.WHAT # ->  (Rat) # Rational number
> "12".Num.WHAT # ->  (Num) # Floating point
> 12.1.Str.WHAT # ->  (Str) # String; "12.1"
> ~(12.1).WHAT  # ->  (Str) # String; "12.1" # Prefix ~
```

But you will get exactly what you ask for:

```
> "12.1".Int;  # -> 12
```

Str / ~

To string:

```
> 12.Str;
> ~12;
```

See section 2.11, “True and False” for the Boolean examples.

There are other ways of stringifying values. See 6.3, “Output” for details.

3.6.3. Preventing Runtime errors with `try`

Run time errors occur when Raku fails to do certain operations. E.g.:

```
> "AS".Int
Cannot convert string to number: ...
```

We can prevent the program termination by prefixing the expression with `try`:

```
> try "AS".Int
Nil
```

Now it is the program's responsibility to handle the error situation. Some possibilities:

```
> my $text = "AS";
> try $possibly-an-int.Int;
Nil

> try $possibly-an-int.Int // 0;
0
```

But beware of zero:

```
> my $zero = 0;
> try $zero.Int;
0
```

When `try` is applied to a non-Failure, it leaves the value intact. So the zero is returned, and we will get into problems if we use this in a test. E.g. like this where the `if` block is not executed, even if we have an integer.

```
my $zero = 0;
if try $zero.Int
{
  ...
}
```

We can prevent this problem with the `.defined` method:

```
> my $zero = 0;
> say "OK" if try $zero.Int.defined;  # -> OK
```

`try` and `$!` will be covered in detail in the «Advanced Raku» course.

3.7. Comparison Operators

We have the usual numeric comparison operators, and their string versions. As well as quite a few others as well:

Numeric	Strings	Smart	Other	What
`==`	`eq`			Equal
`<`	`lt`	`before`		Less than

Numeric	Strings	Smart	Other	What
`<=`	`le`			Less than or equal
`>`	`gt`	`after`		Greater than
`>=`	`ge`			Greater than or equal
`!=`	`ne`			Not equal
`<=>`	`leg`	`cmp`		Three way comparison
			`=:=`	Container equality; see section 3.7.6, “=:=”
			`===`	Value identity; see section 3.7.7, “===”
`=~=`				Approximately equal; see section 5.11, “=~=”

Note that string comparison is case sensitive. It doesn’t recognize letters as such, but compares the Unicode value. So uppercase letters come before the lowercase versions.

Most of them are straight forward, but we’ll discuss the last row (the «Three way comparison» operators) and the «Smart» column in the following sections.

3.7.1. cmp

`cmp` (as in «Compare») is the versatile three-way comparison operator.

It compares strings with string semantics, numbers with number semantics, Pair objects first by key and then by value etc.

```
> say (a => 3) cmp (a => 4);   # -> Less
> say 4 cmp 4.0;               # -> Same
> say 'b' cmp 'a';             # -> More
```

In numeric context, the return values are `-1` (Less), `0` (Same) and `1` (More).

unicmp

There is a `unicmp` version of `cmp` that disregards the case of the characters:

```
> "a" unicmp "B";  # -> Less
> "A" unicmp "b";  # -> Less
```

It doesn’t quite work as expected when we compare lower and upper case versions of the same letter. The lower case version is considered Less than the upper case version:

```
> "A" unicmp "a";  # -> More
> "A" unicmp "A";  # -> Same
> "a" unicmp "A";  # -> Less
```

But this means that we can actually use it for sorting, and get predictable results.

There is a collation aware version of `unicmp` called `coll`. It will be covered in the «Advanced Raku» course.

3.7.2. leg

`leg` (as in «Less, Equal or Greater») is the string only version of `cmp` (see section 3.7.1, “cmp”).

Non-string values are converted to strings before the comparison.

```
> say 'a' leg 'b';  # -> Less
> say 'a' leg 'a';  # -> Same
> say 'b' leg 'a';  # -> More
```

In numeric context, the return values are `-1` (Less), `0` (Same) and `1` (More).

Comparing `cmp` and `leg`:

```
> 11 leg 2;  # -> Less
> 11 cmp 2;  # -> More
```

3.7.3. <=>

`<=>` is the numeric version of `cmp` (see section 3.7.1, “cmp”).

```
> say 1 <=> 2;    # -> Less
> say 2 <=> 2.0;  # -> Same
> say 2 <=> 1;    # -> More
```

Non-numeric values are converted to numbers before the comparison, and you’ll get an error if it wasn’t possible to do the conversion:

```
> 2 <=> "sj"
Cannot convert string to number: ...
```

3.7.4. before

The `before` operator behaves as `cmp` (see section 3.7.1, “cmp”), except that it returns `True` if the first argument comes before (is less than) the second.

```
> 1 before 1;         # -> False
> 1 before 2;         # -> True
> 111 before 21;      # -> False
> "111" before "21"; # -> True
```

3.7.5. after

The `after` operator behaves as `cmp` (see section 3.7.1, “cmp”), except that it returns `True` if the first argument comes after (is greater than) the second.

```
> "ab" after "aaaa";  # -> True
```

Exercise 3.2

Explain why we get `False` from the first, and `True` from the second:

```
> 111 before 21;      # -> False
> "111" before "21";  # -> True
```

3.7.6. =:=

Use the Container Identity Operator `=:=` to find out if both arguments are bound to the same container. If it returns True, it generally means that modifying one will also modify the other.

```
> my $a = 42;
> my $b = $a;
> $a =:= $b;   # -> False
```

Here we apply `=:=` to the two first code blocks from section 2.6.2, “:= (Binding)” to show that it recognizes binding:

```
> my $a = 42;
> my $b := $a;
> $a =:= $b;  # -> True
```

3.7.7. ===

Use the Value Identity Operator `===` to check if both arguments are the same object or value, disregarding any containerization:

```
> my $a = 3;   # -> 3
> my $b := 3;  # -> 3
> $a === $b;   # -> True
> 1 === 1.0;   # -> False
```

When used on values (as done here), `===` behaves the same as `eqv`.

See section 17.9.2, “===” for a description on using === on objects.

We can use a type object to check the type of the variable or value:

```
> my $a = 12; $a.WHAT === Int; # -> True
```

3.7.8. isa

The isa method looks better (avoiding WHAT):

```
> my $a = 12; $a.isa(Int); # -> True
```

We can also use smartmatch to check the type. See section 9.16.2, “With Smartmatch”.

3.8. but (True and False, but …)

We can use the but keyword to change how non-Boolean values are converted to Boolean:

```
> my $a = "Hi" but False;  # -> Hi
> say $a;                  # -> Hi
> say so $a;               # -> False
```

You can read it like «yes, but…»

The but clause is part of the *current value* (and not the variable), and will be lost if we change the value:

```
> my $c = 156 but False;  # -> 156
> say so $c;              # -> False
> $c++;                   # -> 157
> say so $c;              # -> True
```

It is possible (but unwise) to use it with non-Boolean values after the but keyword. This should be all possible combinations (of the basic types string/number/Boolean):

my $x =	$x.Str	$x.Int	$x.Bool	+$x	$x + 0
10 but 0	10	0	True	10	10
10 but 'ten'	ten	ten	True	ten	10
10 but False	10	10	False	10	10
10 but True	10	10	True	10	10
0 but 10	0	10	False	0	0
0 but 'ten'	ten	ten	False	ten	0

my $x =	$x.Str	$x.Int	$x.Bool	+$x	$x + 0
0 but False	0	0	False	0	0
0 but True	0	0	True	0	0
'ten' but 0	ten	0	True	Error	Error
'ten' but 10	ten	10	True	Error	Error
'ten' but False	ten	Error	False	Error	Error
'ten' but True	ten	Error	True	Error	Error
True but 0	True	0	True	1	1
True but 10	True	10	True	1	1
True but 'ten'	ten	1	True	1	1
True but False	False	0	False	0	1
False but 0	False	0	False	0	0
False but 10	False	10	False	0	0
False but 'ten'	ten	0	False	0	0
False but True	True	1	True	1	0

Error means a runtime error (and program termination).

⚠ Some of the values in the table doesn't make much sense. But the problem is the input. As long as we use it *as intended* with a Boolean value after the `but` it works out as expected.

The program used to make this table, using `try` to prevent program termination on error:

File: but

```
say "\$x =            |\$x.Str |\$x.Int |\$x.Bool|+\$x    |\$x+0";
say "----------------+-------+-------+-------+-------+----";

print-it("10 but 0",        10 but 0);
print-it("10 but 'ten'",    10 but 'ten');
print-it("10 but False",    10 but False);
print-it("10 but True",     10 but True);

print-it("0 but 10",        0 but 10);
print-it("0 but 'ten'",     0 but 'ten');
print-it("0 but False",     0 but False);
print-it("0 but True",      0 but True);

print-it("'ten' but 0",     'ten' but 0);
print-it("'ten' but 10",    'ten' but 10);
print-it("'ten' but False", 'ten' but False);
print-it("'ten' but True",  'ten' but True);

print-it("True but 0",      True but 0);
print-it("True but 10",     True but 10);
print-it("True but 'ten'",  True but 'ten');
```

```
print-it("True but False",  True but False);

print-it("False but 0",     False but 0);
print-it("False but 10",    False but 10);
print-it("False but 'ten'", False but 'ten');
print-it("False but True",  False but True);

sub print-it ($label, $expression)
{
  print $label, "\t|";
  print trap-it($expression.Str), "\t|";
  print trap-it($expression.Int), "\t|";
  print trap-it($expression.Bool), "\t|";
  print trap-it(+$expression), "\t|";
  print trap-zero($expression);
  say "";
}

sub trap-it ($expression)
{
  my $result;
  try { $result = $expression.gist; }

  return $!
    ?? "ERR"
    !! $result;
}

sub trap-zero ($expression)
{
  my $result;
  try { $result = ($expression + 0).gist; }

  return $!
    ?? "ERR"
    !! $result;
}
```

The Error Variable `$!` contains the error object (if the code we wrapped in `try` failed). In Boolean context it tells us if we have an error. In string context it gives the error message.

3.8.1. does

The `does` keyword is similar to `but`. The difference is that `does` adds it to the given variable, whereas `but` applies it to a *copy* of it.

So normal assignment works the same:

```
> my $a = "Hi" but  False; say $a.WHAT;  # -> (Str+{<anon|6>})
> my $b = "Hi" does False; say $b.WHAT;  # -> (Str+{<anon|7>})
```

```
> my $a = "Hi"; $a but  False; say $a.WHAT;  # -> (Str)
> my $b = "Hi"; $b does False; say $b.WHAT;  # -> (Str+{<anon|9>})
```

We apply the `but` to a *copy* of `$a`, and this value is discarded as we don't assign it to a variable.

Chapter 4. Control Flow

In this chapter we'll discuss statements used to change the flow of execution from the normal top-down.

4.1. Blocks

A block is a collection of code that is treated as a whole. Blocks are set up inside a pair of curly braces:

```
{
  # This is a block
}
```

4.2. Ranges (A Short Introduction)

A range in Perl 6 is a collection of consecutive incremented integers. A range `1 .. 10` contains all the integers from 1 to 10.

The `..` operator give a range (and not a list):

```
> (1 .. 5).WHAT
(Range)

> say (1 .. 5)
1..5
```

Ranges are lazy, so the individual values will not be calculated until actually needed.

From 10 (but not including it, so from 11) to 1 million:

```
> (10 ^.. 1_000_000)
```

We can exclude the end value as well:

```
> (10 ..^ 1_000_000)
```

Both start and end values excluded:

```
> (10 ^..^ 1_000_000)
```

(Read the `^` character as «up to/from, but not including».)

⚠ The ^ is a part of the range operator, and not the values!

That is why there cannot be any spaces between an ^ and the range operator.

We can use this short form if we want e.g. 10 values:

```
> my @values = ^10;
[0 1 2 3 4 5 6 7 8 9]
```

It starts with zero and counts up.

4.3. loop

The `loop` statement is the classic «for» loop known from other languages. It takes three statements separated by semicolons:

```
> loop (my $i = 0; # The initial value
>       $i < 10;   # The test to decide if the loop should be stopped
>       $i++)      # The incrementer
> { print $i; }
> print "\n";
0123456789
```

Note that `my` can be used to introduce a new variable as the loop counter, as done here, or we can use a variable defined outside the loop. But the side effect is that we change its value.

It is possible to skip the last part (the incrementer):

```
> loop (my $i = 0; $i++ < 10;) { say $i; }
1
2
3
...
```

But this may cause subtle changes (different start value), as shown above.

The statements must be specified in parens.

We can iterate over an array like this:

File: loop-array

```
my @a = <A B C D E F G H I J K L>;

loop (my $i = 0; $i < @a.elems; $i++)
{
  print "|", @a[$i];
}

print "\n";
```

Running it:

```
$ raku loop-array
|A|B|C|D|E|F|G|H|I|J|K|L
```

The first line is an array consisting of single characters; as described in section 2.7.1, “Strings”.

We could have written it like this:

```
my @a = "A" .. "L";
```

A lot of code to iterate over an array, and it is easy to get the indices wrong (usually by 1). But we have a much more efficient way of doing this, as we’ll show in the next section.

4.4. for

A `for` loop is the most used loop type, and it can be used on almost anything.

The «loop-array» program from the previous section can be written much more compact like this:

File: for-array

```
my @a = <A B C D E F G H I J K L>;

for @a -> $elem
{
  print "|", $elem;
}

print "\n";
```

The `->` syntax introduces a local variable (an implicit `my` variable) in the following block, containing each value in the array one after each other.

Running it gives the same result:

```
$ raku for-array
|A|B|C|D|E|F|G|H|I|J|K|L
```

Note the difference; here we iterate over the actual values, and not the indices.

Also note that we get a read only version of the value, so trying to change it will fail:

File: for-array-error (partial)

```
  print "|", $elem; $elem ~= ".";
```

```
Cannot assign to a readonly variable or a value
  in block <unit> at ./for-array-error line 7
```

Exercise 4.1

What is the output from this program?

File: for-array2

```
my @a = <A B C D E F G H I J K L>;

my $elem = 99;

for @a -> $elem
{
  ; # Do nothing
}

say $elem;
```

4.4.1. for as a counter

We can execute a loop a specific number of times as well. The range short form described in section 4.2, “Ranges (A Short Introduction)” is perfect for iterations:

File: for-school

```
for ^5
{
  say "I like school.";
}
```

The output:

```
I like school.
I like school.
I like school.
I like school.
I like school.
```

The indices are 0 to 4 (and not 1 to 5), but as we do not use them in the expression that doesn't really matter.

Exercise 4.2

What is the output from this program:

```
for 5
{
  say "I like school.";
}
```

4.4.2. $_ (Topic Variable)

The «topic variable» `$_` is the default parameter for blocks that don't have an explicit signature:

```
for <a b c> { say $_ }  # sets $_ to 'a', 'b' and 'c' in turn
say $_ for <a b c>;     # same, even though it's not a block
```

It is usually better to use an explicit variable (with the `->` syntax described in section 4.4, "for"), as long as the variable has a good name.

Calling a method on `$_` can be shortened by leaving off the variable name:

```
.say; # same as $_.say
```

We cannot use an explicit block variable, e.g. `-> $val`, outside the block:

```
> $val.say for (1 .. 10) -> $val
===SORRY!=== Error while compiling: Variable '$val' is not declared.
Did you mean '&val'?
------> <BOL>  $val.say for ("aa" .. "bb") -> $val
```

The problem is that `-> $val` is available in the following block only. But we use it long before the block, and the block is missing.

This works:

```
for (1 .. 10) -> $val
{
  $val.say;
}
```

4.4.3. Postfix for

A postfix version of `for` is available, if we only have one expression:

```
say "Country: $_" for @countries;
```

Note that «one expression» can also mean a block:

```
> { .say; .say } for 1 .. 10;
```

Or we can group expressions with parens:

```
> ( .say; .say ) for 1 .. 10;
```

But I don’t advise using it like that.

The last statement in a block doesn’t *need* a trailing semicolon, but it doesn’t *hurt* to add one.

We can iterate over a range:

```
for 1 .. 10 -> $i
{
  say $i;
}
```

If you already have an `$i` variable, it will be hidden inside the `for` body, but will be available again afterwards, with the original value.

The range short form `^10` is handy for a 10 iterations loop. It is the same as `0 ..^ 10` and `0 .. 9`:

```
do-something($_) for 1 .. 10; # 1 .. 10 # 10 iterations
do-something($_) for ^10;     # 0 .. 9  # 10 iterations
```

As long as you **do not** depend on the values being 1 to 10.

4.5. Infinite Loops

These are all the same:

```
.say for 1 .. Inf
.say for 1 .. ∞
.say for 1 .. *
.say for ^Inf
```

Note that we have to use `^Inf` to get a Range. A single `Inf` is just a single value, and the loop would only run once.

Also note that the last one starts with zero.

Or we could use `loop` without arguments (and we can skip the parens), if we do not need a counter:

```
loop { say 'forever' }
```

An infinite loop should have an exit strategy. We will discuss `last` in section 4.17.3, “last”.

4.6. while

The `while` statement executes the block as long as the given condition is true.

File: while

```
my $x = 1;

while $x < 4
{
  print $x++;
}

print "\n";
```

```
$ raku while
123
```

`while` can also be used as a statement modifier:

File: while2

```
my $x = 1;

print $x++ while $x < 4;

print "\n";
```

```
$ raku while2
123
```

The output is the same because the condition is tested *before* the `print` statement is executed.

4.7. until

The `until` statement is a `while` statement with negated test. It will execute the block as long as the expression is false:

File: until

```
my $x = 1;

until $x > 3
{
  print $x++;
}

print "\n";
```

```
$ raku until
123
```

`until` can also be used as a statement modifier:

```
my $i = 0;
say "Hello" until $i++ > 10;
```

4.8. repeat while

The `repeat { … } while` statement has the test at the end. The result is that the block is executed at least once. And as long as the condition holds, another repetition occurs.

File: repeat-while

```
my $x = 5;

repeat
{
  print $x++;
}
while $x < 1;

print "\n";
```

```
$ raku repeat-while
5
```

It is possible to place the `repeat` and `while` statements together at the beginning:

File: repeat-while2

```
my $x = 5;

repeat while $x < 1
{
  print $x++;
}


print "\n";
```

The condition is still evaluated at the end of the loop, even if placed up front.

4.9. repeat until

The `repeat { … } until` statement is a `repeat { … } while` statement with negated test. It will execute the block at least once, and then for as long as the expression is `False`:

File: repeat-until

```
my $x = 5;

repeat
{
  print $x++;
}
until $x > 1;

print "\n";
```

```
$ raku repeat-until
5
```

It is possible to place the `repeat` and `until` statements together at the beginning:

File: repeat-until2

```
my $x = 5;

repeat until $x > 1
{
  print $x++;
}

print "\n";
```

The condition is still evaluated at the end of the loop, even if placed up front.

4.10. Loop Summary

Construct	**See Section**	**Always Executed Once**
`loop`	4.3, “loop”	No
`for …`	4.4, “for”	No
`… for`	4.4.3, “Postfix for”	No
`while …`	4.6, “while”	No
`… while`	4.6, “while”	No
`until …`	4.7, “until”	No
`… until`	4.7, “until”	No
`repeat … while`	4.8, “repeat while”	Yes
`repeat while …`	4.8, “repeat while”	Yes

Construct	See Section	Always Executed Once
`repeat … until`	4.9, “repeat until”	Yes
`repeat until …`	4.9, “repeat until”	Yes

4.11. if

We can conditionally execute a block once with the `if` statement:

```
if $hour == 17
{
  say "Time to go home";
}
```

4.11.1. elsif

We can have several conditions:

```
if $hour == 17
{
  say "Time to go home";
}
elsif $hour == 8
{
  say "Time to go to work";
}
```

4.11.2. else

We can add an `else` block, and it is executed if none of the `if` and `elsif`(s) matched:

```
if $time == 17
{
  say "Time to go home";
}
elsif $time == 8
{
  say "Time to go to work";
}
else
{
  say "Stay put!";
}
```

4.11.3. unless

Use `unless` to negate the if-test;

```
say "Not OK" if not $a;
say "Not OK" unless $a; # The same
```

Note that `unless` doesn't support `else` and `elsif`, as the resulting code would be hard to understand. (No, that isn't false at all.)

4.12. given

We can use `given` to set `$_` for a block:

```
> $_ = 12;
> .say;           # -> 12
> .say given 13;  # -> 13
> .say;           # -> 12

> given 'a' { say $_ }; # sets $_ to 'a'
> say $_ given 'a';     # same, even though it's not a block
```

The old value (if any) is restored after the block or (in this case) prefix statement has been executed.

`given` can also be used to form a switch-like statement (in combination with `when`); it will be covered in the «Advanced Raku» course.

4.13. with

The `with` statement is like `if` but tests for definedness rather than truth. In addition, it topicalizes on the condition (sets `$_` to the value):

```
> say "OK" if 0;         # -> ()
> say "OK" with 0;       # -> OK
> with 12 { .WHAT.say }  # -> (Int)
```

Trying to output an undefined value (`Any`, `Nil` and `Int`) fails in a string:

```
> my $a = Any;
> say "Not OK: $a;
Use of uninitialized value of type Any in string context.
Methods .^name, .perl, .gist, or .say can be used to stringify it to something
meaningful.
```

So we can use with to detect them:

File: with

```
for (0, "2", Any, Nil, Int, pi, "hello") -> $input
{
  with $input { say "OK: $_"; }
  else
  {
    say "Not OK: undefined value";
  }
}
```

```
$ raku with
OK: 0
OK: 2
Not OK: undefined value
Not OK: undefined value
Not OK: undefined value
OK: 3.141592653589793
OK: hello
```

It error message mentioned say as a method. Let us try:

```
my $a; $a.say;  # -> (Any)
```

See section 6.3, “Output” for a discussion of *why* say works on an undefined value, but not when it is interpolated in a string.

So we rewrite the code accordingly:

File: with

```
for (0, "2", Any, Nil, Int, pi, "hello") -> $input
{
  with $input { say "OK: $_"; }
  else        { print "Not OK: "; .say; }
}
```

I have changed the say in the else to print to avoid an extra newline.

```
$ raku with2
OK: 0
OK: 2
Not OK: (Any)
Not OK: Nil
Not OK: (Int)
OK: 3.141592653589793
OK: hello
```

4.13.1. given vs with

`given` (see section 4.12, “given”) and `with` are somewhat similar when used as statement modifiers:

Expression	`given`	`with`
`.say XXX 12`	`12`	`12`
`.say XXX 0`	`0`	`0`
`.say XXX Nil`	`Nil`	`()`
`.say XXX Any`	`Any`	`()`
`.say XXX NaN`	`NaN`	`()`

They differ only when we give an undefined value: `given` returns the undefined value, but `with` returns an empty list.

Do not use `with` as a statement modifier.

4.13.2. orwith

`with` has `orwith` as `if` has `elsif`:

File: orwith

```
for ( 0, "2", 3/11, pi, "hello") -> $input
{
  with   $input.Numeric { say "Number: $input"; }
  orwith $input.Str     { say "Str: $input";  }
  else                  { say "??: <unknown type>"; }
}
```

```
$ raku orwith
Number: 0
Number: 2
Number: 0.272727
Number: 3.141592653589793
Str: hello
```

We write a number, if possible to convert the value to number. If not, we try to convert it to a string. That works for all defined values, so the `else` part is never used.

(If we had passed in e.g. `Any`, that would have been a suitable value for the `else` part, the `with` condition would cause a run time error.)

We can intermix if-based and with-based clauses.

```
# This says "Yes"
if 0 { say "No" } orwith Nil { say "No" } orwith 0 { say "Yes" };
```

4.13.3. without

`without` is related to `with` in the same way that `unless` is related to `if`: negated test.

File: without

```
for (1, "2", Any, Nil, Int, pi, "hello") -> $input
{
  without $input { print "Not OK: "; .say: }
}
```

```
$ raku without
Not OK: (Any)
Not OK: Nil
Not OK: (Int)
```

There is no `else` clause, for the same reason that `unless` doesn't have it.

We can also use `with` and `without` as statement modifiers:

```
> my $variable = 12; say "$_ is of type { .^name } with $variable;
12 is of type Int

> my $answer; say "undefined answer" without $answer;
undefined answer
```

4.14. ?? !!

This is a compact if-then-else:

```
my $x = $y == True ?? 5 !! 4;
```

If the expression to the left of `??` evaluates to `True` the first argument (after the `??`) is used, otherwise the second argument (after the `!!`) is used.

Or, in the usual verbose way:

```
my $x; if $y == True { $x = 5; } else { $x = 4; }
```

4.15. do

`do` is a block construct, returning the last statement inside the block.

We can shorten the previous code line with `do` like this:

```
my $x = do { if $y == True { 5; } else { 4; } }
```

And even more compact:

```
my $x = do { $y == True ?? 5 !! 4; }
```

4.16. when

A `when` block looks similar to an `if` block, but the behaviour is subtly different.

We start with a regular `if` block:

File: if-when (partial)

```
for True, False
{
  if $_  { say "if $_"; } ① ③
  say "if $_ 2"; ② ④
}
```

① True → Executed

② True → Always executed

③ False → Not executed

④ False → Always executed

```
$ raku if-when
if True
if True 2
if False 2
```

A `when` block behaves differently, as the code immediately after it in the same block level is regarded as en implicit `else` block:

File: if-when (partial)

```
for True, False
{
  when $_ { say "when $_"; } ① ③
  say "when $_ 2"; ② ④
}
```

① True → Executed

② True → Not executed because «1» was

③ False → Not executed

④ False → Executed because «3» was not

I had to use `so` to avoid a warning.

```
$ raku if-when
when True
when False 2
```

4.17. Loop Manipulation

We can break out early, skip an iteration, or do it again.

4.17.1. once

A block prefixed with `once` will be executed one time only, even if it is placed inside a loop or a recursive routine:

File: for-once

```
for ^5
{
  once { say "once"; }
  say "many ($_)";
}
```

```
$ raku for-once
once
many (0)
many (1)
many (2)
many (3)
many (4)
```

We can achieve the same thing with the `FIRST` phaser (We'll come back to it in the «Advanced Raku» course.)

4.17.2. next

The `next` command starts the next iteration of the loop:

File: for-next

```
for ^5
{
  next if $_ == 2;

  say "many ($_)";
}
```

```
$ raku for-next
many (0)
many (1)
many (3)
many (4)
```

4.17.3. last

The `last` command immediately exits the loop:

File: for-last

```
for ^Inf
{
  last if $_ == 5;

  say "many ($_)";
}
```

```
$ raku for-last
many (0)
many (1)
many (2)
many (3)
many (4)
```

Be careful with the expression given to `last`. If the value never reaches `5` we have an infinite loop. (As in `last if $_ == 4.5`).

It is also possible to use `die` or `exit` (which will be described in the «Advanced Raku» course) to exit a loop, but they will terminate the program as well.

Exercise 4.3

Write a program that calculates the sum of all the integers from 1 and upwards until the sum reaches a specified upper limit (e.g. 1000), and displays the last integer added to reach (or pass) the limit.

Tip: Use an infinite loop, and use `last` to exit it.

For 1000 the output can be: «Limit 1000 reached (1035) at value 45.»

4.17.4. redo

The `redo` command restarts the loop block without evaluating the conditional again.

File: loop-redo

```
my $sum;

for 1 .. 1000 -> $i
{
  $sum += $i;

  redo if $sum.is-prime;
}

say "Sum: $sum";
```

```
$ raku loop-redo
Sum: 546089
```

What happens here is that we add the numbers from 1 to 1000, and the number a second time (or more) if the sum is a prime number (see section 5.12, “is-prime (Prime Numbers)”).

Without the `redo`, we would get «500500». (This is easily calculated with `sum` (see section 9.16.1, “The «sum» Method”): `sum(1 .. 1000)`.)

4.17.5. LABEL

All the loop constructs (`while`, `until`, `loop` and `for`) can have a label, used to identify them for `next`, `last`, and `redo`. This is useful if we have nested loops, as the statements otherwise would apply to the same scope only.

File: loop-labels

```
FIRST-ONE:
for 1 .. 20 -> $a
{
  NEXT-ONE: for 0 .. 2 -> $b
  {
    next FIRST-ONE if ($a + $b).is-prime;
    next NEXT-ONE  if ($a + $b) % 3;
    say "$a -> $b";
  }
}
```

```
$ raku loop-labels
6 -> 0
8 -> 1
9 -> 0
12 -> 0
14 -> 1
15 -> 0
18 -> 0
20 -> 1
```

Exercise 4.4

What happens of we add a colon after `once` in the «for-once» program in section 4.17.1, “once”?

Chapter 5. Numbers

A number is a numeric value, as e.g. «2», «0» and «3.14».

5.1. Octal, Hex, Binary …

Numbers cannot start with zero, except when specifying the number system (or base):

Number System	Short Form	General Syntax
Decimal	`123`	`:10<123>`
Octal	`0o123`	`:8<123>`
Hexadecimal	`0x12A39F`	`:16<12A39F>`
Binary	`0b10101010`	`:2<10101010>`

```
> say :2<0b10101010>;  # -> 170
```

Note that lower- and uppercase letters are equal in numbers:

```
> :16<12A39F> == :16<12a39f>;  # -> True
```

We can use a procedure-ish syntax as well:

```
> say :2("0b10101010");  # -> 170
```

Placing the value in a variable works, but only with paren syntax:

```
> my $a ="0b10101010";
> say :1<$a>;  # -> Compile time error
> say :1($a);  # -> 170
```

Exercise 5.1

How many number systems (different bases or radixes) are supported by Raku?

Use `REPL`.

5.1.1. base

We can use the `base` method to display a number in other number systems (different bases):

```
> say 1200.base(8);  # -> 2260
> say 170.base(2);   # -> 10101010
> say 256.base(16);  # -> FF
```

Colon Syntax

Arguments to a **function** can be specified with or without parens:

```
> say("12", "34");  # -> 1234
> say "12", "34";   # -> 1234
```

Arguments to a **method** are usually specified with parens. But we can use the special Colon Syntax if we want to omit them:

```
> say 1200.base(8);  # -> 2260
> say 1200.base: 8;  # -> 2260
```

5.1.2. parse-base

`parse-base` is the opposite of `base`:

```
> say "FF".parse-base(16);  # -> 255
```

It round trips:

```
> say "FF".parse-base(16).base(16);  # -> FF
```

5.1.3. Other conversions

We can also use *format strings* to convert a number to other bases, with `printf`, `sprintf` and `fmt`:

Directive	Description	Example
`%b`	Unsigned Integer to Binary	`255.fmt('%b'); # → 11111111`
`%x`	Unsigned Integer to Hexadecimal	`255.fmt('%x'); # → ff`
`%X`	As `%x`, but Uppercase	`255.fmt('%X'); # → FF`

See sections 6.5, “printf” (and 6.5.3, “sprintf” and 6.5.4, “fmt”) for details.

5.2. Unicode Numbers

Unicode has a lot of characters that are regarded as numeric. Use them if you want to cause confusion:

```
½    # This is a single character.
Ⅷ  # A single character (codepoint U+2168).
```

Well. You may not feel that confused, if your terminal or printer support the characters, but what about:

```
> say ৯০৭ + ১; # 907 + 1 (in case you wondered)
908
```

5.3. Not a Number

If it starts with a letter (or an underscore) it is either a procedure call, a predefined value, or an error:

```
> say    # -> error: Missing parameter
> True   # -> True
> False  # -> False
> abcdic # -> error: Undeclared routine
```

`True` and `False` are built-in. See section 2.11, "True and False".

5.4. N_U_M_B_E_R_S

You can add underscores in numbers to make the code more readable. The compiler ignores them. You must have a digit on both sides of an underscore.

```
> my $number1 = 1000000000;     # -> 1000000000
> my $number2 = 1_000_000_000;  # -> 1000000000
> my $number3 = 1_0_0_0 ;       # -> 1000
```

The last one is legal, but stupid.

5.5. Floating Point Numbers

Raku has several numeric types (in addition to integers, which we have discussed already).

5.5.1. pi

`pi` is built in:

```
> say pi;       # -> 3.14159265358979
> say pi.WHAT;  # -> (Num)
```

The term «floating point» is derived from the fact that there is no fixed number of digits before and after the decimal point; that is, the decimal point can float. Raku calls them `Num`.

Exercise 5.2

Display `pi` as a binary number.

```
> pi.say;  # -> 3.141592653589793
```

Use `REPL`.

5.5.2. e

Euler's number `e` (or the unicode version `e`) is also available:

```
> say e;  # -> 2.718281828459045
```

5.5.3. tau

As is `tau` (or the Unicode version `τ`):

```
> say tau;       # -> 6.283185307179586
> say tau / pi;  # -> 2
```

`tau` is the ratio between circumference and radius of a circle.

5.5.4. Floating Point Errors

```
> my $one-third = 1/3;  # -> 0.333333
```

This is as expected (with the actual number of 3's shown as the only surprise).

Adding three of them should give us 0.999999:

```
> say $one-third * 3;  # -> 1
```

But it does not. We do get 1.

5.6. Rational Numbers

Raku has a built in `Rat` (Rational Number) type.

```
> my $one-third = 1/3;  # -> 0.333333
> $one-third.WHAT;      # -> (Rat)
> 0.3.WHAT;             # -> (Rat)
```

Yes, the last one is valid syntax!

The `Rat` type is automatically used for values with a decimal fraction, if possible. Otherwise the floating point type Num is used.

The `Rat` type uses two integers internally; the actual value is the first (the «Numerator») divided by the second (the «Denominator»).

We can use the `nude` («Numerator» + «Denominator»; «Nu» + «De») method to get the values:

```
> (1/3).nude;  # -> (1 3)
> (0.1).nude;  # -> (1 10)
> 0.2.nude;    # -> (1 5)
```

> So where does 0.333333 come from?
>
> The conversion happens when we print the value, as that converts it to a string. This happens as a zero with a point and an infinite number of 3's after it isn't something we can (or should) print.
>
> See section 6.2, "Stringification" for details.

Rational Numers are reduced as much as possible internally:

```
> say (3/9).nude;   # -> (1 3)
> say (8/16).nude;  # -> (1 2)
```

We can add Rational Numbers, as Ruko supports arithmetic with fractions *out of the box*:

```
> my $sum = 1/2 + 1/3;  # -> 0.833333
> say $sum.nude;        # -> (5 6)
```

5.7. narrow

Returns the number coerced to the narrowest type that can hold it without loss of precision.

```
> say (4.0 + 0i).narrow.perl;   # -> 4
> say (4.0 + 0i).narrow.^name;  # -> Int
```

It is also possible to apply a type directly, but you may loose precision:

```
> say pi.Int;  # -> 3
```

5.8. sign

`sign` converts the value to Numeric and returns the sign; `0` if the number is 0, `1` for positive, and `-1` for negative values.

```
> say 6.sign;     # -> 1
> say (-6).sign;  # -> -1
> say "0".sign;   # -> 0
```

5.9. Rounding

We can force a non-integer number to an integer in several ways.

5.9.1. round

Use `round` to round the invocant (converted to Numeric if necessary) to nearest integer:

```
> say 1.7.round;      # -> 2
> say (−0.5 ).round;  # -> 0
> say ( .5 ).round;   # -> 1
```

`round` can take a second argument, specifying a value that we'll round off to (a multiple of):

```
> say 1.07.round(0.1);  # -> 1.1
> say 21.round(10);     # -> 20
```

5.9.2. truncate

Use `truncate` to round the invocant (converted to Numeric if necessary) towards zero.

```
> say 1.2.truncate;   # -> 1
> say truncate -1.2;  # -> -1
```

5.9.3. floor

Use `truncate` to round the invocant (converted to Numeric if necessary) downwards to the nearest integer.

```
> say "1.99".floor;  # > 1
> say "-1.9".floor;  # -> -2
```

5.9.4. ceiling

Use `ceiling` to round the invocant (converted to Numeric if necessary) upwards to the nearest integer.

```
> say "1".ceiling;     # -> 1
> say "-0.9".ceiling;  # -> 0
> say "42.1".ceiling;  # -> 43
```

5.9.5. gcd (Greatest Common Divisor)

Converts both arguments to integers returns the greatest common divisor, the largest number that can integer divide them both).

```
> say 10 gcd 12;  # -> 2
```

5.9.6. lcm (Least Common Multiple)

Converts both arguments to integers and returns the least common multiple, the smallest integer that is evenly divisible by both arguments.

```
> say 10 lcm 12;  # -> 60
> say 10 lcm 2;   # -> 10
> say 2 lcm 3;    # -> 6
```

5.9.7. msb (Most Significant Binary)

`msb` returns `Nil` if the number is 0. Otherwise it returns the position (a zero-based index) from the *right* of the most significant (highest value) digit `1` in the binary representation of the number:

```
> say 0b00001.msb;  # -> 0
> say 0b00011.msb;  # -> 1
> say 0b00101.msb;  # -> 2
> say 0b01010.msb;  # -> 3
> say 0b10011.msb;  # -> 4
```

5.10. NaN (Not a Number)

The value `NaN` is of the `Num` type, and is used to represent a floating point «Not a Number» value. It is used as returned value from some mathematical functions where there is no actual Numeric answer - but a Numeric value is still acceptable.

NaN is defined and coverts to True in Boolean context. It is not numerically equal to any value, including itself.

```
> say cos ∞;      # -> NaN
> say (0/0).Num;  # -> NaN
```

5.10.1. isNaN

Use the isNaN method (or the === operator) to test for NaN:

```
> say (0/0).isNaN;  # -> True
```

We can test for NaN explicitly with the Value identity operator === (see section 3.7.7, "===") if we want to:

```
> say (0/0).Num === NaN;  # -> True
```

5.11. =~=

Use the approximately-equal operator =~= (or the Unicode version ≅) to decide if the two values are numerically almost equal. True if returned if the difference is less than the special dynamic value $*TOLERANCE (which defaults to 1e-15), and False otherwise.

The example we used to explain Rat (See section 5.6, "Rational Numbers") was 1/3. Without Rat this would be the result:

```
> my $b = 0.3333333333333333 * 3;

> say $b;        # -> 0.9999999999999999
> say $b == 1;   # -> False;
> say $b =~= 1;  # -> True
```

Note that the Rat type reduces the need for =~= in practice.

5.12. is-prime (Prime Numbers)

A prime number is a number (integer) that can only be divided by 1 and itself.

Use is-prime to find out if a given value is a prime number:

```
> 7.is-prime;    # -> True
> 1.4.is-prime;  # -> False
```

The last one shows why a method or procedure name cannot start with a digit.

Exercise 5.3

Write a program that adds all the prime numbers (in numerically increasing order) from 1 to 100_000 (both included) showing if the sum is a prime or not.

Display how many of those sums are primes.

5.13. Modulo and variants

Numeric	Int	Description
`%`	`mod`	Modulo operator
`%%`		Divisibility operator

5.13.1. %

Use the modulo operator `%` to get the *remainder* after a division:

```
> say 10 % 3;  # -> 1   # 3 * 3 + 1
> say 11 % 3;  # -> 2   # 3 * 3 + 2
> say 12 % 3;  # -> 0   # 4 * 3 + 0
```

The modulo operator `%` converts the values to Numeric before the division, and can be used on non-integers as well:

```
> say 12 % "3.1";  # -> 2.7
```

5.13.2. mod

`mod` is the Integer version of `%`. The values must be integers:

```
> say 7 mod 3;  # -> 1   # 2 * 3 + 1
> say 8 mod 3;  # -> 2   # 2 * 3 + 2
> say 9 mod 3;  # -> 0   # 3 * 3 + 0
```

5.13.3. %%

The Divisibility operator `%%` is the twin sister of the Modulo operator. The Modulo operator returns the *result* of the division, and the Divisibility operator `%%` return `True` if the remainder is zero:

```
> say 9 %% 3;         # -> True
> say 11 %% 3;        # -> False
> say 9.3 %% "3.1";   # -> True
```

The Divisibility operator `%%` converts the values to Numeric before the division, and can be used on non-integers as well.

The Divisibility operator is a shortcut for: `$a % $b == 0`.

Exercise 5.4

Write a program that adds every number from 1 to 1000 that isn't divisible by 7. Use `next`.

5.14. Other Operators

Raku has a lot of builtin mathematical operators.

5.14.1. sqrt

The square root of a number:

```
> say sqrt(9);         # -> 3
> say sqrt(-1);        # -> NaN
> say sqrt(-1 + 0i);   # -> 0+1i
```

5.14.2. exp

Converts the arguments to `Numeric`, and returns `$base` raised to the power of `$power`. `e` (Euler's Number; see section 5.5.2, "e") is used if `$base` isn't provided.

```
> $power.exp($base);
```

```
> say exp 3;      # -> 20.085536923187668
> say 3.exp;      # -> 20.085536923187668
> say 2.exp(3);   # -> 9
> say exp(2, 3);  # -> 9
```

5.14.3. **

The exponentiation operator `**` converts both arguments to `Numeric` and calculates the left-hand-side raised to the power of the right-hand side:

```
> say 2 ** 3;  # -> 8
> say 3 ** 2;  # -> 9
> say e ** 3;  # -> 20.085536923187664
```

Note the rounding error (even though they should be identical):

```
> say e ** 3 - exp 3;  # -> -3.552713678800501e-15
```

It is also possible to use Unicode superscript digits, e.g.: `2³` is the same as `2 ** 3`.

5.14.4. expmod

`expmod` returns the first argument raised to the power of the second element, and applies the third argument as modulus on that:

```
> say expmod(4, 2, 5);  # -> 1  ## The same as: 4 ** 2 mod 5
> say 7.expmod(2, 5);   # -> 4  ## The same as: 7 ** 2 mod 5;
```

The method form requires integers. The procedure version accepts non-integers as well, but truncates them to integers.

5.14.5. log

`log` returns the Logarithm to the specified base (which defaults to `e`, Euler's Number; see section 5.5.2, "e", if not given) of the given value:

```
> say log(10);     # -> 2.302585092994046
> say log(10, e);  # -> 2.302585092994046
```

The result (`2.302585092994046`) is a number that when raised to the power of the base (`e`) gives the value (`10`), or rather an approximation:

```
> say exp 2.302585092994046;  # -> 10.000000000000002
```

Let us try with another base:

```
> say log(10, pi);                    # -> 2.0114658675880612
> say exp(2.0114658675880612, pi);   # -> 10.000000000000002
```

It returns `NaN` if the base is negative, and throws an exception if it is `1`.

log10

`log10` returns its Logarithm to base 10, that is, a number that approximately produces the original number when raised to the power of 10.

It can be replaced by `log`:

```
> say log10(1001);   # -> 3.0004340774793183
> say log(1001,10);  # -> 3.0004340774793183
```

It returns `NaN` for negative arguments and `-Inf` for `0`.

5.14.6. Trigonometric Functions

This is the complete list. They all take radians:

Function	Description
`sin`	sine
`asin`	arc-sine
`cos`	cosine
`acos`	arc-cosine
`tan`	tangent
`atan`	arc-tangent
`atan2`	arc-tangent (two argument form)
`sec`	secant
`asec`	arc-secant
`cosec`	cosecant
`acosec`	arc-cosecant
`cotan`	cotangent
`acotan`	arc-cotangent
`sinh`	sine hyperbolic
`asinh`	inverse sine hyperbolic
`cosh`	cosine hyperbolic
`acosh`	inverse cosine hyperbolic
`tanh`	tangent hyperbolic
`atanh`	inverse tangent hyperbolic

`sech`	secant hyperbolic
`asech`	inverse secant hyperbolic
`cosech`	cosecant hyperbolic
`acosech`	inverse secant hyperbolic
`cotanh`	hyperbolic cotangent
`acotanh`	inverse hyperbolic cotangent
`cis`	cos(argument) + i*sin(argument)

If you cannot find the function you are looking for in this table, have a look at the «Math::Trig» module.

We'll install and have a closer look at «Math::Trig» in Exercise 12.1.

Exercise 5.5

You have a jar (or cylinder) with internal radius 10cm. It is 50cm high. How many litres of liquid can it hold?

Don't remember the formula? Look it up. E.g. https://www.varsitytutors.com/hotmath/hotmath_help/topics/volume-of-a-cylinder

Exercise 5.6

You are offered two jars (or cylinders), the first one with internal radius 10cm and height 35cm, and the second one with internal radius 35cm and height 10cm. You want the biggest one (in terms of content). Which one do you choose?

Chapter 6. Basic Input and Output

This chapter discusses input from the user, and output to the screen.

(Files are discussed in Chapter 13, *Files and Directories*.)

6.1. Newlines

All functions that reads something (from a file or the terminal) strips off trailing newline character(s) by default. (And `say` adds them back on.)

The newline is marked with one or two characters:

Operating System	**Character(s)**	**In strings**	**Codepoint**
Windows	<CR><LF>	\r\n	10 + 13
Linux	<LF>	\n	10
Mac OSX	<LF>	\n	10
Mac (old)	<CR>	\r	13

The special variable `$?NL` reports what the compiler will use when we print a newline (either implicitly with `say` or explicitly with "\n".

`$?NL` gives the actual newline character(s), so just printing the variable doesn't help.

We can use `ords` (see section 7.1.6.1, "ords") to get the codepoints:

```
> $?NL.ords;   # On Linux we get «10»
(10)
```

The variable is read only:

```
> $?NL = "\n";
Cannot assign to an immutable value in block <unit> at <unknown file> line 1
```

6.1.1. chop

`chop` coerces the invocant (or in sub form, its argument) to a string, and returns it with the last character removed.

```
> say "abcde".chop; # -> abcd
```

The removed character is not obtainable in any way, so `chop` cannot be used to iterate over a string (from the end). Use `comb` instead (see section 7.5, "comb").

`chop` takes an optional parameter, specifying the *number* of characters to remove:

```
> say "abcde".chop(2); # -> abc
```

6.1.2. chomp

`chomp` coerces the invocant (or in sub form, its argument) to a string, and returns it with the last character removed, if it is a logical newline.

This isn't very useful in practice, as the standard reading behaviour removes newline characters by default.

6.2. Stringification

There are three built-in ways for **explicit** stringification of a non-string value: `gist`, `Str` and `raku` (formerly `perl`). (Though the first two are mostly applied **implicitly**, as we'll show in section 6.3, "Output".)

The differences can be summarised like this:

Expression	gist	Str	raku / perl	comment
`pi`	`3.141592653589793`	`3.141592653589793`	`3.141592653589793e0`	[1]
`$a = pi`	`3.141592653589793`	`3.141592653589793`	`3.141592653589793e0`	
`1/3`	`0.333333`	`0.333333`	`<1/3>`	[2]
`$b = 1/3`	`0.333333`	`0.333333`	`<1/3>`	
`Any`	`(Any)`	*Error*	`Any`	[3]
`$c = Any`	`(Any)`	*Error*	`Any`	[4]
`(1..5)`	`1..5`	`1 2 3 4 5`	`1..5`	
`@a = 1..5`	`[1 2 3 4 5]`	`1 2 3 4 5`	`[1, 2, 3, 4, 5]`	
`(1..Inf)`	`1..Inf`	`1..*`	`1..Inf`	[5]
`@b = (1..Inf)`	`...`	`...`	*Error*	[6]

[1] The `pi` constant is a Floating point number (see section 5.5, "Floating Point Numbers"), with the internal type `Num`.

[2] The `1/3` expression gives a Rational Number (see section 5.6, "Rational Numbers", with internal type `Rat`.

[3] The error message is «No such method 'str' for invocant of type 'Any'. Did you mean 'Str'?».

[4] The error message is «Use of uninitialized value $a of type Any in string context».

[5] The `1..Inf` expression is a lazy list. The individual values are only evaluated when they are needed. See section 16.1.1, "Lazy vs Eager" for more information.

[6] The error message is «Cannot .elems a lazy list».

Note the difference in output from lists (lazy or not) when we assign them.

6.2.1. gist

The dictionary definition of «gist» is «the substance or general meaning of a speech or text.»

Every object has a `gist` method (inherited from the base class «Mu», see section 3.2, "^mro (Method Resolution Order)"). Its mission (according to the official documentation) is to «Return a string representation of the invocant, optimized for fast recognition by humans.»

`gist` returns only *partial information* about the object, if it is large. A long list is capped at the 100th element (followed by «...»):

```
> my @a = (1 .. 110); say @a.gist;
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99 100 ...]
```

`gist` handles undefined values:

```
> my $a; say $a.gist;
(Any)
```

6.2.2. Str

`Str` flattens data structures:

```
> my @a = (1,2,3, (4,5,6),7,8,9); say @a.Str;

> say @a.gist
[1 2 3 (4 5 6) 7 8 9]
```

`Str` fails on undefined values:

```
> my $a; say $a.Str
Use of uninitialized value $a of type Any in string context.
```

6.2.3. raku (perl)

Every object has a `raku` method. It can be used to dump the object (recursively), and it shows the objects as Raku actually stores them.

This method is useful for dumping of data structures in a format suitable for passing to a program later on.

```
> say (1/3).raku;  # -> <1/3>
```

Older versions of Raku does not have the `raku` method. Use `perl` instead, if you are unable to upgrade.

6.2.4. Summary

	`gist`	`Str`	`raku`
Flattens data structures	No	Yes	No
Handles undefined values	Yes	No	Yes
Shows everything	No	Yes	Yes

The old name (from before the language rename from «Perl 6» to «Raku») is `perl`, and you will probably see it around for quite some time.

6.3. Output

We have used `say` (and sometimes `print`) to display data on the screen, and said that they are equal, except the added newline with `say`.

That isn't exactly correct. The `put` routine is `print` with an added newline, and `say` is somewhat magical.

Function	**With \n**	**Stringification**	**See section**
`say`	Yes	`.gist`	6.3.1, "say"
`put`	Yes	`.Str`	6.3.2, "put"
`print`	No	`.Str`	6.3.3, "print"
		`.raku`	

We'll start with `say`, then look at `put`, and finally `print`.

6.3.1. say

`say` is the most used output method in Raku (instead of `print`), as it adds a newline at the end.

Note that `say` on a variable or data structure uses `gist`, as stated in the table above.

But as soon as you put it in a string, e.g. `say "$a"` it uses `Str` to sort out the string, and then it uses `gist` on that string (which doesn't do anyting, as it already is a string). This applies to strings inside curlies as well:

```
> my $a; say "The value is: $a";
Use of uninitialized value of type Any in string context.

> say "The value is: { $a }";
Use of uninitialized value of type Any in string context.
```

They must be stringified manually, with `gist` or `raku` (that handle undefined values):

```
> say "The value is: { $a.gist }";
The value is: (Any)

> say "The value is: { $a.raku }";
The value is: Any
```

We can use the «Defined-or» operator `//` (see section 2.12, “//”) if we want control over the output:

```
> say "foo { $a // "Undefined" } bar"
foo Undefined bar
```

It is also possible to use `defined` (see section 3.5.2, “defined”) or `try` (see section 3.6.3, “Preventing Runtime errors with `try`”) to prevent the output of undefined values, depending on what you want to achieve.

6.3.2. put

`put` uses `Str` on the values, regardless of type. It adds a newline at the end.

```
> put ^10;  # -> 0 1 2 3 4 5 6 7 8 9
> say ^10;  # -> ^10
```

Again, beware of interpolation with `say`:

```
>  put "{ ^10 }";  # -> 0 1 2 3 4 5 6 7 8 9
>  say "{ ^10 }";  # -> 0 1 2 3 4 5 6 7 8 9
```

6.3.3. print

`print` (and `say` and `put`) prints to the specified filehandle, and to `$*OUT` if used without one.

`print` is the same as `put`, except that it doesn’t add a newline at the end.

6.3.4. put vs say

Why use `put` instead of `say` (as advocated in the book «Learning Perl 6»):

- say uses gist
- put uses Str

`my @a = 1..5`	**Result**	**Comment**
`say @a`	`[1 2 3 4 5]`	[1]
`say @a, "X"`	`1 2 3 4 5]X`	[1]
`say @a ~ "X"`	`1 2 3 4 5X`	
`say "@a[]"`	`1 2 3 4 5`	
`put @a`	`1 2 3 4 5`	
`put "@a[]"`	`1 2 3 4 5`	
`print @a, "\n"`	`1 2 3 4 5`	
`print "@a[]\n"`	`1 2 3 4 5`	

All of them has a newline at the end.

[1] This is the only cases where say actully uses gist. In all the others string interpolation applies first, using Str. This also applies to implicit output in REPL mode (see section 1.3, "REPL").

Conclusion: If it doesn't matter if you only get a partial data structure, use say. If it does matter, use put.

6.4. Stringifying Numbers

Numbers are problematic. Especially Rational Numbers (of the RAT type). E.g.:

```
> my $third = 1/3;
> say $third;       # -> 0.333333
> say $third.Str;   # -> 0.333333
> say $third.gist;  # -> 0.333333
> say $third.raku;  # -> <1/3>
```

A number with an inifinite number of digits, as 1/3 has, will be truncated. Here we have six digits after the decimal point.

Coercing the value to a Floating Point Number, before stringification, gives a different result:

```
> (1/3).Num.say
0.3333333333333333
```

Coercing a Rational Number to another type, and back again, can also be problematic:

```
> 1/3           <=> 0.33333333333333; # -> More
```

The 1/3 is actually 1/3 when we do the three way comparison (see section 3.7.3, "<=>"), so this makes sense.

```
> (1/3).Str.Num <=> 0.33333333333333; # -> Less
```

Here `1/3` has been stringified to `0.333333` and coerced back to a numeric value.

Here the computations are correct, but the stringification doesn't match up:

```
> my $r = 1/3;               # -> 0.333333
> my $s = .3333333333333;    # -> 0.3333333333333
> say "$r > $s ? {$r > $s}"  # -> 0.333333 > 0.3333333333333 ? True
```

The number of digits *after* the decimal point is 1 more than the number of digits in the denominator, with six as the minium.

```
> say (7412 / 123456789).raku;  # -> <7412/123456789> ## Denominator with 9 digits
> say (7412 / 123456789);       # -> 0.0000600372     ## 10 digits
```

Note that the number of digits are based on the internally reduced value, not the one specified by the programmer:

```
> say (123123123 / 321321321).raku;  # -> <41/107>  ## Denominator with 3 digits
> say (123123123 / 321321321);       # -> 0.383178  ## 6 digits, the minimum
```

6.5. printf

Use `printf` («print formatted») to format values before printing them.

When used as a function, the first argument is the *format* string and the rest is the values to print:

```
> my $x = 1;
> printf("%e\n", $x);  # -> 1.000000e+00
```

I recommend using single quotes on the format string. I had to use double quotes above to get the newline.

The line above works because hash variables are not interpolated in strings unless we add {} at the end. (And remember that we have to add [] after an array variable to get that interpolated.)

```
> my %e = ( A => 14 );
> my $x = 1;
> printf("%e\n", $x);  # -> 1.000000e+00
> printf("%e{}\n", $x);
Your printf-style directives specify 0 arguments, but 1 argument was
  supplied in block <unit> at <unknown file> line 1
```

Now %e{} means the hash variable, and the format string has no directives. Then it gets one argument, and complains.

When used as a method, we invoke it on the *format* string, and pass the values as arguments:

```
> "%s\n".printf($x);
```

The example uses %s which means a string, and it prints the string we pass it, attaching a newline at the end.

The directives are:

Sequence	**Description**
%	A literal percent sign
%b	An unsigned integer, in binary
%c	A character with the given codepoint
%d	A signed integer, in decimal
%e	A floating-point number, in scientific notation
%E	Like e, but using an uppercase "E"
%f	A floating-point number, in fixed decimal notation
%g	A floating-point number, in %e or %f notation
%G	Like g, but with an uppercase "E" (if applicable)
%o	An unsigned integer, in octal
%s	A string (stringification with Str)
%u	An unsigned integer, in decimal
%x	An unsigned integer, in hexadecimal
%X	Like x, but using uppercase letters

Some examples:

```
> printf("%e\n", 1);
1.000000e+00

> my $name = "Tom";
> printf("Hello %s, and welcome to the jungle!\n", "$name");
Hello Tom, and welcome to the jungle!
```

Normal interpolation of variables inside strings reduces the usefulness of `printf`:

```
> my $name = "Tom";
> print "Hello $name, and welcome to the jungle!\n";
Hello Tom, and welcome to the jungle!
```

6.5.1. Parameter Index

If the number of arguments doesn't match with the format string we get an error (as shown in the Warning above):

```
> printf("%s\n", "Tom", 11222);
Your printf-style directives specify 1 argument, but 2 arguments were supplied
  in block <unit> at <unknown file> line 1
```

It is possible to shuffle the order they are used:

```
> printf("Hello %s, and %s.\n", "Tom", "Welcome");
Hello Tom, and Welcome.

> printf('Hello %2$s, and %1$s.' ~ "\n", "Tom", "Welcome");
Hello Welcome, and Tom.
```

The first argument is specified as `1$`, the second one as `2$` and so on.

It is also possible to reuse arguments:

```
> printf('Hello %1$s, and %1$s.' ~ "\n", "Tom");
Hello Tom, and Tom.
```

6.5.2. Flags

We can specify *flags* between the `%` and the letter:

Flag	Description	Example	Result

space	Prefix a non-negative number with a space	`printf '§% d§', 12;`	`§ 12§`
		`printf '§% d§', 0;`	`§ 0§`
		`printf '§% d§', -12;`	`§-12§`
`+`	Prefix a non-negative number with a plus sign	`printf '§%+d§', 12;`	`§+12§`
		`printf '§%+d§', 0;`	`§+0§`
		`printf '§%+d§', -12;`	`§-12§`
number	Add leading spaces so that the value uses at least this number of characters	`printf '§%6s§', 12;`	`§    12§`
`-`	Add trailing spaces (instead of leading)	`printf '§%-6s§', 12;`	`§12    §`
`0`	Use leading zeros, not spaces, for padding	`printf '§%06s§', 12;`	`§000012§`
`#`	Show a leading "0" and the type prefix (hexadecimal with "0x" or "0X", octal with "0o" and binary with"0b" or "0B")	`printf '§%#o§', 12;`	`§014§`
		`printf '§%#x§', 12;`	`§0xc§`
		`printf '§%#X§', 12;`	`§0XC§`
		`printf '§%#b§', 12;`	`§0b1100§`
		`printf '§%#B§', 12;`	`§0B1100§`

It is possible to specify the number as a parameter. The following lines give the same result:

```
> printf '|%6s|', 12;     # -> |    12|
> printf '|%*s|', 6, 12;  # -> |    12|
```

This makes it easy to specify the width as a variable.

It is also possible to specify the maximum number of characters to display. See https://docs.raku.org/routine/sprintf for details.

6.5.3. sprintf

`sprintf` («string print formatted») behaves the same way as `printf` (see the previous section), but it return the string instead of printing it.

We can use `sprintf` and `say` to avoid specifying the newline:

```
> printf('Hello %2$s, and %1$s.' ~ "\n", "Tom", "Welcome");
Hello Welcome, and Tom.

> say sprintf('Hello %2$s, and %1$s.', "Tom", "Welcome");
Hello Welcome, and Tom.
```

`sprintf` is a function only. The corresponding method is `fmt` (see the next section):

```
> my $t1 = $string.fmt($format);
> my $t2 = sprintf($format, $string); # Exactly the same
```

6.5.4. fmt

`fmt` is the method version of the `sprintf` function (see the previous section).

If `fmt` is used without a format parameter, it defaults to `%s`.

Some examples:

```
> say 1200.fmt("%o"); # -> octal
2260

> say 1200.fmt("Octal: %o"); # More verbose.
Octal: 2260

> say 1200.fmt('Decimal: %1$d - Octal: %1$o');
Decimal: 1200 - Octal: 2260
```

6.6. Input from the user

6.6.1. prompt

Use `prompt` to display an optional message and wait for the user to type something.

File: prompt

```
my $name = prompt "What's your name? ";
say "Hi, $name! Nice to meet you!";
```

`prompt` is `say` and `get` (see section 13.6, "get") combined.

Remember the difference between numbers and strings?

- A value in quotes is a string
- A value without quotes is a either a number or an error

So what happens when we have input from the terminal, where we don't use quotes on strings?

Let us try, by applying `^name` on a variable filled with `prompt`:

File: prompt-type

```
loop
{
  my $name = prompt "Enter a value (or return to exit): " or exit;
  say "Value $name is of the type { $name.^name }.";
}
```

```
$ raku prompt-type
Enter a value (or return to exit): Allan
Value Allan is of the type Str.
Enter a value (or return to exit): 12
Value 12 is of the type IntStr.
Enter a value (or return to exit): "12"
Value "12" is of the type Str.
Enter a value (or return to exit):
```

Raku is able to guess the type, except when we specify something that looks like a number - without quotes.

Then the type is `IntStr`, something than can be a string and an integer at the same time - without type conversion as we'd normally need.

6.6.2. Str Inheritance Tree

We find the `IntStr` type in the Inheritance tree for `Str`:

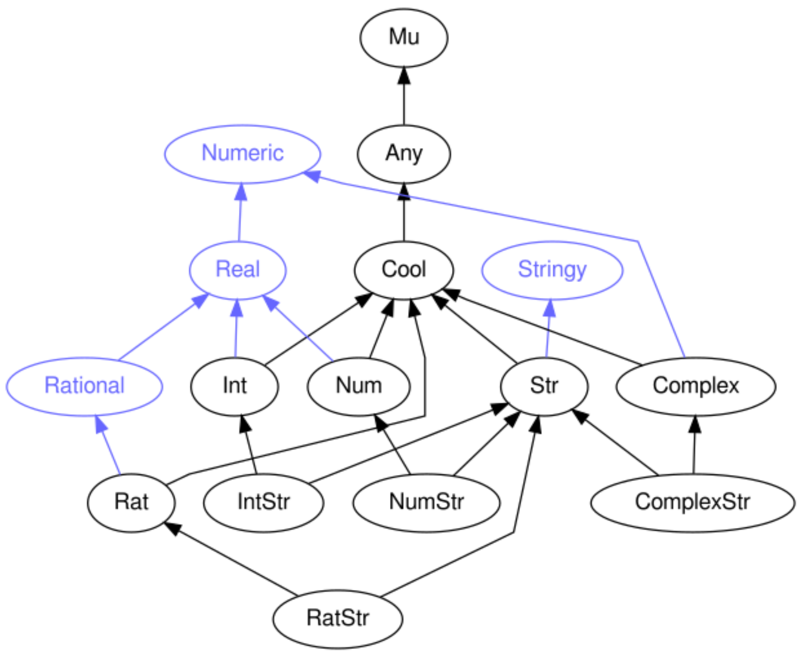


Figure 9. Inheritance Tree for `Str`

The Graph Online: https://docs.raku.org/type/Str#Type_Graph

See section 3.2.1, "Int Inheritance Tree" for the `Int` Inheritance Tree.

```
> say IntStr.^mro;  # -> ((IntStr) (Int) (Str) (Cool) (Any) (Mu))
> say Str.^mro;     # -> ((Str) (Cool) (Any) (Mu))
> say Int.^mro;     # -> ((Int) (Cool) (Any) (Mu))
```

Note that `^mro` (method resolution order) is just that - the **order** of the classes where methods are looked up - for the type it is applied on.

Applying `raku` on the `mro` object makes it clearer that this is just a flat structure (a list):

```
> say IntStr.^mro.raku;  # -> (IntStr, Int, Str, Cool, Any, Mu)
```

We avoided the problem (if we look at it like a problem that is) of an `IntStr` value by using quotes. But that doesn’t work when we have input from the command line (where shell quoting is an issue); see section 10.10.1, “MAIN with typing”.

Exercise 6.1

Write a program that asks for a number, assuming the input is in binary and all the way up to hexadecimal format (i.e base 2, base 3 .. base 16), printing the value in decimal.

Don’t print the value if it fails; e.g «12» isn’t binary.

Chapter 7. Strings

This chapter presents strings and some basic things we can do with them.

Strings are true scalar values, and cannot be treated as a list of characters. (We can *convert* it to a list of characters, as we'll see later, but that is another matter.)

We cannot pick a single character from a string.

7.1. Unicode

All strings are encoded in Unicode.

Note the common naming structure of these functions:

One value	Several values	Description
`uniname`	`uninames`	Get the unicode name(s) of the character(s)
`uniparse`		Get the Unicode character with the given unicode name
`ord`	`ords`	Get the Unicode code point(s) of the character(s)
`chr`	`chrs`	Get the Unicode character(s) with the given code points(s)

The missing several-value edition of `uniparse` is easy to implement with a hyper operator. We will cover both in the «Advanced Raku» course.

7.1.1. chars

Use `chars` to get the number of characters in a string:

```
> say "abc".chars;  # -> 3
```

If used on anything else, it will be converted to a string:

```
> say chars(pi);  # -> 17
```

We can use `chop` and `chars` to get the *first* character in a string:

```
> my $s = "1234567890";
> say $s.chop($s.chars -1);  # -> 1
```

But we'll show better ways later.

Note that the value is the number of Unicode Graphemes, or user visible characters. If it looks like a single character, it is a single Grapheme.

Let us consider the Scandinavian letter «Å». It is present in the Unicode character set (with the code U+00C5), and can be used «out of the box». In Unicode we can also make it up by combining the letter «A» (U+0041) with a «Combining Ring Above» (U+030A) like this:

```
> my $s = "A\c[Combining Ring Above]";     # -> Å
> say $s.chars;                            # -> 1
> say "A\c[Combining Ring Above]" eq "Å";  # -> True
```

The code above shows us that Raku normalizes the string when it reads it in, whether from a string or a file. This means that what you get isn't necessarily what was given.

(Newlines (the end of line marker, which differ on Unix, Windows and Mac) are also normalized, as described in section 6.1, "Newlines".)

7.1.2. Combining Characters

The combining character (or characters) come **after** the base character. (This means that the compiler will have to look at least one byte ahead when it reads strings one character at a time; e.g `for $string.comb → $char { … }`.)

7.1.3. codes

This gives us the number of Unicode Code Points. It is usually the same as `chars`.

```
> say "12øøæåsaåsæ".codes;  # -> 11
> say "12øøæåsaåsæ".chars;  # -> 11

> say "A\c[Combining Ring Above]".codes;  # -> 1
> say "A\c[Combining Ring Above]".chars;  # -> 1
```

As Unicode have a `Å` character. We can try something that Unicode doesn't have:

```
> say "O\c[Combining Ring Above]".codes;  # -> 2
> say "O\c[Combining Ring Above]".chars;  # -> 1
```

7.1.4. uniname

Use `uniname` to get the Unicode name for **the first** character (grapheme) in the string:

```
> say "A\c[Combining Ring Above]".uniname;
LATIN CAPITAL LETTER A WITH RING ABOVE

> say "abc".uniname;
LATIN SMALL LETTER A
```

We can specify any character with the Unicode name:

```
> say "\c[LATIN SMALL LETTER A]"; # -> a
> say "\c[Latin Small Letter a]"; # -> a
```

uninames

Use `uniname` to get the Unicode name for **all the character(s)** in a string:

```
> say "O\c[Combining Ring Above]".uninames;
(LATIN CAPITAL LETTER O COMBINING RING ABOVE)
```

We can use the `raku` method to get a nicer list:

```
> say "O\c[Combining Ring Above]".uninames.raku;
("LATIN CAPITAL LETTER O", "COMBINING RING ABOVE").Seq

> say ‘»ö«’.uninames.raku;
«("RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK", "LATIN SMALL LETTER O WITH DIAERESIS",
"LEFT-POINTING DOUBLE ANGLE QUOTATION MARK").Seq»
```

7.1.5. uniparse

We can use `uniparse` on a Unicode name to get the character:

```
> say "LATIN SMALL LETTER A".uniparse;  # -> a
```

It fails if we pass it something illegal:

```
> uniparse("LATIN SMALL LETTER A WITH VERTICAL LINE BELOW AND ACUTE");
Unrecognized character name [LATIN SMALL LETTER A WITH VERTICAL LINE BELOW AND ACUTE]
  in block <unit> at <unknown file> line 1
```

7.1.6. ord

Use `ord` to get the Unicode code point (a number) of **one character**:

```
> "A".ord;     # -> 65
> "Abba".ord;  # -> 65
```

ords

Use `ords` to get the Unicode code points (numbers) of **all the characters** in a string:

```
> "Abb".ords;  # -> (65 98 98)
```

We can revisit the combined characters we showed in the 7.1.3, “codes” section:

```
> say "A\c[Combining Ring Above]".ords;  # -> (197)
> say "O\c[Combining Ring Above]".ords;  # -> (79 778)
```

Note that Unicode merges the first one, the «A» and the combiner as it exist as a separate Unicode character.

Be very careful with `ord` on what you *think* is a single character:

```
> say "O\c[Combining Ring Above]".ord;  # -> 79
```

7.1.7. chr

Use `chr` to turn an integer into a Unicode character.

```
> say 65.chr;   # -> A  ## Decimal
> say 0x41.chr; # -> A  ## Hexadecimal
```

chrs

Use `chrs` to turn a **list of integers** into a string of Unicode characters:

```
> say <67 97 109 101 108 105 97>.chrs;  # -> Camelia
```

We can have a go at the combined characters again:

```
> say (79, 778).chrs.ords;                    # -> (79 778)
> say "O\c[Combining Ring Above]".chrs.ords;  # -> (79 778)
```

Note that they don’t always round trip.

```
> say 197.chr;              # -> Å
> say (65, 778).chrs;       # -> Å
> say (65, 778).chrs.ords;  # -> (197)
```

Note that the «A» and the combiner is replaced by a separate Unicode character.

Exercise 7.1

Write a program that asks for input (in a loop), and replaces every lower case letter (a-z only) with the upper case version, and vice versa, before printing it on the screen. Other characters are left unchanged.

Use `ord`/`ords` and `chr`/`chrs`.

7.2. join

We can use `join` to glue a list of strings together:

```
> say <1 2 3>.join;  # -> 123
```

If we specify an argument to `join`, that string will be used between the elements. We can generate a CSV-file line like this:

```
> say <12 hello 3.14 bingo 87>.join(";");  # -> 12;hello;3.14;bingo;87
```

7.3. split

This is the opposite of `join`:

```
> say "12;hello;3.14;bingo;87".split(";");  # -> (12 hello 3.14 bingo 87)
```

Note that the text we split on (in this case `;`) is not included in the result.

`split` takes an optional second argument, an integer telling it how many parts to split it into:

```
> "12;hello;3.14;bingo;87".split(";", 2)
(12 hello;3.14;bingo;87)
```

7.4. words

We can use `split` and a space character split a string in words:

```
my @words = $text.split(" ");
```

Note that this doesn't quite work out if we have multiple spaces after each other. (We could have used a Regex as argument to `split` to fix that, and we'll do that when we get to Chapter 11, *Regex Intro*.)

But the `words` method is self-explanatory, and it handles multiple spaces:

```
my @words = $text.words;
```

`words` doesn't handle punctuation characters very well:

```
> "This is it, isn't it? Or perhaps not. 2nd try".words.join("|");
This|is|it,|isn't|it?|Or|perhaps|not.|2nd|try
```

One possible solution would be replacing everything that isn't a letter or digit with a space character, before applying `words`.

It is possible to use the <one two …> Quote Words construct as well. See section 8.3, "<xxx> (Quote Word)".

7.5. comb

We can use `split` to get a list of single characters from a string:

```
> say "12345".split("");        # -> ( 1 2 3 4 5 )
> say "12345".split("").elems;  # -> 7
```

Note the empty first and last element.

`elems` gives us the number of elements in the list.

It is better to use `comb`:

```
> say "12345".comb;        # -> (1 2 3 4 5)
> say "12345".comb.elems;  # -> 5
```

It is possible to get groups of characters:

```
> say "12345".comb(2);  # -> (12 34 5)
```

Exercise 7.2

Write a program that asks for *integer* values in a loop.

If the value is an integer, compute the sum of all the digits.

Tip: Start with «prompt-type» from section 6.6.1, “prompt”.

Exercise 7.3

Write a program that asks for input in a loop.

- Show the *last* character
- Show every second character

7.6. flip

We can use `comb`, `reverse` and `join` to reverse a string:

```
> say "abc123".comb.reverse.join;  # -> 321cba
```

`comb` gives us a list of single characters, `reverse` reverses the order of this list, and `join` merges the list back together as a string.

But it is easier to use the `flip` method to reverse a string:

```
> say "abc123".flip;  # -> 321cba
```

7.7. substr (Partial Strings)

Use `substr` (substring) to get part of a string, starting at the position given as offset (so the first character is at position 0) and returning the rest of it:

```
> say "1234567890".substr(3);  # -> 4567890
```

The length (number of characters) can be specified as well:

```
> say "1234567890".substr(3, 2);  # -> 45
```

We can get the last character of a string (ref Exercise 7.3):

```
> my $s = "123456";
> say $s.substr($s.chars -1);  # -> 6
```

This also works, where `*` means «from the end»:

```
> $s.substr(* -1)
```

7.7.1. substr-eq (Partial Strings)

`substr-eq` is a combination of `substr` and `eq`:

```
say "abc123".substr-eq("c123", 3); # -> False  ## As "123"  ne "c123"
say "abc123".substr-eq("c123", 2); # -> True   ## As "c123" eq "c123"
```

Note that we cannot specify the length, as we could with `substr`.

7.7.2. substr-rw (Partial Strings)

`substr-rw` is a version of `substr` that returns a writable view to the specified part of the string (as opposed to `substr` that merely returns a copy).

```
> my $s = "abc"; $s.substr-rw(1, 1)  = "Q"; $s.say; # -> aQc
> my $s = "abc"; substr-rw($s, 1, 1) = "Q"; $s.say; # -> aQc
```

We are not restricted by the length:

```
> my $s = "abc"; $s.substr-rw(1, 1) = "QQQ"; $s.say; # -> aQQc
```

The second argument decides *how many* characters to remove. We can set it to zero to make an insertion:

```
> my $s = "abc"; $s.substr-rw(1, 0) = "ZZ"; $s.say; # -> aZZbc
```

We can make an *alias* to a substring with binding (see section 2.6.2, “:= (Binding)”):

File: substr-rw

```
my $string = "abc*123*ABC";

my $partial := $string.substr-rw(4, 3);

say "$string - $partial"; ## -> abc*123*ABC - 123

$partial = "9876543210";

say "$string - $partial"; ## -> abc*9876543210*ABC - 987

$string = "123|abc|456";

say "$string - $partial"; ## -> 123|abc|456 - abc
```

Note that even if we replace 3 characters (`123`) with 10 (`9876543210`) the alias is to the same position and length in the original string.

7.8. Changing Case

In Exercise 7.1 we changed letters from upper- to lower case, and vice versa, with `ord` and `chr`, but it is easier to use the builtin functions.

Function	Description
`lc`	Lower Case
`tc`	Title Case
`tclc`	Title Case Lower Case
`uc`	Upper Case
`fc`	Fold Case
`wordcase`	Word Case

7.8.1. lc (Lower Case)

Returns a copy of the string with all the characters converted to lower case.

```
> say "this is IT!".lc;  # -> this is it!
```

7.8.2. tc (Title Case)

Returns a copy of the string with the first character converted to title case (or upper case, if title case isn’t available), and the rest unchanged.

```
> say "this is IT!".tc;  # -> This is IT!
```

«Title Case» is *almost* the same as «Upper Case» on the first letter. See http://unicode.org/faq/casemap_charprop.html for details if you are curious.

7.8.3. tclc (Title Case Lower Case)

Returns a copy of the string with the first character converted to title case (or upper case, if title case isn't available), and the rest converted to lower case.

```
> say "this is IT!".tclc;  # -> This is it!
```

7.8.4. uc (Upper Case)

Returns an uppercase version of the string.

```
> say "this is IT!".uc;  # -> THIS IS IT!
```

7.8.5. fc (Fold Case)

Returns a version of the string using the Unicode «fold case» method. This is recommended for string comparisons only.

```
> say "this is IT!".fc;  # -> this is it!
```

Why «Case Folding» is recommended:

We can convert the strings to *upper case*:

```
> say "Saß".uc  # -> "SASS"
> say "Sass".uc # -> "SASS"
```

The German lower case «double s» "ß" is converted to uppercase "SS" (and the length of the string has changed). So if we compare the strings "Saß".uc and "Sass".uc they are equal.

Converting to *lower case* seems safer, and I haven't found an example that it doesn't work. (Feel free to help me out.) But «Case Folding» is guaranteed to work.

7.8.6. wordcase

Returns a copy of the string with the first character in each word converted to upper case, and the rest converted to lower case.

```
> say "this is IT!".wordcase;  # -> This Is It!
```

Note that `wordcase` accepts two optional arguments:

- `:filter` - A function to use instead of the built in `wordcase`
- `:where` - A Boolean expression that turns the conversion on/off for each word

```
> say "this is IT!".wordcase(:where({ .chars == 2 }) );              # -> this Is It!
> say "this is IT!".wordcase(:filter(&uc), :where({ .chars == 2 })); # -> this IS IT!
```

We specify code inside curlies inside the parens. Procedures are specified without the curlies, but with a `&` prefix.

Why do you think this function doesn't have the normal two letter name?

Exercise 7.4

Rewrite «swap-case» from Exercise 7.1 so that it also converts unicode letters, that is letters other than a-z.

7.9. x (String Repetition Operator)

Use the String Repetition Operator `x` to duplicate the string on the left side the number of times given on the right side:

```
> say "123 " x 2;   # -> 123 123
> say "123 " x pi;  # -> 123 123 123
```

The repetition count must be a number (or something that can be converted to a number), and it will be truncated unless already an integer.

Do **not** use `x` as a multiplication operator. It isn't. Using it on numbers will stringify them:

```
> say 3 x 4;  # -> 3333
> say 4 x 3;  # -> 444
```

But you **can** use the Unicode Multiplication sign `×` (with codepoint «U+00D7»):

```
> say 3 × 4;  # -> 12
> say 4 × 3;  # -> 12
```

7.10. succ

`succ` (Successor) used on a number gives us the number incremented by one.

```
> say pi.succ;    # -> 4.141592653589793
> say 109.succ;   # -> 110
```

But it is much more useful (and magic) on strings:

```
> say 'aa'.succ;  # -> ab
> say 'az'.succ;  # -> ba
> say 'α'.succ;   # -> β
> say 'a9'.succ;  # -> b0
```

If there are no dots (periods or `.`) in the string, the last alphanumeric sequence is incremented. If there are one or more dots, the last alphanumeric sequence *before* the first dot is incremented.

```
> say 'a.a.a.a.a'.succ;   # -> b.a.a.a.a
> say 'img001.png'.succ;  # -> img002.png
```

When it reaches the end of the character range (the digits 0-9, letters a-z or other Unicode ranges), it adds another character (as normal in a numeric situation):

```
> say 99.succ;   # -> 100
> say 'z'.succ;  # -> aa
```

If you already have the highest value, nothings happens:

```
> say True.succ;  # -> True
> say False.succ; # -> True
> Inf.succ;       # -> Inf
```

7.11. pred

`pred` (Predecessor) used on a number gives us the number decremented by one.

```
> say pi.pred;    # -> 2.141592653589793
> say 100.pred;   # -> 99
```

Used on strings:

```
> say 'ab'.pred;  # -> aa
> say 'ba'.pred;  # -> ax
> say 'β'.pred;   # -> α
> say 'b0'.pred;  # -> a9
```

If there are no dots (periods or `.`) in the string, the last alphanumeric sequence is decremented. If there are one or more dots, the last alphanumeric sequence *before* the first dot is decremented.

```
> say 'b.a.a.a.a'.pred;   # -> a.a.a.a.a
> say 'img002.png'.pred;  # -> img001.png
> say 'img000.png'.pred;  # -> imf999.png
```

But it does not decrease the number of characters:

```
> say 'aaaa'.pred; # -> Decrement out of range ...
> say '100'.pred;  # -> 099
```

(This differs from `succ`, that adds another character when necessary.)

If you already have the lowest value, you either get an error (as above) or the same value:

```
> say False.pred; # -> False
> say "a".pred;   # -> Decrement out of range ...
> say -Inf.pred;  # -> -Inf
```

7.12. Quoting

We described single and double quotes in section 2.7.1, “Strings”, but we have many more quoting constructs:

Short	String	Result	Description
single quote	`'ABC$a'`	`ABC$a`	Nothing is interpolated
`Q`	`Q#ABC$a#`	`ABC$a`	Nothing is interpolated
	`Q{ABC$a}`	`ABC$a`	Nothing is interpolated. Note the start and end characters
`q`	`q*ABC$a*`	`ABC$a`	Nothing is interpolated. Note the start and end characters
`q:c`	`q:c/ABC$aX{$a}/`	`ABC12$aX12`	Only closures are interpolated
double quote	`"ABC$a {$a}"`	`ABC12 12`	Variables and closures are interpolated
`qq`	`qq/ABC$a {$a}/"`	`ABC12 12`	Variables and closures are interpolated

`qw`			Quote Words; see section 7.12.1, “qw (Quote Words)”
`qqw`			Quote Words (with interpolation); see section 7.12.2, “qqw (Quote Words with interpolation)”
`qx`			Execute program; see the «Advanced Raku» course
`qqx`			Execute program (with interpolation); see the «Advanced Raku» course

(Given that we have specified `my $a = 12` somewhere.)

A closure is a thingy specified inside curlies.

7.12.1. qw (Quote Words)

This is the same as applying `words` on a single quoted string:

```
> say '1 $aaaaa 17'.words; # -> (1 $aaaaa 17)
> say qw/1 $aaaaa 17/;     # -> (1 $aaaaa 17)
```

7.12.2. qqw (Quote Words with interpolation)

This is the same as applying `words` on a double quoted string:

```
> my $aaaaa = "X";
> say "1 $aaaaa 17"words; # -> (1 X 17)
> say qqw/1 $aaaaa 17/;   # -> (1 X 17)
```

7.13. Multi-line Strings (Heredocs)

Printing a multi-line string isn’t very nice looking, especially if we need embedded newlines:

```
> print "Line 1\nLine2\nline3\n";
Line 1
Line2
line3
```

A much more convenient way is a heredoc:

```
say q:to/END/;
Here is
some multi-line
string
END
```

The contents of the heredoc always begin on the next line. The end can be any literal string, as long as we specify it up front:

```
say q:to/BLAH/;
Here is
some multi-line
string
BLAH
```

If the terminator («END» in our case) is indented, that amount of indention is removed from the string literals.

This heredoc;

```
say q:to/END/;
    Here is
    some multi line
        string
    END
```

produces this output:

```
Here is
some multi line
    string
```

7.13.1. Interpolation in heredocs

Nothing gets interpolated in the heredocs described above. But we can use the `q`, `q:c` and `qq` quoting mechanisms described in section 7.12, “Quoting”:

Start	**Interpolates**	**Content**	**Result**
`q:to/END/;`	*nothing*	`$name and {$age}.`	`$name and {$age}.`
`q:to:c/EOF/;`	closures only (variables in curlies {})	`$name and {$age}.`	`$name and 15.`
`qq:to/EOF/;`	closures and variables	`$name and {$age}.`	`Tom and 15.`

(Where we have declared this: `my $name = "Tom"; my $age = 15;`.)

7.13.2. indent

The `indent` method is used internally by heredocs to manage the indentation, but it can be used directly (on strings only):

```
> "abc";
abc

> "abc".indent: 2;
  abc

> "abc".indent(3);
   abc

> "abc".indent: 10;
          abc
```

Chapter 8. Arrays and Lists

Arrays are mutable, and lists are immutable.

Arrays are created and are shown (on output) by square brackets, and lists are shown with parens:

```
> say [1,2,4,5].WHAT;  # -> (Array)
> say (1,2,4,5).WHAT;  # -> (List)
```

Assigning a list to an array variable converts it to an array:

```
> my @something = (1,2,4,5);  # -> [1 2 4 5]
> say @something.WHAT;        # -> (Array)
```

The difference (and distinction) isn't that important, except when it comes to lazy lists. There is no such thing as a lazy array, so assigning a lazy list to an array forces it to be evaluated.

It is possible to assign a list to a scalar variable:

```
> my $something = (1,2,4,5);  # -> (1 2 4 5)
> say $something.WHAT;        # -> (List)
```

Assigning a list to a scalar variable keeps list type. That means that the variable is read-only:

```
> $something[2] = 99;
Cannot modify an immutable List ((1 2 4 5))
```

8.1. , (List Operator)

Use the `,` (comma) list operator to generate a list:

```
> "rune", "helge", "tom", "jerry";
(rune helge tom jerry)
```

Add parens, if it makes you feel better… (The parens are *only* a Grouping Operator; see 2.12.1, "() (Grouping Operator)").

```
> ("rune", "helge", "tom", "jerry");
(rune helge tom jerry)
```

Strings must be quoted, as shown above, but we can use a short form if the values do not contain spaces:

```
> <rune helge tom jerry>;
(rune helge tom jerry)
```

Note that e.g. `say` can take a list. It will then print all the list values glued together:

```
say "ABC" ~ "123"  # -> ABC123
say "ABC, "123"    # -> ABC123
```

8.2. [ ] (Array Constructor)

Use the `[` and `]` Array Constructor to make an explicit Array:

```
> [1, 2, 3, 4].WHAT
(Array)
```

Note that assigning something to an array variable (the `@` sigil) coerces it to an array:

```
> my @a = <rune helge tom jerry>;  # -> [rune helge tom jerry]
> say @a.WHAT;                     # -> (Array)
> say <rune helge tom jerry>.WHAT; # -> (List)
```

Be careful with precedence. This works, as the binding operator `:=` has lower precedence than the list operator `,`:

```
> my $a := 1, 2, 3; # -> (1 2 3)
> say $a.WHAT;      # -> (List)
> say $a;           # -> (1 2 3)
```

But doing it with the assignment operator doesn't work:

```
> my $a = 1, 2, 3;  # -> (1 2 3)
> say $a.WHAT;      # -> (Int)
> say $a;           # -> 1
```

The first value is assigned to `$a`, and returned. Then we attach `2` and `3` to that value to form a list (which is returned, and printed by REPL). But that list isn't assigned to a variable, and is lost.

The solution, use the array constructor `[]` or the grouping operator `()`:

```
> my $a = [1, 2, 3];  # -> [1 2 3]
> say $a.WHAT;        # -> (Array)

> my $b = (1, 2, 3);  # -> (1 2 3)
> say $b.WHAT;        # -> (List)
```

⚠ Lists are read only, so `$b` is essentially constant:

```
> $a[1] = 4; # -> 4
> say $a;    # -> [1 4 3]
> $b[1] = 4;
Cannot modify an immutable List ((1 2 3))
  in block <unit> at <unknown file> line 1
```

It works for arrays:

```
> my @a = 1, 2, 3;  # -> [1 2 3]
> say @a;           # -> [1 2 3]
```

8.3. <xxx> (Quote Word)

This is the Quote Word syntax. Specify a string like that, and it is converted to a list of partial strings, with space (and space like characters) as delimiters.

```
> my @a = <Peter Paul Mary>;  # -> [Peter Paul Mary]
> say @a.perl;                # -> ["Peter", "Paul", "Mary"]
```

8.4. Empty

Use `Empty` to get an empty list.

All these are equal:

```
my @a;         # No values.
my @b = ();    # Explicit
my @c = Empty; # Also possible
```

8.5. List Elements

You can access an individual item by its index:

```
> say ("a" .. "z")[25];  # -> z

> my @d = <helge tom barry>; say @d[0];  # And NOT $d[0] as in perl5!
helge
```

This works even if we have used scalar assignment:

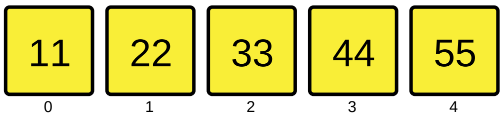
```
> my $a = [1, 2, 3, 4]; say $a[2];  # -> 3
> my $b = (1, 2, 3, 4); say $b[2];  # -> 3
```

8.6. pop / push / shift / unshift

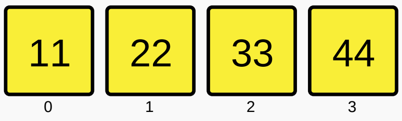
We have some operators that add or remove values from a list:

Function	Description
`pop`	Remove one element from the end
`push`	Add the element(s) at the end
`shift`	Remove one element from the beginning
`unshift`	Add the element(s) at the beginning

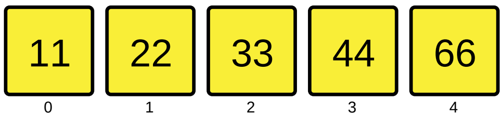
Initial array:
`my @a = <11 22 33 44 55>;`



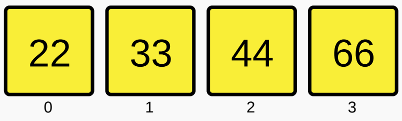
Removing one element (55) from the end:
`@a.pop;` # -> 55



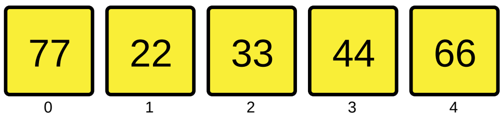
Adding one element at the end:
`@a.push(66);`



Removing one element (11) from the start:
`@a.shift;` # -> 11

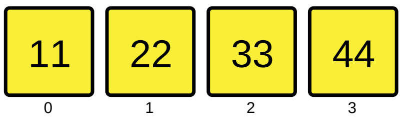


Adding one element at the start:
`@a.unshift(77);`

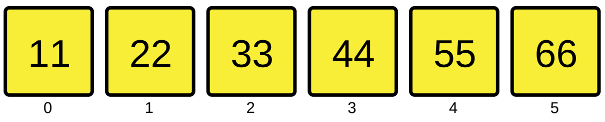


We can add more than one element at the same time with `push` and `unshift`, shown here for `push`:

Initial array:
`my @a = <11 22 33 44>;`



Adding two elements at the end:
`@a.push(55, 66);`



Note that `pop` and `shift` will only remove **one item** at a time.

8.6.1. elems (List Size)

Use `elems` to get the number of elements in a list:

```
> my $number-of-elements = @d.elems;
> my $number-of-elements = elems @d;
```

The value is *read only*, and cannot be used to change the number of elements.

There is no «length» method or function.

We can also get the number of elements by evaluating the list in numeric context:

```
> say +("a" .. "z");  # -> 26
```

end

Use `end` to get the *index* of the last element in a list:

```
> say ("a" .. "z").end;  # -> 25
```

The value is 1 less than the value returned by `elems` (as the first element has index, or offset, 0).

It gives `-1` for an empty array:

```
my @a; say @a.end;  # -> -1
```

Note that the number of elements in a list really is the last defined element in it:

```
> my @a;
> say @a.elems;  # ->  0
> say @a.end;    # -> -1

> my @b = (1);
> say @b.elems;  # ->  1
> say @b.end;    # ->  0


> @b[10] = 's';
> say @a.elems;  # -> 10
> say @a.end;    # -> 11
```

So what can we use them to?

8.6.2. Stack

A stack is a data structure that is easy to implement and does not have much overhead and that is why it is used a lot in low level programming. The last element added to the stack is the first retrieved.

Use `unshift` and `pop` to make a stack:

```
my @stack = ...;

@stack.unshift($customer-id); # Add one element to the stack

my $current = @stack.pop;     # Get one element from the stack
```

8.6.3. Queue

A queue is a data structure where the entries are returned in the order they were added. This is how you generally would treat waiting customers.

Use `push` and `shift` to make a queue:

```
my @queue = ...;

@queue.push($customer-id); # Add one element to the queue

my $current = @queue.shift; # Get one element from the queue
```

8.7. rotate (List Rotation)

List rotations can be done with `push`/`shift` (to the left) and `unshift`/`pop` (to the right), but it is easier to use the built in rotate:

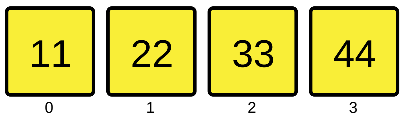
```
(1,2,3,4,5,6).rotate;  # Left. The same as rotate(1)
(2 3 4 5 6 1)

> (1,2,3,4,5,6).rotate(2)
(3 4 5 6 1 2)

> (1,2,3,4,5,6).rotate(-2) # Right
(5 6 1 2 3 4)
```

Note that `rotate` doesn't change the original list, but returns a modified version.

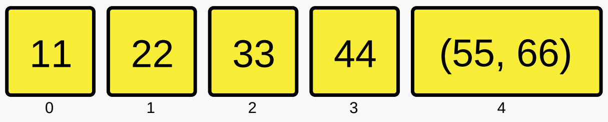
8.8. List of Lists

Raku does not automatically flatten lists (as opposed to Perl 5), so the result of adding (with `push` or `unshift`) a *list* on to a list, is a list with one more item - the second list:

Initial array:
`my @a = <11 22 33 44>`



Adding an array with two elements at the end:
`my @b = <55 66>; @a.push(@b)`



Probably not what you have in mind.

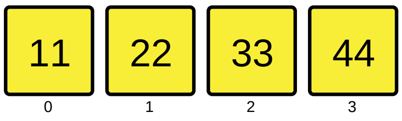
8.9. Flattening Lists

If you want to insert the individual values of a second list to a list, use `prepend` (instead of `unshift` and `append` (instead of `push`):

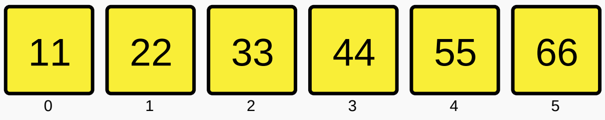
```
> my @list1 = 1,2,3,4,5;
[1 2 3 4 5]

> my @list2 = 8,9;
[8 9]

> @list1.append(@list2); # push the individual values
[1 2 3 4 5 8 9]

> @list1.prepend(@list2); # unshift the individual values
[8 9 1 2 3 4 5 8 9]
```

We can add more than one element at the same time with `append` and `prepend`, shown here for `append`:

Initial array:
`my @a = <11 22 33 44>;`



Adding two elements at the end:
`my @b = <55 66>; @a.append(@b);`



8.10. Array Slice

We can access several items (called an array slice):

```
> my @a = 1,2,3,4,5,6,7,8,9,10,11,12,13;
> @a[0 .. 9]; # The same as [0,1,2,3,4,5,6,7,8,9]
(1 2 3 4 5 6 7 8 9 10)
```

They do not need to be consecutive:

```
> say @a[0,9,2];  # - > (1 10 3)
```

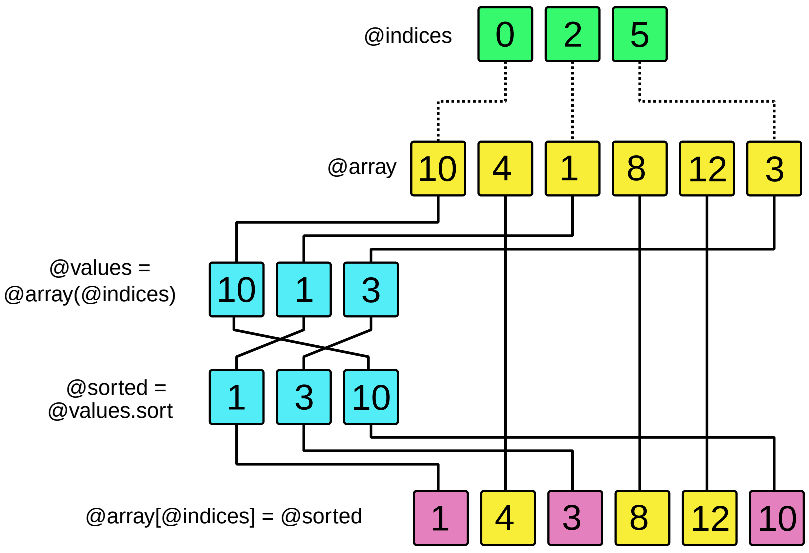
Array slices are writeable, i.e. we can assign values to them:

```
> my @a = <10 9 8 7 6 5 4 3 2 1 0>;

> say @a[2,4,6,8,10,0];        # -> (8 6 4 2 0 10)
> say @a[2,4,6,8,10,0].=sort;  # -> (0 2 4 6 8 10)
> say @a;                      # -> [10 9 0 7 2 5 4 3 6 1 8]
```

We started with a list with the numbers from 10 down to 0. Then we picked some of them, giving a new list of values. We then sorted that list, and assigned the sorted list back (with the `.=` form of the assignment operator =). And finally we show the result.

Another example (with an illustration that *may* help):

```
> my @array   = 10, 4, 1, 8, 12, 3;
> my @indices = 0, 2, 5;
> my @values  = @array[@indices];  # -> (10 1 3)
> my @sorted  = @values.sort;      # -> (1 3 10)

> @array[@indices] = @sorted;      # -> (1 3 10)

> say @array; # -> [1 4 3 8 12 10]
```



We can write the code more compact:

```
> my @array = 10, 4, 1, 8, 12, 3;
> @array[0,2,5].= sort;
> say @array;
[1 4 3 8 12 10]
```

Trying to *remove* values doesn't quite work out:

```
> my @a = 1..10;    # -> [1 2 3 4 5 6 7 8 9 10]
> @a[2,4,5] = ();   # -> ((Any) (Any) (Any))
> say @a;           # -> [1 2 (Any) 4 (Any) (Any) 7 8 9 10]
```

But we can use splice, which we'll cover in the next section.

8.11. splice

Use splice to remove some elements from a list. The removed elements are returned.

We can remove from the given index to the end like this:

```
> @a.splice(5); # Leave the first 5 items, and remove the rest.
```

We can specify the number of items to remove:

```
> @a.splice(5,2); # Leave the first 5 items, remove the next 2, and leave the rest as
well.
```

We can insert another list in place of the elements we removed:

```
> @a.splice(5,2, @list); # Leave the first 5 items, remove the next 2 and replace
                         # with `@list`, and leave the rest as well.
```

Note that the size of the replacement list doesn't have to be the same as the number of removed items.

The replacement values can be specified as scalars as well.

Summary:

code	value of @a	value of @b
`my @a = "a" .. "n"`	`[a b c d e f g h i j k l m n]`	
`my @b = @a.splice(5)`	`[a b c d e]`	`[f g h i j k l m n]`
`my @b = @a.splice(5,2)`	`[a b c d e h i j k l m n]`	`[f g]`
`my @b = @a.splice(5,2, <A B C>)`	`[a b c d e A B C h i j k l m n]`	`[f g]`
`my @b = @a.splice(5,2, 9,9,9,9)`	`[a b c d e 9 9 9 9 h i j k l m n]`	`[f g]`

Note that we have to reset «@a» to the initial value before each `splice`.

It is possible to use `splice` to insert a new value (or several) in a list, without removing anything, by specifying `0` as the second argument:

```
> my @b = @a.splice(10, 0, 3.14);
[]

> say @a;
[a b c d e f g h i j 3.14 k l m n]
```

8.11.1. | (Flattening Operator)

We can flatten a list, by adding a `|` (vertical bar) before it:

```
> my @list1 = 1,2,3,4,5;
[1 2 3 4 5]

> my @list2 = 8,9;
[8 9]

> @list1.push(@list2);
[1 2 3 4 5 [8 9]]

> @list1.push(|@list2); # push the individual values, and not the list
[1 2 3 4 5 8 9]
```

Note that the flattening is performed on the top level only (not recursively).

8.11.2. flat

Or use the more verbose (and explicit) `flat` method:

```
> (1, (2, (3, 4)), 5).flat;
(1 2 3 4 5)
```

The flattening doesn’t flatten arrays recursively, only lists (as shown above):

```
> my @a = (1,2);            # -> [1 2]
> my @b = (1, 2, @a, 1, 2); # -> [1 2 [1 2] 1 2]
> @b.flat;                  # -> (1 2 [1 2] 1 2)
```

We can force it to flatten arrays with a hyper operator

```
> @b».List.flat;   # Unicode version
> @b>>.List.flat;  # ASCII version
```

Hyper Operators will be covered in the «Advanced Raku» course.

8.12. map

Use `map` to apply a bit of code to every element in a list. It leaves the original list unchanged, and returns the modified version:

```
> my @a = 1..10;        # -> [1 2 3 4 5 6 7 8 9 10]
> @a.map({ $^a + 1 });  # -> (2 3 4 5 6 7 8 9 10 11)
```

We use curlies to pass a code block. This block is executed for each value, and the result is placed in the resulting list.

⚠ Note that map only cares about the top level of the list it is used on. See deepmap (in section 8.12.4, “deepmap”) for details.

8.12.1. Placeholder Variables

The twigil (see section 2.2.2, “Twigils”) ^ indicates that it is a placeholder variable. It can be named whatever you want, as long as the twigil is there.

We use the placeholder variable $^a to indicate the current value, in the same way that we use the topic variable (ref) $_ in loops.

Exercise 8.1

What is the result of this:

```
> (1 .. 10).map({ $^a + $^b });
```

See Chapter 10, *Procedures* and section 10.4, “Placeholder Variables” for a more detailed description of Placeholder Variables.

8.12.2. Block Code

We can pass any block of code as parameter. The topic variable holds the current value:

```
> @a.map( { .sqrt } );
```

Note that map is lazy, so the values are not evaluated until they are needed.

8.12.3. * (Whatever Star)

We can use a Whatever Star and skip the block if we just need a single method call:

```
> (1..10).map( *.sqrt );
(1 1.4142135623730951 1.7320508075688772 2 2.23606797749979)
```

We can have simple expressions as well:

```
> (1..10).map(* + 1);  # -> (2 3 4 5 6 7 8 9 10 11)
```

Note that curlies are illegal when we use a Whatever Star.

And we can negate the expression (so the Whatever Star doesn't have to be the first character in the expression):

```
> say (1 .. 25).grep(! *.is-prime);
```

`grep` is described in section 8.20.1, "grep". It selects the values that passes the condition.

8.12.4. deepmap

Use `deepmap` to apply the code to every element in the list, and not only on the top level as done by `map`.

They difference between `map` and `deepmap` can be summarised like this:

```
> my @a = ((1,2),(3,(4,5)));

> @a.map( * +1 );        # -> (3 3)
> @a.deepmap( * +1 );    # ->  [(2 3) (4 (5 6))]
```

`deepmap` traverse the structure recursively, adding one to every element.

`map` sees the first level only, and that is a first a list with two items (1 and 2) followed by another list also with two items (3 and a new list (with the items 4 and 5)). The `+1` coerces the value to Numeric, and the Numeric value of a list is the size. So we get 2 as it has 2 items. The next list also has two items (a value and a new list), so another 2. The `+1` gives the end result (3,3).

There are also `flatmap` (deprecated), `nodemap` and `duckmap`. See the «Advanced Raku» course for details.

8.13. sort

Use `sort` to sort a list. It is type aware, so it will sort numerically when given numbers, and as strings when given strings.

```
> (1, 2, 11, 0, 3, -1).sort;  # -> (-1 0 1 2 3 11)
```

If you mix numbers and strings you will get funny results:

```
> (1, 2, 11, 0, "3", -1).sort;  # -> (-1 0 1 2 11 3)
```

When we sort strings, it uses the character order in the unicode specification, which is the same order as Isolatin and ASCII for english letters (A-Z and a-z). So the upper case letters come before all the lower case ones.

If we have a list of words, some with an initial uppercase letter and some not, we will get the uppercase words before the lowercase ones.

We can tell `sort` how to sort by giving it a custom comparison code block:

```
@words.sort( { $^a.fc cmp $^b.fc } );
```

We apply `fc` (foldcase; see section 7.8.5, “fc (Fold Case)”) to convert all the strings to a caseless version before comparing them.

The inside of the block specifies how to compare any two elements when the compiler does the sorting; `$^a` is the first, and `$^b` is the second. We use these two placeholder variable names (as a single `*` Whatever Star wouldn’t work), and they can be used several times in the expression if required.

`cmp` (see section 3.7.1, “cmp”) is the three way comparison operator, and it does the job for us (or rather, for `sort`).

The curly braces are required when we specify the comparison code inline. It is possible to specify a reference to a procedure doing the job for us like this:

```
.sort( &compare-elements )
```

Do this if the computation has more code than the simple example above.

Note the `&` sigil. It tells the compiler to pass a *reference* to the code. If we skip it, the procedure is executed right away.

See Chapter 10, *Procedures* for an introduction to procedures.

There is a collation aware version of `sort` called `collate`. See the «Advanced Raku» course for details.

8.14. reverse

We can reverse a list with the reverse method:

```
> my @num1 = 1 .. 10;
[1 2 3 4 5 6 7 8 9 10]

> my @num2 = @num1.reverse
[10 9 8 7 6 5 4 3 2 1]
```

We have just used `reverse` to generate a downward counting Range, but this is really something we should use a Sequence to. We’ll discuss them in Chapter 16, *Ranges and Sequences*.

We could have sorted the list:

```
my @num3 = @num1.sort({$^b.fc <=> $^a.fc});
```

8.14.1. Swapping two variables

Swapping two variables in a «normal» programming language requires a temporary variable:

```
my $a = 1;
my $b = 2;

my $tmp = $a; $a = $b; $b = $tmp;
```

We can do it with a list assignment:

```
($a, $b) = ($b, $a);
```

Or with `reverse`:

```
($a, $b) .= reverse;
```

8.15. Array with Limits

We can specify a size limit for an array:

```
> my @d[10] = <rune helge tom jerry>;
[rune helge tom jerry]

> my @d[3] = <rune helge tom jerry>;
Index 3 for dimension 1 out of range (must be 0..2)
```

⚠ Note that the limit is the number of items, and not the index of the last one.

8.16. Typed Array

We can add a type constraint on the *values* of an array, as described in section 3.1, “Strong Typing”:

```
> my Int @values;
> my @values of Int;
```

Use `of` to get the type constraint:

```
> my Int @values; say @values.of;  # -> (Int)
> my @values; say @values.of;      # -> (Mu)
```

8.17. Shaped Array

A shaped array is an array with more than one dimension.

We access an individual cell in a shaped array like this;

```
> my @a; @a[1;2;3]   = 2;
> my @a; @a[1][2][3] = 2; # The same
```

We can access a "row" like this:

```
> @a[1;2;*];  # -> ((Any) (Any) (Any) 2)  ## List
> @a[1][2];   # -> [(Any) (Any) (Any) 2]  ## Array
```

Not the subtle difference in the types.

And the whole array:

```
> @a;  # -> [(Any) [(Any) (Any) [(Any) (Any) (Any) 2]]]
```

8.17.1. Sized Shaped Arrays

We can limit the size of shaped arrays as well:

```
> my @a[3;3;3];
> @a[3;3;3] = 12
Index 3 for dimension 3 out of range (must be 0..2)
  in block <unit> at <unknown file> line 1
```

Normal index rules apply, so the first item has index (offset) 0.

We can assign a shaped array with a list of lists:

```
>  my @a[3;3] = ((1,2,3), (4,5,6), (7,8,9))
[[1 2 3] [4 5 6] [7 8 9]]
```

8.17.2. shape

Returns the shape of the array as a list. Note that this works when we have given the array an explicit shape only.

```
my @foo[2;3] = ( < 1 2 3 >, < 4 5 6 > ); # Array with fixed dimensions
say @foo.shape;                          # -> (2 3)
my @bar = ( < 1 2 3 >, < 4 5 6 > );      # Normal array (of arrays)
say @bar.shape;                          # -> (*)
```

8.17.3. Shaped Arrays Usage

A Shaped Array can be used like a matrix. There are no built-in operators working with matrices, so a module like «Math::Matrix» is a safer bet.

8.18. unique (Lists Without Duplicates)

Use `unique` to get a copy of the list, without duplicates:

```
> (1,1,2,3,4,5,1,6).unique
(1 2 3 4 5 6)
```

If you know that the list is sorted, use `squish` instead of `unique`:

```
> (1,1,2,3,4,5,5,6).squish # OK
(1 2 3 4 5 6)

> (1,1,2,3,4,5,1,6).squish # Wrong usage.
(1 2 3 4 5 1 6)
```

8.18.1. repeated

The `repeated` method does the opposite of `unique` as it only returns duplicates:

```
> (1, 2, 1, 2, 3,4,5,6,1).repeated;
(1 2 1)
```

They will occur once for each time they are duplicated, and the list is not sorted. We can fix duplicates and sort it as well:

```
> (1, 2, 1, 2, 3,4,5,6,1).repeated.sort.squish;
(1 2)
```

8.19. xx (List Repetition Operator)

Use `xx` to repeat the list or value on the left hand side the number of times given on the right hand side:

```
> "abc" xx 3
(abc abc abc)

> "abc " xx 2
(abc abc)
>
```

The number must be an integer, or something that can be coerced to an integer. Zero or negative integers will return an empty string:

```
> "abc" xx 0;  # -> ()
> my $a = (True, False) xx 3;  # -> ((True False) (True False) (True False))
> my $b = |(True, False) xx 3; # -> (True False True False True False)
```

We can generate an infinite list by specifying a `*` (a «Whatever Star») on the right hand side:

```
> my $c = |(True, False) xx *
```

See section 16.3.6, "List Repetition Operator and Sequences" for a use case.

Note the similarity to the Sting Repetition Operator `x` (as described in section 7.9, "x (String Repetition Operator)"

8.20. List Selection

Use `map` to apply changes to all the values, and `grep` to select some of the values.

`grep` returns a list. If you only need the first value, use `first` instead. (See section 8.20.2, "first".)

If you are only interested in the fact that there is at least one match, `any` (which is **not** the same as `Any`, described in section 3.4.1, "Nil & Any") may be useful). See the «Advanced Raku» course for details.

8.20.1. grep

Use `grep` to select some values from a list.

Integers not divisible by 3:

```
> say (1 .. 25).grep(* % 3);
(1 2 4 5 7 8 10 11 13 14 16 17 19 20 22 23 25)
```

Non-prime numbers only:

```
> say (1 .. 25).grep(! *.is-prime);
(1 4 6 8 9 10 12 14 15 16 18 20 21 22 24 25)
```

Integers only:

```
> say (1, 1.5, 2, 2.5, 3, 3.5, 4).grep(Int);
(1 2 3 4)
```

Exercise 8.2

Get all the two digit primes, and count them.

Tip: Use `grep` and `is-prime` (see section 5.12, “is-prime (Prime Numbers)”).

8.20.2. first

The `first` method (and function) is like `grep`, except it only returns the first match. Use this if you only require a single match, as it will be faster.

The first prime number after (and including) 1000:

```
> say (1000 .. Inf).first(*.is-prime);
1009
```

It returns `Nil` if no matches were found:

```
> say (0.1 .. 0.9).first(*.is-prime);
Nil
```

There is no separate «last» method, but use the optional named parameter `:end` to indicate that the search should be from the end of the list, rather than from the start.

The highest prime number lower than 1000:

```
> say (1 ..^ 1000).first(:end, *.is-prime);
997
```

8.20.3. head

Returns the specified number of items from the *beginning* of the list. It defaults to 1 if the size isn’t specified:

```
> (1 .. Inf).head
1

> (-1 .. Inf).head(2)
(-1 0)

> <a b c d 12>.head
a
```

8.20.4. tail

Returns the specified number of items from the *end* of the list. It defaults to 1 if the size isn't specified:

```
> <a b c d 12>.tail
12
```

Don't use `tail` on an infinite list:

```
> (-Inf .. Inf).tail
Cannot tail a lazy list
  in block <unit> at <unknown file> line 1
```

But `head` works (surprisingly, I would say):

```
> (-Inf .. Inf).head
-Inf
```

8.21. min / max

`min` returns the smallest and `max` the largest value in the list.

With numbers:

```
> (1 .. 10).min;  # -> 1
> max 1 .. 10;    # -> 10
> (1 .. *).max;   # -> Inf
```

With strings:

```
> <aa a abc d f ff e>.min;  # -> a
> <aa a abc d f ff e>.max;  # -> ff
```

They can also be used as infix operators, and can be stacked:

```
> say 8 min 10;                          # -> 8
> say 8 min 10 min 2 min 99 min -19;  # -> -19
```

Undefined values are ignored, so the following works:

```
> say (5, Nil, 100, 2).min; # -> 2
```

8.22. Random Values

Random values are important, and they are extremely difficult to implement. Most programming languages settle for pseudo-random numbers, something that looks random but really isn't. Raku is no exception. This is good enough, unless you need real randomness for cryptography.

8.22.1. rand

`rand` as a function gives a pseudo-random number (of type `Num`) between zero (inclusive) and 1 (non-inclusive):

```
> rand;     # -> 0.4214056307236411
> rand;     # -> 0.7753853239550014
```

When used as a method on a value, it returns a pseudo-random number (of type `Num`) between zero (inclusive) and the given value (non-inclusive):

```
> 100.rand; # -> 51.322528184845
```

Exercise 8.3

Write code that chooses a random integer between 10 and 99, both included.

8.22.2. pick

Random values are often used as indexes to lists. Raku has a routine `pick` that can be used on a list (or a range) to get a random element from it:

```
> @colours.pick;   # From an array
> (10 .. 99).pick; # From a range
```

If the value is meant as an array index, use `pick` on the array directly:

```
> ("red", "blue", "green", "yellow").pick
```

We can ask for more values at the same time:

```
> (10 .. 99).pick(10);
```

This gives 10 randomly selected values from the range, **without repetition**.

If you want more than one value, just ask:

```
> ("red", "blue", "green", "yellow").pick(2);
(red yellow)
```

This `pick` call will not repeat the values.

If you ask for more values than it can give, it will give as much as it can (without complaining):

```
> ("red", "blue", "green", "yellow").pick(9);
(green yellow red blue)
```

If you want them all, use this syntax to make it obvious:

```
> <red blue green yellow>pick(*);
(green yellow blue red)
```

This is a handy way of sorting a list in a random order.

Exercise 8.4

Write a piece of code (in REPL) that returns a random prime number between 1 and a specified number.

Use 100 and 1000 as the limit.

pick With Repetition

Repetition is possible, by applying a loop:

```
> say (1 .. 6).pick for ^3
5
2
5
```

This can be used to roll a dice a number of times.

A little trickery to get them on one line:

```
> (1 .. 6).pick.fmt("%d ").print for ^10; say "";
1 5 3 6 2 1 5 6 2 1

> say join(" ", ( (1 .. 6).pick for ^10) );
5 4 3 6 4 4 4 5 6 2
```

Or we could have used the List Repetition Operator `xx` (see section 8.19, "xx (List Repetition Operator)"):

```
> say (1 .. 6).pick xx 10
(6 5 5 4 4 6 6 2 4 2)

> say (1 .. 6).pick xx 10
(6 3 3 6 6 5 5 4 5 2)
```

8.22.3. roll

`roll` is meant to remind you of the roll of a dice. It behaves the same way as `pick`, except that it can repeat the values:

```
> (1..6).roll(3);  # -> (1 4 6)
> (1..6).roll(3);  # -> (1 2 6)
> (1..6).roll(3);  # -> (3 3 6)
```

The count defaults to 1:

```
> (1..6).roll;  # -> 6
```

We can get an infinite lazy sequence if we pass `*` as count:

```
> (1..6).roll(*);  # -> (3 3 6 ...)
```

We can use it on Boolean values, either on a list or the type object:

```
> (True, False).roll;  # -> True
> Bool.roll;           # -> True
```

We can cheat as well:

```
> (True, True, False, False, False,).roll;  # -> True
```

Exercise 8.5

Generate a random string of ten characters usable as a password. Use letters, digits, and some special characters (as e.g. «!» and «@»).

Exercise 8.6

We can check the quality of the random number generator, that the distribution is even.

Write a program that picks 1 million random numbers in the range 1..100, and prints a frequency table.

Also display the minimum and maximum count.

8.22.4. srand

Random numbers are not really random. Raku has a sequence of pseudo random numbers, and the randomness lies in the fact that the compiler starts at a different location in this sequence each time it runs a program, by calling `srand`.

We can mess up this by calling `srand` ourself:

```
> srand(1234567890); say rand; say rand;
0.9168008342654074
0.297372052451493

> srand(1234567890); say rand; say rand;
0.9168008342654074
0.297372052451493
```

Reset again, and it starts at the same place.

Note that the actual values may differ, depending on the Operating System and version of the compiler, but you will get the same sequence each time.

Call `srand` with another integer value, and you'll get another sequence.

```
> srand(112); (1 .. 6).pick.fmt("%d ").print for ^10; say "";
6 1 4 6 6 4 5 4 6 3
> srand(112); (1 .. 6).pick.fmt("%d ").print for ^10; say "";
6 1 4 6 6 4 5 4 6 3
```

`srand` affects `pick` as well:

```
> srand(1); (1..10).pick;  # -> 4
> srand(1); (1..10).pick;  # -> 4
> srand(1); (1..10).pick;  # -> 4
```

8.23. permutations

Use `permutations` to get all possible permutations of a list:

```
> say <a b c>.permutations;
((a b c) (a c b) (b a c) (b c a) (c a b) (c b a))
```

It cares about the *positions*, not the actual values. So duplicate values will result in duplicates in the permutations:

```
> .say for <a b b>.permutations;
(a b b)
(a b b)
(b a b)
(b b a)
(b a b)
(b b a)
```

If used as a function with a numeric value, it will treat that value as a range. E.g. `permutations 3` is the same as `permutations ^3` (which is the same as `permutations 0..2`)

```
> say permutations 3;
((0 1 2) (0 2 1) (1 0 2) (1 2 0) (2 0 1) (2 1 0))
```

Exercise 8.7

Write a program that removes duplicates from the output of `permutations`.

```
> say <a b b>.permutations-without-duplicates; # Doesn't exist.
((a b b) (b a b) (b b a))
```

8.24. combinations

Use `combinations` to get all the possible combinations of zero and more elements from the list:

```
> say <a b c>.combinations;
(() (a) (b) (c) (a b) (a c) (b c) (a b c))
```

Duplicate values in the input list is allowed, and will result in duplicates in the result:

```
> <a b b>.combinations
(() (a) (b) (b) (a b) (a b) (b b) (a b b))
```

This is because `combinations` uses the *positions*, and not the actual values.

We can specify that we only want the combinations with a given number of elements:

```
> <a b c>.combinations(1)
((a) (b) (c))
```

We can specify a range to select the length:

```
 <a b c>.combinations(1..3)
((a) (b) (c) (a b) (a c) (b c) (a b c))
```

As a function we give it the number of items as argument. E.g. `3` will result in the list `0..2`:

```
> combinations 3
(() (0) (1) (2) (0 1) (0 2) (1 2) (0 1 2))
```

We can add the number of elements to select as well, either as a value or a range:

```
> combinations 3,1;    # -> ((0) (1) (2))
> combinations 3,2;    # -> ((0 1) (0 2) (1 2))
> combinations 3,1..2; # -> ((0) (1) (2) (0 1) (0 2) (1 2))
```

8.25. but (Array)

We introduced the `but` operator in section 3.8, “but (True and False, but …)”, and showed how it works on scalar values.

It *does not* work on arrays (`@`):

```
> my @a = <54 12> but False;  # -> [54 12]
> say @a.WHAT;                # -> (Array)
```

But (pun intended) it *does work* if we assign to a scalar:

```
> my $a = <54 12> but False;  # -> (54 12)
> say $a.WHAT;                # -> (List+{<anon|5>})
> say $a[0].WHAT:             # -> (IntStr)
> say $a[1].WHAT;             # -> (IntStr)
> say so $a;                  # -> False

> my $b = [54, 12] does False;  # -> [54 12]
> say $a.WHAT;                  # -> (Array+{<anon|4>})
```

It works on the entire list, on not on the individual elements.

8.25.1. does (Array)

This applies to the `does` operator as well.

Se section 3.8.1, “does” for an introduction.

Chapter 9. Pair and Hashes

Hashes consist of pairs of keys and values.

Before looking at hashes, we'll take a look at the `Pair` type.

9.1. Pair

Hashes (as well as some other types which we will look into in the «Advanced Raku» course) consist of pairs of keys and values. The `Pair` is a built in type, and it is a combination of a single key and value. They can operate alone (a single `Pair`), but are generally more useful in larger numbers.

9.1.1. => (Pair Constructor)

We can construct a `Pair` with the `Pair` constructor `=>`. We can use `Pair`, but don't have to:

```
> Pair(1 => 2).WHAT
(Pair)

> (1 => 2).WHAT # The same, but shorter
(Pair)

> Pair.new(1, 2).WHAT # Without the "fat arrow"
(Pair)
```

⚠ Note that unquoted text is allowed as a key, as long as it doesn't contain spaces:

```
> my $a = (pi => pi);
pi => 3.141592653589793
```

The first «pi» is taken as the a literal text, and the second one is taken as the built in `pi` constant.

We can construct a `Pair` in several ways:

`Pair('key' => 'value')`	As shown above
`('key' => 'value')`	ditto
`Pair.new('key', 'value')`	ditto. This is the canonical way
`'key' => 'value'`	No need for parens
`:key<value>`	The same
`:foo(127)`	Short for `foo => 127`
`:127foo`	The same as `foo => 127`. Works when the value is a number.

If the value is a Boolean value we can shorten the expression:

`:key`	The same as `key => True`
`:!key`	The same as `key => False`

We can turn any variable into a `Pair`, with the variable name as the key:

```
> my $age = 14;
> my $p   = :$age;
> say $p; # -> age => 10
```

9.1.2. key

Use `key` to get the key value (the left side of the fat arrow) of a `Pair`:

```
> my $a = (1 => 2);  # -> 1 => 2
> $a.key;            # -> 1
```

9.1.3. value

Use `value` to get the value (the right side of the fat arrow) of a `Pair`:

```
> $a.value;  # -> 2
```

9.1.4. antipair

Use `antipair` to swap the key and value of a `Pair`:

```
> ("a" => "r").antipair;  # -> r => a
```

9.2. Hash

A hash is a collection if `Pair` objects (zero, one or more), where we use the key as index (lookup).

We can give a hash values when we declare it like this:

```
> my %trans = ("a" => "1", "b" => "9");

> my %population = (Oslo           => 500_000,
                    Paris          => "unknown",
                    "Buenos Aires" => "too many");
```

The keys (the left hand side of `=>`) can be specified without quotes - if they do not contain spaces.

We can skip the parens.

We can populate a hash with a list. It will take the first value as key, the second as the value and so on:

```
> my %a = (11 .. 20)
{11 => 12, 13 => 14, 15 => 16, 17 => 18, 19 => 20}
```

The number of items in the list must be even:

```
{11 => 12, 13 => 14, 15 => 16, 17 => 18, 19 => 20}
> my %a = (11 .. 21)
Odd number of elements found where hash initializer expected:
Found 11 (implicit) elements: ...
```

It is possible to create a hash from two separate lists, one for the keys and the other for the values:

```
> my @keys =  1..10;
> my @vals = 91..100;

> my %hash; %hash{@keys} = @vals;

> say %hash;
{1 => 91, 10 => 100, 2 => 92, 3 => 93, 4 => 94, 5 => 95, 6 => 96, 7 => 97, 8 => 98, 9
=> 99}
```

9.3. Hash Constructor { }

The hash constructor `{ … }` is only required if we assign the hash to a scalar:

```
> my $trans = {"a" => "1", "b" => "9"};
{a => 1, b => 9}
```

Note that if we forget the curlies, we get a list of Pairs:

```
> my $trans = ("a" => "1", "b" => "9");
(a => 1, b => 9)
```

If we assign something to a hash, it is coerced to Pair objects, and then inserted in the hash:

```
> my %hash = 1..10;  # -> {1 => 2, 3 => 4, 5 => 6, 7 => 8, 9 => 10}
```

As long as the number of arguments are even:

```
>  my %hash = 1..11;
Odd number of elements found where hash initializer expected:
Found 11 (implicit) elements:
Last element seen: 11
  in block <unit> at <unknown file> line 1
```

We can use other ways of generating a `Pair`:

```
my %months = :jan('January'), :feb('February'), ...;
```

9.4. Hash Assignment and Values

If we assign to the hash variable, any existing values will be lost. We can add new values, or change the value of an existing one like this:

```
> %population{"Buenos Aires"}  = "too many";
> %population<Oslo> = 500_000;
```

```
> %population{"Oslo"}
500000

> %population<Oslo> # The same
500000

> say %population<Buenos Aires> # An error
((Any) (Any))
```

The last one is the same as:

```
> say ( %population{"Buenos"}, %population{"Aires"} );
((Any) (Any))
```

9.5. keys

The `keys` method gives (a list of) all the keys:

```
for %population.keys -> $city
{
  say "City $city has %population{$_} people";
}
```

We can use keys one a list as well, and it will return the indices:

Even if we have populated the list with Pair objects:

```
(1 => 2, 2 => 3, 4 => 5).keys
(0 1 2)
```

keys gives the keys in random order. If you want order, sort them:

```
> for %population.keys.sort -> $city { ... }
```

The order the keys are returned in is semi random. As long as the hash hasn't been changed, this order remains the same (and that means that calling keys and then values gives the values in the same order.

9.6. values

The values method gives all the values:

```
for %population.values -> $population
{
  say "Unknown City with %population{$_} people";
}
```

Note that values on a list of Pair objects returns everything, as it uses the indices as keys:

```
> (1 => 2, 2 => 3, 4 => 5).values
(1 => 2 2 => 3 4 => 5)
```

There is no way to start with a hash value and get back to the key, except doing a manual search. This is the classical «needle in the haystack problem». We'll have a go at it in section 10.13.4.2, "The Needle in the Haystack Problem", when we have learned about procedures.

9.7. kv (keys + values)

We can use the kv method (for key-value) to get both keys and values at the same time:

```
for %population.kv -> $city, $population
{
  say "City $city has $population people";
}
```

9.8. Typed Hash

We can add a type constraint on the *values* of a hash, as described in section 3.1, “Strong Typing”:

```
> my Int %h;
> say %h.WHAT;  # -> (Hash[Int])
> %h<a> = 12.1;
Type check failed in assignment to %h; expected Int but got Rat (12.1)
```

The constraint can also be specified like this:

```
> my %h of Int;
```

We can also add type constraint on the *keys* of a hash:

```
> my %h{Str};
> say %h.WHAT;  # -> (Hash[Any,Str])
```

We can use both:

```
> my Int %h{Str};
> say %h.WHAT;  # -> (Hash[Int,Str]) # The first is the keys, the second is the values
```

Now we require that the keys are strings, and the values are integers.

9.8.1. keyof

`keyof` returns the type constraint for the *keys* of the invocant.

A hash without a restraint on the keys:

```
> my %h; say %h.keyof;  # -> (Str(Any))
```

A hash with a restraint on the keys:

```
> my %h{Int}; say %h.keyof;  # -> (Int)
```

9.8.2. of

`of` returns the type constraint for the *values* of the invocant.

A hash without a restraint on the values:

```
> my %h; say %h.of;  # -> (Mu)
```

A hash with a restraint on the values:

```
>  my %h of Str; say %h.of;  # -> (Str)
```

9.9. Shaped Hash

A shaped Hash doesn't make as much sense as a shaped array (as described in section 8.17, "Shaped Array").

We can specify the shape like this:

```
> my %hash{10;10;10};
```

The numbers are ignored, but we cannot use more indices than specified (three in this case):

```
> %hash{"A";"E";"C"}     = 1;
> %hash{"A";"E";"C"}     = "A"
> %hash{"A";"E"}         = "A"
> %hash{"A";"E";"J";"O"} = "A"
Type Str does not support associative indexing.
```

The error message is a bit confusing though.

Also note that we can assign any value to the hash, and not only integers (as if we had only specified one integer; e.g. `my %hash{10};` - and yes specifying an integer is taken as the same as `Int`).

The size declaration is pretty useless (except for the upper limit), and can be dropped:

```
> my %hash;
> %hash{"A"; "B"; "C"} = 12;
> %hash{"A"; "B"; "D"} = 13;
```

So what *actually is* a shaped hash? Let us have a look:

```
> say %hash;  # -> {A => {B => {C => 12, D => 13}}}
```

It is a tree-like structure, a hash of hashes of hashes. (I have kept it simple on purpose.)

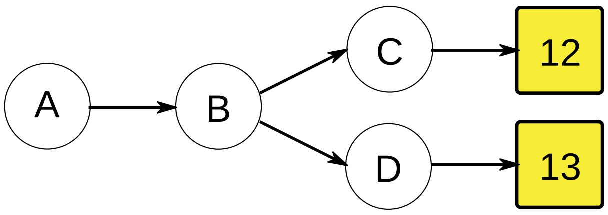


Figure 10. Multihash

We can specify it like this as well:

```
> my %hash;
> %hash{"A"}{"B"}{"C"} = 12;
> %hash{"A"}{"B"}{"D"} = 13;
> say %hash;  # -> {A => {B => {C => 12, D => 13}}}
```

Note the subtle difference when we address a subtree:

```
> %hash{"A"; "B"};   # -> ({C => 12, D => 13})  # A list
> %hash{"A"}{"B"};   # -> {C => 12, D => 13}    # A hash
```

9.10. invert

We can use `invert` to swap the keys and values of a hash:

```
> my %a = (1 => 2, 3 => 4); # -> {1 => 2, 3 => 4}
> my %b = %a.invert;        # -> {2 => 1, 4 => 3}
```

Duplicates values turn into identical keys, but that is ok:

```
> my %a = (1 => 2, 3 => 4, 5 => 2);  # -> {1 => 2, 3 => 4, 5 => 2}
> say %a.invert;                     # -> (2 => 1 2 => 5 4 => 3)
```

We got a list of Pair objects, and the list doesn’t care about duplicates.

Duplicate keys will be squished, when we assign it back to a hash, like this:

```
> my %b = %a.invert;                 # -> {2 => 5, 4 => 3}
```

When we assign a list of Pair objects to a hash, the last one wins when we have duplicate keys. *Which* is the last one is anybody’s guess, as a hash is unordered. (That means that you could get different results each time you run the program, thus getting subtle errors.)

9.11. antipairs

This has the same effect as `invert` on a hash.

But on a list of `Pair` objects they differ:

```
> (1 => 2, 2 => 3, 4 => 5).antipairs
((1 => 2) => 0 (2 => 3) => 1 (4 => 5) => 2)

> (1 => 2, 2 => 3, 4 => 5).invert
(2 => 1 3 => 2 5 => 4)
```

`antipairs` use the list index as key, and the `Pair` as value, and swaps them.

9.11.1. pairs

Use `pairs` to convert the hash to a list of `Pair` objects:

```
> my %a = (1=>2, 2=>3);  # -> {1 => 2, 2 => 3}
> %a.pairs;              # -> (1 => 2 2 => 3)
```

9.12. Hash Slices

Just as we have array slices (see section 8.10, “Array Slice”), we have hash slices:

```
> my %translate = ( one => "ein", two => "zwei", \
  three => "drei" );
{one => ein, three => drei, two => zwei}

> say %translate{"two", "one"}
(zwei ein)

> say %translate<two one> # This does not work on arrays.
(zwei ein)
```

Use `grep` if the selection criteria is more complex:

```
> %translate{%translate.keys.grep(*.chars == 3)}
(zwei ein)
```

9.13. Hash Lookup

How do we check if a value is present in a hash?

```
> my %h;
> %h<a> = 0;     # => 0
> %h<b> = False; # => False
> %h<c> = Nil;   # => (Any)
```

All of them evaluates to `False` in Boolean context.

Use the `:exists` adverb:

```
> %h<a>:exists # => True
> %h<b>:exists # => True
> %h<c>:exists # => True
> %h<d>:exists # => False
```

9.14. Hash Deletion

Use the `:delete` adverb to delete entries from a hash:

```
> my %h = a => 1, b => 2, c => 3
{a => 1, b => 2, c => 3}

> %h<b c>:delete
(2 3)

> %h<a>:delete
1
```

The deleted value or values are returned.

9.15. Hash Duplicate Values

Hashes (obviously) do not allow duplicate values with the same key:

```
> my %hash;
> %hash<M> = 12;
> %hash<M> = "nobody";
> say %hash<M>
nobody
```

But we can get around that by using a list as the value, adding new values to it. We can do this automatically with `push`:

```
> my %hash;
> %hash<M>.push(12);
> say %hash<M>
[12]

> %hash<M>.push("nobody");
> say %hash<M>
[12 nobody]
> say %hash<M>[1];
nobody
```

But we must do this from the start, as the first push will remove any scalar value already there:

```
> my %hash;
> %hash<M> = 12;
> %hash<M>.push("nobody");
> say %hash<M>
[nobody]
```

9.16. Hash Usage

The first code snippet we showed in this chapter is a hash where we have a mapping between cities and the size of their population.

We can try to add those values:

```
my $total = 0;
for %population.values -> $population
{
  $total += $population;
}
```

This gives a run time error because of the string values ("unknown" and "too many"):

```
Cannot convert string to number: base-10 number must begin
with valid digits or '.' in '  too many' (indicated by   )
```

9.16.1. The «sum» Method

We can use the `sum` method instead of looping through the values:

File: population-sum (partial)

```
my $total = %population.values.sum;
```

And again, this fails because of the non-numeric values.

9.16.2. With Smartmatch

We can check that the value is an `Int` with the Smartmatch Operator `~~` (which is introduced in section 11.3, "~~ (Smartmatch Operator)"), before adding it:

File: population-smartmatch (partial)

```
for %population.values -> $population
{
  $total += $population if $population ~~ Int;
}
```

We could have used `$population.WHAT === Int` or `$population.isa(INT)` (see sections 3.7.7, "===" and 3.7.8, "isa", with the same result.

All of them works as long as the **type** is an `Int`. If it is e.g. `Rat`, the comparison will fail - even if the **value** itself is an integer:

```
> say 5.0 ~~ Int;  # -> False
```

9.17. Grep and Smartmatch

We can use `grep` to get rid of illegal values (and the loop), before applying sum:

```
my $total = %population.values.grep(* ~~ Int).sum;
```

We'll get back to this example in the «Advanced Raku» course.

9.18. Hash (method)

We can turn a list into a hash with the `Hash` method, as long as the number of elements are even:

```
> my %hash = (1..10).Hash;
{1 => 2, 3 => 4, 5 => 6, 7 => 8, 9 => 10}
```

Exercise 9.1

What happens if we drop the `.Hash` part?

E.g.

```
> my %hash = (1..10);
```

Exercise 9.2

What happens if we have duplicate values?

E.g.

```
<1 2 1 3 1 4 1 5>.Hash;
```

9.19. but (Hash)

In section 8.25, “but (Array)” we showed that the `but` operator *does not* work on arrays (`@`), but on scalar lists/arrays (`$`).

The same applies to hashes, which doesn’t work:

```
> my %a = (54 => 12) does False;  # -> {54 => 12}
> say %a.WHAT;                    # -> (Hash)
> say so %a:                      # -> True
```

This works, but only on the whole data structure (which is a collection of Pair objects):

```
> my $a = (54 => 12) does False;  # -> 54 => 12
> say $a.WHAT;                    # -> (Pair+{<anon|5>})
> say so $a;                      # -> False
> say $a<54>;                     # -> 12
> say so $a<54>;                  # -> True
```

9.19.1. does (Hash)

This behaves the same as `but`. See section 8.25, “but (Array)” for an details.

Chapter 10. Procedures

Procedures is the traditional basic building block in writing maintainable code, breaking large programs into smaller units. (Object Orientation is another approach, and we'll get back to that in chapter Chapter 17, *Classes*.)

10.1. Procedures Without Arguments

Define a parameterless procedure like this:

```
sub hello
{
  say "Hello";
}

hello;   # Call it like this,
hello(); #  or this.
```

10.2. Procedures With Arguments

With an explicit parameter list:

```
sub add ($first-value, $second-value)
{
  return $first-value + $second-value;
}

say add(1, 2); # Call it like this,
say add 1, 2;  #  or this.
```

Beware of precedence, and use parens to avoid confusion:

```
say add(1, 2), 3;  # -> 33
say add 1, 2, 3;
===SORRY!=== Error while compiling:
Calling add(Int, Int, Int) will never work with declared signature ($first-value,
$second-value)
```

What if we try with a text string?

```
> my $result = add "10", 2;  # -> 12
```

The string "10" is coerced to a number (10), and it works.

It works *because "10" can be converted to a number*.

But if we try something that cannot be converted, we get a run time error:

```
> my $result = add "ten", 2;
Cannot convert string to number: base-10 number must begin
with valid digits or '.' in '  ten' (indicated by   )
  in sub add at <unknown file> line 1
```

10.3. @_

We can use procedures without signatures. Any arguments passed is available in the `@_` variable:

```
> sub test { .say for @_; }
> test
Nil
> test 1, 2, 3
1
2
3
```

`@_` is not available if we have a procedure signature:

```
> sub test ($arg) { .say for @_; }
Placeholder variable '@_' cannot override existing signature
------> sub   test ($a1) { .say for @_; }
```

`@_` flattens lists, but not hashes. Be careful, or even better: don't use it.

10.3.1. $_

We can use `$_` as a procedure variable:

```
> sub x ($_) { .say; .say; }
> x(12)
12
12
```

10.4. Placeholder Variables

We introduced Placeholder Variables in section 8.12.1, "Placeholder Variables". They can be used with any procedure:

File: placeholder

```
sub test
{
  say "Argument 1: $^a"; # -> Argument 1: A
  say "Argument 2: $^b"; # -> Argument 2: B
}

test("A", "B");
```

Placeholder variables pop into existence when we use them. They are assigned to the values in *alphabetical order*, and not the order in which they are first used.

File: placeholder2

```
sub test
{
  say "Argument 2: $^b"; # -> Argument 2: B
  say "Argument 1: $^a"; # -> Argument 1: A
}

test("A", "B");
```

10.4.1. Named Placeholder Variables

We can have Named Placeholder Variables. Specify a colon between the sigil and the name:

File: placeholder-named

```
sub test
{
  say "Argument 1: $:first";   # -> Argument 1: 12
  say "Argument 2: $:second";  # -> Argument 2: 23
}

test(first => 12, second => 23);
```

10.5. Procedures as variables

We can store a reference to a procedure in a variable, and execute it later on:

```
> my &code = sub { say "12345"; }
sub { }

> &code();
12345
```

Or in a scalar (but don’t):

```
> my $code = sub { say "12345"; }
sub { }

> $code()
12345
```

10.5.1. anon

The anonymous procedures above works fine as they don’t have any arguments. If we were to introduce arguments, we would have to name them:

```
> my &code = sub something($arg) { say $arg ~ "123"; }
```

That gives us a name, that we can use to call the procedure with (in the normal way). We can prevent that by using `anon sub`:

```
> my &code = anon sub something($arg) { say $arg ~ "123"; }
> something;
===SORRY!=== Error while compiling:
Undeclared routine: something used at line 1
```

Note that if you drop the assignment, you’ll get an uncallable anonymous function.

10.6. Type Constraints

We can use a type constraint to prevent automatic conversion of strings to numbers (and get a compile time error instead):

File: num-add-err

```
sub add (Numeric $first-value, Numeric $second-value)
{
  return $first-value + $second-value;
}
say add "10", 2;
```

```
$ raku num-add-err
==SORRY!=== Error while compiling ./num-add-err
Calling add(Str, Int) will never work with declared signature (Numeric $first-value,
  Numeric $second-value) at ./num-add-err:7
------> say   add "10", 2;
```

Note that `Numeric` also allows an undefined value:

File: num-add-err2

```
sub add (Numeric $first-value, Numeric $second-value)
{
  return $first-value + $second-value;
}

my Numeric $a; my Numeric $b;

say add $a, $b;
```

```
$ raku num-add-err2
Use of uninitialized value of type Numeric in numeric context
  in sub add at ./num-add-err2 line 3
Use of uninitialized value of type Numeric in numeric context
  in sub add at ./num-add-err2 line 3
0
```

The solution is adding the `:D` adverb (see section 3.5, “:D (Defined Adverb)”):

File: num-add-err3

```
sub add (Numeric:D $first-value, Numeric:D $second-value)
{
  return $first-value + $second-value;
}

my Numeric $a; my Numeric $b;

say add $a, $b;
```

```
$ raku num-add-err3
Parameter '$first-value' of routine 'add' must be an object instance of type
'Numeric', not a type object of type 'Numeric'.  Did you forget a '.new'?
  in sub add at ./num-add3 line 3
  in block <unit> at ./num-add3 line 10
```

Type Constraints give better error messages, and the check is done at compile time, before we have run any code (so we can avoid a program crash in the middle of something that can leave a mess in e.g. the file system or a database).

10.7. return

Use `return` to stop execution of the current procedure or method and give the specified value (if any, `Nil` otherwise) to the caller (as the return value).

Note that if we have set up relevant Phasers (see the «Advanced Raku» course), They will be run before control is returned to the caller.

If we have specified a return value constraint (see section 10.7.2, "Return Value Constraints"), the value will be checked against it. If the check fails, an exception is thrown.

⚠ Note that `return` is implemented as a procedure and not a keyword, so procedure precedence rules applies.

10.7.1. return-rw

`return` gives back a value, not a container, and you cannot change the value.

This will fail:

```
> sub abc { return 123; }
> say ++abc();
```

But we can use `return-rw` (as in «return read write») to get a container that we can change.

This fails as well:

```
> sub abc { return-rw 123; }
> say ++abc(); # -> 124
```

It fails as `return-rw` tries to return the container, and as there isn't one it returns the value `123` instead. That is perfectly legal, but the prefix `++` will fail.

File: return-rw

```
sub abc
{
  my $a = 123;
  return-rw $a;
}

say ++abc(); # -> 124
```

```
$ raku return-rw
124
```

This allows the use of procedure calls as anonymous variables. It may not be very useful in practice.

10.7.2. Return Value Constraints

We can have type constraints on the return value, as well as on the input values as shown in section 10.6, “Type Constraints”.

We can specify the return type constraints in several ways:

```
sub X ($a, $b --> Int)      { $a + $b }
sub X ($a, $b ) returns Int { $a + $b }
sub X ($a, $b ) of Int      { $a + $b }
my Int sub X ($a, $b)       { $a + $b }
```

We get an Exception if we try to return a value that doesn’t fit the restriction.

It is possible to return an explicit value:

```
> sub random( --> 12 ) { rand }
> say random;  # -> 12
```

We can return `Nil` as an error (or «I give in») value, even with a return type constraint:

```
> sub abc ( --> Int) { Nil; }
> abc;  # -> Nil
```

Even if we insist on a defined return value:

```
> sub abc ( --> Int:D) { Nil; }
> abc;  # -> Nil
```

10.8. @*ARGS

We can use the dynamic variable `@*ARGS` to get input from the command line:

File: hello-args

```
say "Hello, @*ARGS[0]!";
```

`@*ARGS` is a list with the arguments, with `@*ARGS[0]` as the first one and so on.

```
> raku hello-args NPW
Hello, NPW!
```

A placeholder variable would have been nice here, but that doesn’t work as we do not have a procedure.

10.9. MAIN

We can use the special `MAIN` procedure and declare procedure arguments, instead of accessing `@*ARGS`:

File: hello

```
sub MAIN ($name)
{
  say "Hello, $name!";
}
```

The compiler will execute any code in the program first, and call the `MAIN` routine afterwards. It is usually not a good idea having any code outside `MAIN`.

Declare `MAIN` with as many arguments as you want, with the names you want. The program will fail with a usage message if you give the wrong number (or types) of arguments to the program:

```
$ raku hello
Usage:
  hello <name>

$ raku hello all
hello, all!

$ raku hello all you
Usage:
  hello <name>
```

10.9.1. unit procedure

We can use `unit procedure` instead, saving us for a block level:

File: hello-unit

```
unit sub MAIN ($name);

say "Hello, $name!";
```

This is useful when we have one procedure only, as we usually do short programs with `MAIN`.

On the other hand, what we gain is a reduction in the number of block levels, and that is usually not an issue in short programs.

10.9.2. A better usage message

The name of the program and the variable name(s) may not say it all. Add a special comment line just above the `MAIN` procedure(s):

File: hello-usage

```
#| Person to greet
sub MAIN ($name)
{
  say "Hello, $name!";
}
```

```
$ raku hello-usage all you
Usage:
 hello-usage <name> -- Person to greet
```

The special comment can be specified *after* the `MAIN` procedure as well:

File: hello-usage2

```
sub MAIN ($name)
{
  say "Hello, $name!";
}
#= Person to greet
```

--doc

If you use the `#|` form, the comment will show up when we ask the compiler to give us the documentation:

```
$ raku --doc hello-usage
sub MAIN(
    $name,
)
Person to greet
```

Doing it on the second form (`#=` *after* the procedure) doesn't work:

```
$ raku --doc hello-usage2
```

10.10. WHY

We can add comments like this for ordinary procedures as well, and use `WHY` to get them:

File: usage

```
#| This is one
sub a1 { ; }

#| This is two
sub a2 { ; }
#= Still two

sub a3 { ; }

say &a1.WHY;
say &a2.WHY;
say &a3.WHY;
```

```
$ raku usage
This is one
This is two
Still two
No documentation available for type 'Sub'.
Perhaps it can be found at https://docs.raku.org/type/Sub
```

Note that we have to prefix the procedure names with `&` so that we don't execute the procedures, and apply `WHY` on the return value.

We can actually do this with any block, as long as it has a name or a pointer to it.

Block comments are part of `pod`, the inline documentation sublanguage. We'll get back to it in the «Advanced Raku» course.

`WHY` is a bridge from one world to the other…

10.10.1. MAIN with typing

We can add strong typing to the arguments we pass to `MAIN`, but have the same problem as described in section 6.6.1, "prompt" that strings are written without quotes, so the compiler assumes that they are strings. A number without quotes is eiter a number - or a string without quotes.

With `prompt` we could avoid this by specifying numbers with quotes to force the compiler to treat them as strings, but that doesn't work in the shell, as the shell uses quotes to group text and removes them before passing the content on to the program.

Spaces are used to separate arguments, and we must quote the string if we want a space inside it.

File: args

```
say "Argument: «{ $_ }»" for @*ARGS;
```

And running it:

```
$ raku args 123 "456 789" '10 11 12' 13
Argument: «123»
Argument: «456 789»
Argument: «10 11 12»
Argument: «13»
```

The quotes are used by the shell, both types (single and double).

So how do we specify quotes so that the shell leaves them alone, and sends them on to the program?

We can try using both single and double quotes at the same time:

```
$ raku args "'456'" '"10"'
Argument: «'456'»
Argument: «"10"»
```

And that works. At least for «bash», the shell I am using. Other shells may do it differently.

Exercise 10.1

Write a program that shows the *type* of the input, and a sorted (alphabetically) list of methods available for objects (or values) of that type.

E.g.

```
$ raku type-methods "3+4i"
3+4i (of type ComplexStr) supports: (abs ACCEPTS acos acosec acosech acosh acotan
acotanh asec asech asin asinh atan atan2 atanh base base-repeating Bool Bridge
BUILDALL ceiling cis Complex conj cos cosec cosech cosh cotan cotanh denominator
DUMP exp FatRat floor gist Int is-prime isNaN log log10 narrow new norm nude Num
numerator Numeric raku polymod pred rand Range Rat Real roots round sec sech sign
sin sinh sqrt Str succ tan tanh truncate unpolar WHICH)
```

Tips: start with a type as e.g. `Int` and try it out in REPL.

The types we get for input (as seen in section 6.6.2, “Str Inheritance Tree”) are:

Normal type	**Input type**
`Int`	`IntStr`
`Num`	`NumStr`
`Rat`	`RatStr`

`Complex`	`ComplexStr`

10.11. IntStr Gotcha

This seems ok, right?

File: intstr-gotcha

```
multi MAIN (Int $number)
{
  say "Integer: $number";
}

multi MAIN (Str $string)
{
  say "String: $string";
}
```

Running it:

```
$ raku intstr-gotcha qwwe
String: qwwe
```

But integers doesn’t work:

```
$ raku intstr-gotcha 12
Ambiguous call to 'MAIN(IntStr)'; these signatures all match:
:(Int $number)
:(Str $string)
  in block <unit> at content/code/intstr-gotcha line 8
```

The problem is that we got the `IntStr` type, and as it inherits from both `Int` and `Str` we were unable to choose between the «MAIN» candidates.

Exercise 10.2

Solve this problem.

10.12. Multiple Dispatch

We can have different versions of a procedure, specified with the `multi` keyword, with different parameter lists (or «signatures»):

```
multi sub do-something ($file1)         { ... }
multi sub do-something ($file1, $file2) { ... }
```

We can skip the `sub` when we specify `multi`, if we want to.

With type constraints:

```
multi add (Numeric $value1, Numeric $value2) { ... }
multi add (Str     $value1, Str     $value2) { ... }
```

10.12.1. Stub Operator

We can specify placeholder code (also called the «stub or «Yada, yada, yada» operator instead of ordinary code.

The code will compile, but will complain if we try to execute it:

Code	Action
`…`	`fail`
`!!!`	`die`
`???`	`warn`

They will be covered in the «Advanced Raku» course.

yada

We can check if a procedure is stubbed with `yada`:

```
sub a { 1; };  say &a.yada;  # -> False;
sub b { ... }; say &b.yada;  # -> True;
sub c { !!! }; say &c.yada;  # -> True;
sub d { ??? }; say &d.yada;  # -> True;
```

10.12.2. The Fibonacci Numbers

This is the first 10 Fibonacci Numbers: «1, 1, 2, 3, 5, 8, 13, 21, 34, 55».

The first value is 1, the second is also 1, and after that each value is the sum of the two preceding values.

I have chosen to present the version of the numbers starting with 1. There is another version that starts with zero (and the indices are off). The answer to «give me the third Fibonacci number» is either 1 or 2.

Here are a couple of programs that prints the given Fibonacci number.

With a loop:

File: fibonacci-loop

```
sub MAIN (Int $n)
{
  say fibonacci $n;
}

sub fibonacci (Int $n)
{
  return 1 if $n == 1 or $n == 2;

  my @fib = (1, 1);

  for 2 .. $n -1 -> $i
  {
    @fib[$i] = @fib[$i -1] + @fib[$i -2]
  }

  return @fib.tail;
}
```

We can use recursion (a procedure that calls itself, repeatedly):

File: fibonacci-recursive

```
sub MAIN (Int $n)
{
  say fibonacci $n;
}

sub fibonacci (Int $n)
{
  return 1 if $n == 1 or $n == 2;

  return fibonacci($n-1) + fibonacci($n-2)
}
```

Much shorter, and it is actually easier to understand than the loop version.

We can use `multi` to factor out the first two values:

File: fibonacci-multi

```
sub MAIN (Int $n)
{
  say fibonacci $n;
}

multi fibonacci (1) { 1 }
multi fibonacci (2) { 1 }

multi fibonacci (Int $n where $n > 2)
{
  fibonacci($n - 2) + fibonacci($n - 1)
}
```

So what can we use the Fibonacci numbers to? Nothing much, except showing off our mathematical knowledge, and the power of Raku.

(We'll show off a little more in section 16.3.1, "The Fibonacci Sequence".)

The recursive version is slower than the loop version. We'll show this, and explain why, in section 15.5, "Timing Fibonacci".

Multiple dispatch is explained in detail in the «Advanced Raku» course. There we present `proto`, and ways to defer execution to other `proto` candidates.

10.13. Procedure Arguments

Values passed to a procedure are read-only by default:

File: increment

```
sub increment ($value)
{
  $value++;
  return $value;
}

my $number = increment(12);
```

```
$ raku increment
Cannot resolve caller postfix:<++>(Int);
the following candidates match the type
but require mutable arguments:
    (Mu:D $a is rw)
    (Int:D $a is rw)
```

10.13.1. is rw

The error message gives a hint: `is rw` («is read write»). This is a trait, that we can add to parameters. Let us try:

File: increment2

```
sub increment ($value is rw)
{
  $value++;
  return $value;
}

my $number = increment(12);
```

```
$ raku increment2
Parameter '$value' expected a writable container,
but got Int value
  in sub increment at increment2 line 3
  in block <unit> at increment2 line 9
```

And this fails as well. The problem is that `is rw` tells the procedure that it can change the variable in the calling code, but we called it with a value. And values cannot be changed:

```
> 12 = 13;
Cannot modify an immutable Int (12)
  in block <unit> at <unknown file> line 1
```

This works:

```
> my $value  = 12;
> my $result = increment($value);
> say $value;  # -> 13
```

But the side effect, that the procedure call changes the value of a variable outside itself without an assignment, is something that should be avoided.

It will fail again if you try to adjust the value passed:

```
> my $result = increment($value + 1);
```

10.13.2. is copy

The `is copy` trait is more fool proof. You get a real variable, with a copy of the value passed to it, and it (the copy) can be changed at will.

File: increment3

```
sub increment ($value is copy)
{
  $value++;
  return $value;
}

say increment(12);
```

```
$ raku increment3
13
```

10.13.3. Optional Arguments

It is possible to specify a default value for an argument, making it optional:

```
sub do-something ($value, $optional = "") { ... }
```

We can have more of them:

```
sub do-something-else ($value, $optional1 = 5,  $optional2 = $value * 2) { ... }
```

It is not possible to assign a value to $optional2 and not $optional1.

```
> do-something-else(11, 101);
```

10.13.4. Named Arguments

A procedure taking many arguments can be a problem. Someone will sooner or later get the order of the arguments wrong.

Named arguments, specified by prefixing the variable with a `:` (colon) in the argument list, removes that problem, as the order is now irrelevant:

```
> sub aaa (:$a, :$b) { return 2*$a + $b; }
> say aaa(a => 2, b => 3);  # -> 7
> say aaa(b => 3, a => 2);  # -> 7
```

Named arguments makes it possible to have many optional arguments, and you can use as many or few as you want, regardless of order:

```
> sub bbb (:$a = 12, :$b = 13, :$c = 12, :$d = 13 )
> {
>   return $a + $b + $c + $d;
> }
> aaa(a => 1);
> aaa(d => 3, a => 4);
```

You can mix normal (or positional) and named arguments, but the positional ones must come first:

```
> sub ccc ($a, $b, :$c, :$b) { ... }
```

Named arguments may remind you of the `Pair` syntax (see section 9.1, “Pair”). That is no coincidence, as they really are `Pair`.

Named Argument Shortcut

Using good variable names may lead to situations like this:

File: named

```
sub cost (:$height, :$width)
{
  return $height * $width * 4;
}

my $height = 12;
my $width  = 512;

say cost(height => $height, width => $width);
```

As long as the variable names are the same, we can shorten this to:

File: named2 (partial)

```
say cost(:$height, :$width);
```

Running them:

```
$ raku named
24576
$ raku named2
24576
```

The Needle in the Haystack Problem

Let us revisit the «needle in the haystack problem» described in passing in section 9.6, “values”, finding the key given a value from a hash.

This requires brute force:

File: haystack

```
sub find-value-in-hash (%hash, $value, :$all = False, :$verbose = False)
{
  say "Looking for $value:";
  for %hash.kv -> $key, $val
  {
    say "- Checking $key" if $verbose;
    if $val eq $value
    {
      say "- Found it: $key -> $val";
      last unless $all;
    }
  }
}

my %haystack = ( A => "Bike",  Q => "Beetle", "#" => "Book", 12 => "Needle",
                17 => "Frog", 29 => "DVD player (defective)", 76 => "Bike");

find-value-in-hash(%haystack, "Beetle");
find-value-in-hash(%haystack, "Bike", :verbose);
find-value-in-hash(%haystack, "Beetle");
find-value-in-hash(%haystack, "Bike", :all);
```

I have used named optional arguments here.

And running it:

```
$ raku haystack
Looking for Beetle:
- Found it: Q -> Beetle
Looking for Bike:
- Checking 17
- Checking 12
- Checking 29
- Checking Q
- Checking A
- Found it: A -> Bike
Looking for Beetle:
- Found it: Q -> Beetle
Looking for Bike:
- Found it: A -> Bike
- Found it: 76 -> Bike
```

10.13.5. Named Mandatory Arguments

We can make a named argument mandatory with the `is required` trait, or the `!` short form:

```
> sub ccc ($a, $b, :$c!, :$d is required) { ... }
```

This gives a nice error message:

```
> ccc(1, 2);
Required named parameter 'c' not passed in sub ccc at ...
```

Default values are meaningless for mandatory arguments. The compiler will not protest, though. So `:$d is required = False` is legal, even if the default value is useless.

10.13.6. Adverbs

It is possible to use an alternative *adverbial syntax* when specifying named arguments in a procedure call:

```
> sub aa (:$a, :$b) { say "A: $a B: $b"; }

> aa(a => 1, b => 2);  # -> A: 1 B: 2
> aa(:a(1), :b(2));    # -> A: 1 B: 2
> aa(:1a,   :2b);      # -> A: 1 B: 2

> aa(a => "r", b => "h");  # -> A: r B: h
> aa(:a<r>   , :b<h>);     # ->  A: r B: h
```

Adverbs works with the built-in functions as well.

So far these named parameters have all taken values. Without any other constraints and no argument value, a named parameter is a Boolean. The adverb form with no value (and no constraint) gets `True` (because that's what Pairs do):

```
> aa(:a, :b);   # -> A: True B: True
```

An `!` in front of the adverb name makes it a `False` value:

```
> aa(:a, :!b);  # -> A: True B: False
```

We can allow *any* named argument by using `%_` (just as we could allow *anything* with `@_`):

```
> sub named { say %_ }
> named( name => 'Tom' );                # -> {name => Tom}
> named( name => 'Phil', age => 12 ); # -> {age => 12, name => Phil}
> named;                                 # -> {}
```

We can combine named and positionals:

```
> sub both { say @_; say %_; }

> both( 12, 13, name => "Tom", age => 45 );
[12 13]
{age => 45, name => Tom}

> both( 12, name => "Tom", age => 45, "19C" );
[12 19C]
{age => 45, name => Tom}
```

They can be intermixed, primarily to add to the confusion.

10.14. * (Slurpy Operator)

If we try to pass a bunch of *scalars* to a procedure expecting an *array* we'll get an error:

```
> sub a (@values) { say "ok"; }
> a(1,2,3,4,5);
===SORRY!=== Error while compiling:
Calling a(Int, Int, Int, Int, Int) will never work with declared signature (@values)
------> <BOL>  a(1,2,3,4,5);
```

We can remedy that in the procedure signature by using a slurpy (or «variadic argument») array, that grabs all the remaining scalar values as a list. Add a `*` before the list argument: `*@values`:

```
> sub a (*@values) { say "ok"; }
> a(1,2,3,4,5);
ok
```

We can of avoid this problem by ensuring that we always call the procedure with a list as argument, but that will not work with arguments passed on the command line (as they will always be scalar values). See the next section for details.

Or we can drop the signature, and access `@_` in the procedure body.

10.14.1. Slurpy MAIN

Arguments passed on the command line are always scalars, but we can use a slurpy array to grab them all by adding a `*` before the list argument: `*@words`:

File: words

```
sub MAIN (*@words)
{
  @words.grep({ .contains("a") }).say;
}
```

Running it:

```
$ raku words absn kakak alala 9099 00 00
(absn kakak alala)

$ raku words
()
```

A slurpy argument allows zero arguments, which is not a good thing.

We can force it to demand *at least* one argument:

```
sub MAIN ($word1, *@words) { ... }
```

Good luck explaining the code...

Using a type constraint gives self-explaining code:

File: words2

```
#| One or more words to search for lines with the letter 'a'
sub MAIN (*@words where @words.elems >= 1)
{
  @words.grep({ .contains("a") }).say;
}
```

(And it certainly doesn't hurt to add an explicit usage comment.)

`contains` does what you think; checks if the string on the left contains the string on the right. See section 11.4.2, "contains (Partial Strings)" for details.

We can polish the `where` condition a little bit:

File: words3 (partial)

```
sub MAIN (*@words where so @words)
```

10.14.2. Random Primes Revisited

Let us revisit the random primes code we wrote in Exercise 8.3:

```
> (1 ..  100).grep(*.is-prime).pick.say;  # ->   13
> (1 .. 1000).grep(*.is-prime).pick.say;  # -> 1861
```

We can rewrite it as a program, taking the upper limit as argument:

File: random-prime

```
sub MAIN ($upper-limit)
{
  (1 .. $upper-limit).grep(*.is-prime).pick.say;
}
```

```
$ raku random-prime aaaaa
Cannot convert string to number: base-10 number must
begin with valid digits or '.' in '  aaa' (indicated by   )
  in sub MAIN at random-prime line 3
  in block <unit> at random-prime line 3
```

We'll get a better error message if we add an `Int` constraint to the input argument:

File: random-prime2 (partial)

```
sub MAIN (Int $upper-limit)
```

```
$ raku random-prime2 aaaaa
Usage:
  random-prime2 <upper-limit>
```

The `Int` constraint ensures that only integers are allowed. But what about negative integers?

```
$ raku random-prime2 -100
Nil
```

No, the prime numbers are all positive.

REPL can help if you didn't know that:

```
> (-17).is-prime
False
```

The `(1 .. -100)` construct will try to generate integers from 1 up to -100. That is impossible, so `Nil` (an empty list) is returned. We then pick one random value from an empty list, and get `Nil`.

If you have heard about Sequences (e.g. `1 … -100`); yes they would have worked here, and no it wouldn't have mattered. We'll cover sequences later.

It is better to make negative input values illegal, and this is possible.

We can add a constraint:

File: random-prime3

```
sub MAIN (Int $upper-limit where * > 0)
{
  (1 .. $upper-limit).grep(*.is-prime).pick.say;
}
```

```
$ raku random-prime3 -100
Usage:
  random-prime2 <upper-limit>
```

Raku has a type `UInt` for «Unsigned Int», and we could of course have used that.

10.14.3. Better Usage Messages

The Usage message doesn't really inform us of the legal values. We can rename the variable to e.g. `$upper-limit-as-a-positive-integer-larger-than-zero`.

But who'd want to type variable names like that?

We can add text to the usage message, like this:

File: random-prime4

```
#| A random prime number between 1 and ...
sub MAIN (Int $upper-limit where * > 0)
{
  (1 .. $upper-limit).grep(*.is-prime).pick.say;
}
```

Place the line just before the procedure.

```
$ raku random-prime4 -100
Usage:
  random-prime2 <upper-limit> -- A random prime \
  number between 1 and the given integer upper limit
```

Exercise 10.3

What is the problem with the random prime number programs?

Rewrite the program to avoid this problem.

10.15. Blocks Revisited

We introduced Blocks in section 4.1, “Blocks”, and defined it as: «A block is a collection of code that is treated as a whole. Blocks are set up inside a pair of curly braces.»

We can assign them to variables, and execute them:

```
> my $block = { "Hello, $_."; };
> say $block("Thomas");
Hello, Thomas.
```

Here we have a single argument, passed in `$_`.

We can use placeholder variables (as described in 10.4, “Placeholder Variables”), just as with procedures. They are useful if we want to pass more than one argument:

```
> my $block = { "Hello, $^a $^b."; };
> say $block("Thomas", "Mann");
Hello, Thomas Mann.
```

10.15.1. ->

We can also pass arguments in named variables to blocks (as with procedures), with a signature between `->` and the block:

```
> my $add = -> $a, $b = 2 { $a + $b };
> say $add(40);
42
```

Optional arguments (with a default value) work, as shown above (with `$b = 2`).

Procedures are essentially named blocks. But they have some extra bells and whistles, mainly multiple dispatch (see 10.12, “Multiple Dispatch”). And a nicer syntax.

10.15.2. <-> / is rw

The variables are read only by default, but we can make them read write with the `is rw` adverb:

```
my $swap = -> $a is rw, $b is rw { ($a, $b) = ($b, $a) };
my ($a, $b) = (2, 4);
$swap($a, $b);
say $a; # -> 4
```

Or we can use the two way arrow `<->` instead of the one way one, but that turns *all* the parameters to read write.

```
my $swap = <-> $a, $b { ($a, $b) = ($b, $a) };
```

10.16. Calling a procedure specified in a variable

We can call a procedure where we have the name in a variable:

```
sub AAA { say "ok"; }
my $sub = "AAA";
&::($sub)();  # -> ok
```

Using a dispatch table is a better solution:

File: dispatch-table

```
sub aaa
{
  say "12345";
}

sub bbb
{
  say "FOOBAR";
}

my %table;

%table{"a"} = &aaa();
%table{"b"} = &bbb();

&(%table<a>);  # Execute "aaa"

my $p = "b";

&(%table{$p}); # Execute "bbb"
```

Calling a *method* stored in variable is also possible. See section 17.21, “Calling a method specified in a variable”.

10.17. Procedures in Procedures

It is possible to place a procedure definition inside another procedure. The result is that the inner procedure is only visible inside the outer one (is in scope), and can only be called by that one.

Chapter 11. Regex Intro

Regular Expressions are a sublanguage, with its own syntax and rules…

> Some people, when confronted with a problem, think «I know, I'll use regular expressions.» Now they have two problems.
>
> — http://regex.info/blog/2006-09-15/247

One way of handling this «problem» is avoiding using regexes in the first place.

This chapter presents some typical Regexes, and then non-regex alternatives where they exist. The non-regex alternatives are generally much faster.

The paradox is that the non-regex alternatives replace quite easy regexes that should be almost impossible to screw up, whereas more advanced regexes that really could use alternatives obviously don't have them.

If you embrace the non-regex versions in this chapter, instead of at least trying to understad the coresponding Regexes, you'll be at a disadvantage when you sometime in the future actually need to make a Regex.

Raku prefers the name «Regex» (and plural «Regexes») instead of «Regular Expressions», as they have diverged from the «regular» origins a long time ago. You may see both names though. (The names «Pattern» and «Rule» have been suggested, but didn't catch on. You may come across them in older blog posts etc.)

11.1. What is a Regex?

A Regex is a way of matching with expression (that can be interoreted in several ways) and not a static text.

We can start with the «hello» program from section 10.9, "MAIN".

File: hello

```
sub MAIN ($name)
{
  say "Hello, $name!";
}
```

Our task is to adapt it, so that the message is different depending on the name:

- It the name is «Steve», «Neve» or «Barry» we print «Go away, <name>!».
- If the name is 5 characters long and the middle character is a wovel we print "Hello, <name>. Whatsup?»
- For all other names we print «Hello, <name>»

File: hello-nonregex

```
sub MAIN ($name)
{
  if $name eq "Steve" or $name eq "Neve" or $name eq "Barry"
  {
    say "Go away, $name!";
  }
  elsif $name.chars == 5 and ($name.substr(2,1) eq "a"
                          or $name.substr(2,1) eq "e"
              or $name.substr(2,1) eq "i"
              or $name.substr(2,1) eq "o"
              or $name.substr(2,1) eq "u")
  {
    say "Hello, $name. Whatsup?";
  }
  else
  {
    say "Hello, $name!";
  }
}
```

So far so good. A lot of code, but it works. But what if we decide to insist on letters only in the name (the one with 5 characters)?

We can rewrite that one as:

File: hello-regex (partial)

```
  elsif $name ~~ /^ \w\w <[aeiou]> \w\w $/
```

- The «/» character marks the beginning and end of a Regex, and whitespace is ignored by default, as are newlines. The «^» character binds to the beginning of the string, and «$» binds to the end.
- «\w» matches a »word character» that is more or less what we want (a letter).
- «<[aeiou]>» is a character group where we match one of the given characters.

Much more compact, and actually easier to understand - when you have some Regex knowledge.

11.2. Making a Regex

The easiest way to generate a Regex is to put it inside two `/`:

```
/abc/
/12345/
```

A stand alone Regex (like these) is matched against `$_` (the topic variable):

```
> $_ = "abc"; say so /abc/;   # -> True
>             say so /abcd/;  # -> False
```

We can use given (see section 4.12, "given") to set $_ for us:

```
> say so /abc/ given "abc";   # -> True
```

11.3. ~~ (Smartmatch Operator)

The «Smartmatch Operator» ~~ is fundamental to Regexes. We can use it to compare almost anything with almost anything else (and not just a Regex).

If we match against $_, as we did in the previous section, we can drop ~~. These are all equal:

```
> $_ ~~ m/1234/;
> $_ ~~ /1234/;
> /1234/;
```

11.3.1. m/…/

Of we prefix the Regex with m (for «match») we can swap the slashes (/) with any other character. This is useful if we have a slash in the Regex itself:

```
> m|/usr/bin/|;
```

We can use characters with an opening and a closing version:

```
> m{123};
```

11.3.2. !~ (Negated Smartmatch Operator)

Use the «Negated Smartmatch Operator» !~ to invert the match.

11.4. Partial Strings

We can check if a given string contains another one:

```
> say so "12345" ~~ /23/; # -> True
> say so "12345" ~~ /33/; # -> False
```

We can skip the slashes on the right side, but that changes the meaning:

```
> say so "12345" ~~ "23"; # -> False
```

We are now smartmatching two strings, and that is the same as a normal string comparison which returns `False` as they are not equal.

We can put the left hand side in a variable:

```
> my $val = "12345";
> say so $val ~~ /23/; # -> True
```

We can do the same with the regex:

```
> my $b = /23/;
> say so "12345" ~~ $b; # -> True
> say $b.WHAT;          # -> (Regex)
```

We can compare with a type:

```
> say so "12345" ~~ Int; # -> False
> say so  12345  ~~ Int; # -> True
```

11.4.1. Regex (type)

We can use the type system:

```
> my Regex $b = /23/;
```

11.4.2. contains (Partial Strings)

Partial strings is so useful that we have a dedicated `contains` function for it:

```
> "12345".contains("23");  # -> True

> "12345".contains("33");  # -> False
```

11.4.3. index (Partial Strings)

Use `index` to get the position of one string in another. It will return the index if found, and `NIL` if not.

```
> "12345".index("1"); # -> 0
> "12345".index("0"); # -> Nil
```

The first character has position (or offset) 0, so be careful with the return value:

```
"12345".index("1").defined; # -> True
"12345".index("0").defined; # -> False

so "12345".index("1"); # -> False
so "12345".index("0"); # -> False
```

Use `contains` if you don't need the position. Then you'll not have to worry about definedness.

11.4.4. rindex (Partial Strings)

`rindex` is similar to `index`, but searches from the right, giving the position of the *last* match:

```
> "121212121".index(1);  # -> 0
> "121212121".rindex(1); # -> 8
```

11.4.5. indices (Partial Strings)

`indices` is similar to `index`, but searches for *all* occurences of one string in another. It returns an empty list if it was not found.

```
> say "banana".indices("a");              # -> (1 3 5)
> say "banana".indices("ana");            # -> (1)
> say "banana".indices("ana", 2);         # -> (3)
> say "banana".indices("b");              # -> (0)
> say "banana".indices("X");              # -> ()
```

If the optional parameter `:overlap` is specified the search continues from the next position in the string, and not after the match as by default:

```
> say "banana".indices("ana");                # -> (1)
> say "banana".indices("ana", :overlap);      # -> (1 3)
> say "aaaaaaaaaa".indices("aaa");            # -> (0 3 6)
> say "aaaaaaaaaa".indices("aaa", :overlap);  # -> (0 1 2 3 4 5 6 7)
```

11.5. Beginning or end of a string

The Regex in the previous section will match regardless of where in the string it was found. We can use an anchor to force the match to only consider the start, end or both of the string.

Anchor	Example	Description
`^`	`/^123/`	Match at the *start* of the string only

`$`	`/345$/`	Match at the *end* of the string only

11.5.1. starts-with (Partial Strings)

Matching from the *start* of the string:

```
> say so "12345" ~~ /^23/;   # -> False
> say so "12345" ~~ /^123/;  # -> True
```

We can use `starts-with` instead:

```
> "12345".starts-with("23");   # -> False
> "12345".starts-with("123");  # -> True
```

11.5.2. ends-with (Partial Strings)

Matching from the *end* of the string:

```
> say so "12345" ~~ /123$/;  # -> False
> say so "12345" ~~ /45$/;   # -> True
```

We can use `ends-with` instead:

```
> "12345".ends-with("123");  # -> False
> "12345".ends-with("45");   # -> True
```

11.5.3. equal

We can use both anchors, and that gives us the same as normal string comparison with `eq` (as we haven't used any Regex special characters yet):

```
> say so "12345" ~~ /^12345$/; # -> True
>        "12345" eq  "12345";  # -> True
```

Or even:

```
> say "12345" ~~ "12345";  # -> True
```

11.6. Regex Metacharacters

Inside a Regex Alphanumeric characters (letters and digits) and underscores (_) are taken literally, spaces are ignored, and every other character is a meta character with a special meaning.

Spaces and newlines inside Regexes are ignored by default, so feel free to add them to enhance readability.

We can get a literal metacharacter by quoting it (with a backslash):

```
> say so "12/34" ~~ /2\/3/; # -> True
```

11.7. $/ (Match Object)

The `$/` object holds the result of the last Regex match (or `Nil` if we have no match):

```
> "12345" ~~ /12/; say $/;  # -> ｢12｣
> "12345" ~~ /67/; say $/;  # -> Nil
```

Note the funny angle brackets. They are used when we print a match object, to remind us that `$/` is a **match object**, and not a string.

Use explicit stringification:

```
> say $/.Str;  # -> 12
> say ~$/;     # -> 12
```

Note that `put` (see section 6.3.4, “put vs say”) does stringification, and hides the problem:

```
> "12345" ~~ /12/; put $/;  # -> 12
```

Passing a match object on to code that expects a string can lead to errors. If the code in question uses an external library through Nativecall the program is almost certain to crash.

Nativecall will be covered in detail in the «Advanced Raku» course.

11.8. Special Characters

A `.` (single period) matches exactly one character:

```
> say so "12345" ~~ /1.3.5/; # -> True
```

We can add a quantifier after any character:

Quantifier	Greedy	Description

`?`	No	Match zero or one time
`+`	Yes	Match one or more times
`*`	Yes	Match zero, one or more times
`** number`	No	Match exactly «number» times
`** min..max`	Yes	Match minimum «min» and maximum «max» times

Note that some of the quantifiers (as indicated) make a match greedy. It will match as much as possible, as long as it manages to match the expression:

```
> say so "1111111111111112345" ~~ /1+2345/; # -> True
> say so "1111111111111112345" ~~ /1+2345/; # -> True

> say so "12345" ~~ /123459/;               # -> False
> say so "12345" ~~ /123459*/;              # -> True

> say so "0111111111111111234" ~~ /01 ** 1..20 234/; # -> True
> say so "0111111111111111234" ~~ /01 ** 1..10 234/; # -> False
```

11.9. Capturing and Grouping

So far we have only shown how to match (or not), but we can divide the match in several parts.

11.9.1. ( ) (Capturing)

We can use parens to capture matches, and reference them as `$0`, `$1` (and so on) later on:

```
> "12345" ~~ /(2)(.4)/;
> say $0.Str;  # -> 2
> say $1.Str;  # -> 34
```

We can use the match object instead of `$0`, `$1` (and so on):

```
> say $/
｢234｣
 0 => ｢2｣
 1 => ｢34｣
```

We can look up an individual match:

```
> say $/[1].Str;  # -> 23456
```

This works as well:

```
> say $[1].Str;  # -> 23456
```

11.9.2. Capture Numbering

Pairs of parens are numbered left to right, starting from zero:

```
> say "0: $0; 1: $1" if 'abc' ~~ /(a) b (c)/;  # -> 0: a; 1: c
```

Captures can be nested, and are numbered according to the level:

```
if 'abc' ~~ / ( a (.) (.) ) /
{
  say "Outer: $0";                  # -> Outer: abc
  say "Inner: $0[0] and $0[1]";    # -> Inner: b and c
}
```

The Match Object:

```
> say $/;
｢abc｣
 0 => ｢abc｣
  0 => ｢b｣
  1 => ｢c｣
```

11.9.3. [ ] (Non-capturing grouping)

If we don’t need capturing, we can skip it. Our Regex needs them for grouping, but we can use non-capturing brackets instead:

```
> my $valid-ipv4 = /^ [\d ** 1..3] ** 4 % '.' $/;
```

Benefits of Non-capturing:

- No clutter of the match object with things that isn’t used
- Faster than capturing

11.9.4. <( )> (Capture Markers)

When we have a match, the match objects contains the whole string, regardless of how many captures (if any) we have used.

We can prevent parts of the match from ending up in the match object by using the `<(` and/or `)>` tokens:

`<(`	Do not capture *before* this token
`)>`	Do not capture *after* this token

```
say 'abc' ~~ / a <( b )> c/;      # -> ｢b｣
say 'abc' ~~ / a  ( b )  c/;      # -> ｢abc｣; 0 => ｢b｣
```

.prematch / .postmatch

It is possible to get the part of the string *before* and *after* the match when we use Capture Markers. Use the `prematch` and/or `postmatch` methods on the match object:

```
> say 'abc' ~~ / a <( b )> c/;  # -> ｢b｣
> say $/.prematch;              # -> a
> say $/.postmatch;             # -> c
```

These methods return strings (and not match objects).

.orig / .target

We can get the original string as well, regardless of Capture Markers:

```
> say 'abc' ~~ / a <( b )> c/;  # -> ｢b｣
> say $/.orig;                  # -> abc
> say $/.target                 # -> abc
```

`orig` return an object, and `target` returns a stringified version of it. As we started with a string (`'abc'`), they both return strings here.

There are other methods that can be used on match objects. See https://docs.raku.org/type/Match#Methods for details.

11.10. Character Classes

We can use a Character class to match different kinds of characters.

We have the following defined with a leading backslash (and called «Backslashed Character Classes»):

Class	Match:	Negated
`\n`	a newline character (see `$?NL` in section 6.1, “Newlines”)	`\N`
`\t`	a tab character	`\T`
`\h`	a horisontal whitespace character	`\H`
`\v`	a vertical whitespace character	`\V`
`\s`	a whitespace character (horisontal or vertical)	`\S`
`\d`	a digit (including unicode digits)	`\D`
`\w`	a word character; a letter, digit or underscore (including unicode letters and digits)	`\W`

They will match exactly one character, unless combined with a quantifier. The negated version matches everything - but the normal one.

`\s` matches newlines as well as spaces:

```
> say so "abc abf"   ~~ /\s/; # -> True;
> say so "abcXabf"   ~~ /\s/; # -> False;
> say so "abcXabf\n" ~~ /\s/; # -> True;
```

We also have more verbosely named Character Classes. The most useful ones are:

Class	**Alias**	**Description**
`<alnum>`	`\w`	`<alpha>` plus `<digit>`
`<alpha>`		Alphabetic characters including `_`
`<blank>`	`\h`	Horizontal whitespace
`<cntrl>`		Control characters
`<digit>`	`\d`	Decimal digits
`<graph>`		`<alnum>` plus `<punct>`
`<lower>`	`<:Ll>`	Lowercase characters
`<print>`		`<graph>` plus `<space>`, but no `<cntrl>`
`<space>`	`\s`	Whitespace
`<upper>`	`<:Lu>`	Uppercase characters
`<xdigit>`		Hexadecimal digit [0-9A-Fa-f]

See https://docs.raku.org/language/regexes#Predefined_character_classes for the complete list.

We haven’t shown a character class representing letters only. We have several among the Unicode Categories that we can use. The most useful (in a very long list) are:

Short	**Long**	**Description**
`<:L>`	`<:Letter>`	
`<:Ll>`	`<:Lowercase_Letter>`	
`<:Lu>`	`<:Uppercase_Letter>`	
`<:N>`	`<:Number>`	Matches digits, unicode digits and things as «½».
`<:P>`	`<:Punctuation>` or `<:punct>`	
`<:S>`	`<:Symbol>`	
`<:Sc>`	`<:Currency_Symbol>`	Matches «£», «$», «€» and others.

See https://docs.raku.org/language/regexes#Unicode_properties for the complete list.

```
> "1234sksjsjsjs1919" ~~ /(<:N>+)/;       # $0 -> ｢1234｣
> "1234sksjsjsjs1919" ~~ /(<:N>+)(<:L>)/; # $0 -> ｢1234｣, $1 -> ｢s｣
```

Counting letters only in a string:

File: letter-count

```
sub MAIN ($string)
{
  my $count = $string.comb.grep(* ~~ /<:L>/).elems;

  say "The string contains $count letters.";
}
```

```
$ raku letter-count 12Aw
The string contains 2 letters.

$ raku letter-count 12Aw#@ß
The string contains 3 letters.
```

Note that we can shorten the selection bit:

File: letter-count2 (partial)

```
my $count = $string.comb.grep(/<:L>/).elems;
```

Further reading on Unicode:

- https://docs.raku.org/language/regexes#Unicode_properties
- https://en.wikipedia.org/wiki/Unicode_character_property

We can negate them, by inserting `!` (an exclamation mark) between `:` (the colon) and the class name:

```
> say so "1234sksjsjsjs1919" ~~ /<:!L>/; # -> True
> say so "abcdefghijklmnopq" ~~ /<:!L>/; # -> False
```

We can combine several Unicode Categories. Either with a `+` to add them (as a set union) or `-` to remove the right hand side (a set difference) inside the angle brackets:

```
> say so "1234sksjsjsjs1919"  ~~ /<:!L-:N>/; # -> False
> say so "1234sksjsjsjs1919." ~~ /<:!L-:N>/; # -> True
```

The first one checks for everything that isn't a letter (`!L`), then removes the digits. That leaves us with nothing. The second one does the same, but leaves us with a single period (`.`).

11.10.1. uniprop

Use `uniprop` to display the Unicode Category for the first character in the given string:

```
> "a".uniprop; # ->  Ll
> "A".uniprop; # ->  Lu
> "ß".uniprop; # ->  Ll
> '$'.uniprop; # ->  Sc
```

We can check for specific properties:

```
say 'a'.uniprop('Alphabetic'); # -> True
```

uniprops

Use `uniprops` to get the values for every charatcter in the string:

```
> "Fix 10!".uniprops;           # -> (Lu Ll Ll Zs Nd Nd Po)
> "Fix 10!".uniprops("Letter"); # ->  (1 1 1 0 0 0 0)
```

11.11. Custom Character Classes

We can specify our own Character Classes with `<[` and `]>`:

```
> "abcdefghijklmn" ~~ /(<[fed]>+)/; # $0 -> ｢def｣
```

Note that a Character Class by itself only matches one character.

We can negate them:

```
> "abcdefghijklmn" ~~ /(<-[fed]>+)/; # $0 -> ｢abc｣
```

We can combine them (as with Unicode Properties) with `-`, and use ranges:

```
> "1234567890" ~~ /(<[1..9] - [5]>+)/; # $0 -> ｢1234｣
```

11.12. Non-greedy

We have shown that some of the Regex Quantifiers are greedy, as they match as much as possible. We can make them non-greedy (or frugal) by adding a `?` (a question mark) after the greedy quantifier:

```
> "12345A" ~~ /(\d*?)/; # $0 -> ｢｣ # zero or more gives zero.
> "12345A" ~~ /(\d+?)/; # $0 -> ｢1｣ # one or more gives zero.
> "12345A" ~~ /(\d+?)A/; # $0 -> ｢12345｣
```

Non greedy is only applied in the rightwards direction, as the last example shows. We still match from the beginning (if possible), and get all the digits.

11.12.1. Usage

We can illustrate this with a simple (as in stupid) parser for html. We want to extract the image tags:

```
> "AAA <img src='12.png' alt='High Noon'> BBB" ~~ /(\<img\s.*\>)/;
0 => ｢<img src='12.png' alt='High Noon'>｣
```

I have added `\s` so that we don't match another tag with a similar name. (There shouldn't bee any, but it is common to comment out html tags by renaming them to something not used. Browsers ignore unknown tags, and so should we.)

That looks promising. But `.*` is greedy, and will run along until the last `>` on the line if there are more of them:

```
> "AAA <img src='12.png' alt='High Noon'> BBB <b>CCC</b> DDD" ~~ /(\<img\s.*\>)/;
 0 => ｢<img src='12.png' alt='High Noon'> BBB <b>CCC</b>｣
```

Non-greedy:

```
> "AAA <img src='12.png' alt='High Noon'> BBB <b>CCC</b>" ~~ /(\<img\s.*?\>)/;
｢<img src='12.png' alt='High Noon'>｣
```

11.13. Backwards References

It is possible to reference things we have already matched with `$0`, `$1` (and so on), inside the Regex.

The «img» tag has no end tag, as opposed to e.g. «b». We can try to write a general regex that matches any tag, and goes on until the matching end tag.

```
> "This is <b>bold <em>and cursive</em> and not</b>." ~~ /\<(.*?)\>(.*)\<\/$0\>/
｢<b>bold <em>and cursive</em> and not</b>｣
 0 => ｢b｣
 1 => ｢bold <em>and cursive</em> and not｣
```

Exercise 11.1

There is a problem with this Regex. What is it?

11.13.1. :ignorecase / :i

We can specify case insensitive matching with the `:ignorecase` (and the short form `:i`) adverb:

```
> say so "abcdefghijkl" ~~ /EFG/;    # -> False
> say so "abcdefghijkl" ~~ /:i EFG/; # -> True
```

11.14. Using a Regex

A Regex is usually created with a leading and trailing slash; e.g. `/abc/`. (We'll cover other ways in the «Advanced Raku» course.)

We have only seen Regexes that matches a string. But they can also be used to change the string we apply them to.

We can apply a Regex to a string in different ways:

Function	Method	Description	See section
`m/.../`		Match against `$_`	11.3.1, "m/…/"
`rx/.../`		A Regex object	
`/.../`	`match`	A Regex object (short form of `rx/.../`)	11.2, "Making a Regex"
`s/.../.../`		in-place substitution	11.15, "String Substitution"
`S/.../.../`	`subst`	non-destructive substitution	11.15.2, "subst (String Substitution)" and 11.15.3, "S/…/…/ (String Substitution)"
`tr/.../.../`		in-place transliteration	11.17, "Transliteration"
`TR/.../.../`	`trans`	non-destructive transliteration	11.17.2, "trans (Transliteration)" and 11.17.4, "TR/…/…/ (Transliteration)"

We can use whatever delimiter we want (instead of `/`) for all of them, except the third one (`/.../`).

Note that opening and closing versions of characters (as e.g. `{` and `[`) only works with matches. Substitution and transliteration use three slashes, and it isn't obvious what the third one should be.

The `match` method is described in the «Advanced Raku» course.

11.15. String Substitution

Use String Substitution to replace one sequence of characters with another one.

11.15.1. s/…/…/ (String Substitution)

The `s/.../.../` operator does the change on the left hand side variable:

```
> my $s = "one two three four";
> $s ~~ s/two/zero/;
> say $s;  # -> one zero three four
```

The substitution is done once by default.

```
> my $s = "one one one one";
> $s ~~ s/one/zero/;
> say $s;  # -> zero one one one
```

:global / :g

We can specify the `:global` (or the `:g` shortform) adverb to do the substitution as many times as possible:

```
> my $s = "one one one one";
> $s ~~ s:g/one/zero/;
> say $s;  # -> zero zero zero zero
```

It does not work recursively:

```
> my $s = "111111111111";
> $s ~~ s:g/11/1/;
> say $s;  # -> 111111
```

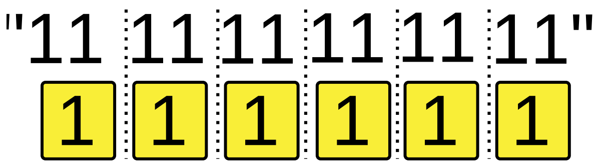


Figure 11. Substitution

Another take of our character counting program:

File: letter-count-remove

```
sub MAIN ($string is copy)
{
  $string ~~ s:g/<-:L>+//;

  say "The string contains { $string.chars } letters.";
}
```

We remove everything that isn't a letter, and count what we have left.

11.15.2. subst (String Substitution)

`subst` returns the calling string where the first string is replaced by the second one (or the original string, if no match was found).

```
> my $s = "one two three four";
> my $t = $s.subst("two", "zero");
> say $s;  # -> one two three four
> say $t;  # -> one zero three four
```

It doesn't change the string or variable it was invoked on, so we can do this:

```
> my $t = "one two three four".subst("two", "zero");
```

The substitution is only done once, unless we use the `:g` (global) adverb:

```
> "10101010101010202020202020202020".subst("10", "X")
X101010101010202020202020202020
```

```
> "10101010101010202020202020202020".subst(:g, "10", "X")
XXXXXXX2020202020202020
```

Assign the new value back, if you want to change the variable we invoked it on:

```
> $variable .= subst($replace, $with);
```

11.15.3. S/…/…/ (String Substitution)

`s/.../.../` changes the string it is used on. If you want to keep the string unchanged, use `S/.../.../` instead:

```
> $_ = "one two three four";
> my $t = S/two/zero/;
> say $t;  # -> one zero three four
> say $_;  # -> one two three four
```

You cannot use smartmatch with the `S/.../.../` operator (not even explicitly on `$_`).

But we can set `$_` implicitly with `given` (see section 4.12, "given"):

```
> my $t = S/two/zero/ given "one two three four";
```

11.15.4. Adverbs

Regex Adverbs) We can use adverbs to change how the Regex works:

Adverb	Short	On	Description
`:continue`	`:c`	M	Where to start the search.
`:exhaustive`	`:ex`	M	All possible matches, including overlapping.
`:global`	`:g`	M	All matches, and not just the first one. See section 11.15.1.1, ":global / :g".
`:ignorecase`	`:i`	R	See section 11.13.1, ":ignorecase / :i".
`:ignoremark`	`:m`	R	Compare base characters only. See below.
`:overlap`	`:ov`	M	As `:exhaustive`, but only one from each starting position.
`:pos`	`:p`	M	Anchor the match from the specified position (substring index).
`:ratchet`	`:r`	R	No backtrace.
`:samecase`	`:ii`	S	Ignore the case when matching, but apply it to the replacement. See below.
`:samemark`	`:mm`	S	As `:ignoremark`, and applies the accents to the replacement. See below.
`:samespace`	`:ss`	S	As `:sigspace`, and takes the whitespace to the replacement. See below.
`:sigspace`	`:s`	R	Make whitespace significant. See below.

The «On» column means:

- M - on matches only
- R - on all Regexes
- S - on Substitution only

Adverbs not shown with «See below» are not described in this book. See https://docs.raku.org/language/regexes#Adverbs for details.

:ignoremark / :m

Regex: Compare base characters only, ignoring accents:

```
> say so /:ignoremark abc/ given "åbc";  # -> True
> say so /:ignoremark abc/ given "øbc";  # -> False
> say so /:ignoremark obc/ given "øbc";  # -> True
```

:samemark / :mm

Substitution only: As `:ignoremark`, and applies the accents to the replacement.

```
> say S:samemark:global/a/o/ given "åbäcà";  # -> o  böcò
```

:samecase / :ii

Substitution only: Ignore the case when matching, but apply it to the replacement. So «abc» and «Abc» will match with «abc», «Abc», «ABC» and so on. The replacement string will have the same case as specified on the match:

```
> say S:samecase/abc/def/ given "xABCxabcABx";  # -> xDEFxabcABx
```

Compare with normal substitution:

```
> say S/abc/def/ given "xABCxabcABx";  # -> xABCxdefABx
```

:sigspace / :s

Regex: Make whitespace significant.

```
say so "abc abc" ~~ /abc abc/; # -> False (as «/abcabc/» does not match)
say so "abc abc" ~~ /:sigspace abc abc/;
```

Note that the first space (between the adverb and the first «a»letter in the regex) is there as a delimiter, and it is ignored.

:samespace /:ss

Substitution only: As `:sigspace`, and takes the whitespace to the replacement.

```
say S:samespace/a ./c d/ given "a b";  # -> c d
say S:samespace/a ./c d/ given "a\tb"; # -> c\td
```

11.16. Substitution Tuning

We can specify how many replacements we want with the `:x` adverb:

```
$str.subst(/foo/, "no subst", :x(0)); # targeted substitution. Number of times to
substitute. Returns back unmodified.
$str.subst(/foo/, "bar", :x(1)); #replace just the first occurrence.
```

We can specify which match we want to replace with the `:nth` adverb:

```
$str.subst(/foo/, "bar", :nth(3)); # replace nth match alone. Replaces the third foo.
Returns Hey foo foo bar
```

11.17. Transliteration

Transliteration is the process of replacing one character with another.

11.17.1. tr/…/…/ (Transliteration)

The `tr/.../.../` operator does the change on the left hand side variable:

```
> my $s = "1234567890";
> $s ~~ tr/129/ABx/;  # -> AB345678x0
```

If we do not specify enough replacement characters, the last one is (re)used:

```
> my $s = "1234567890";
> $s ~~ tr/129/A/;  # -> AA345678A0
```

We can remove characters as well:

```
> my $s = "1234567890";
> $s ~~ tr:delete/129//; # -> 3456780
```

11.17.2. trans (Transliteration)

Use `trans` to change one character with another one, given as a `Pair`:

```
> say "abcabc".trans("a" => "1");  # -> 1bc1bc
```

We can do several transliterations at the same time:

```
> say "abcabc".trans("a" => "1", "b" => "9");  # -> 19c19c
```

We can use a hash as well:

```
> my %trans = ("a" => "1", "b" => "9");
> say "abcabc".trans(%trans); # -> 19c19c
```

Ranges can also be used:

```
> "secret text".trans( ['a' .. 'z'] => ['b' .. 'z', 'a'] );
```

11.17.3. Rotate 13

Exercise 11.2

The oldest famous encryption algorithm is «Rotate 13», known from the Roman empire.

The same function encrypts and decrypts, as there are 26 characters in the (roman) alphabet. (The Romans didn't have «j» and «v», so they only had 24 characters, but we'll ignore that historic anomaly.)

Implement «rotate13».

Handle a-z and A-Z only. All other characters are left unchanged.

The string "Hello, raku programmers!" translates to "Uryyb, enxh cebtenzzref!" and vice versa.

11.17.4. TR/…/…/ (Transliteration)

The `tr/.../.../` operator changes the string it is used on. If you want to keep the string unchanged, use `TR/.../.../` instead:

```
> $_ = "one two three four";
> my $t = TR/oe/xx/;
> say $t;  # -> xnx twx thrxx fxur
> say $_;  # -> one two three four
```

11.18. trim / trim-leading / trim-trailing

Removing leading and/or trailing spaces is a common task. So Raku has functions for it:

Function	Description
`trim`	Remove leading and trailing spaces
`trim-leading`	Remove leading spaces
`trim-trailing`	Remove trailing spaces

```
> say "X" ~ "  123  " ~ "X";                # -> X  123  X
> say "X" ~ "  123  ".trim ~ "X";           # -> X123X
> say "X" ~ "  123  ".trim-trailing ~ "X";  # -> X  123X
> say "X" ~ "  123  ".trim-leading ~ "X";   # -> X123  X
```

I have used them as methods, but they work as a functions as well.

Exercise 11.3

Write the Regex versions of `trim-leading`, `trim-trailing` and `trim`:

Method	Regex Usage
`$y = $x.trim-leading`	`$x ~~ /XXXX/; $y = $0.Str;`
`$y = $x.trim-trailing`	`$x ~~ /XXXX/; $y = $0.Str;`
`$y = $x.trim`	`$x ~~ /XXXX/; $y = $0.Str;`

11.19. split and grep

Note that `split` (see section 7.3, "split") and `grep` (see section 8.20.1, "grep") can take a Regex as argument (instead of a normal string):

```
my @words = $text.split(/\s/);
```

This solves the multiple spaces problem we pointed out in section 7.3, "split".

Any two digit number, where the second one is «2»:

```
> (1..100).grep: /^(\d)2$/;  # -> (12 22 32 42 52 62 72 82 92)
```

11.20. Comments

We can place comments in Regexes. Inline comments are not supported, as you are encouraged to use newlines.

Instead of:

```
"12345" ~~ /(2)(.4)/;
```

Write it like this:

```
"12345" ~~ /(2)   # A literal "2"
             (.4)  # Any character followed by a literal "4"
            /;
```

Chapter 12. Modules

Modules is a useful encapsulation technique, splitting a big task in smaller parts that are easier to implement - and thus making the whole thing possible.

If somebody else has written a module doing what you need, or a part of it, use that instead of reinventing the wheel yourself. This will save you time and effort.

The quality and maturity of modules differ quite a lot, and some of them may not be maintained any more. Which module to use if there are more to choose from is a topic for a book of its own. It may be better to write the code yourself than using a badly designed module.

12.1. Precompilation

Modules are compiled when they are installed. When a program uses a module, the precompiled version is loaded - and this speeds up the compilaten of the program.

It is not possible to precompile programs, but you can move the code (or at least most of it) to one or more modules. You should do this for larger applications, but not because of any (more or less imaginary) startup speed gains.

12.2. Module Administration with zef

Modules, Administration) You need a module manager to install, list, update and remove modules. `zef` is the only module manager that should be used.

The old module manager `panda` is not maintained, and should not be used.

12.2.1. zef list

Use the `zef list --installed` command to get a list of installed modules.

This is a very abridged list:

```
$ zef list --installed
===> Found via /usr/local/share/perl6/site ①
App::Mi6:ver<0.2.2>:auth<cpan:SKAJI> ②
Bailador:ver<0.0.15>:auth<github:Bailador> ③
Linenoise:ver<0.1.1>:auth<Rob Hoelz> ④
Shell::Command ⑤
p6doc:ver<1.002001> ⑥
zef:ver<0.5.3>:auth<github:ugexe>:api<0> ⑦
```

① `zef` tells us where it found the modules.

② A module with a version («ver») and author («auth»), prefixed with a single colon. The author is a CPAN user name.

③ As above, but the author is a github project name.

④ The author as a text string.

⑤ No author or version on this one.

⑥ No author on this one.

⑦ The author is a GitHub user name. Note the new tag «api». We'll discuss it in the «Advanced Raku» course.

> Modules may have a version (`:ver`) and author (`:auth`) part. This makes it possible to have several versions of a module installed, and you can choose which of them (or more) to use. (See section 12.3, "Using Modules (use)" for details.)

12.2.2. CPAN vs GitHub

Raku modules was initially hosted on GitHub only, but support for CPAN (the «Comprehensive Perl Archive Network») was added in 2018.

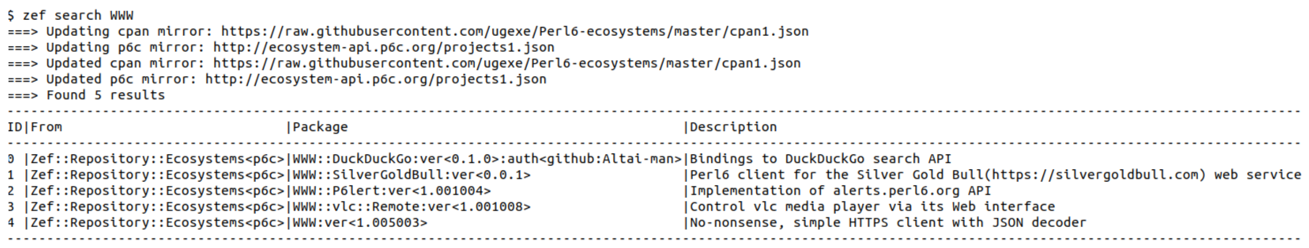
GitHub is a single server, and the Raku community suffered when it went offline. CPAN is a distributed network of sites, so problems on a single server will not affect the module repository.

`zef` supports both GitHub and CPAN. Note that the «auth» field in the module name is just a text field, so a module can be hosted on CPAN even if the «auth» field uses «github:».

12.2.3. zef search

Use `zef search` to search for modules with the given string in their name or, description.

E.g. `zef search www`:



```
$ zef search WWW
===> Updating cpan mirror: https://raw.githubusercontent.com/ugexe/Perl6-ecosystems/master/cpan1.json
===> Updating p6c mirror: http://ecosystem-api.p6c.org/projects1.json
===> Updated cpan mirror: https://raw.githubusercontent.com/ugexe/Perl6-ecosystems/master/cpan1.json
===> Updated p6c mirror: http://ecosystem-api.p6c.org/projects1.json
===> Found 5 results
--------------------------------------------------------------------------------------------------------------------------------------------------------------------
ID|From                            |Package                                          |Description
--------------------------------------------------------------------------------------------------------------------------------------------------------------------
0 |Zef::Repository::Ecosystems<p6c>|WWW::DuckDuckGo:ver<0.1.0>:auth<github:Altai-man>|Bindings to DuckDuckGo search API
1 |Zef::Repository::Ecosystems<p6c>|WWW::SilverGoldBull:ver<0.0.1>                   |Perl6 client for the Silver Gold Bull(https://silvergoldbull.com) web service
2 |Zef::Repository::Ecosystems<p6c>|WWW::P6lert:ver<1.001004>                        |Implementation of alerts.perl6.org API
3 |Zef::Repository::Ecosystems<p6c>|WWW::vlc::Remote:ver<1.001008>                   |Control vlc media player via its Web interface
4 |Zef::Repository::Ecosystems<p6c>|WWW:ver<1.005003>                                |No-nonsense, simple HTTPS client with JSON decoder
--------------------------------------------------------------------------------------------------------------------------------------------------------------------
```

Figure 12. zef search www

The output is rather wide, but here is the last one:

Field	Description	Value
ID	Just an internal counter	4
From	Which repository it was found in	`Zef::Repository::Ecosystems<p6c>`
Package	The name of the package (module)	`WWW:ver<1.005003>`
Description	A short description	No-nonsense, simple HTTPS client with JSON decoder

Note that the local list of available modules is updated first. This step can also be done manually with `zef update`.

⚠ If you installed Raku and zef as root, you will probably have to prefix the zef commands with sudo: `sudo zef ...`

12.2.4. zef install

Use `zef install` to install a module. It will download the specified module, run the tests and install it if the tests passed. If the module has dependencies (other modules) that isn't installed, they will be installed first.

E.g. `zef install WWW`:

```
$ zef install WWW
===> Searching for: WWW
```

It starts with updating the local list of available modules:

```
===> Updating cpan mirror: https://raw.githubusercontent.com/ugexe/Perl6-
ecosystems/master/cpan1.json
===> Updating p6c mirror: http://ecosystem-api.p6c.org/projects1.json
===> Updated cpan mirror: https://raw.githubusercontent.com/ugexe/Perl6-
ecosystems/master/cpan1.json
===> Updated p6c mirror: http://ecosystem-api.p6c.org/projects1.json
```

Then it checks for dependencies, recursively:

```
===> Searching for missing dependencies: HTTP::UserAgent, IO::Socket::SSL
===> Searching for missing dependencies: DateTime::Parse, Encode, IO::Capture::Simple,
Test::Util::ServerPort, OpenSSL
```

Then it gives a warning for two modules that don't follow the rules:

```
===> Extraction: Failed to find a META6.json file for
Encode:ver<0.0.2>:auth<github:sergot> -- failure is likely
===> Extraction: Failed to find a META6.json file for IO::Capture::Simple -- failure
is likely
```

Then it tests all the modules:

```
===> Testing: DateTime::Parse:ver<0.9.1>
===> Testing [OK] for DateTime::Parse:ver<0.9.1>
===> Testing: Encode:ver<0.0.2>:auth<github:sergot>
===> Testing [OK] for Encode:ver<0.0.2>:auth<github:sergot>
...
```

And finally installs them:

```
===> Installing: DateTime::Parse:ver<0.9.1>
===> Installing: Encode:ver<0.0.2>:auth<github:sergot>
===> Installing: OpenSSL:ver<0.1.21>:auth<github:sergot>
===> Installing: IO::Socket::SSL:ver<0.0.1>:auth<github:sergot>
===> Installing: IO::Capture::Simple
===> Installing: Test::Util::ServerPort:ver<0.0.1>:auth<github:jonathanstowe>
===> Installing: HTTP::UserAgent:ver<1.1.46>:auth<github:sergot>
===> Installing: WWW:ver<1.005003>
```

Note that some modules have many dependencies, and those dependencies may have dependencies of their own. If one of those dependencies have non-passing tests, nothing will be installed.

Failed tests doesn't *necessarily* mean that the module in question is broken, as Raku is in constant evolution, and module authors may not keep up with every change. But it may also m,ean exactly that, that the module *is* broken.

It is possible to force installation even if the tests failed:

```
zef install --force WWW
```

But it isn't a good idea without knowing *why* the tests failed.

12.2.5. zef depends

Use `zef depends` for a list of dependencies for a given module. This list is recursive, i.e. it follows the dependencies of the dependencies and so on.

12.2.6. zef upgrade

Use `zef upgrade` to upgrade to the newest versions of the specified module(s). Note that this function is beta.

It will try to upgrade all the installed modules, if used without arguments.

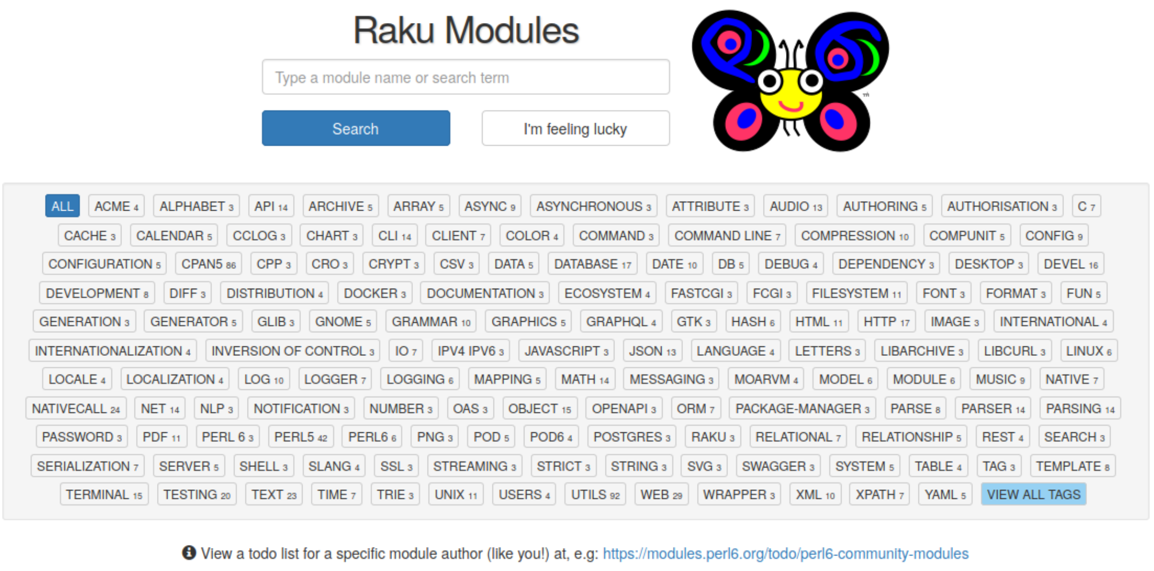
12.2.7. zef uninstall

Use `zef uninstall` to remove an installed module.

Note that it will only remove what you ask for. Any modules originally installed because of dependecies will not be affected.

12.2.8. Web Search

`zef search` isn't the best way to browse modules. But we can use the Raku Modules website at https://modules.raku.org/:


Raku Modules

Type a module name or search term

Search | I'm feeling lucky

ALL ACME 4 ALPHABET 3 API 14 ARCHIVE 5 ARRAY 5 ASYNC 9 ASYNCHRONOUS 3 ATTRIBUTE 3 AUDIO 13 AUTHORING 5 AUTHORISATION 3 C 7
CACHE 3 CALENDAR 5 CCLOG 3 CHART 3 CLI 14 CLIENT 7 COLOR 4 COMMAND 3 COMMAND LINE 7 COMPRESSION 10 COMPUNIT 5 CONFIG 9
CONFIGURATION 5 CPAN5 86 CPP 3 CRO 3 CRYPT 3 CSV 3 DATA 5 DATABASE 17 DATE 10 DB 5 DEBUG 4 DEPENDENCY 3 DESKTOP 3 DEVEL 16
DEVELOPMENT 8 DIFF 3 DISTRIBUTION 4 DOCKER 3 DOCUMENTATION 3 ECOSYSTEM 4 FASTCGI 3 FCGI 3 FILESYSTEM 11 FONT 3 FORMAT 3 FUN 5
GENERATION 3 GENERATOR 5 GLIB 3 GNOME 5 GRAMMAR 10 GRAPHICS 5 GRAPHQL 4 GTK 3 HASH 6 HTML 11 HTTP 17 IMAGE 3 INTERNATIONAL 4
INTERNATIONALIZATION 4 INVERSION OF CONTROL 3 IO 7 IPV4 IPV6 3 JAVASCRIPT 3 JSON 13 LANGUAGE 4 LETTERS 3 LIBARCHIVE 3 LIBCURL 3 LINUX 6
LOCALE 4 LOCALIZATION 4 LOG 10 LOGGER 7 LOGGING 6 MAPPING 5 MATH 14 MESSAGING 3 MOARVM 4 MODEL 6 MODULE 6 MUSIC 9 NATIVE 7
NATIVECALL 24 NET 14 NLP 3 NOTIFICATION 3 NUMBER 3 OAS 3 OBJECT 15 OPENAPI 3 ORM 7 PACKAGE-MANAGER 3 PARSE 8 PARSER 14 PARSING 14
PASSWORD 3 PDF 11 PERL 6 3 PERL5 42 PERL6 6 PNG 3 POD 5 POD6 4 POSTGRES 3 RAKU 3 RELATIONAL 7 RELATIONSHIP 5 REST 4 SEARCH 3
SERIALIZATION 7 SERVER 5 SHELL 3 SLANG 4 SSL 3 STREAMING 3 STRICT 3 STRING 3 SVG 3 SWAGGER 3 SYSTEM 5 TABLE 4 TAG 3 TEMPLATE 8
TERMINAL 15 TESTING 20 TEXT 23 TIME 7 TRIE 3 UNIX 11 USERS 4 UTILS 92 WEB 29 WRAPPER 3 XML 10 XPATH 7 YAML 5 VIEW ALL TAGS

View a todo list for a specific module author (like you!) at, e.g: https://modules.perl6.org/todo/perl6-community-modules

Figure 13. modules.raku.org

Exercise 12.1

Click on some of the tags to get a sense of structure.

Do some searches.

12.3. Using Modules (use)

We tell the prorgram that we'll use a module with the `use` keyword:

```
use DBIish;      # Top level namespace
use My::Module; # Two levels of namespaces
```

If we don't specify a version, Raku will load the newest one - if there are more than one version installed. (The notion of «newest» is based on the version number only, so if you have two modules with the same name (differentiated by the «auth» field) which is legal, the one with the higest version number will be used. Updating the modules (with `zef upgrade`) may change what is considered the newest version.)

If you want to ensure that a specific version of the module is used, specify it like this:

```
use DBIish:ver<0.5.17>;
```

Insisting on a specific version that isn't installed will fail:

```
use DBIish:ver<0.5.18>;
Could not find DBIish:ver<0.5.18> at line 1 in:
    /home/arne/.perl6
    /usr/local/share/perl6/site
    /usr/local/share/perl6/vendor
    /usr/local/share/perl6
    CompUnit::Repository::AbsolutePath«94631019616016»
    CompUnit::Repository::NQP«94631041192904»
    CompUnit::Repository::Perl5«94631041192864»
  in any statement_control at /usr/local/share/nqp/lib/Perl6/Grammar.moarvm line 1
```

It is helpful in displaying a list of locations for installed modules.

«CompUnit::Repository» mainly handles precompilation. See the «Advanced Raku» course for more information.

You can check if a module is installed with REPL:

Not installed:

```
> use Data::TextOrBinary
Could not find Data::TextOrBinary at line 1 in: ...
```

Installed:

```
> use WWW
Nil
```

Or on the command line:

```
$ raku -MData::TextOrBinary -e "say 'ok';"
===SORRY!===
Could not find Data::TextOrBinaryat line 1 in: ...
```

```
$ raku -MWWW -e ""say 'ok';"
ok
```

You can drop the `-e` part, but that will give you REPL mode if the module is installed. And you can use the module:

```
$ raku -MWWW
>
```

Exercise 12.2

Install the module «Math::Trig» from CPAN, and write a short program using something from it.

12.4. Writing Modules

In chapter Chapter 15, *Writing a Module* we'll show how to place modules locally, bypassing the need for module installation with `zef`.

In the «Advanced Raku» course we'll show how to write a module following the rules, so that it can be uploaded to CPAN for public usage and installation with `zef`.

Chapter 13. Files and Directories

13.1. Reading Files

The normal way of reading a file is open it first, read the content, and close the file. We can of course do that, but we don't have to.

13.1.1. IO.lines

Use `IO.lines` on a file name to get the content, as a lazy list of lines.

Read a specified file, and display lines with an «a» in them:

File: echo-file-contains

```
sub MAIN ($file-name)
{
  for $file-name.IO.lines -> $line
  {
    say $line if $line.contains("a");
  }
}
```

⚠ The implicit filehandle is closed when we have read the entire content of the file. So beware of situations with early exit of the loop (e.g. a program that looks for a certain string in the file, and does a `last` when it is found).

Filehandles are automatically closed when they go out of scope, but in a large program the end of the scope may be a long way off.

We can use `grep` (and move `contains` there) to avoid the loop:

File: echo-file-contains2

```
sub MAIN ($file-name)
{
  .say for $file-name.IO.lines.grep( *.contains("a") );
}
```

I have used a loop to get newlines after each line we display, but we can avoid that as well by using `join` on the list:

File: echo-file-contains3

```
sub MAIN ($file-name)
{
  $file-name.IO.lines.grep( *.contains("a") ).join("\n").say;
}
```

Remember that we can (and indeed must) omit the curlies in the `grep` argument if the first argument is a Whatever Star.

We could have written it like this instead: `grep({.contains("a")})`.

Take a little time looking at the three versions. Which of them is easiest to understand? And if you like another one better, why?

13.1.2. limit

If you are only interested in a certain part of the file, specify the maximum number of lines to read like this:

```
> $file-name.IO.lines(10);
```

13.1.3. IO.words

This is identical to `IO.lines`, but the content is returned one word at a time (instead of one line at a time).

Note that `words` may not do what you want, as descibed in section 7.4, "words".

13.1.4. lines

We have used `lines` as a method on an `IO`-object in the previous sections, but we can use `lines` as a procedure without arguments as well. This will read the content of the file(s) specified on the command line:

File: echo-all

```
.say for lines;
```

No need for `MAIN`, and it will handle as many files as we want:

```
$ raku echo-all /etc/*
```

We will get a warning if one or more files is a directory:

```
'NPW18/' is a directory, cannot do '.open' on a directory in block <unit> at echo-all
line 3
```

Note that used like this we have no way of getting the *file names*, or when one file ends and the next begins. (But see section 13.6.1, “$*ARGFILES” for a workaround.)

If we invoke the program without arguments, it will wait for input, and copy it back verbatim. Use <Control-c> to exit.

We can pipe input to it if we want, and these lines are equal:

```
$ raku echo-all file1.txt file2.txt
$ cat file.txt file2.txt | raku echo-all
```

We can make the `echo-file-contains3` program shorter:

File: echo-grep

```
lines.grep( *.contains("a") }).say;
```

We can use a slurpy array to get them file names:

```
sub MAIN (*@files)
{
  lines.grep({ .contains("a") }).say;
}
```

We have added `MAIN` for the usage message only. The arguments (in `@files`) are ignored.

⚠ Note that the slurpy argument allows zero arguments, so this program will behave in the same way as «echo-grep». That means that the usage message will never be triggered (and as such the use of `MAIN` is useless).

See section 10.14, “* (Slurpy Operator)” for information on how to fix it.

13.2. slurp

Use `slurp` to read the whole file at once:

```
> my $content = slurp "/home/raku/bin/echo-file";
```

All strings in Raku are in Unicode, but it is possible to read (and convert) files with other encodings:

```
> my $contents = slurp "/home/arne/echo.c", enc => "latin1";
```

More about supported encodings: https://docs.raku.org/routine/encoding

13.3. open / close

We can open the file (with `open`), do something with it, and close it (with `close`) afterwards:

This program will read a file specified as argument, and print all the lines containing the letter «a»:

File: echo-file-MAIN

```
sub MAIN ($file-name)
{
  my $fh = open $file-name;

  for $fh.lines -> $line
  {
    say $line if $line.contains("a");
  }

  $fh.close;
}
```

13.3.1. lines

I have used `lines` on an open file handle to read the content, one line at a time.

Exercise 13.1

Write a file conversion program. Input is in latin1 (iso-latin-1), and output is in Unicode (utf-8).

Hint: Do not try writing to a file (as we have not shown how to do that yet). Writing to the screen (STDOUT) is ok, and we can use the shell to save it for us like this:

```
$ raku isolatin2unicode isolatinfile > unicodefile
```

13.4. INPUT OUTPUT - IO

On a Unix-like system we have the following predefined filehandles:

Name	Filehandle	Description
STDIN	`$*IN`	Standard input

STDOUT	`$*OUT`	Standard output
STDERR	`$*ERR`	Standard error

13.4.1. note

`note` prints to STDERR (the same as `$*ERR.say`). Do not use it on a filehandle!

Without \n	With \n	To	Stringification
	`note`	*ERR	`.gist`

`note` may surprise you in REPL:

```
> note False
False
True
```

The `False` output was sendt to STDERR. As nothing was displayed on STDOUT by the code, REPL displayed the result of the last statement. `note` was successful (as we haven't closed STDERR), and returned `True`.

13.5. Writing Files

We can expand the file conversion program to write to a file, if given. The first argument to the program is the file name to read from, as before, and a second one (if given) is the file name to write to.

If we don't specify the second argument, it will print to the screen as before:

```
$ raku isolatin2unicode4 isolatinfile unicodefile
$ raku isolatin2unicode4 isolatinfile > unicodefile
```

We can use a filehandle, open the file in write mode, and use say on the filhandle:

```
> my $fh = open :w, '/tmp/some-file.txt';
> $fh.say("Hello");
> $fh.close;
```

Note that `$fh.say` requires parens or colon syntax (e.g. `$fh.say: "Hello"`).

Remember the adverbial syntax (see section 10.13.6, "Adverbs"); `:w` is the same as `w => True`.

But we'll use the `spurt` command instead. It is the opposite of `lines`, as it writes all the text we give it to the specified file.

When we create a file, the file permissions is copied from the system umask on Unix like systems. Windows doesn't support umask values or file permissions.

It is not possible to change the mode in the `open` or `spurt` calls, but see section the description of `chmod` in the «Advanced Raku» course.

13.5.1. spurt

Use `spurt` to write to a file, with automatic open and close.

File: isolatin2unicode4

```
sub MAIN ($file-in, $file-out = "")
{
  $file-out
    ?? spurt $file-out, slurp $file-in, enc => "latin1"
    !! say slurp $file-in, enc => "latin1";
}
```

We could have written `$file-out = Nil` instead, but an empty string is ok.

No error checking of any kind, so what can go wrong?

spurt overwrite

Note that `spurt` happily overwrites existing files, without warning.

We can instruct it to fail if the file exists:

```
spurt $out, :createonly, slurp $in, enc => "latin1";
```

You can add parens if you are confused:

```
spurt($out, :createonly, slurp($in, enc => "latin1"));
```

spurt append

`spurt` also have append mode, where the text is added to the end of an existing file:

```
spurt $file-out, :append, slurp $file-in, enc => "latin1";
```

This is useful when writing to log files.

13.5.2. prompt Revisited

`prompt`, as presented in 6.6.1, “prompt” is the same as `$*IN.get` with an optional text output.

```
sub prompt-reimplemented ($message = "")
{
  $*OUT.say $message if $message;
  return $*IN.get;
}
```

File: prompt

```
my $name = prompt "What's your name? ";
say "Hi, $name! Nice to meet you!";
```

13.6. get

`get` reads a single line from the specified filehandle. It returns `Nil` if no more input is available.

Read one line from standard input:

```
my $line = $*IN.get;
```

Read one line from a file:

```
my $fh = open 'filename';
my $line = $fh.get;
$fh.close;
```

13.6.1. $*ARGFILES

A standalone `get` (without using it on a filehandle) behaves just like `lines`.

It will read from the files given on the command line, and if none are given from `$*IN` instead.

The magic is performed by `$*ARGFILES` behind the scenes.

We can use `handles` on `$*ARGFILES` to get a the filehandles for each argument:

File: argfile-handle

```
say $_ for $*ARGFILES.handles;
```

Running it:

```
$ raku argfile-handle person args ack6
person -> IO::Handle<"person".IO>(opened)
args -> IO::Handle<"args".IO>(opened)
ack6 -> IO::Handle<"ack6".IO>(opened)
```

Specifying a non-existing file causes the program to crash:

```
$ raku argfile-handle person sjsjsjsjsjs
person -> IO::Handle<"person".IO>(opened)
Failed to open file /home/raku/sjsjsjsjsjs: No such file or directory
  in block <unit> at ./argfile-handle line 3
```

⚠ Note that the handles are reported as open, even though they are not - if we try to read from them. The error message we get if we try to read is «Cannot do 'get' on a handle in binary mode», and that is wrong.

What actually happens is that the handles are open when the `handles` method is called, but they are closed afterwards:

File: argfile-handle2

```
my @handles = $*ARGFILES.handles;

for @handles { say $_ }
```

Running it:

```
$ raku argfile-handle2 person args ack6
IO::Handle<"person".IO>(closed)
IO::Handle<"args".IO>(closed)
IO::Handle<"ack6".IO>(closed)
```

We can use `handles` to write a zip program, that takes one line from each file given as argument, and continues until everything has been written:

File: zip-merge

```
my @handles = $*ARGFILES.handles;

.open for @handles;

while @handles
{
  my $handle = @handles.shift;
  say $handle.get;
  @handles.push($handle) unless $handle.eof;
}
```

Note that we have to open the files manually for this to work.

```
$ raku zip-merge person-say3 repeat repeat2
```

The not-so-subtle naming of the program may help you remembering the `zip` operator (See the «Advanced Raku» course.)

We cannot use that, as it would stop when the file with the smallest number of lines is finished. But we can use the `roundrobin` operator (Which we'll cover in the «Advanced Raku» course).

It is extremely hard to get it right, but this one-liner works:

File: roundrobin

```
$*ARGFILES.handles.eager».open».lines.&roundrobin.flat.map: *.put
```

`>>` is a Hyper Operator (Which we'll cover in the «Advanced Raku» course). It works in parallel on the elements. I'll leave it as an exercise to the reader to figure out *why* it works.

(This code snippet is written by Brad Gilbert. See https://stackoverflow.com/questions/53639771/perl-6-argfiles-handles-in-binary-mode)

If you use `$*ARGFILES` inside the special `MAIN` function, it will read from `$*IN`. (This applies to version 6.d (and later).

13.7. Temporary Files

Temporary files should be placed in a suitable location, so as not to clutter a directory with normal content. A directory meant for temporary files may also employ automatic cleanup from time to time.

Use normal file operations (create, write, read, remove).

13.7.1. tmpdir

Use the `$*SPEC.tmpdir` or `$*TMPDIR` dynamic variables to locate the system temporary directory. They will default to the current directory if the system was unable to locate it.

It is good practice to delete temporary files afterwards, but program crashes or premature termination may happen before the programs do their clean up.

13.7.2. unlink

Use `unlink` to remove a file, link or symbolic link. Directories can only be removed with `rmdir` (see section 13.10.7, "rmdir").

```
> unlink(<A B C D>);
> unlink "A".IO, "B".IO, "C".IO, "D".IO;
```

Because of file system limitations, the return list includes all files, except those that caused problems (missing permissions, or a directoy).

As a metod it returns `True` on success, and a `X::IO::Unlink` failure otherwise:

```
> "A".IO.unlink;
```

Removing a non-existing file gives `True`.

We can get the error message in this compact way:

```
> say .exception.message without 'bar'.IO.unlink;
Failed to remove the file [⋯] illegal operation on a directory
```

13.7.3. getc

Use `getc` to read a single character from the specified filehandle. The subroutine form defaults to $*ARGFILES if used without a handle, and that again defaults to STDIN if no files were specified on the command line.

It returns `Nil` if no more input is available, and throws an exception if used on a filehandle in binary mode.

File: getc

```
my $char;

repeat
{
  print "> "; $char = getc;
  say "Character: $char";
}
while $char ne "Q";
```

It goes in a loop, until we enter the «Q» character.

Running it:

```
$ raku getc
> asdQw
Character: a
> Character: s
> Character: d
> Character: Q
```

It doesn’t do anything before we press return, as the terminal is set to Buffered. Then it gets all the characters, printing the «>» prompt after each one as we can see.

On a Linux system we should have been able to turn off buffering for a program with the `stdbuf` command like this:

```
$ stdbuf --input=0 raku getc
```

But it doesn’t work, on my computer at least.

Unicode and *combining characters* (see section 7.1.2, “Combining Characters”) are also an issue, as they come 'after* the base character. This means that `getc` will wait for at least two characters, before returning the first one. If the second one isn’t a combining character, the first one is returned. If it is a combining character, `getc` will continue reading characters as long as they all are combining character (as we can have many of them) - or we encounter the end of the file.

13.7.4. readchars

Use the `readchars` method to read up to the specfied number of characters (graphemes) from the filehandle. It will throw an excecption if the filehandle has been opened in binary mode.

File: readchars

```
my $file = $*TMPDIR.add('foo.txt');

$file.IO.spurt: "This is a test...\n" x 25;

given $file.IO.open
{
  say "A:" ~ .readchars: 5; # OUTPUT:
  say "B:" ~ .readchars: 90;
  say "C:" ~ .readchars;
  .close;
  $file.unlink;
}
```

If we don't specify the number of characters, an implementation specific number is used. Rakudo uses `$*DEFAULT-READ-ELEMS` which is `65536`.

13.8. File tests

We really should have added error detection (and recovery) in our programs. Trying to *read* from a non-existing file will fail, and terminate the program.

There are quite a few file test we can apply to a file, through an `IO`-object (`"file-name".IO.d`), on a filehandle (`$fh.d`) or with smartmatch (`"file-name".IO ~~ :d`):

Method	Description	Return value	Non-existing
`d`	Is it a *directory*?	`True`/`False`	`fail`
`e`	Does it *exists*?	`True`/`False`	`False`
`f`	Is it a *file*?	`True`/`False`	`fail`
`l`	Is it a *symlink*?	`True`/`False`	`fail`
`r`	Is it *readable*?	`True`/`False`	`fail`
`rw`	Is it *readable* and *writeable*?	`True`/`False`	`fail`
`rwx`	Is it *readable*, *writeable* and *executable*?	`True`/`False`	`fail`
`s`	File size (in bytes)	Integer	`fail`
`w`	Is it *writeable*?	`True`/`False`	`fail`
`x`	Is it *executable*?	`True`/`False`	`fail`
`z`	File size zero?	`True`/`False`	`fail`

Most of them will `fail` (with `X::IO::DoesNotExist`) if the file doesn't exist.

Note that `s` and `z` can give non-zero results if used on a directory, but this is depending on the operating system.

Examples (given that the file exists, and that we have done `my $fh = "/tmp/A".IO.open` first):

IO method	Filehandle method	Smartmatch	Result
`"/tmp/A".IO.d`	`$fh.d` *(see note)*	`"/tmp/A".IO ~~ :d`	`False`
`"/tmp/A".IO.e`	`$fh.e`	`"/tmp/A".IO ~~ :e`	`True`
`"/tmp/A".IO.f`	`$fh.f`	`"/tmp/A".IO ~~ :f`	`True`
`"/tmp/A".IO.s`	`$fh.s`	`"/tmp/A".IO ~~ :s`	126976

Note that we cannot *open a directory*, so checking if a *filehandle* is a directory can never return `True`.

13.8.1. File Tests in Signatures

Let us revisit «echo-file-contains3» from section 13.1.1, "IO.lines". If we specify a non-existing file we get a run time error:

```
$raku echo-file-contains3 SSSS
Failed to open file /home/raku/SSSS: No such file or directory ...
```

We can add a type check, and a `multi` like this:

File: echo-file-contains4

```
multi sub MAIN ($file-name where $file-name.IO.f)
{
  $file-name.IO.lines.grep( *.contains("a") ).join("\n").say;
}

multi sub MAIN (*@args)
{
  say "Oh, no! Please specify one file.";
}
```

The second `multi MAIN` cathes situations where we have specified anything else than one argument that is an existing file.

The error message given by «echo-file-contains3» is quite good, and program termination can be the right thing to do.

13.9. Binary Files

Newline conversion (see section 6.1, "Newlines") and Unicode normalization (see section 7.1, "Unicode") screws up reading (and writing) of binary files.

When we read in Unicode mode, the program will chocke on illegal sequences. We can avoid that by using the `utf8-c8` encoding, as it passes bytes through unchanged.

But binary files should be read in binary mode:

```
my $buffer = slurp $filename, :bin;
```

13.9.1. Buf

When we read from a file in binary mode, we get a `Buf` (Buffer) in return.

We can either use `slurp`:

```
my $buffer = slurp $filename, :bin;
for @$buffer { ... }
```

Or do it manually, with `open` and `read`.

read

`read` reads the specified number of bytes:

```
if my $fh = open $path, :bin
{
  my Buf $buffer = $fh.read( $count );
  my $third_byte = $buffer[2];
}
```

Note that `read` works for non-binary files as well, but doing so can cause the breaking up of Unicode characters.

Hex-dumping a file:

File: file-show

```
constant NL  = 9252.chr; # This is the unicode "N/L" symbol
constant BOX = 9617.chr; # This is a unicode gray box

sub MAIN ($file where $file.IO.r)
{
  my $fh = open $file, :bin;

  while my Buf $buf = $fh.read(10)
  {
    my $ascii = "";
    my $elems = @$buf.elems;
    for @$buf -> $byte
    {
      print $byte.fmt("%02X ");
      if $byte eq any(10,13)
      {
        $ascii ~= NL;
      }
      else
      {
        $ascii ~= 31 < $byte < 127 ?? $byte.chr !! BOX;
      }
    }
    print "   " x 10 - $elems; # Fill the last line
    say "| $ascii";
  }

  $fh.close;
}
```

13.9.2. Blob

Binary data can also be stored in a `Blob` («Binary Large OBject»).

```
> my $blob = Blob.new([1, 2, 3]);
Blob:0x<01 02 03>

> my $blob = Blob.new([255, 2, 3]);
Blob:0x<ff 02 03>
```

The values are in the range `0 .. 255`. Values outside that range will be truncated (by applying `% 256` on them):

```
> my $blob = Blob.new([256, 2, -1]);
Blob:0x<00 02 ff>
```

We can rewrite «file-show» to use a `Blob`:

Change this line:

```
while my Buf $buf = $fh.read(10)
```

to this:

```
while my $buf = Blob.new($fh.read(10))
```

(It is available as «file-show-blob».)

Exercise 13.2

Write a file comparison program. It takes two file names, and does a binary comparison.

Usage:

```
./file-equal enum-red enum-redX
No such file enum-redX

$ raku file-equal enum-red enum-red
The files are equal

$ raku file-equal enum-red enum-fixed
Files differ (different sizes)

$ raku file-equal hello-usage hello-usage2
Files differ
```

13.9.3. Writing a Binary File

Writing a binary file with `Buf` and `write`:

File: write-buf

```
unit sub MAIN ($file-name);

if my $fh = open $file-name, :w, :bin
{
  my $buf = Buf.new: 82, 97, 107, 117, 100, 111, 10;
  $fh.write: $buf;
}
```

And as a `blob`:

File: write-blob

```
unit sub MAIN ($file-name);

if my $fh = open $file-name, :w, :bin
{
  my $blob = Blob.new: 82, 97, 107, 117, 100, 111, 10;
  $fh.write: $blob;
}
```

A little test:

```
$ raku write-buf X1
$ raku write-blob X2
$ raku file-equal X1 X2
The files are equal
```

`write` can be used on non-binary files (text files), but this can cause illegal Unicode sequences. Or illegal sequences in whatever text encoding you want to use. (Note that this can be used to write text files with encodings not supported by Raku, but it is a better idea to implement the support instead.)

13.9.4. Detecting Binary Files

There is no built in way of detecting if a file is binary or not, but we can use the module `Data::TextOrBinary`.

Install the module `Data::TextOrBinary` if it isn't installed alredy. (Possibly with «sudo zef» instead, depending on your setup):

```
zef install Data::TextOrBinary
```

See also section 12.2, "Module Administration with zef".

File: binary

```
use Data::TextOrBinary;

sub MAIN ($file)
{
  if $file.IO.d
  {
    say "Directory.";
  }
  elsif $file.IO.e
  {
    is-text($file.IO)
      ?? say "Text file."
      !! say "Binary file.";
  }
  else
  {
    say "File doesn't exist.";
  }
}
```

The module works by reading the first 4096 bytes from the file, and then looking for characters that doesn't appear in text files.

We can specify the number of bytes to read with the «test-bytes» argument:

```
my $text = is-text($filename.IO, test-bytes => 8192);
```

See https://github.com/jnthn/p6-data-textorbinary for details.

Using it:

```
$ raku binary axxxx
File doesn't exist.

$ raku binary num-add-err
Text file.

$ raku binary _old/
Directory.

$ raku binary /bin/false
Binary file.
```

Note that the program may report some pdf files as text (and this book is an example of that), if the binary content isn't placed up front. Increasing the `test-bytes` value fixes this.

13.10. Directories

Directories are places where we store files. And directories.

13.10.1. %*ENV<PATH>

The most important directories are the path, the «PATH» environment variable, available as `%*ENV<PATH>`. It is a string containing a colon separated list of directories where the shell looks for a program to execute, in the specified order.

```
> say %*ENV<PATH>;
/home/arne/bin:/home/arne/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

With better formatting:

```
> say %*ENV<PATH>.split(":").join("\n");
/home/arne/bin
/home/arne/.local/bin
/usr/local/sbin
/usr/local/bin
/usr/sbin
/usr/bin
/sbin
/bin
/usr/games
/usr/local/games
/snap/bin
```

13.10.2. dir

Use `dir` without arguments to give us the entire content (without the special directories «.» and «..») of the current directory, as a list of `IO`-objects:

```
> dir
("src".IO "RakuExplained.html".IO)
```

We can specify a directory:

```
> chdir "/"
> dir "home"
("home/arne".IO)

> dir "/home"
("/home/arne".IO)
```

If we give a relative directory (not starting with «/») we get a list realtive to the current directory.

13.10.3. indir

Use the `indir` command to execute code in the specified directory, with the current directory set to it.

We can use `indir` to list all the programs available for us (in the path). The path is not recursive, so directories inside the directories specified in the path are ignored.

We start with the path, splitting it (on the colon separator) into each part and iterate over it:

```
for %*ENV<PATH>.split(":") -> $directory
{
```

Inside the loop we start with skipping non-existing directories, as we are allowed to have junk in the path (and indeed usually have):

```
  next unless $directory.IO.d; # Is this a directory?
```

Then we have a new loop. We use `indir` on the directory (`$directory`) and runs the `dir` command in it (specified as `&dir`, a reference). The `dir` command gives us a list of files (and directories) in this directory. We add `sort` to give a sorted list. It is a case sensitive sort, but that's ok:

```
  for indir($directory, &dir).sort -> $file
  {
```

The we skip directories, as we are only looking for files:

```
    next if $file.d;
```

And finally we display the file (wuith full path) if it is executable by the current user:

```
    say "$directory/$file" if $file.x;
  }
}
```

Note that `say` converts the `IO.Path` objects to strings for us.

The whole program:

File: list-path

```
for %*ENV<PATH>.split(":") -> $directory
{
  next unless $directory.IO.d; # Is this a directory?

  for indir($directory, &dir).sort -> $file
  {
    next if $file.d;
    say "$directory/$file" if $file.x;
  }
}
```

Run it to have a look at the programs available for you.

13.10.4. $*CWD

Raku keeps the current directory in the special `$*CWD` variable. It is not advisable to change it manually.

`indir` does indeed change the value of `$*CWD` automatically, but will also change it back after running the command in the specified directory.

Exercise 13.3

The Unix program `which` gives us the full path for a given program. E.g.:

```
$ which less
/usr/bin/less

$ which pwd
/bin/pwd
```

Write this program in Raku (and call it «which6»). Use «list-path» as a starting point.

It should report the first match only.

Exercise 13.4

Having duplicate programs (programs with the same name) in your path may indicate a problem, as the first one (in the path) will be executed - and that may not be the one you want.

Write a program traversing the path, reporting duplicates.

```
$ raku check-path
fido
- /usr/bin/fido
- /opt/bin/gecco/fido
false
- /bin/false
- /home/raku/fakebin/false
```

Use «which6» as a starting point.

Note that most shells have several built-in commands that are used instead of similar programs in the path.

Exercise 13.5

Extend «check-path», comparing the files to see if they are the same. Either a symbolic link, a hard link, or an exact copy.

Tip: Reuse «file-equal» from Exercise 13.2.

It is ok to assume that we have only two versions of the same program. But if you want to allow for more than two versions, it is ok to only compare the second and third with the first one.

Sample output (abridged):

```
$ raku check-path-duplicates
print
- /usr/local/bin/print
- /usr/bin/print - Not equal
touch
- /usr/bin/touch
- /bin/touch - Equal
Found 19 duplicates and 12 different programs.
```

13.10.5. Recursion with indir

`indir` can be used recursively, with some care.

Here is a program that lists every readable file in every readable directory starting with the current one, recursively:

File: indir-loop

```
unit sub MAIN;

my @dirs = ".";

while (@dirs)
{
  check-dir(@dirs.shift);
}

sub check-dir ($dir)
{
  say "Reading dir: $dir";
  for indir($dir, &dir).sort -> $current
  {
    next unless $current.IO.r; # Skip files/directories we cannot read

    $current.IO.d
      ?? @dirs.push("$dir/$current")
      !! say "File: $dir/$current";
  }
}
```

It sorts the files (and directories), and goes for width first (showing all the files in a directory, before traversing the directories). We do this with a list of directories to check, and add new ones to the end when we encounter a new one.

I have chosen to program this as a loop, and not recursively, as I want to list the files before the directories.

Here we have a recursive version:

File: indir-recursive

```
unit sub MAIN;

check-dir(".");

sub check-dir ($dir)
{
  say "Reading dir: $dir";
  for indir($dir, &dir).sort -> $current
  {
    next unless $current.IO.r; # Skip files/directories we cannot read

    $current.IO.d
      ?? check-dir("$dir/$current")
      !! say "File: $dir/$current";
  }
}
```

The sorting order is not very nice, but we can fix that by sorting the files before the directories:

```
  for indir($dir, &dir).sort({ +$^a.IO.d ~ $^a cmp +$^b.IO.d ~ $^b }) -> $current
```

The sort applies `IO.d` to the file, and converts the result to a number (0 or 1) with a `+` prefix. Then it attachs the file name. The result is that all files are prefixed with «0», and directories with «1». So we get the files (in sorted order), before the directories (also in sorted order). The output should be exactly the same as for «indir-loop».

The whole program is available as «indir-recursive2».

Exercise 13.6

The Unix programs «grep» can be used to search for strings in files, but the syntax for the command line arguments is not user friendly.

The «ack» program was written to make this task easier, and it has features suitable for programmers. (Look it up, as it really is useful.)

Write a program «ack6» that searches all non-binary files recursively from the current directory, looking for the specified string.

Tip: Start with «indir-recursive2».

13.10.6. mkdir

Use `mkdir` to create one directory:

```
> mkdir "misc";
> mkdir "misc".IO;
> "misc".IO.mkdir;
```

It can take an optional permission argument (or mode), that is best specified in octal form, e.g. `mkdir "misc", 0o777`. This value is ignored on Windows.

```
> mkdir "misc", 0o777;
[a]
```

Note that the mode value will be OR'ed with the system «umask value», as done with the «mkdir» program. There is no way to override this.

It can create a path as well (similar to the Unix «mkdir -p» command):

```
> mkdir "a/b/c/d/e";
```

13.10.7. rmdir

Use the `rmdir` function to remove one or more directories. It will only remove empty directories, and returns a list of diretories actually removed:

```
> rmdir(<a b c d e>);
[a b]
```

When used as a method it will return `True` if it was able to remove the directory, and throw an `X::IO::Rmdir` exception if the directory cannot be removed.

```
> "a".IO.rmdir;  # -> True
> "c".IO.rmdir;  # -> Failed to remove the directory ...
```

Chapter 14. Date and Time

Raku has very good built in support for dates and times.

14.1. time

Use `time` to get the time in whole seconds since 1.1.1970 (the beginning of time in Unix, also known as the Epoch):

```
> time;  # -> 1542530698
```

The returned value is an `Int`.

This is the traditional Unix way of doing it, and it has been standardised by POSIX.

One second is a long time compared with CPU cycles, so it is pretty useless when timing code.

14.2. now

Use `now` to get the current time in seconds (with a fractional part) since 1.1.1970:

```
> now;  # -> Instant:1532015558.371171
```

The returned value is an object of type `Instant`.

14.3. Leap Seconds

If you care about leap seconds, note that `now` handles them, and `time` does not.

Leap Seconds are «extra» seconds that are inserted from time to time to correct for discrepancies.

The result is a slight difference between the values used by `time` and `now`. We can actually show it:

```
> say "{ time } - { now }";
1542532489 - Instant:1542532526.274499

> say "{ time } - { now.Int }";
1542532500 - 1542532537
```

It looks like we have a 37 second discrepancy. Let us make sure:

File: time-leap

```
my $diff; # We add all the differences

for ^100
{
  my $time = time;
  my $now  = now;

  $diff += ($now.Int - $time)
}

say "Number of leap seconds added after 1.1.1970: " ~ round($diff.sum/100);
```

I run the loop a 100 times to avoid the problem of `time` getting one value, and `now` getting the next second.

```
$ raku /time-leap
Number of leap seconds added after 1.1.1970: 37
```

If you want to know more about POSIX time (and its absence of leap seconds) start here: https://en.wikipedia.org/wiki/Unix_time

If you don't want to care, that's fine. As long as you do not compare values from `time` and `now` with each other!

14.4. Instant

Note that we must have an Instant object (and not a POSIX time value) if we want to deduce dates and such things from it.

14.4.1. Date

Use the `Date` method to get a `Date` object from an `Instant`.

A `Date` object stringifies to a date string with year, month and day like this:

```
> say now.Date;   # -> 2018-11-18
> say Date.today; # -> 2018-11-18
```

The year uses 4 digits, and the month and day 2 digits each.

We can make a `Date` object for any date by specifying it as a string, an array of `Int`s or a list of named arguments:

```
> my $date = Date.new("2018-10-01");
> my $date = Date.new(2018, 10, 1);
> my $date = Date.new(year => 2018, month => 12, day => 10);
```

There are a lot of methods we can use on `Date` objects. This is the most useful (and we start with `my $d = Date.new(2018,10,1)`):

Method	**Result**	**Description**
`year`	2018	The year
`month`	10	The month (1..12)
`day`	1	The day of the month (1..31)
`is-leap-year`	False	2018 is not a leap year
`day-of-month`	1	The same as `day`
`day-of-week`	1	The day in the week (1=Monday .. 7=Sunday)
`day-of-year`	274	The 274th day of the year
`days-in-month`	31	The number of days in the month
`week-number`	40	The week number (1..53)
`week-year`	2018	The year that the week number belongs to (see note below)
`week`	(2018 40)	A list with `week-year` and `week-number`
`weekday-of-month`	1	The number of times this day has occured this month (including this one)
`yyyy-mm-dd`	2018-10-01	The same as `Str` and `gist`

A week spanning two years belongs to the year that has most of it (or that has the Thursday). The week may belong to the previous year (in January) or the next year (in December):

```
> Date.new(2017,1,1).week;   # -> (2016 52)
> Date.new(2018,12,31).week; # -> (2019 1)
```

The following methods returns a new `Date` object (and we start with `my $d = Date.new(2018,10,10)`):

Method	**Result**	**Description**
`earlier(days => 2)`	2018-10-08	Subtract the given number of days
`earlier(week => 1)`	2018-10-01	Subtract the given number of weeks
`earlier(month => 2)`	2018-08-10	Subtract the given number of months
`earlier(year => 2)`	2016-10-11	Subtract the given number of years
`later(days => 2)`	2018-10-12	Add the given number of days
`later(week => 1)`	2018-10-17	Add the given number of weeks
`later(month => 2)`	2018-12-10	Add the given number of months

`later(year => 2)`	2020-10-10	Add the given number of years
`truncated-to('year')`	2018-01-01	Truncate to the first day of the year
`truncated-to('month')`	2018-10-01	Truncate to the first day of the month
`truncated-to('week')`	2018-01-08	Truncate to the first day of the week
`succ`	2018-10-11	The next day
`pred`	2018-10-09	The previous day

The argument to `earlier` and `later` can be given in singular or plural form: (day or days, week or weeks, month or months, year or years).

The methods taking an argument takes only one, but they can be stacked:

```
> my $date = Date.new(2018,10,10).later(years => 10).later(days => 4);
```

Exercise 14.1

The Unix command «cal» shows the current month if invoked without arguments. The current date is highlighted.

Implement it, as «cal6». Don't support arguments, and skip the highlighting.

Sunday is supposed to be the seventh day of the week, so fix that at the same time.

```
    January 2019
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
```

Exercise 14.2

Extend «cal6» so that it takes optional values for month and year. E.g.

```
$ raku cal6-param --month=2 --year=2020
```

14.4.2. DateTime

A `DateTime` object has the date (as a `Date` object) and in addition fields for second, minute and hour. The time is kept in UTC format (formerly known as GMT, or UK standard time) internally.

We can get a new `DateTime` object for the current time with `DateTime.now` - or an `Instant` object (with the `now` function):

```
> say DateTime.now;   # -> 2018-11-18T23:46:22.609982+01:00
> say now.DateTime;   # -> 2018-11-18T22:46:22.609982Z
```

If we print the values, we get different results, as shown above. The time is the same, but the way they are shown differ. We'll get back to that (time zones).

First we'll take a look at the `DateTime` constructor:

```
my $date-time = DateTime.new(year     => 2018,
                             month    => 10,
                             day      => 20,
                             hour     => 16,
                             minute   => 1,
                             second   => 10);

my Date $date = Date.now;
my $date-time2 =  DateTime.new($date,  hour     => 16,
                                       minute   => 1,
                                       second   => 10);

my $date-time3 =  DateTime.new(now);  # An Instant object
my $date-time4 =  DateTime.new(time); # An integer

my $date-time5 =  DateTime.new("2018-11-18T23:46:22.609982+01:00");
my $date-time6 =  DateTime.new("2018-11-18T22:46:22.609982Z");
```

Note that the seconds are the only field that can have a fractional part.

The date part is on the same format as for the `Date` object: «year-month-date» (with 4, 2 and 2 digits). Then a «T» before the time part. The time part: «hour:minute:seconds» (with 2, 2 and 2 digits). The seconds can have a fractional part as well. And the last part is the time zone. It is either the letter «Z» (as in «Zulu») indicating UTC - or a description of the timezone as an offset «hours:minutes» (2 and 2 digits). The time is shown in the local time zone, and we can get the UTC time by subtracting this difference

`DateTime` objects support the same methods as `Date` objects (the two tables in section 14.4.1, "Date"), as well as (and we start with `DateTime.new("2018-11-18T23:46:22.609982+01:00")`):

Method	**Result**	**Description**
`hour`	22	The hour
`minute`	46	The minutes
`second`	22.609982	The seconds, with fraction if any
`whole-second`	22	The seconds, truncated to an integer
`timezone`	3600	The time zone, as offset from UTC in seconds
`offset`	3600	The same as `timezone`

`offset-in-minutes`	60	The time zone, as offset from UTC in minutes
`offset-in-hours`	1	The time zone, as offset from UTC in hours
`Str`	2018-11-18T23:46:22.609982+01:00	The date and time as a string
`Instant`	`Instant`	An `Instant` object representing the time and date
`posix`	`Int`	The POSIX value (seconds since 1.1.1970)
`Date`	`Date`	The date part of the object. Beware of time zones
`utc`	`DateTime`	A new `DateTime` object, with the time zone set to UTC
`in-timezone`	`DateTime`	A new `DateTime` object, in the specified time zone
`local`	`DateTime`	A new `DateTime` object, with the local time zone

In addition to extended parameter support for these:

Method	**Result**	**Description**
`earlier(second => 2)`	2018-11-18T23:46:20.609982+01:00	Subtract the given number of seconds
`earlier(minute => 2)`	2018-11-18T23:44:22.609982+01:00	Subtract the given number of minutes
`earlier(hour => 2)`	2018-11-18T21:46:22.609982+01:00	Subtract the given number of hours
`later(second => 2)`	2018-11-18T23:46:24.609982+01:00	Add the given number of seconds
`later(minute => 2)`	2018-11-18T23:48:22.609982+01:00	Add the given number of minutes
`later(hour => 2)`	2018-11-19T01:46:22.609982+01:00	Add the given number of hours
`truncated-to('second')`	2018-11-18T23:46:22+01:00	Truncate to whole seconds
`truncated-to('minute')`	2018-11-18T23:46:00+01:00	Truncate to whole minutes
`truncated-to('hour')`	2018-11-18T23:00:00+01:00	Truncate to whole hours
`truncated-to('day')`	2018-11-18T00:00:00+01:00	Truncate to whole days

The value `2` is just an example.

The argument to `earlier` and `later` can be given in singular or plural form: (second or seconds, minute or minutes, hour or hours, day or days, week or weeks, month or months, year or years).

We also have `clone` that gives a copy of the `DateTime` object, with new values for the specified fields (if any). The fields are the same as for the `new` call:

```
> my $much-later = DateTime.now.clone(year => 2045);
```

14.4.3. Time Zones

The `new` constructor takes an optional argument `timezone => <seconds>`, where we specify the time zone as offset from UTC (or GMT) in **seconds**. If we don't specify it, the value of the dynamic variable `$*TZ` («Time Zone») is used.

```
> say $*TZ; # -> 3600 (in Oslo, Norway)
```

Note that `DateTime.now` adds the time zone for us, but `now.DateTime` does not.

14.4.4. Custom Formatter

We can also specify a custom formatter when we use `DateTime.new` or `DateTime.now`. Its job is to format the date and time when it is stringified:

File: datetime

```
sub custom-formatter (DateTime $dt)
{
  sprintf '%02d.%02d.%04d %02d:%02d:%02d', $dt.day, $dt.month, $dt.year, $dt.hour,
$dt.minute, $dt.whole-second;
}

say DateTime.now;
say DateTime.now(formatter => &custom-formatter);
say DateTime.now(formatter => &custom-formatter).later(year => 1);
```

Running it shows that the custom formatter is inherited by the new object:

```
$ raku datetime
2018-11-20T12:18:36.963003+01:00
20.11.2018 12:18:36
20.11.2019 12:18:36
```

14.4.5. From POSIX to Instant

We can get from a POSIX value to an Instant:

```
> my $intant = Instant.from-posix: $posix-time;
```

We could have used this fact in «time-leap»:

File: time-leap2

```
my $time = time;
my $now  = Instant.from-posix: $time;

say "Number of leap seconds added after 1.1.1970: " ~ $now.Int - $time;
```

And we get the same result as previously in this chapter:

```
$ raku text/code/time-leap2
Number of leap seconds added after 1.1.1970: 37
```

14.5. Timing

We can time the execution of a program or a part of it.

14.5.1. Timing Programs

On a Unix-like systems we can use the «time» program to see how long it takes to run a program:

```
$ time raku random-prime 10
7

real    0m0,138s
user    0m0,164s
sys     0m0,031s

$ time raku random-prime 10000
8089

real    0m1,419s
user    0m1,458s
sys     0m0,024s
```

(The «real» value is the actual time, «user» is the part of this used by the compiler, and «sys» is the time used in system calls (in the operating system).)

The «random-prime» program was introduced in section 10.14.2, "Random Primes Revisited". The important point is that the higher number you give as argument, the longer time it takes to finish.

Note that we time everything; the starting up of raku, reading the source file (and any modules), compiling the program and finally executing it.

The actual values will differ each time you run the program, depending on what else is going on at the same time. Another computer can give quite different values, so be careful with comparisons.

This method isn't very useful when we want to compare things.

14.5.2. Timing Code

We can time the execution of code inside the compiler itself:

```
my $start = now;
# do-something;
say "Time used: { $start - now }";
```

If we are only going to time one block, we can write this as:

```
# do-something;
say "Time used: { now - INIT now }";
```

`INIT` is a program execution phaser (or a block) that is executed just after the program has been compiled, and before it is executed.

We can use phasers to execute code automatically at different times. There are quite a lot of them. See https://docs.raku.org/language/phasers for details.

As when using «time», the code should be timed several times, as the running times will differ. The actual values are not very useful, but they can be compared with other values - and used to compare different implementations.

We can make a function of it:

File: time-me (partial)

```
sub time-me (&code, $iterations = 100)
{
  my @time;

  for ^$iterations
  {
    my $start = now;
    &code();
    my $stop = now;
    @time.push($stop - $start);
  }

  return @time.sum / @time.elems;
}
```

Then some tests, where we increment an anonymous state variable (which we will decribe in section 16.6.2, "$ / @ / % (Anonymous State Variable)". They keep their value between calls, essentially working as counters in this code:

```
sub a
{
  $++ for 10000;
}

sub b
{
  ++$ for 10000;
}

say "a: " ~ time-me(&a, 10000);
say "b: " ~ time-me(&b, 10000);
```

Running it:

```
> $raku time-me
a:4.481019762283224e-05
b:4.395750332005312e-05
```

Scientific notation is hard to read, especially to compare. So we add an optional multiplier:

```
sub time-me (&code, $iterations = 100, $multiplier = 1)
```

```
return $multiplier * @time.sum / @time.elems;
```

```
say "a:" ~ time-me(&a, 10000, 1000);
say "b:" ~ time-me(&b, 10000, 1000);
```

Running it gives much nicer (human readable) values:

```
$ raku time-me
a:0.044137168141592915
b:0.04315050652893645
```

We now have a very basic timing framework. We could turn it into a module, and indeed we shall. In Chapter 15, *Writing a Module*.

Chapter 15. Writing a Module

We made a very basic timing framework in section 14.5.2, “Timing Code”. We can turn it into a module, but that would have reinvented the wheel as there already are two modules available doing this:

- Test::Performance
- Benchmark

But we’ll do it anyway…

15.1. unit module

The normal way of specifying a module is just using the `module` keyword followed by a block:

```
module xxxx
{
  # Code here;
}
```

But we can save one block level by using `unit module` instead:

```
unit module xxxx;
# Code here;
```

Just as when we specify a `procedure` (see 10.9.1, “unit procedure”) or a `class` (see 17.11.2, “unit class”) .

15.2. is export

We have to mark the procedures we want to make available for external use with `is export` after the signature:

```
sub time-me (&code, :$iterations = 100, :$multiplier = 1) is export { ... }
```

15.3. pm6

Raku modules have the filename extention «pm6» (as in «Perl Module 6». The language rename to Raku will have an impact on this and other filename extensions.) Note that «pm» was used as well in the past, and some older modules may still do so. It is recommended to use «pm6» only, as that will give a nicer error message if someony tries to use the module from Perl 5.

File: lib/Time-Code.pm6

```
use v6.c;

unit module Time-Code;

sub time-me (&code, :$iterations = 100, :$multiplier = 1) is export
{
  my @time;

  for ^$iterations
  {
    my $start = now;
    &code();
    my $stop = now;
    @time.push($stop - $start);
  }

  return $multiplier * @time.sum / @time.elems;
}
```

15.4. use lib

Use `use lib` to specify additional locations where the compiler should look for modules.

It is normal (and recommended) to use e.g. `use lib "lib"` while developing a module, as that makes it easier to test it on the fly. (The test framework, which we'll discuss later, uses this technique as `zef` runs the tests before installing the module (and wouldn't fint the module otherwise).

You have to run the program from the directory where «lib» is located for this to work.

Note that this can be a security problem, if you run the program from another location, where you have a «lib» directory as well:

```
$ pwd    # -> /home/raku
$ raku code/chapter12/check-path
```

If the «check-path» program has a `use lib "lib"` statement, it will tell the compiler to look in the «/home/raku/lib» directory (and not «/home/raku/code/chapter12»).

And a program using it (also from section 14.5.2, "Timing Code"):

File: time-me-module

```
use lib "lib";
use Time-Code;

sub a
{
  $++ for 10000;
}

sub b
{
  ++$ for 10000;
}

my $iterations = 1000;
my $multiplier = 1000;

say "a: " ~ time-me(&a, :$iterations, :$multiplier);
say "b: " ~ time-me(&b, :$iterations, :$multiplier);
```

Running it, and the old version without a module:

```
$ raku time-me-module
a: 0.04731108696221918
b: 0.04730277517929529

$raku time-me
a: 0.04462623946451714
b: 0.0447062423500612
```

The module version of the timing framework is slightly slower (about 5%).

15.5. Timing Fibonacci

We can time the Fibonacci Number procedures (see section 10.12.2, “The Fibonacci Numbers”) and the Sequence (see section 16.3.1, “The Fibonacci Sequence”).

File: fibonacci-time

```
use lib "lib";
use Time-Code;

my $fibonacci := (1, 1, { $^a + $^b } ... Inf);

sub MAIN (Int $n, :$iterations = 100, :$multiplier = 1)
{
  say "Fib     $n: " ~ time-me({ &fibonacci($n) }, :$iterations, :$multiplier);
  say "Fib Rec $n: " ~ time-me({ &fibonacci-recursive($n) }, :$iterations,
:$multiplier);
  say "Fib Mul $n: " ~ time-me({ &fibonacci-multi($n) }, :$iterations, :$multiplier);
  say "Fib Seq $n: " ~ time-me({ $fibonacci[$n] }, :$iterations, :$multiplier);
}

sub fibonacci (Int $n)
{
  return 1 if $n == 1 or $n == 2;

  my @fib = (1, 1);

  for 2 .. $n -1 -> $i
  {
    @fib[$i] = @fib[$i -1] + @fib[$i -2]
  }

  return @fib.tail;
}

sub fibonacci-recursive (Int $n)
{
  return 1 if $n == 1 or $n == 2;

  return fibonacci-recursive($n-1) + fibonacci-recursive($n-2)
}

multi fibonacci-multi (1) { 1 }
multi fibonacci-multi (2) { 1 }
multi fibonacci-multi (Int $n where $n > 2)
{
  fibonacci-multi($n - 2) + fibonacci-multi($n - 1)
}
```

```
$ raku fibonacci-time --mul=1000 12
Fib     12: 0.17109426818938775
Fib Rec 12: 0.28062656376560663
Fib Mul 12: 4.763127095776541
Fib Seq 12: 0.05405135328930629
```

Computing the twelfth Fibonacci number takes almost the same time with a loop and with recursion. The `multi` version is way slower.

And the Sequence is blindingly fast in comparison. (But the problem is that we run the timing 100 times, and the last 99 of them we retrieve a cached value from the Sequence.)

We can run the code one time:

```
$ raku fibonacci-time --mul=1000 --iter=1 12
Fib     12: 1.686145939074689
Fib Rec 12: 0.8542231594091491
Fib Mul 12: 8.122685764523933
Fib Seq 12: 1.4559518704264895
```

Do not trust these numbers (the recursive version is suddenly faster than the loop version), as we ran the code only once. But the Sequence version got a more realistic timing.

```
$ raku fibonacci-time --mul=1000 20
Fib     20: 0.19579672840493245
Fib Rec 20: 7.5705065387054296
Fib Mul 20: 220.8083508932559
Fib Seq 20: 0.06156730344760013
```

When we compute the 20th number, the recursive version slows down considerably, and the `multi` version is even slower.

Exercise 15.1

Why is the recursive version slower than the loop version?

Note that the timing module doesn't return or show the returned value from the procedure we are timing. So run the code normally as well, to make certain that the value returned is correct (or at least that we get the same error for all of them - so that they are *equally wrong*).

15.6. Dictionaries

In this section you will need a dictionary of legal words. Ubuntu Linux has the following dictionaries:

- /usr/share/dict/american-english (the «wamerican» package)
- /usr/share/dict/british-english (the «wenglish» package)
- /usr/share/dict/ngerman (the «wngerman» package)

The english dictionaries have entries like «Abe’s» that we’ll ignore, as they contain non-word characters.

Locate and download a dictionary file if you don’t have one installed. It must be a text file, with one word per line. It doesn’t matter which language you choose, as long as you are familiar with the chosen language.

Exercise 15.2

Write a module «Dictionary» that loads a specified dictionary file (with full path), and returns a hash of all the words.

Write a short test program.

«A palindrome is a word, number, phrase, or other sequence of characters which reads the same backward as forward, such as *madam* or *racecar* or the number *10801*.» (Source: https://en.wikipedia.org/wiki/Palindrome)

Exercise 15.3

Write a program using the «Dictionary» module that prints the palindromes in the dictionary.

Exercise 15.4

Write a program using the «Dictionary» module that checks if the reverse version of every word in the dictionary is also a valid word.

«An anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once. For example, the word *anagram* can be rearranged into *nag a ram*, or the word binary into *brainy*.» (Source: https://en.wikipedia.org/wiki/Anagram)

Exercise 15.5

Write a program using the «Dictionary» module that checks for anagrams of the word specified as argument to the program.

Exercise 15.6

It is easy to rewrite the anagram checking program to check all the words in the dictionary, instead of a single one given as argument.

Is that a good idea?

Chapter 16. Ranges and Sequences

Read the introduction to Ranges in section 4.2, “Ranges (A Short Introduction)” if you haven’t done so already.

A Range is rather limited, but a Sequence has (almost) no limitations.

16.1. Ranges

The Range Operator `..` gives a range of consecutive increasing integers:

```
> say (1 .. 5).WHAT;  # -> (Range)
> say (1 .. 5);       # -> 1..5
```

16.1.1. Lazy vs Eager

Values are normally calculated when we define them. Raku calls this «Eager», and adds a second type called «Lazy».

Lazy data structures consist of values of some kind, but the individual values are not calculated until they are actually needed (as in accessed).

Ranges (and Sequences, which we’ll present shortly) are lazy, so values will not be calculated until they are actually needed.

This makes it possible to have an endless (or infinite) Range:

```
> say (1 .. Inf).WHAT;  # -> (Range)
```

16.1.2. is-lazy

Use `is-lazy` if you are unsure if a value, variable or data structure is lazy or eager:

```
> say (1 .. Inf).is-lazy;  # -> True
> say "A String".is-lazy;  # -> False
```

If we assign a Range to an array, it will be evaluated:

```
> my @range = 1 .. 10;  # -> [1 2 3 4 5 6 7 8 9 10]
> say @range.is-lazy;   # -> False
> say @range.WHAT;      # > (Array)
```

We can try with the infinite Range:

```
> my @range = 1 .. Inf;  # -> [...]
> say @range.WHAT;       # -> (Array)
> say @range.is-lazy;    # -> True
```

This gives a lazy list. This is the default only when it cannot be eager.

It is also possible to force an expression to be lazy, with the `lazy` keyword.

16.2. lazy

Use the `lazy` keyword to force an expression to be lazy.

It can be used on almost anything, but a loop is the most useful construct.

File: lazy

```
my $numbers := lazy for ^Inf { $_ };

say $numbers[0];
say $numbers[10];

.say for $numbers;
```

This gives us an infinite loop. The counter variable (from `0` to `Inf`) is - as usual - available in the block as `$_`. The last expression inside the block is the value in the lazy list.

This would make more sense if we did something inside the block, e.g. like: `{ "dummy$_" }` or `{ pi * $_ - e }`.

See 16.7.1, “lazy vs gather/take” for a more complicated example.

16.2.1. infinite

We can use `infinite` to check if a `Range` is infinite, or rather that the start and/or end was declared infinite (with `Inf`, `*` or `∞`):

```
> say (1 .. Inf).infinite;  # True
```

Note that we cannot use `infinite` on `@array`, as it is coerced to a list when we assign it.

16.2.2. List Coercion

We can coerce a Range to a List:

```
> say (1 .. Inf).WHAT;          # -> (Range)
> say (1 .. Inf).List.WHAT;     # -> (List)
> say (1 .. Inf).List.is-lazy;  # -> True
> say (1 .. 100).is-lazy;       # -> False
> say (1 .. 100).List.is-lazy;  # -> False
```

16.2.3. eager

Use `eager` to force the values in Range (or Sequence or a lazy List) to be calculated. It will return the values as a list.

```
> (1 .. Inf).eager
```

This works (or hangs, depending on your point of view). The expression will run forever, without any visible result.

Exercise 16.1 If we sit down and wait for the infinite list to crash, what would happen first:

- out of memory (too many elements in the array)?
- value too large for an integer?

16.2.4. is-int

Use `is-int` (implemented for Ranges only!) to tell us if the Range contains integers only:

```
> say (1 .. 10).is-int;     # -> True
> say ('A' .. 'Z').is-int;  # -> False
```

16.2.5. Ranges on Strings

This works:

```
> .print for ("a" .. "z"); say ""; # Add a newline at the end.
abcdefghijklmnopqrstuvwxyz
```

Or characters:

```
> say ("aa" .. "bb").WHAT;   # -> (Range)
> say ("aa" .. "bb").eager;  # -> (aa ab ba bb)
```

Did you expect "aa" .. "az", "ba", "bb" ?

The range operator doesn’t now about letters. Everything is a Unicode character, so it will give you exactly what you ask for. (It iterates the first character until reaching the target, then it does the same wuth the seond, and so on.)

Note that Raku’s idea of how to *count* from «aa» to «bb» may not suit your needs.

16.2.6. minmax

It is also possible to construct a Range with the «minmax» operator.

It takes two values, and returns a Range starting from the lowest to the highest of the values, regardless of the given order:

```
# numeric comparison
10 minmax 3;     # 3..10

# string comparison
'10' minmax '3'; # "10".."3"
'z' minmax 'k';  # "k".."z"
```

The order is decided with the «cmp» operator.

If the lowest and highest values coincide, the operator returns a Range made by the same value:

```
1 minmax 1;  # 1..1
```

When applied to Lists, the operator evaluates the lowest and highest values among all available values:

```
(10,20,30) minmax (0,11,22,33);       # 0..33
('a','b','z') minmax ('c','d','w');   # "a".."z"
```

Similarly, when applied to Hashes, it performs a cmp way comparison:

```
my %winner = points => 30, misses => 10;
my %loser = points => 20, misses => 10;
%winner cmp %loser;      # More
%winner minmax %loser;
# ${:misses(10), :points(20)}..${:misses(10), :points(30)}
```

16.3. Sequences

Seqences are generated with `...`, as opposed to `..` that generates ranges.

```
> say (1 .. Inf).WHAT;   # -> (Range)
> say (1 .. Inf)[^10];   # -> (1 2 3 4 5 6 7 8 9 10)

> say (1 ... Inf).WHAT;  # -> (Seq)
> say (1 ... Inf)[^10];  # -> (1 2 3 4 5 6 7 8 9 10)
```

Every range you can construct in Raku can also be made as a sequence, but not the other way round.

Sequences (and Ranges) are lazy. The values will only be calculated when needed.

```
> say (1..10).WHAT;          # -> (Range)
> say (1..10).reverse;       # -> (10 9 8 7 6 5 4 3 2 1)
> say (1..10).reverse.WHAT;  # -> (Seq)
```

Raku can generate sequences for us:

```
> (1, {$_ * 2} ... *)[^10];  # -> (1 2 4 8 16 32 64 128 256 512)
> (2, {$_ - 2} ... *)[^10];  # -> (2 0 -2 -4 -6 -8 -10 -12 -14 -16)
```

Or we can give it enough values to understand the pattern:

```
> (1, 2, 4 ... Inf)[^10];  # -> (1 2 4 8 16 32 64 128 256 512)
> (2, 4 ... Inf)[^10];     # -> (2 4 6 8 10 12 14 16 18)
> (2, 0 ... -Inf)[^10];    # -> (2 0 -2 -4 -6 -8 -10 -12 -14 -16)
> (1 ... -10)[^10];        # -> (1 0 -1 -2 -3 -4 -5 -6 -7 -8)
```

16.3.1. The Fibonacci Sequence

We can have pretty advanced rules for generating the values.

Remember the Fibonacci Numbers (from section 10.12.2, "The Fibonacci Numbers")? Here they are as a Sequence:

```
> say (1, 1, * + * ... *)[^10];            # -> (1 1 2 3 5 8 13 21 34 55)
> say (1, 1, { $^a + $^b } ... Inf)[^10];  # -> (1 1 2 3 5 8 13 21 34 55)
```

The `* + *` parts means that the third value is computed with two placeholder values (the first `*` and the second `*` and adding them together (the `+`). The placeholders are to the left of the current value, so in this case the first and second value (and that is why they must be specified explicitly). The `... *` part means that this will go on forever (and here `*` is the same as `Inf`).

The second example uses explicit placeholder variables, and here we can do almost anything with the values.

16.3.2. Binding vs Assignment

It is recommended to use binding (`:=`), and not the usual assignment (`=`) on sequences.

```
> my $fibonacci := 0, 1, * + * ... *;
> say $fibonacci.is-lazy;  # -> True
> say $fibonacci[10];      # -> 55
```

The Fibonacci sequence can either start with 1, as we have done until now, or with 0. The mathematicians disagree about what is correct. 1 is the most usual start value though.

Binding only works for scalars. If you want the variable to look like an array, use assignment:

```
> my @fibonacci = 0, 1, * + * ... *;
> @fibonacci.is-lazy
True

> @fibonacci[10]
55
```

Binding works better on lazy data structures (as sequences), as assignment may cause the data structure to be evaluated (or be «un-lazified» so to speak).

If you wonder. This is legal (but not a good idea):

```
> my $a = (1..10)
1..10

> $a[8]
9
```

The list (or range in this case) will be read only.

16.3.3. lazy

Use `lazy` to force a Range (or Sequence) to stay lazy as long as possible:

```
> my @range = (1 .. 10);
(1 2 3 4 5 6 7 8 9 10)

> my @range = (1 .. 10).lazy;
[...]
```

Note that the assignment to an array causes the Range to be evaluated. Assign it to a scalar to avoid that:

```
> my $range = (1 .. 10);
1..10

> $range.WHAT;
(Range)
```

Summary:

Eager Data type	Lazy Data type
`my @x = (1 .. 10)`	
`my @x = (1 ... 10)`	
	`my $x = (1 .. 10)`
`my $x = (1 ... 10)`	
	`my $x := (1 .. 10)`
`my $x := (1 ... 10)`	

Infinite ranges and sequences are **always** lazy. So if we change `10` with `Inf`, everything would end up in the «Lazy Data type» column.

16.3.4. Memory Friendly

The memory footprint of an infinite Ranges and Sequences is low, as it will only generate the values as needed:

```
> say (1..Inf)[10] # Parens required, as «Inf[10]» isn't a thing.
9
```

Note that `pick` on a lazy Range (or List or Sequence) forces it to be evaluated. This will not work if it is infinite:

```
> (1 .. Inf).pick;  # -> Nil
```

No luck!

16.3.5. A Flip-Flop Sequence

Is it possible to make a flip-flop sequence? Something that changes its mind each time we ask it about something?

Raku has a flip-flop operator called `ff` and a variant called `fff`. They will be covered in the «Advanced Raku» course. They do not have the same meaning as what we describe here.

```
> my $flip-flop := (True, False, !* ... *);

> (True, False, !* ... *)[^10]
(True False True False True False True False True False)
```

Or this:

```
> my $flip-flop := (True, {! $_ } ... *);
```

This sequence will never reach infinity (or rather; "never tries to reach inifinity"), but the generator works even so:

```
> my $i = 0;
> say "I like potatos: " ~ $flip-flop[$i++] for ^10;
```

We cannot use shift on a sequence, only iterate:

File: flip-flop-sequence

```
my $flip-flop := (True, False, !* ... *);

say "I like potatos: $_." for $flip-flop
```

```
> raku flip-flop-sequence
I like potatos: True.
I like potatos: False.
I like potatos: True.
I like potatos: False.
...
```

The program will run forever. Use <Control-c> to stop it.

16.3.6. List Repetition Operator and Sequences

We can get the same Sequence with the List Repetition Operator `xx` (see section 8.19, "xx (List Repetition Operator)") when used like this:

```
> my $flip-flop = |(True, False) xx *

> $flip-flop.WHAT
(Seq)

> $flip-flop.is-lazy
True
```

The `|` before the list to flatten it is essential, as we otherwise would get an infinite list of sublists with `(True, False)`.

16.4. state

A state variable declared with `state` (instead of the normal `my`).

It is only initialised the first time the program comes to the `state` line, and this line will be ignored after that so that it keeps the old value.

We can implement Flip.Flop with a state variable, and wrap everything in a procedure.

We can access the Flip-Flop Sequence through a procedure keeping track of the index. The initialization of a state variable is only done once, and it will keep the value between calls:

File: flip-flop-sequence-wrapped

```
my $flip-flop := (True, False, !* ... *);

sub flip-flop
{
  state $index = 0; # Only executed once!
  return $flip-flop[$index++];
}

say flip-flop for ^10;
```

We don't need the sequence at all:

File: flip-flop-procedure

```
sub flip-flop
{
  state $state = False; # Only executed once!
  $state = ! $state;
  return $state;
}

say flip-flop for ^10;
```

16.5. Truly Random Flip-Flop

The first version of «flip-flop» is predictable, as it always starts with `True`.

It is easy to fix that, so that it is completely random if the first value is `True` or `False`:

File: flip-flop-procedure2

```
sub flip-flop
{
  state $state = (True, False).pick; # Only executed once!
  $state = ! $state;
  return $state;
}

say flip-flop for ^10;
```

16.6. Flip-Flop Problems

«flip-flop» behaves as expected if used in one context. Example of the opposite:

File: flip-flop-problems

```
sub flip-flop
{
  state $state = (True, False).pick; # Only executed once!
  $state = ! $state;
  return $state;
}

sub free-lunch
{
  say "I'm { flip-flop() ?? "for" !! "against" } free lunches";
}

sub free-dinner
{
  say "I'm { flip-flop() ?? "for" !! "against" } free dinners";
}

for ^5
{
  free-dinner;
  free-lunch;
}
```

```
I'm against free dinners
I'm for free lunches
I'm against free dinners
I'm for free lunches
I'm against free dinners
I'm for free lunches
I'm against free dinners
I'm for free lunches
I'm against free dinners
I'm for free lunches
```

The program doesn't change position at all, but is *against* free dinners, and *for* free lunches all the time.

Suggestions?

16.6.1. Flip-Flop Redesign

A complete redesign is needed.

The obvious approach is to make a class, and use instances (objects) of that class.

We'll do just that, in the next chapter.

(We are not finished with ranges yet.)

16.6.2. $ / @ / % (Anonymous State Variable)

The anonymous state variable `$` can be used instead of an explicit `state $xxxx` (see section 16.4, "state"), but you can (obviously) only have one.

File: state

```
sub something
{
  my $ = 0; # Only executed once!
  return $++;
}

say something for ^10;
```

The program will print the numbers 0 to 9, each on its own line.

If we are happy with zero as the initial value, we can actually skip the declaration. The anonymous state variable will magically pop into existence when used.

File: state2

```
sub something
{
  return $++;
}

say something for ^10;
```

The array `@` and hash `%` versions are also available. So in a way you can have as many variables as you want:

```
> %<name> = 12;
> %<city> = "Oslo";
```

This may look similar to matches (see section 11.9.1, “( ) (Capturing)”), where we can use `$[0]` to get the first match. But the first value in the anonymous state array is `@[0]`.

16.7. gather / take

Let us take a look at another way of making sequences; with `gather` and `take`.

`gather` takes a block as argument, and collects (or «gathers») the values specified with `take`.

File: gather1

```
my @a = gather
{
  take 1; take 5; take 42;
}

say @a;
```

```
$ raku gather1
[1 5 42]
```

Note that all the values are computed at once. We could have done it like this instead, with the same result:

```
my @a = 1, 5, 43;
```

We want a lazy sequence, and that can be accomplished with binding to a scalar instead:

File: gather2

```
my $a := gather
{
  take 1 while 1;
}

say "1: $a[1]";
say "40: $a[40]";
say "4: $a[4]";

my $count = 0;
for $a -> $item
{
  last if $count++ >= 10;
  say $item;
}
```

```
$ raku gather2
1: 1
40: 1
4: 1
1  1  1  1  1  1  1  1  1  1
```

The three `say` statemements before the loop use this as a lazy list, expanding it as we need the values.

The loop on the other hand makes a sequence, and it forgets the values as soon as they are consumed (the loop enters the next iteration). So after the loop, `$a` is exhausted and will fail if we try to access it.

We can drop the curlies (if we want to) as we have only one expression in the block:

```
my $a := gather take 1 while 1;
```

Assigning the sequence to an array is a bad idea, as it will try to evaluate an infinite sequence. The code runs forever:

```
my @a = gather take 1 while 1;
```

Let us revisit the Flip-Flop sequence from section 16.3.5, “A Flip-Flop Sequence”:

File: flip-flop-sequence

```
my $flip-flop := (True, False, !* ... *);

say "I like potatos: $_." for $flip-flop
```

Here it is, with gather/take:

File: flip-flop-gather

```
my $flip-flop := gather loop { take True; take False; }

say "I like potatos: $_." for $flip-flop;
```

We can use a state variable instead:

File: flip-flop-gather2

```
my $flip-flop := gather loop { state $state = False;
                               $state = ! $state;
                               take $state; }

say "I like potatos: $_." for $flip-flop
```

The loop is still there, so this version makes for more code without any obvious benefits.

But we can add the random start value here as well:

File: flip-flop-gather3

```
my $flip-flop := gather loop { state $state =  (True, False).pick;
                               $state = ! $state;
                               take $state; }

say "I like potatos: $_." for $flip-flop
```

And that is something we haven't shown for a sequence before.

So let's try to do that with a sequence.

We can shorten the definition (in «flip-flop-sequence»):

```
my $flip-flop := (True, False, !* ... *);   # Old
my $flip-flop := (True, !* ... *);          # New
```

Next we randomize the first value:

```
my $flip-flop := ( (True, False).pick, !* ... *);
```

And it works, as this infinite loop shows:

File: flip-flop-sequence2

```
my $flip-flop := ( (True, False).pick, !* ... *);

say "I like potatos: $_." for $flip-flop
```

16.7.1. lazy vs gather/take

Note that `lazy` (see 16.2, “lazy”) can be used instead of `gather`/`take` - if the code has only one `take`.

A `lazy` version of *flip-flop-gather2* looks like this

File: flip-flop-lazy

```
my $flip-flop := lazy loop { state $state = False;
                             $state = ! $state;
                           }

say "I like potatos: $_." for $flip-flop;
```

Here is the original again, for easy comparison:

File: flip-flop-gather2

```
my $flip-flop := gather loop { state $state = False;
                               $state = ! $state;
                               take $state; }

say "I like potatos: $_." for $flip-flop
```

16.7.2. A Deck of Cards

A Deck of Cards consist of 52 cards, 13 (values 1 to 13) of each of the four types Spade, Club, Heart and Diamond.

Unicode has characters for the types:

```
for <♠ ♣ ♥ ♦> -> $type # spade, club, heart, diamond
```

But they are difficult to print, so we’ll stick with the first letter of the names.

File: deck

```
my @deck;

for <S C H D> -> $type  # Spade, Club, Heart, Diamond
{
  @deck.push("$type$_") for 1 .. 13;
}

say @deck.join(",");
```

```
$ raku deck
S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,
H1,H2,H3,H4,H5,H6,H7,H8,H9,H10,H11,H12,H13,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13
```

Exercise 16.2

Rewrite the «deck» program using `map` instead of the two loop.

Exercise 16.3

Rewrite the «deck» program using `gather`/`take`.

A deck of cards isn't much use if it is sorted. We can use `pick(*)` to shuffle the cards (return the values in random order):

File: deck-random

```
my @deck;

for <S C H D> -> $type  # Spade, Club, Heart, Diamond
{
  @deck.push("$type$_") for 1 .. 13;
}

say @deck.pick(*).join(",");
```

```
$ raku deck-random
C6,S4,S6,H6,S1,C12,D8,C7,C8,S2,S5,C5,D13,H3,S7,S3,S11,H8,C4,H11,D9,D4,C10,D10,H9,D5,D3
,H12,C11,D11,H7,D12,D6,S13,H4,C9,C2,S8,D2,D7,H10,C3,D1,S10,S9,H1,H2,H13,H5,C1,S12,C13

$ raku deck-random
S6,H7,S5,D8,S1,H12,C1,H13,C13,C11,S10,C5,H8,D5,H10,D3,H2,S4,H11,C6,C3,D1,D12,D11,D2,S7
,H5,D13,D9,S11,C12,C7,H4,S8,H1,D6,H3,C9,D7,D10,H9,S3,S9,H6,C10,S13,D4,S2,C4,S12,C2,C8
```

Note that using `pick` (see 8.22.2, “pick”) to draw a selection of cards from our deck isn’t a good idea, as it leaves the cards in the deck. (So that we can draw them again later.)

Use a loop (`for @deck ..`) or use `shift` on it.

Another approach is using `grab`, which will be described in the «Advanced Raku» course.

16.7.3. take-rw

Use `take-rw` to return the given item to the enclosing `gather` block, without introducing a new container. It is binded to the position in the original array, and can be changed:

File: flip-flip

```
my @flip-flop =  |(True, False) xx 8;

say @flip-flop;

sub flipflop (@list)
{
  gather for @list
  {
    take-rw $_;
  }
}

for flipflop(@flip-flop)
{
  $_ = True if $_ == False;
}

say @flip-flop;
```

Running it:

```
$raku flip-flip
[True False True False True False True False True False True False True False True
False]
[True True True True True True True True True True True True True True True True]
```

Use with care. It will fail (with the «Cannot assign to an immutable value» error message) if we give it a list, as we cannot change immutable values:

```
for flipflop((True, False, True, False))
{
  $_ = True if $_ == False;
}
```

16.8. Closures

We define a lexical variable (a `my`-variable) in a block. This variable would normally die when the block ended, but the trick is that we use it inside a procedure (also out of scope) that we have a reference to. This reference is stored outside this block.

See https://en.wikipedia.org/wiki/Closure_(computer_programming) for details.

> We have used the term Closure earlier in this book to mean text inside curly braces in strings. That is a Raku name that is somewhat unfortunate. But on the other hand, it is a kind of closure, as it evaluates the expression inside it and returns it to the outside where it lives on after the closure has died.

16.8.1. Flip-Flop Closure

We'll have a look at Closures before discussing Classes (in the next chapter).

File: flip-flop-closure

```
sub gen-flip-flop ①
{
  my $state = Bool.pick; ②

  sub flip-flop ③
  {
    $state = ! $state; return ! $state;
  }
  return &flip-flop; ④
}

my $lunch  = gen-flip-flop; ⑤
my $dinner = gen-flip-flop;

say "Lunch: "  ~ $lunch(); ⑥
say "Dinner: " ~ $dinner();
say "Dinner: " ~ $dinner();
say "Lunch: "  ~ $lunch();
say "Lunch: "  ~ $lunch();
```

① We set up an outer procedure.

② We place the state variable (now a usual `my`-variable)

③ and the flip-flop procedure inside it.

④ This outer procedure returns a pointer to the inner procedure (by using the `&` prefix).

⑤ Then we set up two instances of the generator. The variables contain pointers to procedures.

⑥ Use the parens-syntax to call the procedures.

Exercise 16.4

Is it possible and/or sensible to use `take-rw` (see section 16.7.3, “take-rw”) to generate a closure?

16.8.2. Optional Start Value

The initial value is completely random. We can modify it to take an optional start value:

File: flip-flop-closure2 (diff)

```
sub gen-flip-flop (Bool $state is copy = Bool.pick)
```

```
my $dinner = gen-flip-flop(True);
```

16.8.3. Closure vs Class

Every closure can be replaced by a class, so feel free to uses classes instead.

But the opposite is not true; classes can do so much more than what a closure can.

Chapter 17. Classes

Classes is the fundamental building block in *Object Oriented Programming*.

The class has the description, and the instances (filled with values) are called objects.

If we have class called «Person» (see the following section), we can have several objects - one for each person we want.

17.1. class

We define a class with the `class` keyword.

```
class Person
{
  # Place the content here
}
```

17.1.1. has

Class variables are declared with the `has` keyword:

```
class Person
{
  has Bool $.is-taxpayer;
  has Bool $!is-happy;
}
```

! (Private Attribute)

The exclamation point (`!`) between the sigil (`$`) and the name (in this case `is-taxpayer`) means that the attribute (variable) is private and cannot be accessed outside of the class.

(We *can* in fact give other classes explicit access; see section 17.12.1, “trusts”.)

. (Public Attribute)

Use a dot (`.`) instead of the exclamation point (`!`) to make the attribute (variable) public.

Flip-Flop implemented as a class:

```
class Flip-Flop
{
  has Bool $!state = Bool.pick;
}
```

17.1.2. new

We need two flip-flop instances, and use the built-in new method supported by every class to get them. We omit giving a value to the class variable, and it will defalt to a random value of `True` or `False`:

```
class Flip-Flop
{
  has Bool $!state = Bool.pick;
}

my $lunch  = Flip-Flop.new();
my $dinner = Flip-Flop.new();
```

17.2. method

Methods are a way of doing something with the objects.

The flip flop class needs a method giving us the state, and flips the stored value. I have called it «flip-flop»:

```
class Flip-Flop
{
  has Bool $!state = Bool.pick;

  method flip-flop
  {
    $!state = ! $!state;
    return $!state;
  }
}
```

17.2.1. . (Method Call)

Then we'll have to rewrite the «free-lunch» and «free-dinner» procedures.

We use a `.` and the method name (`.flip-flop`) on the object (`$lunch`):

```
sub free-lunch
{
  say "I'm { $lunch.flip-flop ?? "for" !! "against" } free lunches";
}

sub free-dinner
{
  say "I'm { $dinner.flip-flop ?? "for" !! "against" } free dinners";
}
```

```
free-lunch;   # True
free-dinner;  #        True
free-dinner;  #        False
free-dinner;  #        True
free-lunch;   # False
free-lunch;   # True
free-dinner;  #        False
free-lunch;   # False
```

I have added the result as comments, to show that the program works as expected.

The complete program:

File: flip-flop-class

```
class Flip-Flop
{
  has Bool $!state = Bool.pick;

  method flip-flop
  {
    $!state = ! $!state;
    return $!state;
  }
}

my $lunch  = Flip-Flop.new();
my $dinner = Flip-Flop.new();

sub free-lunch
{
  say "I'm { $lunch.flip-flop ?? "for" !! "against" } free lunches";
}
sub free-dinner
{
  say "I'm { $dinner.flip-flop ?? "for" !! "against" } free dinners";
}

free-lunch;   # True
free-dinner;  #       True
free-dinner;  #       False
free-dinner;  #       True
free-lunch;   # False
free-lunch;   # True
free-dinner;  #       False
free-lunch;   # False
```

17.2.2. Colon Syntax

There is an alternate colon syntax for calling methods, so that they look like procedure calls:

```
$lunch.flip-flop
flip-flop($lunch:); # The same
```

If the methods takes arguments, just add them:

```
$obj.method(1);
method($obj:, 1); # The same
```

17.3. Named Arguments

The default constructor (the `new` method) supports named arguments only. So if we want to override the random initial value, we must do it like this:

```
my $lunch  = Flip-Flop.new(state => False);
my $dinner = Flip-Flop.new;
```

But that doesn't work. The initial value is still random.

17.4. Public Class Variables

The builtin default constructor only works with public variables, so we have to change the variable from private to public:

```
has Bool $.state = Bool.pick;  # public
```

Note that public variables are visible to the public (the program code outside the class):

```
say $lunch.state;
```

So this is generally **not a good idea**.

We can *read* public object variables, but we *cannot change* them.

17.4.1. is rw

We can allow public variables to be changed with the `is rw` (read write) trait, like this:

```
has Bool $.state is rw = Bool.pick;
```

In our case, that would be a bad idea.

17.5. self

If you need access to the current object inside a method, use `self`.

See the next section for an example.

17.6. Custom «new»

If it is a problem that we can access object variables from the outside, make them private and write a custom `new` constructor:

File: flip-flop-class-new (partially)

```
class Flip-Flop
{
  has Bool $!state;

  method new (:$start = Bool.pick)
  {
    self.bless(state => :$start);
  }
}
```

I have changed the variable to private, and moved the default value to the `new` method so that it can cope with and without an argument.

I have also changed the name used in the constructor from «state» to «start», as that is a better name from a user point of view. Inside the class, «state» makes sense though.

```
my $lunch  = Flip-Flop.new(start => False);
my $dinner = Flip-Flop.new;
```

17.6.1. bless

`bless` is a low-level object constructor. It creates a new object of the same type as the invocant (given with `self`), and fills in the named parameters.

It is useful for custom constructors (as the `new` method in the previous section). But a custom `Build` method (see the next section) is usually better.

17.7. Custom BUILD

We can try with `BUILD` instead, which is called by the default `new`:

File: flip-flop-class-BUILD (partially)

```
class Flip-Flop
{
  has Bool $!state;
  submethod BUILD (:$!state = Bool.pick) { }
}
```

`BUILD` has to cope with a missing value as well, so we move the default initialization there. `BUILD` has no body, as variables with the same names are mapped automatically; as I have kept the name state.

`submethod` is a `method`, but it will not be inherited by child classes. See the 17.13.3, “submethod” and 17.13, “Inheritance” sections for more information.

17.8. Wrong Start Value

The «flip-flop» method flips the value, before returning it. The consequence is that the first value we get is the opposite of what we specified as the start value.

The normal solution would have been saving the value before changing it, but as we have a Boolean value, we can get away with inverting (flipping) the returned value.

```
method flip-flop
{
  $!state = ! $!state;
  return ! $!state;
}
```

A lot of exclamation marks!

The complete program:

File: flip-flop-class-new2

```
class Flip-Flop
{
  has Bool $!state;

  method flip-flop
  {
    $!state = ! $!state;
    return ! $!state;
  }

  method new (:$start = Bool.pick)
  {
    self.bless(state => $start);
  }
}

my $lunch  = Flip-Flop.new(start => False);
my $dinner = Flip-Flop.new;

sub free-lunch
{
  say "I'm { $lunch.flip-flop ?? "for" !! "against" } free lunches";
}

sub free-dinner
{
  say "I'm { $dinner.flip-flop ?? "for" !! "against" } free dinners";
}

free-lunch;
free-dinner;
free-dinner;
free-dinner;
free-lunch;
free-lunch;
free-lunch;
free-dinner;
```

17.9. Object Comparison

If we compare two objects, what are we comparing?

File: flip-flop-test (partial)

```
my $lunch  = Flip-Flop.new(start => False);
my $dinner = Flip-Flop.new(start => False);

say $dinner eq $lunch; # -> False
```

Answer: We chech if they are *the same* object. And in this case they obviously are not.

17.9.1. eqv (Equivalence)

We can compare them with the Equivalence Operator `eqv` instead.

This operator returns `True` if the two arguments have the same type, structure and values:

File: flip-flop-test (partial)

```
say $dinner eqv $lunch; # -> True
```

The objects may have the same values, but they are not the same object. Think of having two children named «Fred» in the same kindergarten. They are not the same kid, regardless of the same name.

```
my $a = Person.new(name => "Arne")
my $b = Person.new(name => "Bente")

say $a eq $b;  # -> False

my $c = $a ①
say $a eq $c  # -> True ②

$c.name = "Charlie";
say $c.name # -> Charlie
say $a.name # -> Charlie
```

① $c points to the same object as $a.

② So $a and $c is *the same object.*

We'll get back to this class in the next section.

Have this in mind before using `unique` (see section 8.18, "unique (Lists Without Duplicates)") on a list of objects.

17.9.2. ===

Use the Value Identity Operator `===` to check if both arguments are the same *object*, disregarding containerization:

```
> my class A { };
> my $a = A.new;
> say $a === $a;          # -> True
> say A.new === A.new;    # -> False
> say A === A;            # -> True
> my $b := $a;
> say $a === ba;          # -> True
```

See section 3.7.7, “===” for a description on using === on values.

17.10. A Person Class

The «Person» class hinted at in the previous section can look like this:

File: person

```
class Person
{
  has Str $.name; ①
  has Str $.birtdate; ②

  has Person $.father; ③
  has Person $.mother; ④
  has Person $.spouse; ⑤
  has Person @.child; ⑥

  method new (:$name, :$birthdate) ⑩
  {
    self.bless(:$name, :$birthdate); ⑪
  }
}

my $tom   = Person.new(name => "Tom",  birthdate => "12 Jan 1970"); ⑦ ⑧
my $lisa  = Person.new(name => "Lisa", birthdate => "21 Mar 1969");

my $john  = Person.new(name => "John",  birthdate => "5 Apr 1998");
my $peter = Person.new(name => "Peter", birthdate => "23 Oct 2001");

my $mary  = Person.new(name => "Mary",  birthdate => "12 Mar 2000");

say $tom.birthdate; # 12 Jan 1970 ⑨
say $mary.name;     # Mary ⑨
```

① A public variable, so the default «new» will take care of it for us.

② As above. Also a string, as we haven’t discussed dates (date objects) yet.

③ A person has one (or no) father.

④ A person has one (or no) mother.

⑤ A person has one (or no) spouse.

⑥ A person has none or more children.

⑦ And finally we set up 5 Person objects.

⑧ We use the default «new» constructor, and ignore the relationship fields.

⑨ Just to show that it actually works.

⑩ I have written a constructor, so that we cannot specify the relationship fields.

⑪ We can use this short form when the field has the same name as the variable.

I could have made the relationship fields private (e.g. «$!father» instead of «$.father»), instead of writing a new constructor, to make it impossible to set them in the «new» call.

But that would have limited what we could do later on, so we'll stick with public.

Exercise 17.1

Write the following methods: «set-father», «set-mother», and «set-spouse» to set the respective fields. Disregard the `@.child` field.

Add the following code to check that it works (executes without errors):

```
$tom.set-spouse($lisa);
$lisa.set-spouse($tom);

$john.set-father($tom);
$john.set-mother($lisa);

$peter.set-father($tom);
$peter.set-mother($lisa);

$peter.set-spouse($mary);
$mary.set-spouse($peter);

say $tom.spouse.name;
say $john.mother.name;
```

The data structure/dependencies should look like this:

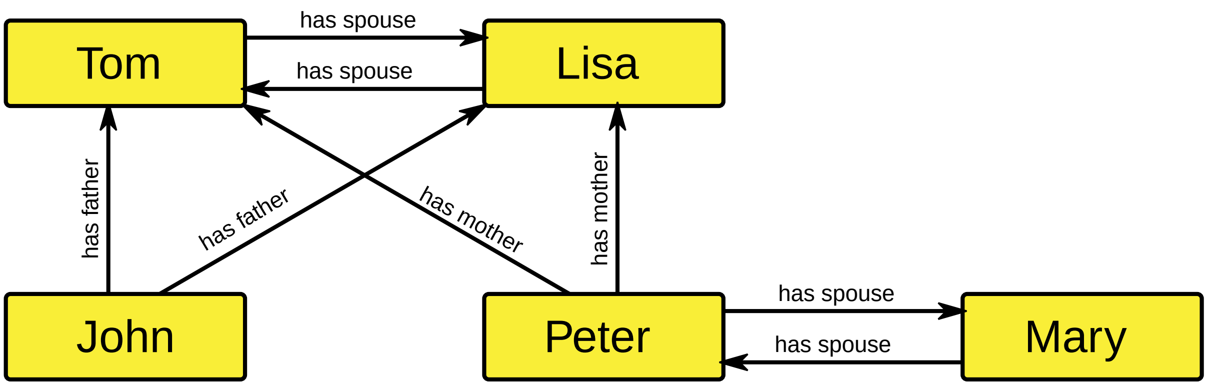


Figure 14. Persons

If Lisa had had a father, we could have used «$john.mother.father.name» to get his name. She has not (in our data structure), so the program would terminate.

```
$tom.spouse.name; # -> lisa
$tom.father.name; # -> program termination
```

Not very robust. And this shows that we really shouldn't access fields directly (but use methods). We'll get back to that later.

Being a spouse is a one-to-one relationship in our class, and we set up as a two way relationship in our data structure by specifying both directions separately. It is possible to screw up things like this:

```
$peter.set-spouse($mary);
$mary.set-spouse($tom);  # Tom vs Peter

$peter.spouse.name;               # -> Mary
$peter.spouse.spouse.name;        # -> Tom
$peter.spouse.spouse.spouse.name; # -> Lisa
```

Exercise 17.2

Simplify (from a user perspective, as it will add code to the module) «set-spouse» to set up both relationships, so that we can do this:

```
$tom.set-spouse($lisa);
# $lisa.set-spouse($tom);

$peter.set-spouse($mary);
# $mary.set-spouse($peter);
```

Add this line to check that it works:

```
say $lisa.spouse.name;  # Tom
```

Exercise 17.3

Write the method «add-child» to add to the «child» field.

Add the following code to check that it works (executes without errors):

```
$tom.add-child($john);
$tom.add-child($peter);

$lisa.add-child($john);
$lisa.add-child($peter);
```

Write the method «show-children« that shows them, and use it like this:

```
$tom.show-children;
$mary.show-children;
```

This should give this output:

```
Tom has a child named John.
Tom has a child named Peter.
Mary has no children.
```

The list of children is just that, a list.

```
$tom.add-child($peter);
$tom.add-child($peter);
$tom.add-child($peter);
$tom.add-child($peter);
```

The easiest way of preventing duplicates is to apply `unique` on the list, after adding the new child to it:

File: person-children2 (partial)

```
method add-child (Person $child)
{
  @!child.push($child);
  @!child.=unique;
}
```

This removes duplicate *objects* only (see 17.9, “Object Comparison”), so it is ok to have several persons with the same name (though confusing in practice, as we all know).

Exercise 17.4

Simplify the parent - child relationship set up, as we did for spouses.

Rewrite «set-father» and «set-mother» to call «add-child» for us, so that we can remove the «add-child» calls in the program.

17.11. Output

What happens if we try to print an object?

We can try. I have written a little program doing that (available as «person-say»). All it does is define the «Person» class, set up «Tom» and then these lines:

```
say $tom;
say $tom.perl;
say $tom.gist;
say $tom.Str;
```

The first three `say`s behave the same, and gives:

```
Person.new(name => "Tom", birthdate => "12 Jan 1970", father => Person, mother =>
Person, spouse => Person, child => Array[Person].new())
```

But the last one is truly non-useful: `Person<94782701358256>`.

Stringification of objects gives a unique value for that object (so that comparison will work), and the number is the memory address of the object.

But it is actually up to us, as programmerer. We can add custom `Str` and `gist` methods if we don't like the default behaviour. (And I certainly do not.)

But before we do that, I'll take a little detour. I didn't show the file «person-say» as it had the entire class definition that I would have shown again and again in this chapter (if I hadn't cheated with my examples).

17.11.1. A Class as a Module

We should move it into a module, as we did with our testing framework in Chapter 15, *Writing a Module*.

File: lib/Person.pm6

```
class Person
{
  has Str $.name;
  has Str $.birthdate;

  has Person $.father;
  has Person $.mother;
  has Person $.spouse;
  has Person @.child;
}
```

File: person-say2

```
use lib "lib";
use Person;

my $tom = Person.new(name => "Tom",  birthdate => "12 Jan 1970");

say $tom;
say $tom.perl;
say $tom.gist;
say $tom.Str;
```

Running it gives the same result as running «person-say».

`use lib "lib"` tells the compiler to add the «lib» directory to the list of locations to look for modules. See section 15.4, “use lib” for more information.

17.11.2. unit class

We can use `unit class` instead, saving us for a block level if we have the class in a separate file (as we do now):

```
unit class Person;

# Place the content here
```

17.11.3. custom Str and gist

And now back to the question of output. We can provide custom versions of `Str` and `gist`:

File: lib/PersonX.pm6

```
unit class Person;

has Str $.name;
has Str $.birthdate;

has Person $.father;
has Person $.mother;
has Person $.spouse;
has Person @.child;

method gist
{
  return "{ $.name } ({ $.birthdate }-)";
}

method Str
{
  return $.gist;
}
```

Note that I have renamed the class file (to «PersonX»), but not the class (still «Person»). That is legal, but not necessarily smart.

File: person-say3

```
use lib "lib";
use PersonX;

my $tom = Person.new(name => "Tom",  birthdate => "12 Jan 1970");

say $tom;
say $tom.perl;
say $tom.gist;
say $tom.Str;
```

Running it gives `Tom (12 Jan 1970-)` for all of them except `.perl` (which continues to give a `Person.new(...)` code block).

(The dash after the birth date indicates that the person is still alive. We should be careful with how we display data about persons.)

If you don't like code replication, let one method (`gist`) do the job:

```
method Str
{
  return $.gist;
}
```

If you think it is a *bad idea* to stringify objects, use one of the Stub Operators (see section 10.12.1, “Stub Operator”) to «reward» programmer stupidity:

```
method gist
{
  ...;
}
```

Do this in a module, release it, and wait for users to complain aboat it.

We have to drop the explicit `return` as the compiler will be confused and assume that `...` is the Sequene Operator. `???` and `!!!` can be used with `return`, as they have no double meaning.

17.12. Private Methods

Methods can be private, just prefix them with `!`:

```
method !explode { ... }
```

They are not callable *outside* the class itself. Inside call them like this:

```
self!explode;
```

17.12.1. trusts

We can allow other classes to access private methods and attributes in a class.

The class that we want to trust must be declared already, and we give it access with the `trusts` keyword:

File: class-trust

```
class Owner { ... }; ①

class Car ①
{
  trusts Owner;

  has Str $.type = "no name";

  method !sell { say "The $.type is sold." } ②
}

class Owner ①
{
  has Str $.name = "no name";
  has Car $!car = Car.new(type => "Volvo X1"); ④

  method sell-car { $!car!Car::sell; } ⑥
  method get-car  { $!car; }
}

my Owner $o = Owner.new(name => "Tom Jones"); ③

$o.sell-car; ⑤

say $o.get-car.type; ⑦

# $o.get-car!sell;

# $o.get-car!Car::sell;
```

① The `Owner` class references the `Car` class, so `Car` must be declared first. This poses a problem as `trusts Owner` requires that `Owner` must be declared first. But we can get away with a stubbed class.

② The `Car` class has a private method `sell` that it has allowed `Owner` to call.

③ We set up a new `Owner` object,

④ and this sets up a new `Car` as well, linked from `Owner`.

⑤ Then we sell the car, from the `Owner` object.

⑥ We must prefix the method (`sell`) with the full class name (`Car::sell`).

⑦ We can access the car type, as that is a public attribute.

```
# $o.get-car!sell;
```

Uncommenting this line in the program gives an error:

```
===SORRY!=== Private method call to sell must be fully qualified with the package
containing the method
```

We can try fixing the problem (using the class name), and uncomment this line instead:

```
# $o.get-car!Car::sell;
```

It fails as well:

```
===SORRY!=== Cannot call private method 'sell' on package Car because it does not
trust GLOBAL
```

The syntax is correct, but the main program isn't trusted. The error message gives a hint, so we can try:

```
class Car
{
  trusts Owner;
  trusts GLOBAL;
  ...
}
```

And with this change it works. We can now sell the car from itself (the `Car` class), the `Owner` class, and the main program - but not from any other class (if we had any).

17.12.2. trusts (method)

Use `trusts` as a method to get a list of classes that the invocant trusts.

We can add the following code to the program:

```
print "Owner trusts:"; print " " ~ .^name for Owner.^trusts; say "";
print "Car trusts:";   print " " ~ .^name for Car.^trusts; say "";
```

The result:

```
Owner trusts:
Car trusts: Owner
```

17.13. Inheritance

Inheritance is the primary mechanisms for code reuse in classes. A class (called child class) can inherit from one or more classes (called parent or base classes).

Everything in a parent class, except submethods (see below) and private methods, is inherited. If the child class defines an attribute or method with the same name, this version is used instead.

17.13.1. is

Inheritance is specified with the `is` keyword.

Let us revisit our «Person» class. We can add a couple of new classes, reusing it like this:

```
class Adult is Person
{
  has Str $.employer;
}

class Child is Person
{
  has Str $.school;
}

class Pensioner is Person
{
  ;
}
```

A class can inherit from more than one parent classes, either by inheriting from a class that itself uses inheritance (and so on).

Or directly, like this:

```
class SeaPlane is Plane is Boat
{
  has Int $.pontoons;
}
```

Inheritance loops, where we end up inheriting from ourself, is a known problem. The fact that we cannot inherit from a class that we haven't already declared (just as with `trusts`; see section 17.12.1, "trusts") makes it hard to make this mistake.

A Stubbed class will not work:

File: inheritance-loop

```
class Mammal { ... }; ①
class Person is Mammal { has Str $.name; } ②
class Mammal is Person { has Str $.city; } ③
```

① We have to stub it, as «is Person» will fail

② This fails, as it tries to inherit from a stubbed class

③ We'll never get this far

If we remove line 2, we get an error because of the `is Person` bit. We cannot add inheritance on a stubbed class:

```
===SORRY!=== Redeclaration of symbol 'Mammal'
```

So it is impossible to set up a circular inheritance.

17.13.2. also is

We can use `also is` in the class body, instead of `is` in the head:

```
class SeaPlane
{
  also is Plane;
  also is Boat;

  has Int $.pontoons;
}
```

> ⚠ The problem with multiple inheritance is what to do if both the classes we inherit from adds a method with the same name? Which of them should we use? Raku has no mechanisms for the programmer to influence what it chooses to do. This is the main reason it is advised to use Roles (see section 17.14, "Roles") instead.

17.13.3. submethod

`submethod` is a `method`, but it will not be inherited by child classes. (The name refers to the fact that `submethods` are scoped in the same way as procedures (`sub`).

It is often used for custom `BUILD` methods (as described in section 17.7, "Custom BUILD").

They can also be useful if we want to ensure that the method must be specified for a child class, as the parent version is unsuitable, and you want to have it as a public method.

```
class Person
{
  has Str $.birthdate; # On the form "yyyy-mm-dd"
  submethod age { Int((now.Date - Date.new($!birthdate))/365) }
}

class Woman is Person
{
  ;
}
```

Asking a Woman about her age causes an exception.

Exercise 17.5

The way we calculate a person's age is wrong (as we assume that every year has 365 days). Fix it.

17.13.4. rebless

Use the `Metamodel::Primitives.rebless` method to cange the *type* of an object. This only works if the new type is a subtype of the objects original type.

Let us revisit the Person/Woman example, stripped down to the bare minimum:

File: rebless

```
class Person          { ; }
class Woman is Person { ; }

my $tom   = Person.new;
my $lisa  = Woman.new;

say $tom.WHAT;   # -> (Person)
say $lisa.WHAT;  # -> (Woman)

Metamodel::Primitives.rebless($tom, Woman);

say $tom.WHAT;   # -> (Woman)
```

17.14. Roles

Roles let us attact attributes and methods to classes, without inheritance. This is useful when the content of the role is the only thing the classes have in common.

They behave as a sort of macro, and one they are added to the class (or mixed in, in normal OO terminology) they become part of the class - and we have no way of detecting that they were added as a role instead of being declared in the class itself.

An example:

```
role Doors
{
  has Int $.number-of-doors;
}
```

17.14.1. does (Objects)

Use the `does` keyword to add the role to the class:

```
class Car does Doors
{
  has Int $.wheels;
  has Bool $.has-automatic-transmission;
}

class House does Doors
{
  has Int $.floors;
  has Int $.rooms;
}
```

It is possible to apply a `role` to an object (at runtime):

```
> role Windows {}
> my $c = Car.new;    # -> Car.new
> $c does Doors;      # -> Car+{Doors}.new
> $c does Windows;    # -> Car+{Doors}+{Windows}.new
```

17.14.2. also does

We can use `also does` in the class body, instead of `does` in the head:

```
class Car
{
  also does Doors;
  has Int $.wheels;
  has Bool $.has-automatic-transmission;
}

class House
{
  also does Doors;
  has Int $.floors;
  has Int $.rooms;
}
```

17.14.3. but (Objects)

In section 3.8, “but (True and False, but …)” we showed how `but` (and `does`) can be used with scalar values on other scalar values. Applying anything except a Boolean value doesn’t really work out.

We can apply a role in the same way.

File: but-role

```
my $a = 41 but Doors;
say $a ~ " " ~ $a.doors();

my $b = 42;
say $b ~ " " ~ ( $b.^can("doors") ?? $b.doors() !! "-" );
```

The `but` keyword is similar to `does`. The difference is that `does` adds it to the given variable, class or object, whereas `but` applies it to a *copy* of it. When dealing with objects, we use `does` inside the class, and `but` on objects.

^can

`$b` doesn't have the role, and calling `.doors` on it would terminate the program. We can avoid that by checking that the method exist (with the `.^can` method), before calling it.

Running it:

```
$ raku but-role
41 No doors
42 -
```

Applying Roles to objects or values with `but` doesn't work in `REPL`.

17.15. Multiple Dispatch

We can have different versions of a method, specified with the `multi` keyword, (as we can with procedures; see section 10.12, "Multiple Dispatch") with different parameter lists (or «signatures»):

```
multi method do-something ($file1)         { ... }
multi method do-something ($file1, $file2) { ... }
```

Multiple dispatch come in addition to the built in dispatch mechanisms we get with classes, as we can use the same method name on objects of different classes and have them behave differently.

File: class-multi

```
class Fly
{
  # Attributes
  method kill { say "Fly killed"; }
}

class Process
{
  # Attributes
  method kill { say "Process killed"; };
}

my $a = Fly.new;
my $b = Process.new;

.kill for $a, $b;
```

Running it:

```
$ raku class-multi
Fly killed
Process killed
```

Don’t forget the `method` keyword, as a `multi` on its own is short for `multi sub`.

17.16. A Fallback Method

We can define a method with the special `FALLBACK` name, and it will be used if we call a non-existing method:

File: fallback

```
class Stupid
{
  method FALLBACK ($name)
  {
    say "You invoked $name, but it doesn't exist.";
  }
  method hello
  {
    say "Hi.";
  }
}

my Stupid $s = Stupid.new;

$s.some-method-that-doesn't-exist;
$s.hello;
$s.hi;
```

Running it:

```
$ raku fallback
You invoked some-method-that-doesn't-exist, but it doesn't exist.
Hi.
You invoked hi, but it doesn't exist.
```

If we don't have `FALLBACK` method, calling non-existing methods will give a run time error.

Having a `FALLBACK` method will hide this error message, if we actually intended to call an existing method but made a typing error.

We can have arguments as well. Simply specify them after the argument taking the method name (which is usually called `$name`).

Different signatures are supported with `multi method`:

File: fallback-multi

```
class Stupid
{
  multi method FALLBACK ($name, Str $person)
  {
    say "Hi, $person.";
  }
  multi method FALLBACK ($name)
  {
    say "You invoked $name, but it doesn't exist.";
  }
  method hello
  {
    say "Hi.";
  }
}

my Stupid $s = Stupid.new;

$s.some-method-that-doesn't-exist;
$s.some-method-that-doesn't-exist("Tom");
$s.hello;
$s.hi;
```

Running it:

```
$ raku fallback-multi
You invoked some-method-that-doesn't-exist, but it doesn't exist.
Hi, Tom.
Hi.
You invoked hi, but it doesn't exist.
```

We can have a catch all, using a slurpy argument (see section 10.14.1, “Slurpy MAIN”):

File: fallback-cathcall

```
class Stupid
{
  multi method FALLBACK ($name, *@arguments)
  {
    say "Method: $name";
    say "- Arg: $_" for @arguments;
  }
}

my Stupid $s = Stupid.new;

$s.some-method-that-doesn't-exist(<1 2 3>);
$s.hello;
$s.hi(706);
```

Running it:

```
$ raku fallback-catchall
Method: some-method-that-doesn't-exist
- Arg: 1
- Arg: 2
- Arg: 3
Method: hello
Method: hi
- Arg: 706
```

17.17. .?

We can use the special `.?` method invocation syntax if we are unsure if the method is available for the object. It is executed if it is, and `Nil` is returned if it isn't.

```
class A {};
my $a = A.new;
> say $a.foo-bar;   # -> No such method 'foo-bar' for invocant of type 'A'
> say $a.?foo-bar;  # -> Nil
```

17.18. .+

If we have a subclass wthat has redefined a method in the base class, invoking that method on an object of the child class uses the child class version. We can get it to invoke them all with the `.+` syntax:

File: all-methods

```
class A
{
  method hi { say "Hi!"; }
}

class B is A
{
  method hi { say "Hello!"; }
}

my $x = B.new;

$x.hi;
say "....";
$x.+hi;
```

Running it:

```
> raku all-methods
Hello!
....
Hello!
Hi!
```

An exception is thrown if the method doesn't exist.

We have ignored the return values, but they are available. The last one gives a list, and here the value is `(True True)`.

The order of the calls are from the current class, and nesting out. We can use the `^mro` method (see section 3.2, "^mro (Method Resolution Order)") to see the Method Resolution Order:

File: all-methods-mro (partial)

```
my $x = B.new;

say $x.^mro;  # -> ((B) (A) (Any) (Mu))
```

17.19. .*

As `.+`, except that it returns an empty list if the method doesn't exist (instead of throwing an exception).

17.20. handles (Delegation)

Delegation is another way to set up relationships between classes. We make methods from another class available in the current class (a sort of *import*).

See https://en.wikipedia.org/wiki/Delegation_(object-oriented_programming) for more information.

File: delegation

```
class Baby
{
  has $.name;
  method cry($times) { say "Waah! " x $times; }
}

my $tim = Baby.new(name => 'Tim');
say $tim.name;  # -> Tim
$tim.cry(5);    # -> Waah! Waah! Waah! Waah! Waah!

class BabySitter
{
  has $.name;
  has Baby $.baby handles (baby_name => 'name'); ① ②
}

my $teenager = BabySitter.new(name => 'Lisa', baby => $tim); ③

say $teenager.name;       # -> Lisa
say $teenager.baby_name;  # -> Tim ③
```

① The «BabySitter» class has a «Baby» attribute,

② and we make a method «baby_name» available. It is delegated (by `handles`) to the `name` method in the «Baby» class.

③ Calling it.

Note that we had to rename the method (to «baby_name»), as they both use «name».

If the method names don't collide, we can do this instead:

```
has Baby $.baby handles 'name';
```

We can inherit several methods as well:

```
has Baby $.baby handles <name bedtime diaper-type>';
```

The alternative to delegation is accessing the objects directly. E.g.

File: delegation-no (partial)

```
say $teenager.baby.name;  # -> Tim
```

17.21. Calling a method specified in a variable

We can call a method specified as a string (in a variable) by putting the variable in double quotes (to get it interpolated), and adding `()` to get it called:

```
> say "{pi}.{$_}: " ~ pi."$_"() for <Int Real Str>;
3.141592653589793.Int: 3
3.141592653589793.Real: 3.141592653589793
3.141592653589793.Str: 3.141592653589793
```

17.22. Stubbed Class

We have to declare a class, before we can use (or reference) it. That can be a problem if we want classes to reference each other.

One solution is a forward class declaration (also called «Stubbed class», as we use a 10.12.1, “Stub Operator”). It looks like this:

```
class Parent { ... }
```

Stubbed Classes are fine, as long as we specify the content of the class, before using it.

The other solution is to ensure that we specify them in the right order, so that they are not referenced befroe we have declared them.

We must use Stubbed Classes when we have mutual dependencies (or a circular data structure):

File: stubbed-class

```
class Parent { ... }

class Child { has Parent $.parent; }

class Parent { has Child $.child; }
```

Appendix 1. Docker

(Docker Docker (a light weight container technology) is the easiest way to obtain and run Raku (except on Windows), at least just for testing. (And certainly if you do not want to install Raku on your system.)

Since Raku is in very active development, with monthly releases, it may be prudent to check a program with a newer version via Docker before upgrading the locally installed version of Raku.

If you don't want to install Rakudo Star, using Docker is a good alternative.

> I don't recomment dunning Docker on Windows, as it requires Win 10 Pro. Using VirtualBox is a workaround.
>
> See https://docs.docker.com/docker-for-windows/install/ if you want to have a go anyway.

Installing Rakudo Star with Docker

There are several Docker Images with Rakudo publicly available:

Image Name	Operating System	URL (for more information)
rakudo-star	Ubuntu (Linux)	https://github.com/perl6/docker
jjmerelo/alpine-perl6	Alpine (Linux)	https://hub.docker.com/r/jjmerelo/alpine-perl6/
moritzlenz/perl6-regex-alpine	Alpine (Linux)	https://hub.docker.com/r/moritzlenz/perl6-regex-alpine/
jjmerelo/rakudo-nostar	Debian (Linux)	https://hub.docker.com/r/jjmerelo/rakudo-nostar
jjmerelo/perl6-doc	Debian (Linux)	https://github.com/perl6/doc (and section 1.8.2, "Local documentation")

The Ubuntu version is a rather large Ubuntu system, and the Alpine one is a much more compact distribution. Alpine should be faster, and use less memory. But it has some limitations, which we'll get back to later.

We can download and run a Docker Image in one operation (assuming that Docker has been installed first):

```
$ docker run -it rakudo-star
Unable to find image 'rakudo-star:latest' locally
latest: Pulling from library/rakudo-star
693502eb7dfb: Already exists
081cd4bfd521: Pull complete
c3439586dbe8: Pull complete
Digest: sha256:eac1ce2634c62857ee7e5e3f23b215 ...
Status: Downloaded newer image for rakudo-star:latest
To exit type 'exit' or '^D'
>
```

First it checks if the Image has been downloaded, and as it hasn't it does so. Then it does a checksum test, before running the Container.

The last line is the REPL prompt.

If you get an error message (permission denied), run the program as root:

```
$ sudo docker run -it rakudo-star
```

Or fix the permissions:

```
$ sudo usermod -a -G docker $USER
```

You have to log out and in again for this change to take effect. Reboot if that doesn't work.

Or if you want to use Alpine:

```
$ docker run -it jjmerelo/alpine-perl6
To exit type 'exit' or '^D'
>
```

Docker downloads the specified Docker Image the first time you run this command, and will use the local copy after that.

Use `docker pull` to check if there is a newer version available, and download that:

```
$ docker pull rakudo-star
$ docker pull jjmerelo/alpine-perl6
```

Docker Shell

We have shown how to use Docker to run Raku in REPL mode.

But it is possible to log in to the container, so that you can run programs. Use this command:

```
$ docker run -it -v $(pwd):/opt rakudo-star bash
```

This will give you a bash (shell) prompt, inside the Docker file system, and the directory where you ran the command is available as /opt.

This can be used to test local programs, simply by going to that directory and running them.

Note that this will not work with Alpine (as «bash» isn't available in the Container). So use Ubuntu if you want to run «bash».

It is also possible to run a local program directly. E.g. the «hello-world» program located in the current directory:

```
$ docker run -it -v $(pwd):/opt rakudo-star /opt/hello-world
Hello, World!
```

Be aware that the current directory (inside Docker) will not be set to `/opt`, and this can cause problems if the program assumes that it is run from the directory it is located in.

Appendix 2. Solutions

Solutions to the exercises in the book.

Chapter 1

Exercise 1.1

Installation is described in the chapter. Ensure that `raku` is in your path, and run:

```
> raku -v
This is Rakudo version 2019.11 built on MoarVM version 2019.11
implementing Perl 6.d.
```

If `raku` isn't available, you can try the old name `perl6`. If that works, either use that instead - or create a symbolic link (on a Unix likesystem).

If you get longer version numbers, you have a development version (typically a result of a git pull command):

```
> raku -v
This is Rakudo version 2018.01-210-gf1b7cc4d9 built on MoarVM version
2018.01-97-g22d2db5e0 implementing Perl 6.c.
```

The compiler may be in an inconsistent state, and strange errors may pop up.

Exercise 1.2

You didn't really expect a written solution?

Exercise 1.3

The number of keywords available in Raku:

```
$ p6doc list | wc
    855    1711   12499
```

Chapter 2

Exercise 2.1

```
> say 12 + 10 * 4; # -> 52
```

It is the same as:

```
> say 12 + (10 * 4); # -> 52
```

Exercise 2.2

Legal variable names:

```
my $don't-do-it;  # Ok
my $dog;          # Ok
my $dog2;         # Ok
my $dog-3;        # Error
```

Exercise 2.3

We can rewrite the code:

```
> ! 1 == 15;    # -> False
> not 1 == 15;  # -> True
```

In the first one the`!` negation has higher precedence than the comparison `==`, so we get `(! 1) == 15` or `False == 15` or `0 == 15` or `False`.

In the second one the `not` negation has lower precedence than the comparison `==`, so we get `not (1 == 15)` or `not (False)` or `True`.

Chapter 3

Exercise 3.1

The *largest* number we can store in an `Int`:

We can try:

```
> my $a = 10 ** 100;
> my $b = 10 ** 1000;
> my $c = 10 ** 10000;
```

`**` is the exponential operator, and the last one gave us a number with 10001 digits. Probably big enough.

It is estimated that there are between 10^{78} and 10^{82} (or `10 ++ 78 and 10 ++ 82` in Raku) atoms in the known, observable universe. (Source: https://www.universetoday.com/36302/atoms-in-the-universe/.)

Just to be sure that we actually get integers. (The fact that the output looks like it is a hint as well.)

```
> (10 ** 10000).WHAT;  # -> (Int)
```

We can try with even bigger numbers:

```
> (10 ** 10000000000).WHAT;  # -> (Failure)
> (10 ** 1000000000).WHAT;   # Hangs
```

Conclusion: The limit is somewhere between those to. (We'll get back to Failures in the «Advanced Raku» course.)

The last one hangs as it takes a lot of time to compute that integer value.

So integers have a limit, but it is so large that for all practical purposes they are limitless.

Exercise 3.2

Why we get `False` from the first and `True` from the second:

```
> 111 before 21;      # -> False
> "111" before "21";  # -> True
```

`before` is like `cmp` in that it compares numbers as numbers, and string as strings.

The first is simply a numeric comparison, and 21 is lower than 21.

In the second we compare two strings, and «1» comes before «2».

Chapter 4

Exercise 4.1

The output is «99».

Because the loop variable is lecically scoped to the block, and goes away afterwards - without affecting the global variable which is now visible again.

Exercise 4.2

The output from this program:

```
for 5
{
  say "I like school.";
}
```

```
I like school.
```

`5` is a single value, so we iterate over a one element list.

Exercise 4.3

A program that calculates the sum of all the integers from 1 to a specified upper limit (e.g. 1000), displaying the last integer added to reach (or pass) the limit.

File: sum

```
my $sum   = 0;
my $limit = 1000;

for 1 .. Inf -> $current
{
  $sum += $current;

  if $sum >= $limit
  {
    say "Limit $limit reached ($sum) at value $current.";
    last;
  }
}
```

```
$ raku sum
Limit 1000 reached (1035) at value 45.
```

Exercise 4.3

Adding a colon after `once` turns it into a label. That label is not used, but that doesn't matter. The following block (or statement) is executed for every iteration of the loop.

Chapter 5

Exercise 5.1

The easiest way to detect the bases or radixes available in Raku, try a high'ish value in `REPL`:

```
> :100<123>
===SORRY!=== Error while compiling:
Radix 100 out of range (allowed: 2..36)
------> :100<123>  <EOL>
```

So the answer is everything from «base 2» to «base 36».

The value 36 is the sum of the 26 letters in the english alphabet and the 10 digits.

»Base 1» (also called «Unary numerical system») is not supported. It would have been easy; the unary value "1" is 1, "11" is 2, and so on. Note that this number system doesn't have a null value.

Exercise 5.2

Display `pi` as a binary number:

```
> say pi.base(2)
11.001001000011111101101010
```

Note that `sprintf` and `fmt` (which we'll introduce in chapter 6; sections 6.5.3, "sprintf" and 6.5.4, "fmt") don't work (as they truncate the value to an `Int`):

```
> say pi.fmt("%b");
11

> say sprintf("%b", pi);
11
```

Exercise 5.3

Computing the sums of all the prime numbers (in numerically increasing order) from 1 to 100_000 (both included) showing if the sum is a prime or not. And displaying how many of those sums also are primes.

File: prime-prime

```
my $sum;
my $prime = 0;

for 1 .. 100_000
{
  next unless .is-prime;

  $sum += $_;

  if $sum.is-prime
  {
    $prime++;
    say "Yes: primes(1 .. $_).sum is also a prime ($sum)";
  }
  else
  {
    say "No:  primes(1 .. $_).sum is not a prime ($sum)";
  }
}

say "Number of primes: $prime.";
```

Running it gives a lot of output, but the last line tells us that:

```
Number of primes: 571.
```

Exercise 5.4

A program that adds every integer from 1 to 1000 that isn't divisible by 7, using `next`:

File: sum-seven

```
my $sum;

for (1 .. 1000)
{
  next if $_ %% 7;
  $sum += $_;
}

say "Sum: $sum\n";
```

Running it:

```
$ raku sum-seven
Sum: 429429
```

Exercise 5.5

A cylinder with internal radius 10cm and height 50cm high can contain how many litres of liquid?

The formula for the volume is: «V = πr²h», where «r» is the radius, and «h» is the height.

If «r» and «h» are given in centimeters, the result is in cubic centimeters. 1 litre is equal to 1000 cubic centimeters.

That gives us this code:

File: cylinder

```
my $r = 10;
my $h = 50;

say "Litres: ", π * $r * $r * $h / 1000;
```

Running it:

```
$ raku cylinder
Litres: 15.707963267948966
```

Exercise 5.6

Which cylinder can contain the most liquid; the one from the prior exercise, or one with radius 35cm and height 10cm?

File: cylinder-compare

```
my $r = 10;
my $h = 35;

say "R:10,H:25,Litres: ", π * $r * $r * $h / 1000;

$r = 35;
$h = 10;

say "R:35,H10,Litres: ", π * $r * $r * $h / 1000;
```

Running it:

```
$ raku cylinder
R:10,H:25,Litres: 10.995574287564278
R:35,H10,Litres: 38.48451000647496
```

Choose the last one.

Chapter 6

Exercise 6.1

A program that asks for a number, and assumes that the input is in hexademinal and all the way down to binary, printing the value in decimal:

File: number-base

```
my $input;
my $decimal;

repeat
{
  $input = prompt "Enter a number (or return to exit): ";

  for 2 .. 16 -> $base
  {
    my $decimal = try $input.parse-base($base);

    say "Base $base gives decimal: $decimal" if $decimal;
  }
} while $input;
```

Chapter 7

Exercise 7.1

A program that asks for input (in a loop), and replaces every lower case letter (a-z only) with the upper case version, and vice versa, before printing it on the screen. Other characters are left unchanged.

File: swap-case

```
my $in;

constant $a = ord "a";
constant $z = ord "z";
constant $A = ord "A";
constant $Z = ord "Z";

constant $diff = $A - $a;

repeat while $in
{
  $in = prompt "Type a text (or return to exit): ";

  for ords $in
  {
     if    $a <= $_ <= $z { print chr($_ + $diff); }
     elsif $A <= $_ <= $Z { print chr($_ - $diff); }
     else                 { print chr($_);         }
  }
  print "\n";
}
```

Exercise 7.2

A program that asks for integer values in a loop, displaying the sum of all the digits.

File: digit-sum

```
loop
{
  my $value = prompt "Enter an integer value (or return to exit): " or exit;

  next unless $value.Int;

  say "The digit sum of {$value} is { $value.comb.sum }.";
}
```

Exercise 7.3

A program that asks for input in a loop, and shows the *last* character:

We cannot use `chop`. It does remove the last character, but returns the original string without this character (e.g. `say "1234".chop; #→ 123`)

File: last-char (partial)

```
loop
{
  my $value = prompt "Enter a string (or return to exit): " or exit;

  say "The last character in '{$value}': { $value.comb[* -1] }.";

  ...
}
```

Also, show every second character:

File: last-char (partial)

```
  print "Every second character: ";

  my $count = 0;
  for $value.comb
  {
    next if $count++ % 2;
    print $_;
  }
  say "\n"; # An extra newline (i.e. 2 of them).
```

Exercise 7.4

A version of «swap-case» from Exercise 7.1 that also handles unicode letters, that is letters other than a-z.

File: swap-case2

```
my $in;

repeat while $in
{
  $in = prompt "Type a text (or return to exit): ";

  my @in    = $in.comb;
  my @lower = $in.uc.comb;
  my @upper = $in.lc.comb;

  for ^@in.elems -> $i
  {
    if    @lower[$i] eq @upper[$i] { print @in[$i]; }     # Not a letter
    elsif @lower[$i] eq @in[$i]    { print @upper[$i]; }  # Lower case -> Upper
    elsif @upper[$i] eq @in[$i]    { print @lower[$i]; }  # Upper case -> Upper
  }

  print "\n";
}
```

Chapter 8

Exercise 8.1

```
(1 .. 10).map({ $^a + $^b }); # -> (3 7 11 15 19)
```

Because we specify 2 placeholder arguments, it takes 2 at a time.

If we give it an odd number of elements, it will fail:

```
> my @a = 1..11;  # -> [1 2 3 4 5 6 7 8 9 10 11]
> @a.map({ $^a + $^b });
Too few positionals passed; expected 2 arguments but got 1 in block  at
<unknown file> line 1
```

Exercise 8.2

Get all two digit prime numbers:

File: primes

```
my @primes;
for 10 .. 99 -> $candidate
{
  @primes.push($candidate) if $candidate.is-prime;
}
say "Found { @primes.elems } primes: { @primes }.";
```

```
$ raku primes
Found 21 primes: 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97.
```

It is possible to use `grep` instead of a loop:

File: primes2

```
my @primes = (10 .. 99).grep(*.is-prime);
say "Found { @primes.elems } primes: { @primes }.";
```

`grep` will apply the block/code to each element in the list, and will include it (the element) in the return value if the block gives a `True` value.

This is the same:

```
my @primes = (10 .. 99).grep: *.is-prime;
my @primes = grep { .is-prime }, (10 .. 99);
```

Exercise 8.3

A random integer from 10 to 99, both included.

```
> rand(99-10+1).Int + 10;
```

But is it really correct?

```
> rand(90).Int + 10; # -> [0 .. 89] + 10 -> [10 .. 99]
```

Yes it is.

But it is not instantly obvious that it is. So mere mortal programmers will get it wrong.

Conclusion: This is hard!

Exercise 8.4

A random prime number, between 1 and the given value.

```
> (1 ..  100).grep(*.is-prime).pick.say;  # ->   13
> (1 .. 1000).grep(*.is-prime).pick.say;  # -> 1861
```

Exercise 8.5

Generate a random string of ten characters usable as a password. Use letters, digits, and some special characters (as e.g. «!» and «@»).

```
> (0..9, 'A' .. 'Z', 'a' .. 'z', '!', '@', '#', '|', '&', '%').flat.roll(10).join;
b@RO5pFSAo
```

We had to use `flat` to get a one dimensional array.

Exercise 8.6

A frequency table for 1 million random numbers in the range 1..100.

File: random-frequency

```
my @frequency;

@frequency[$_]++ for (1..100).roll(1_000_000);

say "$_: @frequency[$_]" for 1..100;

print "\n";

say "Max: { max(@frequency) }";
say "Min: { min(@frequency) }";
```

Exercise 8.7

Remove duplicates from the output of `permutations`:

`unique` doesn't work, as it relies on scalar values. We have a list of lists.

File: permutations-unique

```
my @all = <a b b>.permutations;
my %seen;

print "(";

for @all -> @current
{
  my $string = @current.join;
  next if %seen{$string};
  print "(" ~ @current ~ ")";
  %seen{$string} = True;
}

say ")";
```

We used a hash (`%seen`) and add the sublist as a string when we encounter a new sequence.

We can do it without a hash. `sort` works:

```
> <a b b>.permutations.sort
((a b b) (a b b) (b a b) (b a b) (b b a) (b b a))
```

File: permutations-unique2

```
my @all = <a b b>.permutations.sort;
my $old = "";

print "(";

for @all -> @current
{
  my $string = @current.join;
  next if $string eq $old;
  print "(" ~ @current ~ ")";
  $old = $string;
}

say ")";
```

Running it:

```
$ raku permutations-unique2
((a b b)(b a b)(b b a))
```

Chapter 9

Exercise 9.1

What happens if we drop the `.Hash` part?

```
> my %hash = (1..10).Hash;  # -> {1 => 2, 3 => 4, 5 => 6, 7 => 8, 9 => 10}
> my %hash = (1..10);       # -> {1 => 2, 3 => 4, 5 => 6, 7 => 8, 9 => 10}
```

Nothing. The first line is overkill; both the assignment to a hash (`%hash`) and the final `.Hash` coerces the list to a Hash.

We must drop the assignment, or assign to a scalar to avoid this:

```
> my $hash = (1..10).Hash;  # -> {1 => 2, 3 => 4, 5 => 6, 7 => 8, 9 => 10}
> my $not  = (1..10);       # -> 1..10
```

The last one is shown without parens (a list) and square brackets (an array), and is obviously another type:

```
> say $not.WHAT;  # -> (Range)
```

Exercise 9.2

What happens if we have duplicate valuesin a hash:

```
> <1 2 1 3 1 4 1 5>.Hash;
{1 => 5}
```

They are squished. The last one wins.

Chapter 10

Exercise 10.1

Showing a sorted list of methods available for objects (or values) of the given type.

We have enough knowledge to have a go:

```
> Int.^methods;   # -> (new Capture Int Num Rat ...)
```

That was easy. Now we add sorting:

```
Int.^methods.sort({$^a.fc cmp $^b.fc});
No such method 'fc' for invocant of type 'Method' ...
```

So `^methods` gives us a list of objects of type `Method`, and `sort` doesn't stringify them. So we have to do that manually.

Surely the `name` method is it? We can check:

```
> Int.^methods[0];            # -> new
> Int.^methods[0].WHAT;       # -> (Method)
> Int.^methods[0].name;       # -> new
> Int.^methods[0].name.WHAT;  # -> (Str)
```

Then we wrap it in a `Map`:

```
say map(*.name, $var.^methods).sort({$^a.fc cmp $^b.fc});
```

The complete program:

File: type-methods

```
sub MAIN ($var)
{
  print "$var (of type { $var.^name }) supports: ";
  say map(*.name, $var.^methods).sort({$^a.fc cmp $^b.fc});
}
```

Note that an integer argument will get the type "IntStr", regardless of what we try to do with it.

Exercise 10.2

Multiple dispatch cannot choose between `Int` and `Str` candidate in «MAIN», when we give it an integer, as `IntStr` inherits from both of them.

We must make the candidates different:

File: intstr-gotcha2

```
multi MAIN (Int $number)
{
  say "Integer: $number";
}

multi MAIN (Str $string where /\D/)
{
  say "String: $string";
}
```

I have added a clause on the `Str` candidate that demands at least one non-digit character.

Exercise 10.3

Problem: Say that we specify 1000 as the upper limit. The programs makes a (lazy) Range. Then `grep` converts it to a an eager list, applies `is-prime` on every value to sort out the prime numbers, and finally `pick` gives us one of them.

Solution: Pick a random integer, and continue doing so until we have a prime number.

File: random-prime-smart

```
#| A random prime number between 1 and ...
sub MAIN (Int $upper-limit where * > 0)
{
  my $candidate;

  repeat
  {
    $candidate = (1 .. $upper-limit).pick;
  }
  until $candidate.is-prime;

  say $candidate;
}
```

Chapter 11

Exercise 11.1

What is the problem with this regex:

```
/\<(.*?)\>(.*)\<\/$0\>/
```

The greedy match in the middle is the problem.

```
> "This is <b>bold</b> and not <b>and bold again</b>." ~~ /\<(.*?)\>(.*)\<\/$0\>/
｢<b>bold</b> and not <b>and bold again</b>｣
 0 => ｢b｣
 1 => ｢bold</b> and not <b>and bold again｣
```

Solution: Make it non-greedy:

```
>  "This is <b>bold</b> and not <b>and bold again</b>." ~~ /\<(.*?)\>(.*?)\<\/$0\>/
｢<b>bold</b>｣
 0 => ｢b｣
 1 => ｢bold｣
```

Exercise 11.2

The rotate13 encryption code

```
"This is it".trans( ['a' .. 'z'] => ['n' .. 'z', 'a' .. 'm'] ) ①
            .trans( ['A' .. 'Z'] => ['N' .. 'Z', 'A' .. 'M'] ) ②
            .say; ③
```

① You knew that «n» is the 14th letter in the alphabet, right? We map «a» to «n», «b» to «o» and so on. We wrap the right side, so that «n» map to «a», «o» map to «b» ans so on.

② The same for upper case letters.

③ And end with printing the new string.

```
> "a".ord;            # -> 97
> "z".ord;            # -> 122
> "z".ord - "a".ord;  # -> 25
```

So 26 letters.

Halfway (the one starting the second half):

```
> chr("a".ord + 13);  # -> n
```

Shorter:

```
("a" .. "z").elems;  # -> 26
```

```
("a" .. "z")[13];  # -> n
```

Other ways of doing this?

```
my @old_lc = "a" .. "z";
my @new_lc = @old_lc[13 .. 25, 0 .. 12];
my @old_uc = "A" .. "Z";
my @new_uc = @old_uc[13 .. 25, 0 .. 12];

"This is it".trans(@old_lc => @new_lc).trans(@old_uc => @new_lc).say;
```

The new arrays have two values, each a list, because of the comma (which is a list generating operator).

This doesn't matter here, but we can fix it (on both of them!) by attaching a `.flat` at the end, like this: `my @new_lc = @old_lc[13 .. 25, 0 .. 12].flat;`.

With `rotate`:

```
$string.trans( ['a' .. 'z'] => [('a' .. 'z').list.rotate: 13] )
       .trans( ['A' .. 'Z'] => [('A' .. 'Z').list.rotate: 13] )
       .say;
```

Note that `rotate` doesn't work on Ranges, so we must explicitly convert them to Lists.

And as a stand alone program:

File: rotate13

```
sub MAIN ($string)
{
  $string.trans( ['a' .. 'z'] => [('a' .. 'z').list.rotate: 13] )
         .trans( ['A' .. 'Z'] => [('A' .. 'Z').list.rotate: 13] )
         .say;
}
```

We have to convert the range to a list, as `rotate` doesn't work with ranges.

Exercise 11.3

The regex versions of the trim-family:

Method	Regex
`$y = $x.trim-leading`	`$x ~~ /^\s*(.*)/; $y = $0.Str;`
`$y = $x.trim-trailing`	`$x ~~ /^(.?)\s$/; $y = $0.Str;`
`$y = $x.trim`	`$x ~~ /^\s*(.?)\s$/; $y = $0.Str;`

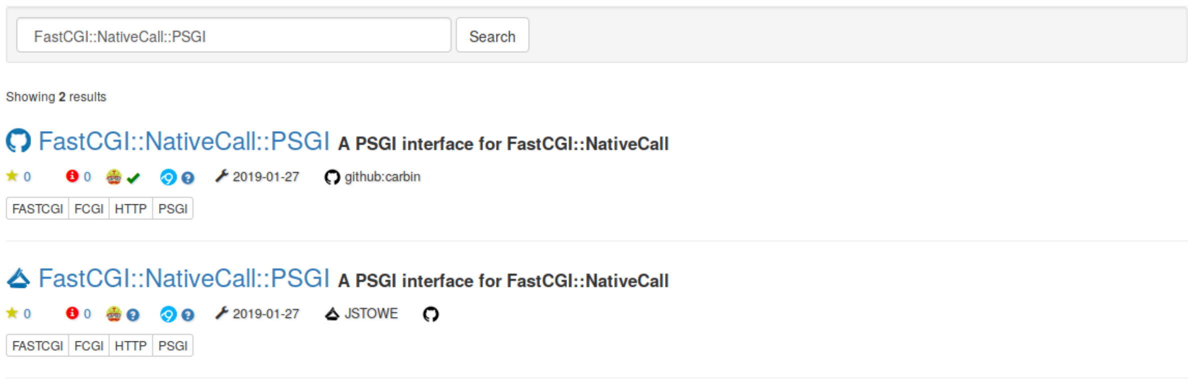
The Regex versions are obvious, right? There is no way we could make a mistake?

Note that the trim-family are not implemented as Regexes, and should be much faster.

Chapter 12

Exercise 12.1

Note that you can get two matches for a single module; one from GitHub and one from CPAN - even if they are identical (the same version):


FastCGI::NativeCall::PSGI

Search

Showing 2 results

FastCGI::NativeCall::PSGI A PSGI interface for FastCGI::NativeCall

0 0 2019-01-27 github:carbin

FASTCGI FCGI HTTP PSGI

FastCGI::NativeCall::PSGI A PSGI interface for FastCGI::NativeCall

0 0 2019-01-27 JSTOWE

FASTCGI FCGI HTTP PSGI

Figure 15. modules-fastcgi

Exercise 12.2

Installation of «Math::Trig» (possibly with «sudo zef» instead, depending on your setup):

```
$ zef install Math::Trig
===> Testing: Math::Trig
===> Testing [OK] for Math::Trig
===> Installing: Math::Trig
```

Reading the documentation, locally with «p6doc»:

```
$ p6doc Math::Trig
No Pod found in
/usr/local/share/perl6/site/sources/2E5819C0155B871E56783BE021F254FAFAD7A597
```

That didn't work. So on to the web:

- Go to https://modules.raku.org/
- Write «Math::Trig» in the Search box, and click on «Search»
- We got 1 match. Click on the link to go to https://github.com/perlpilot/p6-Math-Trig
- The documentation is useless (the file «README.md»), so click on «lib/Math» and «Trig.pm»
- No inline documentation, and no comments either. So we have to read the actual source code

Decide on a procedure to use. We can go for «deg2rad».

Writing a program:

File: math-trig

```
use Math::Trig;

for (0, 45 ... 360) -> $degree
{
  say "Degree $degree = { deg2rad($degree) } Radian.";
}
```

Running it:

```
$ raku math-trig
Degree 0 = 0 Radian.
Degree 45 = 0.7853981633974483 Radian.
Degree 90 = 1.5707963267948966 Radian.
Degree 135 = 2.356194490192345 Radian.
Degree 180 = 3.141592653589793 Radian.
Degree 225 = 3.9269908169872414 Radian.
Degree 270 = 4.71238898038469 Radian.
Degree 315 = 5.497787143782138 Radian.
Degree 360 = 6.283185307179586 Radian.
```

Chapter 13

Exercise 13.1

A file conversion program. Input is in latin1 (iso-latin-1), and output to the screen is in Unicode (utf-8).

File: isolatin2unicode

```
sub MAIN ($file-name)
{
  say slurp $file-name, enc => "latin1";
}
```

File: isolatin2unicode2

```
sub MAIN ($file-name)
{
  .say for $file-name.IO.lines(enc => "latin1");
}
```

File: isolatin2unicode3

```
.say for lines(enc => "latin1");
```

Exercise 13.2

A file comparison program (cippled edition).

File: file-equal

```
sub MAIN ($file1, $file2)
{
  (say "No such file $file1"; exit) unless $file1.IO.e;
  (say "No such file $file2"; exit) unless $file2.IO.e;

  my $size1 = $file1.IO.s;
  my $size2 = $file2.IO.s;

  (say "Files differ (different sizes)"; exit) unless $size1 == $size2;

  (say "Unable to read file $file1"; exit) unless my $fh1 = open $file1, :bin;
  (say "Unable to read file $file2"; exit) unless my $fh2 = open $file2, :bin;

  my Buf $buf1;
  my Buf $buf2;

  while $buf1 = $fh1.read(1)
  {
    $buf2 = $fh2.read(1);

     (say "Files differ"; exit) if $buf1[0] != $buf2[0]
  }

  $fh1.close;
  $fh2.close;

  say "The files are equal";
}
```

Exercise 13.3

A Raku version of the Unix «which» program:

File: which6

```
sub MAIN ($program) ①
{
  for %*ENV<PATH>.split(":") -> $dir
  {
    next unless $dir.IO.d; # Is this a directory?

    for indir($dir, &dir).sort -> $file
    {
      next if $file.d;
      next unless $file.x; ②
      if $program eq $file ③
      {
        say "$dir/$file"; ④
        exit; ⑤
      }
    }
  }
}
```

① We start with «list-path», as told, and wrap it in a `MAIN` sub.

② The second change; the test for executable is moved up.

③ Have we found the program?

④ If so, print the full path (including the program)

⑤ And stop

We could have used `last` instead of `exit`. That would have terminated the inner loop only, and the program would continue looking for the program in the other directories in our path. We can avoid that by using a label:

```
OUTER:
for %*ENV<PATH>.split(":") -> $dir
```

```
last OUTER; # was 'exit'
```

But `exit` works just fine, as long as program termination is ok.

❗ This program doesn't cope with partial paths, e.g. a file with an embedded directory separator (as in «bin/emacs»). But that is probably ok, as it will report nothing.

We can extend it, so that it is possible to get *all* matches, and not only the first:

The following changes:

```
sub MAIN ($program, :$all = False)
```

```
exit unless $all;
```

```
$ raku which6-all which
/usr/bin/which

$ raku which6-all -all which
/usr/bin/which
/bin/which
```

The complete program is available as «which6-all».

The next exercise looks closer at the duplicate program problem.

Exercise 13.4

A program that traverses the path, reporting duplicate programs.

File: check-path

```
sub MAIN
{
  my %programs;
  for %*ENV<PATH>.split(":") -> $dir
  {
    next unless $dir.IO.d; # Is this a directory?

    for indir($dir, &dir).sort -> $file
    {
      next if $file.d;
      next unless $file.x;

      %programs{$file}.push("$dir/$file");
    }
  }

  for %programs.keys.sort -> $program
  {
    if %programs{$program}.elems > 1
    {
      say $program;
      for @(%programs{$program}) { say "- $_"; }
    }
  }
}
```

Running it on my machine shows quite a lot of duplicates. The first one shown is the one that is first in the path, and the one actually chosen to execute.

Exercise 13.5

The program is the same as «check-path», as given in the previous exercise, except for the last line of code (`for @ …`) that we replace with quite a lot of new code. And I have declared two counters as well.

The code from the «files-equal» program has been modified to return `True` if the files are equal, and `False` if not.

The new part of the `if`-block gets the first version (in `$program-a`), and iterates over the rest (in `$program-b`) and compares the two versions. (Note that if we have more than two versions, they will only be compared with the first one.

And we finish off by a count of how many duplicates and different versions we found.

File: check-path-duplicates

```
sub MAIN
{
  my %programs;
  for %*ENV<PATH>.split(":") -> $dir
  {
    next unless $dir.IO.d; # Is this a directory?

    for indir($dir, &dir).sort -> $file
    {
      next if $file.d;
      next unless $file.x;

      %programs{$file}.push("$dir/$file");
    }
  }

  my $equal = 0; my $not-equal = 0;

  for %programs.keys.sort -> $program
  {
    if %programs{$program}.elems > 1
    {
      say $program;
      my $program-a = @(%programs{$program})[0];
      say "- $program-a";
      for 1 .. %programs{$program}.elems -1 -> $current
      {
        my $program-b = @(%programs{$program})[$current];
        print "- $program-b";

        files-equal($program-a, $program-b)
```

```
          ?? (say " - Equal";     $equal++;)
          !! (say " - Not equal"; $not-equal++;);
      }
    }
  }

  say "Found $equal duplicates and $not-equal different programs."
}

sub files-equal ($file1, $file2)
{
  (say "No such file $file1"; exit) unless $file1.IO.e;
  (say "No such file $file2"; exit) unless $file2.IO.e;

  my $size1 = $file1.IO.s;
  my $size2 = $file2.IO.s;

  return False unless $size1 == $size2;

  (say "Unable to read file $file1"; exit) unless my $fh1 = open $file1, :bin;
  (say "Unable to read file $file2"; exit) unless my $fh2 = open $file2, :bin;

  my Buf $buf1;
  my Buf $buf2;

  while $buf1 = $fh1.read(1)
  {
    $buf2 = $fh2.read(1);

      return False if $buf1[0] != $buf2[0]
  }

  $fh1.close;
  $fh2.close;

  return True;
}
```

Exercise 13.6

A program «ack6» that searches all non-binary files recursively from the current directory, looking for the specified string:

File: ack6

```
use Data::TextOrBinary; ①

sub is-binary ($file)
{
  return True if $file ~~ /\.[pdf|PDF|svg|SVG]$/; ①

  return ! is-text($file.IO);
}

unit sub MAIN ($search, :$verbose = False); ②

check-dir(".");

sub check-dir ($dir)
{
  say "Reading dir: $dir" if $verbose;
  for indir($dir, &dir).sort({ +$^a.IO.d ~ $^a cmp +$^b.IO.d ~ $^b }) -> $current
  {
    next unless $current.IO.r; # Skip files/directories we cannot read

    my $next = "$dir/$current"; ③

    if $current.IO.d
    {
      check-dir($next);
    }
    elsif is-binary($next) ④
    {
      say "Binary: $next (skipped)" if $verbose;
      next;
    }
    else ⑤
    {
      say "File: $next" if $verbose;

      for "$next".IO.lines -> $line ⑥
      {
        say "$next: $line" if $line.contains($search); ⑥
      }
    }
  }
}
```

① We use `Data::TextOrBinary` (from section xx) to detect if the files are binary. Note that the library doesn't always detect pdf files as binary, so I have added a test for some common file types that could cause problems.

② I have kept the debug output from «indir-recursive2», but it must be activated with the «--verbose» flag.

③ Simply to get the file with path (relative to the initial direcrtory) to save typing.

④ Skip binary files.

⑤ A file to search.

⑥ We read the content, line by line, and prints the line if we have a match

Note that **all** matching lines in a file will be shown. We can fix that easily:

```
unit sub MAIN ($search, :$verbose = False, :$first-only = False);

...

for "$next".IO.lines -> $line
{
  if $line.contains($search)
  {
    say "$next: $line";
    last if $first-only;
  }
}
```

The whole program is available as «ack6-first«.

```
$ raku ack6-first --first-only zef
```

Chapter 14

Exercise 14.1

A «cal» clone:

File: cal6

```
unit sub MAIN ();

my @month = ("", "January", "February", "March", "April", "May", "June", "July",
             "August", "September", "October", "November", "December");

my $now   = now.Date.truncated-to('month');
my $year  = $now.year;
my $month = $now.month;

say "    @month[$month] $year";
say "Mo Tu We Th Fr Sa Su";

print "   " x $now.day-of-week - 1;

loop
{
  printf('%2d ', $now.day);
  print "\n" if $now.day-of-week == 7;
  $now.=later(days => 1);
  last if $now.day == 1;
}
print "\n";
```

Exercise 14.2

Extended to accept optional values for month and year:

File: cal6-param

```
unit sub MAIN (:$year = now.Date.year, :$month = now.Date.month);

my @month = ("", "January", "February", "March", "April", "May", "June", "July",
             "August", "September", "October", "November", "December");

my $now   = Date.new($year, $month, 1);

say "    @month[$month] $year";
say "Mo Tu We Th Fr Sa Su";

print "   " x $now.day-of-week - 1;

loop
{
  printf('%2d ', $now.day);
  print "\n" if $now.day-of-week == 7;
  $now.=later(days => 1);
  last if $now.day == 1;
}

print "\n";
```

Chapter 15

Exercise 15.1

Why is the recursive version of the Fibonacci procedure slower than the loop version?

- Recursion (calling a procedure a lot of times) takes time
- The recursive version recalculates the Fibonacci numbers repeatedly. Caching could have helped.

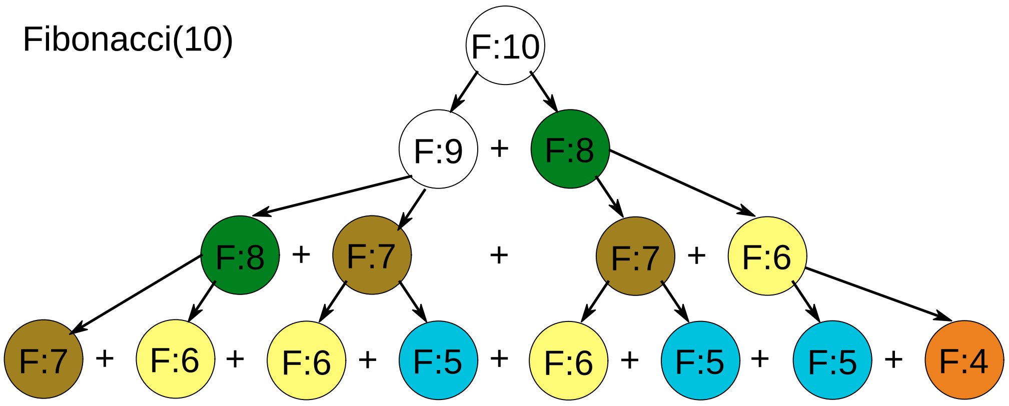


Figure 16. Fibonacci 10 Call Tree (first 4 levels)

Here we have `F:8` in two places, and both of them will be calculated (with recursion). And it gets worse; `F:7` is calculated three times, and `F:6` 4 times. An so on.

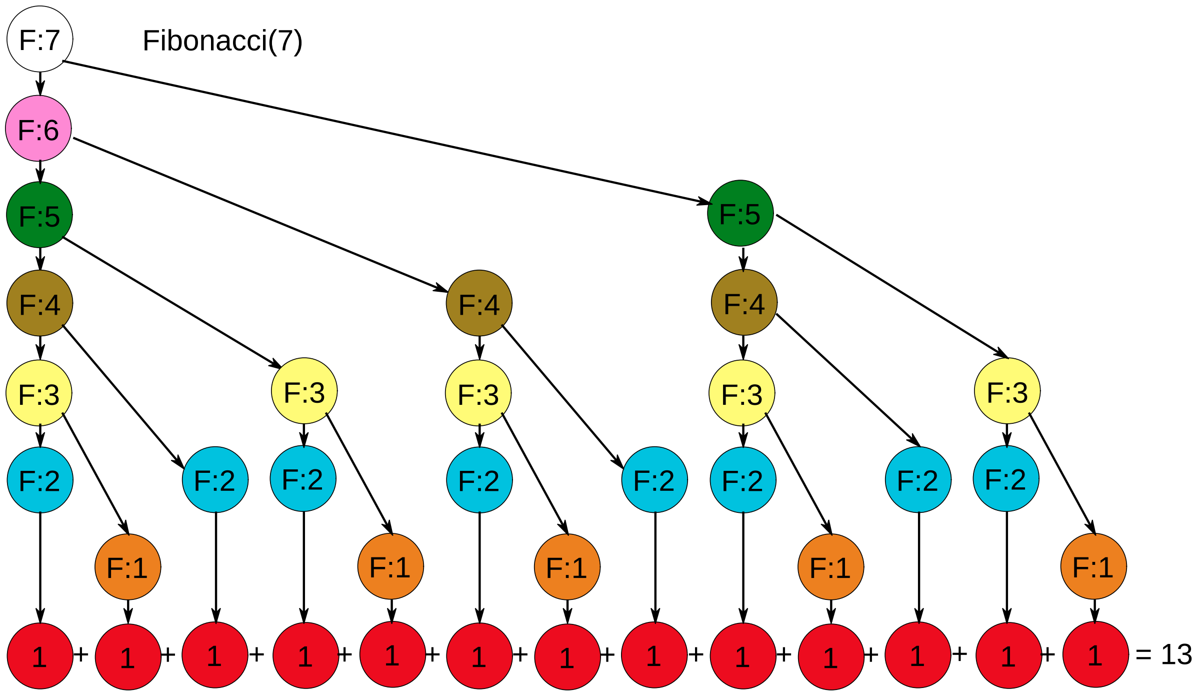


Figure 17. Fibonacci 7 Call Tree (sorted)

Here we see what happens when we recursively calculate the 7th Fibonacci Number. The sequence of the calls is each column from top to bottom, with the leftmost column first.

Note that a Sequence caches the values, so the Fibonacci Sequence (as shown in section 16.3.1, “The Fibonacci Sequence” is way more efficient.)

Exercise 15.2

A module «Dictionary.pm6» that loads a specified dictionary file (with full path), and returns a hash of all the words:

File: lib/Dictionary.pm6

```
use v6;

unit module Dictionary;

sub get-dictionary ($file where $file.IO.r) is export
{
  return $file.IO.lines.grep(* !~~ /\W/).Set;
}
```

I have chosen a `Set` - a write once version of a hash where the values can only be `True`, and the absence of the key gives `False` - as that gives shorter code.

`Set` will be covered in the «Advanced Raku» course.

We can use a hash instead, but the code is more complicated:

```
  my %hash;
  $file.IO.lines.grep(* !~~ /\W/).map({ %hash{$_} = True; });
  return %hash;
```

Note that I am updating the hash *inside* the `map`, and discard the return value of the chained operation.

And a simple test:

File: dictionary-test

```
use lib "lib";
use Dictionary;

my $dict = get-dictionary("/usr/share/dict/british-english");

say $dict<friend>;   # -> True
say $dict<friendQ>;  # -> False
```

Note that we get `(Any)` and not `False` for non-existing values if we use a hash. This doesn't matter in this case, as `(Any)` boolifies to `False`.

Exercise 15.3

Use «Dictionary.pm6» to look for palindromes in the dictionary.

File: dictionary-palindrome

```
use lib "lib";
use Dictionary;

my $dict = get-dictionary("/usr/share/dict/british-english");

say "Palindromes:";

for $dict.keys.sort -> $word
{
  say $word if $word eq $word.flip;
}
```

Exercise 15.4

Use «Dictionary.pm6» to check if the reverse version of every word in the dictionary is also a valid word.

File: dictionary-reverse

```
use lib "lib";
use Dictionary;

my $dict = get-dictionary("/usr/share/dict/british-english");

for $dict.keys.sort -> $word
{
  my $reverse = $word.flip;

  if $dict{$reverse}
  {
    say "$word -> $reverse";
  }
}
```

Exercise 15.5

Use «Dictionary.pm6» to check for anagrams of the word given as parameter to the program.

File: anagram

```
use lib "lib";
use Dictionary;

unit sub MAIN (Str $word where $word !~~ /\W/);

my $dict = get-dictionary("/usr/share/dict/british-english");

my @permutations = $word.comb.permutations;
  # This gives a list of lists (with single characters). Not a list of words.

print "Anagrams:";

for @permutations -> @chars
{
  my $candidate = @chars.join;
  next if $candidate eq $word;
  print " $candidate" if $dict{$candidate};
}
print "\n";
```

Exercise 15.6

Modifying the program Exercise 15.5 is easy.

The problem is that the program will probably take a very long time to run through a large dictionary.

Chapter 16

Exercise 16.1

`(1 .. Inf).eager` will fail eventually.

As we discovered in Exercise 3.2, integers are limitless.

So the answer is that the compiler would have run out of memory long before the values themselves would be a problem.

Exercise 16.2

A Deck of Cards using `map` instead of explicit loops.

File: deck-map

```
my @deck = (1 .. 13).map({ "S$_", "C$_", "H$_", "D$_" }).flat;

say @deck.join(",");
```

```
$ raku deck-map
S1,C1,H1,D1,S2,C2,H2,D2,S3,C3,H3,D3,S4,C4,H4,D4,S5,C5,H5,D5,S6,C6,H6,D6,S7,C7,H7,D7,S8
,C8,H8,D8,S9,C9,H9,D9,S10,C10,H10,D10,S11,C11,H11,D11,S12,C12,H12,D12,S13,C13,H13,D13
```

I had to append `.flat` as we would have gotten a list of lists otherwise.

Note that the order isn't the same. But as we usually don't rely on order, this shouldn't be a problem.

We can get the original order, but this doesn't look very nice:

File: deck-map2

```
my @deck = <S C H D>.map({ $_~"1", $_~"2", $_~"3", $_ ~"4", $_ ~"5", $_~"6",
                           $_~"7", $_~"8", $_~"9", $_~"10", $_~"11", $_~"12", $_~"13"
}).flat;

say @deck.join(",");
```

Things like `$_1` is taken as a variable name, so I had to quote it.

```
$ raku deck-map2
S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12,S13,C1,C2,C3,C4,C5,C6,C7,C8,C9,C10,C11,C12,C13,
H1,H2,H3,H4,H5,H6,H7,H8,H9,H10,H11,H12,H13,D1,D2,D3,D4,D5,D6,D7,D8,D9,D10,D11,D12,D13
```

⚠ You may have wondered if it is possible to use two `map` inside each other. The answer is yes, but it isn't possible to get at the outer`map` iterated value.

The mechanism for passing arguments to blocks (as described in section 10.15.1, “->”) does not work.

Exercise 16.3

Using `gather`/`take` to generate a deck of cards.

File: deck-gather

```
my $deck := gather
{
  for <S C H D> -> $type  # Spade, Club, Heart, Diamond
  {
    take "$type$_" for 1 .. 13;
  }
}

say $deck.join(",");
```

Note that using `gather`/`take` gave us a lazy data structure, but we got rid of that advantage by collecting all the values at once (done by the `join`).

Exercise 16.4

Question: Is it possible and/or sensible to use `take-rw` (see section 16.7.3, “take-rw” to generate a closure?

Possible: Yes.

Sensible: Perhaps. If the values in questions are objects that represent some sort of underlying data structure (a kind of binding) - and copying the object destroys that.

Objects are the topic of Chapter 17, *Classes*, but we'll not discuss closures there.

Chapter 17

Exercise 17.1

The three metods to set a person's spouse, father and mother:

File: person-spouse (partial)

```
method set-spouse (Person $spouse)
{
  $!spouse = $spouse;
}

method set-father (Person $father)
{
  $!father = $father;
}

method set-mother (Person $mother)
{
  $!mother = $mother;
}
```

Exercise 17.2

We *could* do this:

```
has Person $.spouse is rw; ①

method set-spouse (Person $spouse)
{
  $!spouse = $spouse;
  $spouse!spouse = self; ②
}
```

① As we try to change an object attribute from outside the object (from the spouse). We could have made this work by adding `is rw` on the spouse field

② My spouse has a spiuse, and that is me.

But that means that everyone can meddle with the inside of our objects, and we certainly don’t want that!

So instead we add a new method, and use that in addition to the old one:

File: person-spouse2 (partial)

```
method !set-spouse-internal (Person $spouse) ①
{
  $!spouse = $spouse;
}

method set-spouse (Person $spouse) ②
{
  $!spouse = $spouse;
  $spouse!set-spouse-internal(self); ③
}
```

① We rename the existing «set-spouse», as we need that. And we make it internal (with the `!`).

② The new «set-spouse».

③ This is the new part. We cannot set the value directly, so we ask «set-spouse-internal» to do it for us. Note the syntax: `!` (an exclamation point) instead of `.` (a period).

Exercise 17.3

File: person-children (partial)

```
method add-child (Person $child)
{
  @!child.push($child); ①
}

method get-children ②
{
  @!child;
}

method show-children
{
  @!child ③
    ?? ( say "$.name has a child named { .name }." for @!child ) ④
    !! say "$.name has no children."; ⑤
}
```

① Add the new child to our list of children.

② I have added this one, even though it wasn't part of the exercise.

③ Do we have any children?

④ Yes; print them in a loop.

⑤ No; say so.

Exercise 17.4

File: person-parenthood (partial)

```
method set-father (Person $father)
{
  $!father = $father;
  $father.add-child(self);
}

method set-mother (Person $mother)
{
  $!mother = $mother;
  $mother.add-child(self);
}
```

Exercise 17.5

The way we calculate a person’s age is wrong (as we assume that every year has 365 days):

```
submethod age { Int((now.Date - Date.new($.birthdate))/365) }
```

There are no easy way to do this, so we’ll have to do some coding.

I have written it as a stand alone program, to make it easier to test:

File: age

```
sub MAIN ($birthdate)
{
  my $now  = now.Date;
  my $then = Date.new($birthdate);

  my $delta-year  = $now.year  - $then.year;
  my $delta-month = $now.month - $then.month;
  my $delta-day   = $now.day   - $then.day;

  $delta-year-- if $delta-month <= 0 && $delta-day < 0;

  say $delta-year;
}
```

Note that we cannot use the `day-of-year` method, as a leap year would screw up the calculation.

Appendix 3. Beware of

There are a number of things that is important to beware of

A3.1 length

There is no «length» method.

Use `chars` (see 7.1.1, “chars”) on strings and `elems` (see 8.6.1, “elems (List Size)”) on lists et al.

A3.2 Objects are not strings

Match objects should be converted to strings before doing anything with them.

As should IO Objects.

```
> $*TMPDIR
"/tmp".IO

> $*TMPDIR.say
"/tmp".IO

> $*TMPDIR ~ "/myfile"
/tmp/myfile
```

They **may or may not** be converted to strings automatically.

A3.3 See also

See also https://docs.raku.org/language/traps

A3.4 Syntax Summary

A summary of extra special characters used with procedures; in the head, body or invocation (calling).

With variables

Syntax	Where	Description	See section
`:$a`	Head	A named argument	10.13.4, “Named Arguments”
`:$a!`	Head	Mandatory named argument	10.13.5, “Named Mandatory Arguments”
`:$a`	Invocation	A named argument	10.13.4, “Named Arguments”
`$:a`	Body	A named placeholder variable	10.4.1, “Named Placeholder Variables”

`$a:`	Invocation	A method called with procedure syntax	17.2.2, “Colon Syntax”

With Names Arguments

Syntax	Where	Description	See section
`:a`	Invocation	Named parameter is `True`	10.13.6, “Adverbs”
`:!a`	Invocation	Named parameter is `False`	10.13.6, “Adverbs”

Short forms of $, @ and %

Syntax	Description	See section
`$<aa>`	Looking up a named match	See the «Advanced Raku» course
`%<aa>`	looking up a hash value in the state hash variable `%`	16.6.2, “$ / @ / % (Anonymous State Variable)”

Others

Syntax	Where	Description	See section
`FOOBAR:`	Inside or just before a loop	A label	4.17.5, “LABEL”
`:12("12345")`	anywhere	A base12 number	5.1, “Octal, Hex, Binary …”
`:13<12345>`	anyhwere	A base13 number	5.1, “Octal, Hex, Binary …”
`FOO.BAR: 12, 13;`	method call	Passing arguments	17.2.2, “Colon Syntax”

Appendix 4. Raku Background and History

Perl version 1 to 4 were released from 1987 to 1993.

Perl version 5 was released in 1994. It was a complete rewrite, and all the new features killed off Perl 4 usage. Perl 5 is in active development today.

More information: https://en.wikipedia.org/wiki/Perl

Perl version 6 was conceived in 2000. It started with an invitation to the perl community to propose changes. The process was extremely time comsuming, as everything was (and still are) done by volunteers.

It was intended as the new version of Perl, but over the years it became clear that Perl 5 will continue to live - and it is being actively maintained.

https://en.wikipedia.org/wiki/Perl_6

https://en.wikipedia.org/wiki/Raku_(programming_language))

The decision to remame the language as Raku was taken by the lead developers in October 2019, but it wil ltake some time before the change has been 100% implemented.

6.a and 6.b

The alpha and beta versions were called «6.a» and «6.b»

They should not be used, so please upgrade if you have one of them.

6.c

The first stable version (version 6.c) was released on 25. December 2015.

6.d

The next version (6.d) was released in november 2018.

6.e

This version has not been released yet, but work is under way.

About Versions

For Perl 5 (and most other interpreted languages) you have one version, and that's it. If you upgrade to a newer version, that is what you have. This is true for installed modules as well.

In Raku you can have several versions of a module installed. A program will use the newest (the one with the highest version number) by default, but you can choose to explicitly use a specific version.

This applies to the language itself as well. 6d has some new features that break compatibility with 6c. You can use the old 6c semantics by explicitly doing:

```
use v6.c;
```

If you do that, the code is locked to that version forever. If you use modules, be aware that newer versions (because you upgrade them) may use features from 6d, and may have a `use v6d;` statement.

That works out, as different parts of the prgram can end up using different versions of the same module. This is however a possible developer nightmare.

The Rakudo implementation gets new features between major versions, but breaking changes are hidden behind a «PREVIEW» tag. So if you want to use something that will be part of the «6.e» version use this:

```
use v6.e.PREVIEW;
```

Don’t use that in production code. When «6.e» has been released, you should update the use statement:

```
use v6.e;
```

⚠ Note that if you specify a version, the code will be run under that version.

If you *don’t* specify a version, you will get the latest (excluding the «PREVIEW» versions).

Index