Learn Julia the Hard Way

Chris von Csefalvay

Contents

Chapter 1

Author

Chris von Csefalvay

The author is the sole and exclusive holder of all commercial rights in and over the entirety of the content. The contributions of all contributors below are acknowledged with the utmost gratitude.

The contents of this book, where applicable, represent the author's views or, where appropriate, that of the respective contributor(s). The contents of this book do not necessarily represent the views of any company, organisation or charity that the author is affiliated with.

Chapter 2

With kind contributions from

- M.Schauer
- visr
- Paulo Roberto de Oliveira Castro
- Tom White
- Svaksha
- Louis Luangkesorn

Chapter 3

With thanks to

- My amazing wife, Katie, who has been nothing but awesome throughout the creation of this manuscript.
- Peter Boardman, for his kind permission to use his Wikibook Introducing Julia as a foundation for several chapters in this book.

Chapter 4

License

Contributions

Furthermore, by contributing to this project, you grant the Author (Chris von Csefalvay) an irrevocable licence to all content you have contributed. Under this licence, the Author will be entitled to retain the profits of any use of the final document, including your contributions, as long as he provides a general attribution to you. As a contributor, please be aware that one day, the contents of this book may be published. By contributing, you waive any and all rights other than the right to be acknowledged as a contributor to the finished work.

License terms

Chapter 5

Introduction

Chapter 6

Preface

Chapter 7

Chapter 0: The Setup

First things first – we'll have to setup your computer to work with Julia. Fortunately, this is quite easy regardless of the OS you use! Less easy is the mental preparation, namely getting your head around the idea of a functional programming language. Therefore, the first part of this chapter will attempt a brief course in re-educating object-oriented programmers. You should definitely read it if you have been mainly using an object-oriented paradigm. In fact, you should read it anyway.

The OOP Re-Education Camp

When writing code, we follow some basic concepts about building up a complex structure from simple building blocks. The highest, most abstract level of these concepts is that of programming paradigms. A paradigm can be understood as a very broad guiding idea of writing code.

A **procedural** or **imperative** programming style uses a series of instructions in sequence. This is how most of us have started out programming: one instruction after another. Eventually, it gets more complicated, and subroutines have to get involved to achieve modularity, but ultimately, the idea of procedural programming is to take one step after another, in sequence.

Object-oriented programming, the currently dominant programming trend, is different. OOP has great reliance on the idea of objects, based on prototypes (often called *classes*), which are created (*instantiated*, to use OOP lingo) at a given time. These objects have particular data fields (called *attributes* or *instance variables*) and their own bits of code (*methods*) operating on the instance variables, the object and the world at large. Every object is an instance of a class, that is, it derives its basic form from a pre-defined blueprint. Of course,

that's a very simplistic description of an extremely complex idea, but do bear with me for the moment.

Functional programming is different. The main point of functional programming is, as the name suggests, to execute functions. In this way, it is very close to mathematics - so close, in fact, that it is based to a great extent on the concept of a *lambda calculus*, devised by Alonso Church, well before there were computers! Instead of classes, some functional languages, such as Julia, have the concept of *types*. A *type* is, quite simply, the kind of structure that a particular bit of information is. For instance, we are all familiar with strings and integers as types, but various other types exist, and there is nothing that keeps you from inventing your own (such as a type representing geographic coordinates).

Functional programming is older than OOP – Lisp, the great big ancestor of all things functional (and many things that are not!), was devised in the 1950s by John McCarthy. However, since then, they have mostly been used in academia to teach computing concepts to first-year undergraduates who can't wait to progress on to something they can use in the real world. One of the reasons for this was the need to represent changing data. Objects have *state*, that is, at a given time, their instance variables or attributes have a particular value, which can change depending on what happens in the real world: an object that represents a bank account, for instance, might have a `balance` attribute that may change if the account holder credits money or withdraws from it. Object-oriented languages are more efficient in representing state and making sure that concurrent transactions affect the object in the right order. Equally, functional languages were seen as less efficient when compared to imperative languages, such as C. This restricted functional languages to a primarily academic role. That is, until it was realised that functional languages can be useful for distributed systems. This spawned a new interest in the new generation of functional languages, such as Haskell, Clojure and Scala. The latter is used by Twitter extensively to manage long-running distributed processes, such as its message queuing and caching layers, while Clojure is used by a number of financial institutions and Big Data intensive users because of the ability to use Java libraries and access Hadoop/HBase easily. Functional programming is no longer a purely academic pursuit.

Julia marries the idea of functional programming to field of statistical or scientific programming. For this reason, much of Julia should come naturally to those more inclined to mathematical thought than the more systemic perspective that OOP people generally have.

As an OOP programmer, you might find yourself at a loss at times. This is normal. Experimenting with a new programming paradigm is always an endeavour that requires some mental restructuring. However, it is worthwhile – not only are multi-paradigm coders generally in higher demand, your OOP code might improve by learning functional tricks as most OOP languages do have some minimal facilities for functional programming strategies. After all, we earn our keep by finding the most appropriate solutions – whether that's

speed, reliability or taking advantage of distributed system power – for the problems we encounter. It can't ever hurt to have another arrow in our quiver.

Things to get used to

1. You might be used to things being called as methods, whereas in Julia, they will be called as functions: so e.g. Python's `array.push(x)` would correspond to Julia's `push!(array, x)`.
2. Data constructs are expressed through types. These can be as complex as classes, or sometimes even more so!
3. Not strictly an OOP versus functional thing, but Julia is 1-indexed, not 0-indexed. The _n_th element of an indexable collection has the index *n*, not *n-1*.

Installing Julia

Binaries

If you use Windows, Mac OS X, Ubuntu, Fedora/RHEL/CentOS or a generic Linux distribution that does packages, the download page for the Julia language is the easiest way to obtain an installer or package version of Julia. For the purposes of this book, we will assume you're using version v0.4. Julia is a young language, and it is developing quite a lot (although the core features have been stable for a pretty long time). There is, currently, no guarantee that the final version – or, indeed, the next version! – will look anything like it currently does.

Package managers

If you're using a *NIX distribution, you might be able to use your distro's package manager to get Julia. These are a better option for folks who have hardware or dependency issues while compiling Julia from source. Do remember that if you have installed and update your local copy of Julia via the package manager's nightly builds, you need to be aware that they run from a different directory than when you compile Julia from source.

1. **Ubuntu**: For Ubuntu users, there is a PPA (Personal Package Archive) provided for Julia. PPA's are a personal collection of the latest (stable) version of the software not included in Ubuntu by default, packaged and tested by community maintainers and/or upstream developers maintaining the software for the respective Linux distros. So all you need to do is get a shell and run the following commands:

```
sudo add-apt-repository ppa:staticfloat/juliareleases
sudo add-apt-repository ppa:staticfloat/julia-deps
sudo apt-get update
sudo apt-get install julia
```

2. **Fedora**: In Fedora:

```
sudo dnf copr enable nalimilan/julia
sudo yum install julia
```

3. **Arch**: Under Arch Linux, simply install the package, which is available in the 'community' repository:

```
sudo pacman -S julia
```

Compiling from source

Julia changes rapidly, with often more than a dozen average changes per day (!) to the source code. Therefore, some people tend to simply pull the most recent nightly build from Git and recompile it every few days. This is a great way to stay ahead of the curve, and also to come across more bugs than one would want to when learning a new language. By all means do use nightly builds when you are confident with Julia, but I do not recommend doing so for the time you are trying to navigate your way through it.

If you do want to use nightly builds (a.k.a PPA's), simply pull the master branch from the Julia GitHub repo and compile by typing make into the Terminal. To speed up the process, you might wish to use a number of concurring processes (`make -j n`, where n is the number of concurrent processes you are going to use).

IJulia

There is an implementation of the vastly successful IPython notebook environment for Julia. The IPython system, now known as the Jupyter project, is a great way to interactively use Julia in a notebook environment that users of Mathematica or similar software might be used to. It also allows literate coding, mixing Markdown notes and formatting with executable code. To install IJulia, you will need an existing IPython installation, either a distribution like Anaconda or by installing Python and the IPython notebook environment by downloading the Python distribution for your operating system. Once that is in place, install Julia using the instructions above.

To launch Julia, open a Terminal and type `julia`, which will drop you into the REPL:

```
$ julia

               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "help()" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.4.0-dev+2251 (2014-12-23 15:46 UTC)
 _/ |\__'_|_|_|\__'_|  |  master/4814557 (fork: 21 commits, 2 days)
|__/                   |  i686-linux-gnu

julia>
```

Now, you can install IJulia using the `Pkg.add("IJulia")` command. Congratulations, you now have your very own notebook environment for Julia!

You can fire up the Julia REPL and launch IJulia by:

```
using IJulia
notebook()
```

Alternatively, to launch your environment, open a new terminal tab to use the profile IPython parameter:

```
$ ipython notebook --profile julia
```

CoLaboratory

CoLaboratory is the brainchild of the IPython team. Leveraging the experience and success of IPython, it aims at creating a similar experience, in the cloud, for R and Julia. CoLaboratory is based on Chrome Apps and Google Docs, using the latter as a storage platform. CoLaboratory is a great tool for pair programming and exploring Python, R or Julia together, or for working together on complex projects.

A good introduction to CoLaboratory is this tutorial by Eric Hare and Andee Kaplan. CoLaboratory can be a great alternative to your own local Python installation, especially if scientific programming in the 'notebook' context is your thing (migrants from *Mathematica* are certain to welcome this feature!).

JuliaBox

JuliaBox is a project supported by Julia's core team, Juliabox allows you to code, store and share your work with other people. However, it is still in *beta*

and mainly used for classroom work by MIT, Stanford and CUNY. Logins are time limited and there are performance limitations. JuliaBox uses a virtual docker instance that it uses to control and execute your work in a segregated environment. If you want to play around with Julia without the need to set up anything on your own computer, JuliaBox is a good alternative. Since notebooks created on JuliaBox are effectively IJulia notebooks, the interface will be quite familiar.

A word on IDEs

As previously noted, this book diverges from Zed Shaw's framework a little. As such, I will not take you through the intricacies of setting up an IDE or a text editor. To cut a rather long story short – use what works for you! Fortunately, most text editors do have some form of support for Julia:

- Vim: julia-vim
- Emacs: Julia highlighting is provided via Emacs Speaks Statistics (ESS) through this package.
- Textmate: a Julia TextMate Bundle is available, which includes syntax support and somewhat rudimentary bundle features
- Sublime Text: Sublime-IJulia integrates IJulia into Sublime Text - installation is a little complex at this time, but worth it!
- Notepad++: Syntax highlighting is available for Notepad++.
- Light Table: supports Julia out of the box and with more IDE-like features through Juno.

Ready teddy?

Open up Julia by launching IJulia, opening the Julia app provided with the OS X version or calling Julia from the terminal (usually, julia). You are greeted by the Julia REPL.

```
               _       _ _(_)_     |  A fresh approach to technical computing
              (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
               _ _   _| |_  __ _    |  Type "help()" for help.
              | | | | | | |/ _` |   |
              | | |_| | | | (_| |   |  Version 0.4.0-dev+2214 (2014-12-20 03:45 UTC)
             _/ |\__'_|_|_|\__'_|   |  Commit ddaff3c (4 days old master)
            |__/                     |  x86_64-apple-darwin13.3.0

julia>
```

Your version may differ, as will the architecture (final line). If you seek help on Julia forums, always be sure to mention what build you have (the final three lines).

Congratulations. Your adventure begins here.

Chapter 8

Chapter 1: Let's get printing!

In this exercise, we will familiarise ourselves with the Julia REPL and print a few things.

Get to know and love the REPL

The great thing about REPLs is that they make learning a language a lot easier - especially a language that is at least partly intended to allow you to quickly prototype complex ideas in a few lines of code, manipulate your code, iterate until you get the desired results, then flesh it out or tidy it up.

Using the REPL isn't complicated, but it might be unusual at first if you have not used much of a similarly constructed REPL (such as Prelude, the Haskell REPL). It helps to remember a few commands and tricks for the future.

Modes

Julia's REPL has four 'modes', each indicated in the prompt.

Julia

Prompt: `julia>`

In this mode, you interact with the Julia engine directly. Pressing Return/Enter executes the command in the current line and prints the result. You can also access the result of the last operation in the ans variable. If you don't wish for Julia to print the result, conclude your line with a ; (semicolon).

Help

Prompt: `help>`

By typing ? at the beginning of a line, you can enter the help mode. Entering anything searches the Julia documentation for that word:

```
help?> besselj
INFO: Loading help data...
Base.besselj(nu, x)
   Bessel function of the first kind of order "nu", J_\nu(x).
```

To leave help mode, use the Backspace key.

Shell

Prompt: `shell>`

To execute shell commands, enter `;`. The REPL prompt will change to `shell>`, and anything you enter will be executed as a shell command.

To leave shell mode, use the Backspace key.

Search

Prompt: `(reverse-i-search)`

By pressing `^R` (Ctrl+R), you can initiate a reverse search of your history, including from previous sessions. It will show you any commands that match the pattern you have entered.

Key bindings

The Julia REPL uses a few key bindings that might be very familiar to those who have used *nix based systems frequently in the past. Most importantly, to exit the REPL, you can use `^D` (Ctrl+D), which will also close your shell, and you can abort a current operation using `^C` (Ctrl+C).

There are many more key bindings that the Julia REPL recognises, but these should be enough to get you off the ground.

Autocompletion and Unicode entry

Julia's autocomplete feature recognizes the `Tab` key trigger, your new best friend. Julia also knows Unicode math, so this is valid Julia:

```
julia> 2*2.5*
15.707963267948966
```

Pressing the `Tab` key on a LaTeX symbol name autocompletes to Unicode symbols:

```
julia> \sqrt[Tab]2
julia> √2
1.4142135623730951
```

For all Unicode completions, check out the Unicode conversion table in the Julia documentation. Remember, you can also use Unicode symbols in saved code.

Let's say something!

In the following, we'll be exploring a few ways to say hello to Julia. Each of these has their place in the coder's arsenal, and while you will eventually use the REPL a little less and your text editor a little more, you will probably use the REPL quite a bit to test out new ideas. Think of the REPL as your lab and the text editor as your drawing board – scientists who spend all their time in the lab eventually go mad, while those who are always at the drawing board rarely discover much!

Using the REPL

A REPL interface repeats everything you enter. This makes saying hello to the world rather simple – simply declare a variable containing the string literal "Hello, Julia!" or, even simpler, just declare the string literal. Let's see both of these in action in the REPL.

```
julia> "Hello, Julia!"
"Hello, Julia!"

julia> v = "Hello, Julia!"
"Hello, Julia!"

julia> v
"Hello, Julia!"
```

Using `println()`

Now for some actual coding. Time to invoke our first real function. The `println` function prints the string representation of an object.

```
julia> println("Hello, Julia!")
Hello, Julia!
```

You might notice that the "Hello, Julia!" string is not printed in bold type. This is to indicate that rather than part of the REPL's print cycle, it is a system output.

Using string concatenation

String concatenation is just a fancy name for putting strings together. In this case, we create two variables that represent strings, then use the string() function to put them together. The `string()` function concatenates each of its positional arguments.

```
julia> what = "Hello"
"Hello"

julia> whom = "Julia"
"Julia"

julia> string(what, ", ", whom, "!")
"Hello, Julia!"
```

Julia also allows for variables to be called within string literals. So the above is equivalent to:

```
julia> "$what, $whom\!"
"Hello, Julia!"
```

And this is true even for maths (or any function!)

```
julia> "2 plus 2 is $(2+2)."
"2 plus 2 is 4."
```

Running Julia programs

To run a Julia program, you have two options – either include it in another program or the REPL, or specify it as a positional argument in the command line.

Let's open a file in our favourite text editor, call it `hello.jl` (the commonly accepted file name for a Julia program), and enter

```
println("hello world")
```

We now have two ways to launch it.

By opening the REPL, we can `include` the file. This will evaluate everything in the file, then return the result of the last valid expression.

```
julia> include("hello.jl")
"Hello, Julia!"
```

Alternatively, you can merely grace your command line with your greeting to Julia by launching Julia with the appropriate argument:

```
$ julia hello.jl
"Hello, Julia!"
```

Chapter Conclusion

The journey of a thousand miles begins with a single step. If REPLs are something new to you (they are rarely used with a number of OOP languages, and most people using it in Python are using functional(ish) paradigms), then this chapter was two steps at the very least. Give yourself a pat on the back and a cookie, and play around with Julia. See you soon with the next chapter!

Chapter 9

Chapter 2: Variables

We have, of course, encountered variables in the previous chapter, where we used some for string substitution. For a functional programming language, mastery of how variables work is essential. Julia is extremely flexible when it comes to variables, but wielding this flexibility will require finesse.

Assigning variables

To assign a variable, simply use `=` (a single equals sign). That, really, is *it*. Nice, huh?

```
julia> m = 2.32
2.32

julia> m
2.32
```

Accessing an undefined variable yields an exception, of course:

```
julia> n
ERROR: n not defined
```

Julia does not require you to explicitly declare variables before assignment (indeed, there is no useful way to do so).

Variable naming

In general, you can use just about anything you can type as a variable name. This includes Unicode characters from a quite astounding range. In case you ever wanted to give your code some Christmas spirit, you are free to do so. In the following, we will be proving, using the comparison operator (>), that winter is not warmer than summer:

```
julia>   = -12
-12

julia>   = 27
27

julia>   >
false
```

The above is, believe it or not, perfectly valid Julia. Whether it is perfectly sensible Julia, too, is a different question. In general, using Unicode variable names for anything but commonly accepted scientific symbols, such as , is not the best idea, lest you might leave readers of your code trying to figure out which of the thirty or so similar looking Unicode symbols you meant.

The stylistic convention of Julia is to use lowercase variable names, with words separated by _ (underscore), and only if necessary for the sake of legibility. Variables may not start with numbers and exclamation marks.

Goodwin's dream

In 1894, an amateur mathematician from Indiana, Edward J. Goodwin, proposed a way to square the circle (a feat proven to be impossible by the Lindeman-Weierstrass theorem proven a decade or so earlier). He lobbied long enough for his 'new mathematical truth' to be introduced as a Bill in the Indiana House of Representatives, in what became known as the *Indiana Pi Bill* of 1897. This inferred a value of approximately 3.2 for . Fortunately, the intervention of Purdue professor C.A. Waldo helped to defeat the already much-ridiculed bill in the Senate, and the matter was laid to rest.

Julia, on the other hand, allows you to redefine the value of . The following is entirely correct Julia:

```
julia>
  = 3.1415926535897...

julia>   = 3.2
```

```
Warning: imported binding for  overwritten in module Main
3.2

julia>
3.2
```

It is, on the other hand, really bad practice to redefine set constants. Therefore, should you encounter the opportunity to redefine , learn from the sad case of Mr. Goodwin and try to resist the temptation.

Assigning variables to variables

It is important to understand what the effect of assigning a variable to a variable is. It is perfectly valid Julia to have multiple variables point at each other. However, doing so creates a 'shallow copy' – that is, a reference to the same memory address of where the original copy is located. If you modify one variable pointing at it, the value of the other changes, too. In the following example, we will be creating an array `m`, then set `n` be equal to `m`. We then carry out an operation that changes the value of `m`, marked by the exclamation mark (don't worry if that part is new to you, it will be explained in Exercise 3: Collections – for now, all you need to know is that `pop!` takes the last element of an indexable collection, returns it and removes it from the collection). When we then call `n`, we see that its value, too, has been affected by the operation – that is because it did not so much have a value but acted merely as a reference to 'whatever is in m'.

```
julia> m = [1,2,3,4]          # Creating array
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> n = m                  # Setting n to point to m
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> pop!(m)                # Altering m
4

julia> n                      # n is also changed as a result.
```

```
3-element Array{Int64,1}:
 1
 2
 3
```

We will be considering a way to get around this by 'deep copying' a value in chapter [X].

Literals

Literals are ways to enter various data types, whether into the REPL or a program.

Strings

We have already encountered string literals. To enter a string literal, simply delimit it by `"` (double quotation marks). Unlike Python or JavaScript, Julia does **not** accept single quotation marks.

One-dimensional arrays

1D array literals are delimited by square brackets (`[]`). Each element they contain has to be either a variable or an otherwise valid literal, but they do not all have to be the same type (something true for most collections within Julia). Values are separated by a `,` (comma):

```
julia> arr1 = [1, 2, "sausage", ]
4-element Array{Any,1}:
     1
     2
      "sausage"
      = 3.1415926535897...
```

Ranges and range arrays

A range, in Julia, is simply a shorthand for a sequence of numbers that are all spaced equally. A range can be created using the `range()` function as `range(start, end)`, but it is usually denoted in a shorthand literal, `start:end`. Ranges are interpreted as arrays, and you can create *range arrays*, arrays that are formed from a range, by simply enclosing the range notation in brackets:

```
julia> [0:10]
11-element Array{Int64,1}:
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
```

As you can see, a range in Julia includes both its start and end element. A range doesn't have to be created between integers – `[0.5:3.5]` would return an array of `[0.5, 1.5, 2.5, 3.5]`.

An array may have an optional middle `step` attribute. To obtain an array of the numbers from 0 to 30 in steps of 10, you would enter the range array literal [0:10:30]:

```
julia> [0:10:30]
4-element Array{Int64,1}:
  0
 10
 20
 30
```

Multidimensional arrays

In some languages, multidimensional arrays can be created by enclosing one-dimensional arrays in another array, and so on. Unfortunately, that is not the case in Julia, so `[[1,2],[3,4]]` would yield the one-dimensional array `[1,2,3,4]`. Rather, to create a multidimensional array, separate values by empty space and rows by a `;` (semicolon):

```
julia> md_array = [1 1 2 3 5; 8 13 21 34 55; 89 144 233 377 610]
3x5 Array{Int64,2}:
  1    1    2    3    5
  8   13   21   34   55
 89  144  233  377  610
```

Alternatively, you can enter columns, using square brackets (remember not to let your ingrained reflexes from Python take over and put a comma between the arrays!):

```
julia> [[1,2,3] [4,5,6]]
3x2 Array{Int64,2}:
 1  4
 2  5
 3  6
```

When entering a multidimensional array, each row has to have the same length. Failing to do so raises an error:

```
julia> md_sparse_array = [1 1 2; 8 13 21 34 55]
ERROR: argument count does not match specified shape
 in hvcat at abstractarray.jl:802
```

Tuples

Tuples are similar to one-dimensional arrays, in that they consist of a number of values. They are, however, unlike array literals in that they are immutable – once assigned, you cannot change the values in a tuple. Tuples are delimited by `()` (round brackets) and values are separated by commas.

```
julia> fibo_tuple = (1, 1, 2, 3, 5)
(1,1,2,3,5)
```

To demonstrate the difference between arrays and tuples, consider the following:

```
julia> fibo_arr = [1, 1, 2, 3, 5]
5-element Array{Int64,1}:
 1
 1
 2
 3
 5

julia> fibo_arr[2] = 0
0

julia> fibo_arr
5-element Array{Int64,1}:
 1
 0
 2
 3
 5
```

```
julia> fibo_tuple[2] = 0
ERROR: `setindex!` has no method matching setindex!(::(Int64,Int64,Int64,Int64,Int64), ::Int6
```

In this listing, we have created an array with the first five non-zero elements of the Fibonacci sequence. We then have used an accessor (`fibo_arr[2]` is an accessor that retrieves the second element of the array `fibo_arr`) to change the second value in the array to zero. Calling the array shows that this was successful. On the other hand, trying to do the same with the tuple `fibo_tuple` of the same values that we declared earlier yields an error. This is because tuples are immutable.

Dicts

Dicts are what in some other languages are known as *associative arrays* or *maps*: they contain key-value pairs. A dict is delimited by square brackets. Individual mappings (key-value pairs) are separated by commas. Keys are separated from values by `=>` (a double-arrow).

```
julia> statisticians = Dict("Gosset" => "1876-1937", "Pearson" => "1857-1936", "Galton" => "1
Dict{ASCIIString,ASCIIString} with 3 entries:
  "Galton"  => "1822-1911"
  "Pearson" => "1857-1936"
  "Gosset"  => "1876-1937"
```

Sets

Sets are similar to other collections in that they contain various values. However, unlike arrays and tuples, they are unordered and unique-constrained: no element may occur multiple times within the set.

Sets do not have a specific notation, but rather are created using a syntax we will use a great deal for creating all kinds of objects: by using the type (`Set`) and entering the values to constitute the set in round brackets *and* square braces:

```
julia> stooges = Set(["Moe", "Curly", "Larry"])
Set(ASCIIString["Moe","Larry","Curly"])
```

Sets do accept duplicates at time of construction, but the resulting set will still only contain one of each unique element:

```
julia> beatles = Set(["Lennon", "McCartney", "Harrison", "Starr", "Lennon"])
Set(ASCIIString["Starr","Harrison","McCartney","Lennon"])
```

Sets are unordered, meaning that two sets containing the same elements are equal:

```
julia> Set(["Marsellus", "Jules", "Vincent"]) == Set(["Jules", "Vincent", "Marsellu
true
```

An empty set is created by

```
julia> Set([])
Set{Any}()
```

Conclusion

In this chapter, we have learned how to construct a handful of literals for the main data types and assign them to variables. This should give us a promising start for our next chapter, in which we are going to explore collections.

Chapter 10

Chapter 3: Types

Julia's type system

A type system describes a programming language's way of handling individual pieces of data and determining how to operate on them based on their type. Julia's type system is primarily *dynamic*, meaning that there is no need to tell Julia what type a particular value is. This is useful, in that you can write fairly complex applications without ever needing to specify types. You might, then, be tempted to disregard types as an advanced feature that you cannot be bothered right now. However, a good understanding of types is extremely helpful to mastering a functional language.

Julia's dynamic system is augmented by the ability to specify types where needed. This has two advantages. First, type specification leads to more efficient code. It will make your code more stable, much faster and much more robust. At the same time, unlike in a statically typed language, you do not need to get your head around types at the very beginning. Thus, you can treat this chapter not so much as a tutorial exercise but as a reference you can come back to every now and then.

It's important to understand that in Julia, *types belong to values, not variables.* It's also important to understand the hierarchy of types in Julia.

Types may be *abstract* or *concrete*. You can think of an abstract type as a type that is intended solely to act as supertypes of other types, rather than types of particular objects. Thus, no object that is not a type can have an abstract type as its type.

Concrete types are intended to be the types of actual objects. They are always subtypes of abstract types. This is because concrete types cannot have subtypes, and also because you can't create types that don't have supertypes (`Any` is the default supertype for any type you create). Here is useful to mention an

interesting property of Julia's type system: any two types always have a common ancestor type.

If this feels incredibly convoluted, just bear with it for now. It will make much more sense once you get around to its practical implementations.

Declaring and testing types

Julia's primary type operator is `::` (double-colons). It has three different uses, all fairly important, and it's crucial that you understand the different functions that `::` fulfills in the different contexts.

Declaring a (sub)type

In the context of a *statement*, such as a function, `::` appended to a variable means 'this variable is always to be of this type'. In the following, we will create a function that returns `32` as `Int8` (for now, let's ignore that we don't know much about functions and we don't quite know what integer types exist – these will all be explained shortly!).

```
julia> function restrict_this_integer()
         x::Int8 = 32
         x
       end
restrict_this_integer (generic function with 1 method)

julia> p = restrict_this_integer()
32

julia> typeof(p)
Int8
```

As we can see, the `::` within the function had the effect that the returned result would be represented as an 8-bit integer (`Int8`). Recall that this *only works in the context of a statement* – thus simply entering `x::Int8` will yield a `typeassert` error, telling us that we have provided an integer literal, which Julia understands by default to be an `Int64, to be assigned to a variable and shaped as an`Int8' – which clearly doesn't work.

Asserting a type

In every other context, `::` means 'I assert this value is of this particular type'. This is a great way to check a value for both abstract and concrete type.

For instance, you are provided a variable `input_from_user`. How do you make sure it has the right kind of value?

```
julia> input_from_user = 128
128

julia> input_from_user::Int
128

julia> input_from_user::Char
ERROR: type: typeassert: expected Char, got Int64
```

As you can see, if you specify the correct abstract type, you get the value returned, whereas in our second assertion, where we asserted that the value was of the type `Char` (used to store individual characters), we got a `typeassert` error, which we can catch later on and return to ensure that we get the right type of value.

Remember that a type hierarchy is like a Venn diagram. Every `Int64` (a concrete type) is also an `Int` (an abstract type). Therefore, asserting `input_from_user::Int64` will also yield `128`, while asserting a different concrete type, such as `Int32`, will yield a `typeassert` error.

Specifying acceptable function inputs

While we have not really discussed function inputs, you should be familiar with the general idea of a function – values go in, results go out. In Julia, you have the possibility to make sure your function only accepts values that you want it to. Consider creating a function that adds up only floating point numbers:

```
function addition(x::Float64, y::Float64)
    x + y
end
```

Calling it on two floating-point numbers will, of course, yield the expected result:

```
julia> addition(3.14, 2.71)
5.85
```

But giving it a simpler task will raise an error:

```
julia> addition(1, 1)
ERROR: `addition` has no method matching addition(::Int64, ::Int64)
```

The real meaning of this error is a little complex, and refers to one of the base features of Julia called *multiple dispatch*. In Julia, you can create multiple functions with the same name that process different types of inputs, so e.g. an `add()` function can add up `Int` and `Float` inputs but concatenate `String` type inputs. Multiple dispatch effectively creates a table for every possible type for which the function is defined and looks up the right function at call time (so you can use both abstract and concrete types without a performance penalty). What the error complaining about the lack of a method matching `addition(::Int64)` means is that Julia cannot find a definition for the name `addition` that would accept an `Int64` value.

Getting the type of a value

To obtain the type of a value, use the `typeof()` function:

```
julia> typeof(32)
Int64
```

`typeof()` is notable for treating tuples differently from most other collections. Calling `typeof()` on a tuple enumerates the types of each element, whereas calling it on, say, an `Array` value returns the `Array` notation of type (which looks for the largest common type among the values, up to `Any`):

```
julia> typeof([1, 2, "a"])
Array{Any,1}

julia> typeof((1, 2, "a"))
(Int64,Int64,ASCIIString)
```

Helpfully, the `isa()` function tells us whether something is a particular type:

```
julia> isa("River", ASCIIString)
true
```

And, of course, types have types (specifically, `DataType`)!

```
julia> typeof("River")
ASCIIString (constructor with 2 methods)

julia> typeof(ans)
DataType
```

Exploring the type hierarchy

The `<:` operator can help you find out whether the left-side type is a subtype of the right-side type. Thus, we see that `Int64` is a subtype of `Int`, but `ASCIIString` isn't!

```
julia> Int64 <: Int
true

julia> ASCIIString <: Int
false
```

To reveal the supertype of a type, use the `super()` function:

```
julia> super(ASCIIString)
DirectIndexString
```

Composite types

Composite types, known to C coders as `structs`, are more complex object structures that you can define to hold a set of values. For instance, to have a Type that would accommodate geographic coordinates, you would use a composite type. Composite types are created with the `type` keyword:

```
type GeoCoordinates
    lat::Float64
    lon::Float64
end
```

We can then create a new value with this type:

```
julia> home = GeoCoordinates(51.7519, 1.2578)
GeoCoordinates(51.7519,1.2578)

julia> typeof(home)
GeoCoordinates (constructor with 2 methods)
```

The values of a composite object are, of course, accessible using the dot notation you might be used to from many other programming languages:

```
julia> home.lat
51.7519
```

In the same way, you can assign new values to it. However, these values have to comply with the type's definition in that they have to be *convertible* to the type specified (in our case, `Float64`). So, for instance, an `Int64` input would be acceptable, since you can convert an `Int64` into a `Float64` easily. On the other hand, an `ASCIIString` would not do, since you cannot convert it into an `Int64`.

Creating your very own immutable

An *immutable* type is one which, once instantiated, cannot be changed. They are created the same way as composite types, except by using the `immutable` keyword in lieu of `type`:

```
immutable GeoCoordinates
    lat::Float64
    lon::Float64
end
```

Once instantiated, you cannot change the values. So if we would instantiate the immutable `GeoCoordinates` type with the values above, then attempt to change one of its values, we would get an error:

```
julia> home.lat = 51.75
ERROR: type GeoCoordinates is immutable
```

Type unions

Sometimes, it's useful to have a single alias for multiple types. To do so, you can create a *type union* using the constructor `Union`:

```
julia> Numeric = Union(Int, Float64)
Union(Float64, Int64)

julia> 1::Numeric
1

julia> 3.14::Numeric
3.14
```

From start to finish: creating a custom type

When you hear LSD, you might be tempted of the groovy drug that turned the '70s weird. It also refers to one of the biggest problems of early computing in

Britain – making computers make sense of Britain's odd pre-decimal currency system before it was abandoned in 1971. Under this system, there were 20 shillings (`s`) in a pound (`£` or `L`) and twelve pence (`d`) in a shilling (so, 240 pence in a pound). This made electronic book-keeping in its earliest era in Britain rather difficult. Let's see how Julia would solve the problem.

Type definition

First of all, we need a type *definition*. We also know that this would be a *composite* type, since we want it to hold three values (known in this context as 'fields') - one for each of pounds, shillings and pence. We also know that these would have to be integers.

```
type LSD
    pounds::Int
    shillings::Int
    pence::Int
end
```

You don't strictly need to define types, but the narrower the types you define for fields when you create a new type, the faster compilation is going to be - thus, `pounds::Int` is faster than `pounds`, and `pounds::Int64` is faster than `pounds::Int`. At any rate, avoid not defining any data types, which Julia will understand as referring to the global supertype `::Any`, unless that indeed is what you want your field to embrace.

Constructor function

We have a good start, but not quite there yet. Every type can have a *constructor function*, the function executed when a new instance of a type is created. A constructor function is *inside the type definition* and *has the same name as the type*:

```
function LSD(l,s,d)
    if l < 0 || s < 0 || d < 0
        error("No negative numbers, please! We're British!")
    end
    if d > 12 || s > 20
        error("That's too many pence or shillings!")
    end
    new(l,s,d)
end
```

Don't worry if this looks a little strange – since we haven't dealt with functions yet, most of this is going to be alien to you. What the function `LSD(l,s,d)` does is to, first, test whether any of `l`, `s` or `d` are negative or whether there are more pence or shillings than there could be in a shilling or a pound, respectively. In both of these cases, it raises an error. If the values do comply, it creates the new instance of the `LSD` composite type using the `new(l,s,d)` keyword.

The full type definition, therefore, would look like this:

```
type LSD
    pounds::Int
    shillings::Int
    pence::Int

    function LSD(l,s,d)
        if l < 0 || s < 0 || d < 0
            error("No negative numbers, please! We're British!")
        end
        if d > 12 || s > 20
            error("That's too many pence or shillings!")
        end
        new(l,s,d)
    end
end
```

As we can see, we can now create valid prices in the old LSD system:

```
julia> biscuits = LSD(0,1,3)
LSD(0,1,3)
```

And the constructor function makes sure we don't contravene the constraints we set up earlier

```
julia> sausages = LSD(1,25,31)
ERROR: That's too many pence or shillings!
 in LSD at none:11

julia> national_debt = LSD(-1000000000,0,0)
ERROR: No negative numbers, please! We're British!
 in LSD at none:8
```

We can, of course, use dot notation to access constituent values of the type, the names of which derive from the beginning of our definition:

```
julia> biscuits.pence
3
```

Type methods

Let's see how our new type deals with some simple maths:

```
julia> biscuits = LSD(0,1,3)
LSD(0,1,3)

julia> gravy = LSD(0,0,5)
LSD(0,0,5)

julia> biscuits + gravy
ERROR: `+` has no method matching +(::LSD, ::LSD)
```

Ooops, that's not great. What the error message means is that the function `+` (addition) has no 'method' for two instances of type `LSD` (as you remember, `::` is short for 'type of'). A 'method', in Julia, is a type-specific way for an operation or function to behave. As we will discuss it in detail later on, most functions and operators in Julia are actually shorthands for a bundle of multiple methods. Julia decides which of these to call given the input, a feature known as *multiple dispatch*. So, for instance, `+` given the input `::Int` means numerical addition, but something rather different for two Boolean values:

```
julia> true + true
2
```

In fact, `+` is the 'shorthand' for over a hundred methods. You can see all of these by calling `methods()` on `+`:

```
julia> methods(+)
# 117 methods for generic function "+":
+(x::Bool) at bool.jl:36
+(x::Bool,y::Bool) at bool.jl:39
+(y::FloatingPoint,x::Bool) at bool.jl:49
+(A::BitArray{N},B::BitArray{N}) at bitarray.jl:848
```

…and so on. What we need is there to be a method that accommodates the type `LSD`. We do that by creating a method of `+` for the type `LSD`. Again, the function is less important here (it will be trivial after reading the chapter on *Functions*), what matters is the idea of creating a method to augment an existing function/operator to handle our new type:

```
julia> function +{LSD}(a::LSD, b::LSD)
         newpence = a.pence + b.pence
         newshillings = a.shillings + b.shillings
```

```
    newpounds = a.pounds + b.pounds
    subtotal = newpence + newshillings * 12 + newpounds * 240
    (pounds, balance) = divrem(subtotal, 240)
    (shillings, pence) = divrem(balance, 12)
    LSD(pounds, shillings, pence)
end
```

When entering it in the REPL, Julia tells us that `+` now has one more method:

```
+ (generic function with 118 methods)
```

Indeed, `methods(+)` shows that the new method for two `LSD`s is registered:

```
julia> methods(+)
# 118 methods for generic function "+":
+(x::Bool) at bool.jl:36
+(x::Bool,y::Bool) at bool.jl:39
...
+{LSD}(a::LSD,b::LSD) at none:2
```

And now we know the price of biscuits and gravy:

```
julia> biscuits + gravy
LSD(0,1,8)
```

Representation of types

Every type has a particular 'representation', which is what we encountered every time the REPL showed us the value of an object after entering an expression or a literal. It probably won't surprise you that representations are methods of the `Base.show()` function, and a new method to 'pretty-print' our `LSD` type (similar to creating a `__repr__` or `__str__` function in a Python class's declaration) can be created the same way:

```
function Base.show(io::IO, money::LSD)
    print(io, "£$(money.pounds), $(money.shillings)s, $(money.pence)d.")
end
```

`Base.show` has two arguments: the output channel, which we do not need to concern ourselves with, and the second argument, which is the value to be displayed. We declared a function that used the `print()` function to use the output channel on which `Base.show()` is called, and display the second argument, which is a string formatted version of the `LSD` object.

Our pretty-printing worked:

```
julia> biscuits + gravy
£0, 1s, 8d.
```

Our new type is looking quite good!

What next for `LSD`?

Of course, the `LSD` type is far from ready. We need to define a list of other methods, from subtraction to division, but the general concept ought to be clear. A new type is easy to create, but when doing so, you as a developer need to keep in mind what you and your users will do with this new type, and create methods accordingly. Chapter [X] will discuss methods in depth, but this introduction should help you think intelligently about creating new types.

Conclusion

In this chapter, we learned about the way Julia's type system is set up. The issue of types will be at the background of most of what we do in the future, so feel free to refer back to this chapter as frequently as you feel the need to. In the next chapter, we will be exploring collections, a category of types that share one important property – they all act as 'envelopes' for multiple elements, each with their distinct type.

Appendix: Julia types crib sheet

This is a selection of Julia's type tree, omitting quite a few elements. To see the full thing, you can use Tanmay Mohapatra's `julia_types.jl`.

Chapter 11

Chapter 4: Collections

A taxonomy of collections

We have already encountered some collections in the previous chapter – arrays, tuples, dicts and sets. This chapter examines what we can do with collections and how to best use them for maximum effectiveness. A collection may be *indexable*, *associative* or neither. This refers primarily to the way we access individual elements within the collection.

Think of an *indexable* collection as a shopping list – the only way to identify individual elements is by pointing out their position. If you want to refer to, say, *1 pint of milk*, you refer to it as *the fifth entry on my shopping list*. Elements of an indexable collection are accessed using the square bracket notation `collection[index]`, e.g. `shopping_list[5]`. Unusually for most programming languages and in sharp contrast to other languages like Python, indices begin with 1, rather than 0. An indexable collection is also only equal to a collection with the same elements if they are in the same order – thus `[1, 3, 5, 7] == [3, 7, 1, 5]` yields, as one would expect, `false`.

Associative collections, on the other hand, resemble a page from a phone book instead (if any of you actually still remember what one of those things is!). You wouldn't say that your phone number is on page 217, left column, fifth from the bottom. Rather, you would have a *key* (your name), by reference to which someone can find your phone number (the *value*). Associative arrays do follow the key-value pair form. They are, therefore, not accessed in the `collection[index]` form but rather in the `collection[key]` form. An associative collection is not indexable, therefore the order of entries does not matter: two associative collections will be equal as long as they contain the same key-value pairs, order regardless.

Collections that are neither associative nor indexable are a somewhat complex case. Sets are the only frequently used collection that is netiher associative nor

indexable. They are also special because of the uniqueness constraint, that is, a set may contain each value once and only once. A set does not support the commonly used methods of access, but it does support many of the collection manipulation functions, such as `push!()` and `pop!()`. The latter, in case you were wondering, returns items in a random order, since the absence of an index means sets are not ordered. Sets are not indexable, consequently two sets that contain the same elements will be considered equal, order regardless.

In addition, collections may be *mutable* or *non-mutable*. Put quite simply, a mutable collection is one where you can change particular values after creation, while in an immutable collection, you cannot do so. The typical immutable collections are, of course, tuples – once you have assigned a value to a tuple, you cannot change that value (although you can assign some other tuple or indeed an entirely different value to the variable that holds the tuple). Sets again represent a special case – they are what could be best described as *pseudo-immutable* – there is no way to access values in a way that could change them, since you normally access an element of a set by its value (which is sufficient in a set thanks to the uniqueness constraint).

	Mutable	**Immutable**
Indexable	Arrays	Tuples
Associative	Dicts	
Non-indexable and non-associative		Sets

Indexable collections

Access

Elements of an indexable collection can be accessed using the square bracket notation, by their ordinal:

```
julia> prime_array = [2, 3, 5, 7, 11]
5-element Array{Int64,1}:
  2
  3
  5
  7
 11

julia> prime_array[3]
5
```

In Julia, a range of numbers is written as `start:end` or `start:steps:end`. You can use a range to access a range of elements:

```
julia> prime_array[2:3]
2-element Array{Int64,1}:
 3
 5
```

A range always returns a collection, even if it has the length 1. This is exemplified by the difference between `prime_array[3]`, the call we made above, and `prime_array[3:3]`, which returns

```
julia> prime_array[3:3]
1-element Array{Int64,1}:
 5

julia> prime_array[3:3] == 5
false
```

You can access the last element of an indexable collection using `[end]`:

```
julia> prime_array[end]
 11
```

Incidentally, `end` behaves like a number – so `prime_array[end-1]` returns the penultimate element of the collection.

Setting

If the indexable collection you are using is also mutable (e.g. an array), any of these methods will act as a pseudo-variable and will allow you to assign a value to it. Thus `prime_array[end] = 12` would change the last element of `prime_array` to 12. You also aren't restricted to a single element: calling `prime_array[2:4] = 0` would result in

```
julia> prime_array
5-element Array{Int64,1}:
  2
  0
  0
  0
 11
```

And, of course, you can use an array or another indexable collection to replace values:

```
julia> prime_array[2:4] = [3,5,7]
3-element Array{Int64,1}:
 3
 5
 7

julia> prime_array
5-element Array{Int64,1}:
  2
  3
  5
  7
 11
```

Unpacking

Indexable collections can *unpack*: that is, they can be assigned in a single line to as many distinct variables as they have elements. This is a very useful convenience feature, and is much used in functional programming:

```
julia> actors = ["Ian McKellen", "Martin Freeman", "Elijah Wood"]
3-element Array{ASCIIString,1}:
 "Ian McKellen"
 "Martin Freeman"
 "Elijah Wood"

julia> gandalf, bilbo, frodo = actors
3-element Array{ASCIIString,1}:
 "Ian McKellen"
 "Martin Freeman"
 "Elijah Wood"

julia> gandalf
"Ian McKellen"
```

Unpacking can also be used to swap the contents of variables:

```
julia> firstname = "Irving"
julia> lastname = "Washington"

julia> firstname, lastname = lastname, firstname
("Washington","Irving")

julia> lastname
"Irving"
```

Common functions

`push!`, `pop!` and `append!`

`push!` appends the value to the end of the collection. `pop!` takes the last element of the list, returns it and removes it from the collection.

```
julia> array = [1,2,3,4]
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> push!(array, 5)
5-element Array{Int64,1}:
 1
 2
 3
 4
 5

julia> pop!(array)
5

julia> array
4-element Array{Int64,1}:
 1
 2
 3
 4
```

`append!`, somewhat unusually, puts the elements of a collection to the end of another collection:

```
julia> array2 = [5,6,7]
3-element Array{Int64,1}:
 5
 6
 7

julia> append!(array, array2)
7-element Array{Int64,1}:
 1
 2
```

```
3
4
5
6
7
```

`shift!` and `unshift!`

`shift!` ans `unshift!` are the front equivalent of `pop!` and `push!`. Similarly to `pop!`, `shift!` retrieves the first element of the collection and removes it from the collection (which *shifts* it):

```
julia> shift!(array)
1
```

Similarly to `push!`, `unshift!` puts an element to the front of the collection:

```
julia> unshift!(array, 8)
7-element Array{Int64,1}:
 8
 2
 3
 4
 5
 6
 7
```

`find` functions

There's a set of functions starting with find — such as `find()`, `findfirst()`, and `findnext()` — that you can use to get the index or indices of values within an indexable collection that match a specific value, or pass a test. These functions share three properties.

1. Their result is the *index or indices of the value sought or tested for*, with the _n_th element's index being *n*, not *n-1* as you might be used to from other languages.
2. You can use these functions in two principal forms: you can test for a value or you can test for a function, in which case the results will be the values for which the function returns `true`.
3. The `find` functions' way of telling you they haven't found anything is returning zero, since there is no element of index zero.

Let's try to find things within the following `Array`: `primes_and_one = [1,2,3,5,7,11,13,17,19,23]`

`findfirst()` `findfirst()` finds the first occurrence of a value and returns its index (or zero):

```
julia> findfirst(primes_and_one, 5)
4
```

As noted above, we can feed the `find` functions a function as well – it will return values for which the function would return a `true` value. We have not really discussed functions, but the general idea should be familiar to you. A function of the form `x -> x == 13` results in `true` of the value of `x` is 13 and `false` otherwise. Let's try to see which prime number is the first to equal 13 (don't expect big surprises):

```
julia> findfirst(x -> x == 13, primes_and_one)
7
```

You might have noticed that unlike in the case where you were searching for a particular value, *where you're searching by a function, the function comes first.* This is a little idiosyncrasy, but has to do with the way Julia determines what to do based on what it is provided with. A lot more of this will be explored in Chapter [X].

`find()` `find()` returns an array of results. Thus, for instance, let's use the `isinteger` function to see which of our primes are integers (yet again, the result should not come as a shock):

```
julia> find(isinteger, primes_and_one)
10-element Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
```

`findnext()` `findnext()` returns results from a given index onwards. Thus, if you want to know the index of the first odd number after 3 in the list of primes, you would proceed as follows (using the function `isodd`, which, as you could guess, returns `true` for odd integers and `false` otherwise):

```
julia> findnext(isodd, primes_and_one, 4)
4
```

Wait, why `4`? As you might remember, Julia is 1-indexed, not 0-indexed. Therefore, an index 'begins before' the number. The number after the first index is the first number in the sequence and so on. As such, the number after the third item in a collection is the item next to (= following) the index `4`, not `3`.

As you might have noticed, when you use a function as an argument, you do not use the parentheses you would normally use to call a function. That is because `function()` means `call this function` while `function` is merely a reference to the function object itself.

Get elements, not indices So far, we've only been getting indices. How do we get the actual elements? The answer is, of course, by using our magical `[]`(square brackets) syntax. We'll also use this as a good opportunity to introduce a very useful function, `isprime()`, which returns `true` for primes and false otherwise:

```
julia> find(isprime, primes_and_one)
9-element Array{Int64,1}:
  2
  3
  4
  5
  6
  7
  8
  9
 10

julia> primes_and_one[find(isprime,primes_and_one)]
9-element Array{Int64,1}:
  2
  3
  5
  7
 11
 13
 17
 19
 23
```

Filtering

The `filter()` function works quite similar to `find`, except in this case returns only the elements that satisfy the condition (it is, effectively, a shorthand for the previous listing).

```
julia> filter(isodd, primes_and_one)
9-element Array{Int64,1}:
  1
  3
  5
  7
 11
 13
 17
 19
 23
```

`filter()` can be used in-place, by using the `!` after the name of the function. Thus, using `filter!()`, alters the actual array rather than returning a filtered copy. Note, however, that functions ending with `!` modify the object, so, obviously, the type they act on must be mutable – you would, therefore, not be able to `filter!()` a tuple, even though you would be able to `filter()` it.

Sorting

The `sort()` function sorts an array lexicographically, generally in an ascending order:

```
julia> sort([-3, 2, 1, 7])
4-element Array{Int64,1}:
 -3
 1
 2
 7
```

You can specify the sort criterion using `by` – in this case, we will be using the absolute value function `abs` (remember not to use the parentheses symbolising function call, just the name of the function):

```
julia> sort([-3,2,1,7], by=abs)
4-element Array{Int64,1}:
  1
  2
```

```
 -3
  7
```

You can change the order of sorting using `rev`:

```
julia> sort([-3,2,1,7], by=abs, rev=true)
4-element Array{Int64,1}:
  7
 -3
  2
  1
```

And, for the great joy of algorithm nerds like me, you can choose the sort algorithm using `alg`. Julia currently supports three sorting algorithms (`InsertionSort`, `QuickSort` and `MergeSort`).

```
julia> sort([-3,2,1,7], by=abs, rev=true, alg=MergeSort)
4-element Array{Int64,1}:
  7
 -3
  2
  1
```

For mutable indexable collections, such as arrays, you can use `sort!()`, which sorts 'in place'. Of course, you can also sort non-numeric elements, or indeed anything for which the `isless()` function is defined, which sorting uses internally.

```
julia> sort(["Bayes", "Laplace", "Poisson", "Gauss"])
4-element Array{ASCIIString,1}:
 "Bayes"
 "Gauss"
 "Laplace"
 "Poisson"
```

Counting

`count()` tells you the number of instances in the collection that satisfy the criterion:

```
julia> count(isodd, primes_and_one)
9
```

`all()` and `any()`

`all()` and `any()` implement two of the mathematical concepts known as *quantifiers*, with `all()` representing the universal quantifier `\forall`, while `any()` implements the existential quantifier. These functions test whether all or any, respectively, of a collection satisfies a certain criterion, and return a single truth value.

Existence of a particular value

To find out whether an array has a particular value among its elements, you can use `in()`:

```
julia> in(2, primes)
true
```

Somewhat strangely, in the `in()` syntax, the needle comes before the haystack, i.e. `in(value, array)`, where `value` denotes the value you are looking for.

Particular types

Arrays

Arrays (the ones we used in our examples so far in this section) are mutable indexable collections. The type `Array{T,N}` indicates an `N`-dimensional array which elements' types are subtypes of `T`. For instance, `Array{Number, 2}` is an 2-dimensional array. Its elements' types descend from `Number` (e.g.`Int`, `Float64`).

Access in multidimensional arrays How do we access elements in a multidimensional array, a special form of indexable collection? Simple – in a multidimensional array, indexes go down each row, then from left to right. Therefore, this array

```
julia> md_array = ["A" "B"; "C" "D"]
2x2 Array{ASCIIString,2}:
 "A"  "B"
 "C"  "D"
```

would be indexed as follows:

```
md_array[1] = "A"
md_array[2] = "C"
md_array[3] = "B"
md_array[4] = "D"
```

This is a little counterintuitive and different from the usual row/column notation, where you would use `array[row][column]`. To retrieve a cell by row and column, use `array[row, column]`:

```
julia> md_array[1,2]
"B"
```

This generalizes for higher dimension arrays.

Tuples

A tuple is an ordered sequence of elements, like an array. A tuple is represented by parentheses and commas, rather than the square brackets used by arrays. The important difference between arrays and tuples is that *tuples are immutable*: you can't change the elements of a tuple, or the tuple itself, after creating it.

Tuples are generally used for small fixed-length collections — they're ubiquitous across Julia, for example as argument lists. Where a function returns multiple values, which, as we see, is pretty often the case, the result is a tuple.

A corollary of immutability is that none of the `!` functions work on tuples - but at the same time, using functions such as `push()` is perfectly acceptable, since it returns *a copy of the tuple with the added element*, which does not alter the original tuple.

Associative collections: dicts

An associative collection is a kind of non-indexed collection that stores (usually) pairs of values. The indexable collections you have encountered correspond to real-life examples such as a shopping list or a sequential list of train stations. Associative collections, on the other hand, have a *key* and a *value* (for this reason, they are sometimes referred to as *key-value pairs*). Julia, like many other programming languages, implements associative collections in an object called a `dict` (short for `dictionary`), which corresponds to 'maps', 'hash tables' or 'dictionaries' in other programming languages.

A dict, as we have seen, is usually created using the dict literal

```
dict = Dict("a" => 1, "b" => 2, "c" => 3)
```

The key of a key-value pair is *unique*, meaning that while several keys might point at the same value (and a key might point at a collection as a value), you cannot have duplicate keys (in database terminology, you might have heard this referred to as a *one-to-many relationship*).

Creating dicts

Other than the `Dict()` literal, there are three more ways to create a dict.

First, you can create a dict using the *comprehension syntax* for dicts. An example is

```
[i => sqrt(i) for i = 1:2:15]
```

This creates a dict with the square root of every odd number from 1 to 15. In this case, `i` can be any iterable – while ranges are the most frequently used, there is no reason why

```
Dict(i => sqrt(i) for i = [2, 5, 6, 8, 12, 64])
```

would not be equally valid.

Secondly, you can also create an empty dictionary. `Dict()` will construct an empty dictionary that permits any elements, while `Dict{type1, type2}()` will create an empty dictionary that permits any elements with keys of `type1` and values of `type2`.

Finally, earlier versions of Julia used to support what is sometimes referred to as *zip creation* of a dict, namely entering two equal-length tuples, one for keys and one for values. This is now regarded as deprecated – it still works, but you should not use it. Instead, the correct syntax is `Dict(zip(ks, vs))`:

```
ks = ("a", "b", "c")
vs = ("1", "2", "3")

julia> Dict(zip(ks,vs))
Dict{ASCIIString,ASCIIString} with 3 entries:
  "c" => "3"
  "b" => "2"
  "a" => "1"
```

Access

Just like items in an indexable arrays are keyed by their index, items in a dict are identified by their key and retrieved using the square bracket syntax:

```
julia> statisticians = Dict("Gosset" => "1876-1937", "Pearson" => "1857-1936", "Galton" => "1
Dict{ASCIIString,ASCIIString} with 3 entries:
  "Galton"  => "1822-1911"
  "Pearson" => "1857-1936"
```

```
  "Gosset"  => "1876-1937"

julia> statisticians["Gosset"]
"1876-1937"
```

One drawback of the bracket syntax is that if there is no entry for the key provided, Julia will raise an error:

```
julia> statisticians["Kendall"]
ERROR: key not found: "Kendall"
 in getindex at dict.jl:644
```

An alternative form of accessing a dictionary is using the `get()` function, which accepts a default value:

```
julia> get(statisticians, "Pearson", "I'm sorry, I don't know when this person live
"1857-1936"

julia> get(statisticians, "Kendall", "I'm sorry, I don't know when this person live
"I'm sorry, I don't know when this person lived."
```

An advantage of this is that you can create a default value, which the function will return if it cannot find the key requested. Unlike in some other programming languages, a default is *not optional* for Julia:

```
julia> get(statisticians, "Kendall")
ERROR: `get` has no method matching get(::Dict{ASCIIString,ASCIIString}, ::ASCIISt
```

Get or create (`get!()`)

Because dicts are mutable, `get!()` can try to access a value by its key and create it if not found, then return the new value. Its syntax is identical to `get()`:

```
julia> get!(statisticians, "Kendall", "I'm sorry, I don't know when this person li
"I'm sorry, I don't know when this person lived."

julia> statisticians
Dict{ASCIIString,ASCIIString} with 4 entries:
  "Galton"  => "1822-1911"
  "Pearson" => "1857-1936"
  "Kendall" => "I'm sorry, I don't know when this person lived."
  "Gosset"  => "1876-1937"
```

`pop!()`

`pop!()` returns the key-value array matching the key. If the key does not exist, it returns an optional default value or throws an error:

```
julia> pop!(statisticians, "Gosset")
"1876-1937"

julia> statisticians
Dict{ASCIIString,ASCIIString} with 3 entries:
  "Galton"  => "1822-1911"
  "Pearson" => "1857-1936"
  "Kendall" => "1907-1983"
```

Change values

To change a value, access it via the bracket syntax, then assign it the new value:

```
julia> statisticians["Kendall"] = "1907-1983"
"1907-1983"

julia> statisticians
Dict{ASCIIString,ASCIIString} with 4 entries:
  "Galton"  => "1822-1911"
  "Pearson" => "1857-1936"
  "Kendall" => "1907-1983"
  "Gosset"  => "1876-1937"
```

Checking for existence of a key or a key-value pair

To check for the existence of a key without retrieving it, you can use `haskey()`:

```
julia> haskey(statisticians, "Galton")
true

julia> haskey(statisticians, "Bayes")
false
```

You can also check for the existence of a key-value pair in a dict using the `in()` function you might be familiar with from arrays. Note that the notation of a key-value pair in this case will be in the form of a tuple `(key, value)` rather than using the associative array symbol `=>`:

```
julia> in(("Bayes", "1702-1761"), statisticians)
false
```

Retrieving keys or values

To retrieve all keys of a dict, use `keys()`. This will retrieve an object of type `KeyIterator`, which does just what the name suggests - it iterates through the keys of an array. This will be useful later on when we want to iterate through the dictionary by keys:

```
julia> keys(statisticians)
KeyIterator for a Dict{ASCIIString,ASCIIString} with 3 entries. Keys:
  "Galton"
  "Pearson"
  "Kendall"
```

You can retrieve values, predictably, by using the `values()` function:

```
julia> values(statisticians)
ValueIterator for a Dict{ASCIIString,ASCIIString} with 3 entries. Values:
  "1822-1911"
  "1857-1936"
  "1907-1983"
```

Sorting

You may have noticed that dicts are unordered. Even a dict generated by reference to a range, such as the one seen above, will not be in any particular order:

```
julia> [i => sqrt(i) for i = 1:2:15]
Dict{Int64,Float64} with 8 entries:
  7  => 2.6457513110645907
  9  => 3.0
  13 => 3.605551275463989
  3  => 1.7320508075688772
  11 => 3.3166247903554
  5  => 2.23606797749979
  15 => 3.872983346207417
  1  => 1.0
```

This is because dicts are not indexable, therefore there is no ordering that would make inherent sense. However, sometimes, we like dictionaries sorted. Disappointingly, sorting dicts is not as easy as sorting arrays: `sort(statisticians)` tells us that `'sort' has no method matching sort(::Dict{ASCIIString,ASCIIString})`. Therefore, you have to write

your own sort function that first converts `statisticians` from a dict into an array of 2-element tuples. This is because the `sort()` function has no defined methods for dicts, but it can sort arrays, including tuples, where it sorts by the first element in the tuple. Then, it iterates through the result and represents it as a dict again:

```
result = Dict{ASCIIString, ASCIIString}
for (k,v) in sort(collect(statisticians))
    result[k] => v
    println(result)
end
```

This yields the expected result:

```
Galton => 1822-1911
Kendall => 1907-1983
Pearson => 1857-1936
```

If you want the output to be in-place or yield an actual dict, you will have to augment your code a little:

```
result = Dict()
for (k::ASCIIString,v::ASCIIString) in sort(collect(statisticians))
    setindex!(result, v, k)
end
```

```
julia> result
Dict{Any,Any} with 3 entries:
  "Galton"  => "1822-1911"
  "Pearson" => "1857-1936"
  "Kendall" => "1907-1983"
```

Merging

The function `merge()` merges two or more dicts.

```
julia> mathematicians = Dict("Gauss" => "1777-1855", "Leibniz" => "1646-1716", "Abel" => "180
Dict{ASCIIString,ASCIIString} with 3 entries:
  "Abel"    => "1802-1829"
  "Leibniz" => "1646-1716"
  "Gauss"   => "1777-1855"

julia> merge(mathematicians, statisticians)
```

```
Dict{ASCIIString,ASCIIString} with 6 entries:
  "Abel"    => "1802-1829"
  "Galton"  => "1822-1911"
  "Leibniz" => "1646-1716"
  "Gauss"   => "1777-1855"
  "Pearson" => "1857-1936"
  "Kendall" => "1907-1983"
```

Its bang counterpart, `merge!()`, merges them in place, overwriting the first dict mentioned while leaving the second intact.

```
julia> merge!(mathematicians, statisticians)
Dict{ASCIIString,ASCIIString} with 6 entries:
  "Abel"    => "1802-1829"
  "Galton"  => "1822-1911"
  "Leibniz" => "1646-1716"
  "Gauss"   => "1777-1855"
  "Pearson" => "1857-1936"
  "Kendall" => "1907-1983"

julia> mathematicians
Dict{ASCIIString,ASCIIString} with 6 entries:
  "Abel"    => "1802-1829"
  "Galton"  => "1822-1911"
  "Leibniz" => "1646-1716"
  "Gauss"   => "1777-1855"
  "Pearson" => "1857-1936"
  "Kendall" => "1907-1983"

julia> statisticians
  Dict{ASCIIString,ASCIIString} with 3 entries:
  "Galton"  => "1822-1911"
  "Pearson" => "1857-1936"
  "Kendall" => "1907-1983"
```

Non-indexable non-associative collections: sets

You might be familiar with the idea of sets from maths/set theory. A set is a non-indexable, non-associative and non-mutable collection that also has unique elements. No element may occur twice, so an element's value identifies it conclusively.

Creating sets

To create a set, use the `Set()` constructor function. You can create a set that accepts any data type

```
julia> primes = Set()
Set{Any}({})
```

– or you can specify what sort of data types it would accept:

```
julia> primes = Set{Int64}()
Set{Int64}({})
```

You can create and fill sets in one go by listing elements surrounded by curly braces `{}`, and if you surround the elements with square brackets `[]` instead of curly braces `{}` Julia will guess the type:

```
julia> mersenne_primes_set = Set([3, 7, 31, 127])
Set([7,31,3,127])
```

Set operations

Sets have some unique functions that accommodate certain problems well-known from set theory: the functions `union()`, `intersect()` and `setdiff()` each, respectively, implement the union, intersection and difference of sets. Let's see how we can use this to find some similarities between the cast of two blockbusters, *The Lord of the Rings* and *The Matrix*.

First, let's create two sets with some actors from each movie:

```
lotr_actors = Set(["Elijah Wood", "Ian McKellen", "Viggo Mortensen", "Hugo Weaving"])
matrix_actors = Set(["Keanu Reeves", "Lawrence Fishburne", "Hugo Weaving"]
```

To find shared actors, we can use `intersect()`:

```
julia> intersect(lotr_actors, matrix_actors)
Set(ASCIIString["Hugo Weaving"])
```

To find actors who only starred in *Lord of the Rings* but not in *The Matrix*, we can use `setdiff()`, which shows all elements that are in the first `Set` but not the second:

```
julia> setdiff(lotr_actors, matrix_actors)
Set(ASCIIString["Elijah Wood","Ian McKellen","Viggo Mortensen"])
```

Finally, we can see actors who played in either of the movies, by using `union()`:

```
julia> union(lotr_actors, matrix_actors)
Set(ASCIIString["Elijah Wood","Ian McKellen","Hugo Weaving","Keanu Reeves","Lawren
```

Collections and types

Until now, we have generally created collections using literals, and with precious little regard to the types of information that go in them. While types will be discussed in quite a bit of detail later on, what we do know about types is that they are individual categories of data.

Julia operates what is called *type inference*: unless you tell it explicitly what type something is, it tries to figure it out best as it can. We see this in operation when we create a new collection. When a collection is created and Julia is not told that this is going to be a collection containing elements of only a particular kind or particular kinds of values, it makes an educated guess. The REPL tells us this much:

```
julia> mersenne_primes = [3, 7, 31, 127, 8191, 131071]
6-element Array{Int64,1}:
      3
      7
     31
    127
   8191
 131071
```

Upon creating the array, the REPL reports to us that it's an array consisting of six elements, all of type `Int64` – a type of signed 64-bit integer (don't worry if that means quite little to you just yet, we will be discussing various integer types in Chapter [X]). It also, helpfully, reports to us that we've got a 1-dimensional array.

Type inference and dissimilar types

What, however, if I want to play it a little wild and mix it up? Consider the following array:

```
julia> not_really_mersenne_primes = [3, 7, "potato", 127,  , "wallaby"]
6-element Array{Any,1}:
     3
     7
```

```
 "potato"
127
  = 3.1415926535897...
"wallaby"
```

As you have guessed, Julia is at a loss as to what to do with this, since we've got a mix of integers, strings and a constant thrown in for good measure. Therefore, it tells us that it has inferred the type of the collection to be `Any` – a type that applies to all objects.

Type inference and empty collections

The other marginal case is that of the empty set. Julia has a dedicated type, `None` – a subtype of `Any` – that applies to the empty set:

```
julia> empty_set = []
0-element Array{None,1}
```

Chapter 12

Chapter 5: Strings

A string is a sequence of one or more characters, and one of the most frequently used types in programming. It is therefore fitting that we acquaint ourselves with the idea of operating on strings.

String and character literals

You might be familiar by now with string and character literals from the introductory chapter, which introduced some literals, or from other programming languages. A string literal is surrounded by *double quotes*: `" string "`. Within the string, you can escape a double-quote using a backslash:

```
"This string contains a \" double quote \" "
```

Strings are *immutable* and *indexable* – indices return the characters at the index position, starting from 1.

The difference between string and character literals

String and character literals are differentiated by two indiciae:

- strings may have a length other than one while a `Char` type object necessarily has the length one (or potentially zero),
- strings are introduced and terminated by double quotation marks `""`, `Char` type objects are introduced by single apostrophes `''`.

The second of these tends to be somewhat vexing for many programmers who are used to the equivalence of `''` and `""` in languages that do not necessarily have an implemented type or class for characters mirroring `Char`. So while for instance in Python, `'a' == "a"` holds, this is not the case in Julia:

```
julia> typeof("a")
ASCIIString (constructor with 2 methods)

julia> typeof('a')
Char

julia> "a" == 'a'
false
```

Heredocs and multiline literals

Multiline literals allow you to keep longer spans of text within a single string, with line breaks. They are introduced, similarly to Python, by triple double quotation marks `"""`:

```
multiline_declaration = """
    We hold these truths to be self-evident,
    that all men are created equal,
    that they are endowed by their Creator with certain unalienable Rights,
    that among these are Life, Liberty and the pursuit of Happiness.

    That to secure these rights, Governments are instituted among Men,
    deriving their just powers from the consent of the governed...
"""

julia> println(multiline_declaration)
    We hold these truths to be self-evident,
    that all men are created equal,
    that they are endowed by their Creator with certain unalienable Rights,
    that among these are Life, Liberty and the pursuit of Happiness.

    That to secure these rights, Governments are instituted among Men,
    deriving their just powers from the consent of the governed...
```

As you can see, the use of the `"""` or 'heredoc' format has preserved the line breaks and structure of the text, a rather helpful feature where longer texts are concerned.

Regex literals

Regular expressions (regexes) are special strings that represent particular patterns. They are useful in matching and searching text, and a good knowledge of regex should be essential knowledge for any good functional programmer.

To construct a regex literal, preface the string with `r`:

```
julia> regex_literal = r"a|e|i|o|u"
r"a|e|i|o|u"
```

This is a regex literal that matches (English) wovels. Julia recognises regex literals as the type `regex`:

```
julia> typeof(regex_literal)
Regex
```

String operations

Substrings

Because strings are indexable, we can use *ranges* to select a part of a string, something we generally refer to as a *substring* or *string subsetting*:

```
julia> declaration = "When in the Course of human events"
"When in the Course of human events"

julia> declaration[1:4]
"When"
```

You might recall that a range might actually have a `step` attribute, which we can use to obtain every _n_th letter within a text. Let's see every odd-numbered letter within the first few words of the Declaration of Independence:

```
julia> declaration[1:2:end]
"We nteCus fhmneet"
```

You might remember that `end`, which we used above to extend the range across the entire length of the string, behaves like a number. Therefore, you can use it to create a substring that excludes the last, say, five letters:

```
julia> declaration[1:end-5]
"When in the Course of human e"
```

Concatenation, splitting and interpolation

Concatenating and repeating

In most programming languages, maths and string operations correspond, so you can use `+` to concatenate and `*` to repeat a string. This is *not* the case in Julia. `+` has no method for `ASCIIString`s. What you would expect `+` to do is accomplished by `*`:

```
julia> "I" * " <3 " * "Julia"
"I <3 Julia"
```

So how do you multiply a sequence of text? Easy – use the `^` operator. This is useful if you happen to have been set the old school punishment of 'lines' (writing the same sentence all over again).

```
julia> "I will not say bad things about functional languages again. " ^ 10
"I will not say bad things about functional languages again. I will not say bad thi
```

`split()`

The `split()` function separates a piece of text at a particular character, which it also removes. The result is an array of the chunks. By default, split() will separate at spaces, but you can provide any other string – not even necessarily a single character, as the third example shows:

```
julia> split(declaration)
7-element Array{SubString{ASCIIString},1}:
 "When"
 "in"
 "the"
 "Course"
 "of"
 "human"
 "events"

julia> split(declaration, "e")
6-element Array{SubString{ASCIIString},1}:
 "Wh"
 "n in th"
 " Cours"
 " of human "
 "v"
 "nts"
```

```
julia> split(declaration, "the")
2-element Array{SubString{ASCIIString},1}:
 "When in "
 " Course of human events"
```

If you provide `""` as the string to split at, Julia will split the text into individual letters.

You may also use a regex to split your text at:

```
julia> regex_literal = r"a|e|i|o|u"
julia> split(declaration, regex_literal)
12-element Array{SubString{ASCIIString},1}:
 "Wh"
 "n "
 "n th"
 " C"
 ""
 "rs"
 " "
 "f h"
 "m"
 "n "
 "v"
 "nts"
```

Needless to say, since strings are immutable, the original string is not affected by the application of `split()`.

Interpolation

String interpolation refers to the incredibly useful capability of including variable values within a string. As you might remember, we have used `*` above to concatenate strings:

```
julia> love = "<3"
"<3"

julia> "I " * love * " Julia"
"I <3 Julia"
```

While this is technically correct, it is much faster by using string interpolation, in which case we would refer back to the variable `love` as `$(love)` within the string. Julia knows this means it is to replace `$(love)` with the contents of the variable `love`:

```
julia> "I $(love) Julia"
"I <3 Julia"
```

You can put anything within the parentheses in string interpolation – anything Julia knows how to handle. For instance, including an expression in a string, you get

```
julia> "Three plus four is $(3+4)."
"Three plus four is 7."
```

If, and only if, you are referring to a variable, you can omit the parentheses (but not if you are referring to an expression):

```
julia> "I $love Julia"
"I <3 Julia"
```

Regular expressions and finding text within strings

As it has been mentioned, the main utility of regular expressions (Regexes) is to find things within long pieces of text. In the following, we will introduce the three main regex search functions of Julia - `match()`, `matchall()` and `eachmatch()`, with reference to a bit of the Declaration of Independence:

```
declaration = "We hold these truths to be self-evident, that all men are created e
```

If you are familiar with regular expressions, plod ahead! However, if

```
(GIR 0AA)|((([A-Z-[QVX]][0-9][0-9]?)|(([A-Z-[QVX]][A-Z-[IJZ]][0-9][0-9]?)|(([A-Z-[
```

looks like gobbledygook to you or you feel your regex fu is a little rusty, put down this book and consult the Regex cheatsheet or, even better, Jeffrey Friedl's amazing book on mastering regexes.

Finding and replacing using the `search()` function

If you are only concerned with finding a single instance of a search term within a string, the `search()` function returns the range index of where the search expression appears:

```
julia> search(declaration, "Government")
241:250
```

`search()` also accepts regular expressions:

```
julia> search(declaration, r"th.{2,3}")
9:13
```

To retrieve the result, rather than its index, you can pass the resulting index off to the string as the subsetting range, using the square bracket `[]` syntax:

```
julia> declaration[search(declaration, r"th.{2,3}")]
"these"
```

Ah, so that's the word it found!

Where a search string is not found, `search()` will yield `0:-1`. That is an odd result, until you realise the reason: for any string `s`, `s[0:-1]` will necessarily yield `""` (that is, an empty string).

Finding using the `match()` family of functions

The problem with `search()` is that it retrieves one, and only one, result – the first within the string passed to it. The `match()` family of functions can help us with finding more results:

- `match()` retrieves *either the first match or nothing* within the text,
- `matchall()` returns *an array of all matching substrings*, and
- `eachmatch()` returns an *iterator over all matches*.

The `match()` family of functions needs a regular expression literal as a search argument. This is so even if the regular expression does not make use of any pattern matching beyond a simple string. Thus,

```
julia> match(r"truths", declaration)
```

is valid, while

```
julia> match("truths", declaration)
```

yields an error:

```
ERROR: `match` has no method matching match(::ASCIIString, ::ASCIIString)
```

Understanding `RegexMatch` objects

Most regex search functions return an object of type `RegexMatch`. As the name reveals, a `RegexMatch` is a composite type representing a match. As such, it encapsulates (to use a little more OOP terminology than one would normally be allowed to in a book on functional programming) four values, the first three of which will be of immediate interest to us:

- `RegexMatch.match` is the matched substring,
- `RegexMatch.captures` is an array of types that represent the type of what the regex would capture,
- `RegexMatch.offset` is generally an `Int64` that represents the index of the first character of the matched string where there is a single match (e.g. when using `match()`).

To illustrate, let's consider the result of a `match()` call, which will be introduced in the next subsection:

```
m = match(r"That .*?,", declaration)

julia> m.match
"That to secure these rights,"

julia> m.captures
0-element Array{Union(SubString{UTF8String},Nothing),1}

julia> m.offset
212
```

`match()`

`match()` retrieves the first match or nothing - in this sense, it is rather similar to `search()`:

```
julia> match(r"That .*?,", declaration)
RegexMatch("That to secure these rights,")
```

The result is a `RegexMatch` object. The object can be inspected using `.match` (e.g. `match(r"truths", declaration).match`).

`matchall()`

`matchall()` returns an array of matching substrings, which is sometimes a little easier to use:

```
julia> matchall(r"That .*?,", declaration)
2-element Array{SubString{UTF8String},1}:
 "That to secure these rights,"
 "That whenever any Form of Government becomes destructive of these ends,"
```

eachmatch()

`eachmatch()` returns an object known as an iterator, specifically of the type `RegexMatchIterator`. We have on and off encountered iterators, but we will not really deal with them in depth until chapter [X], which deals with control flow. Suffice it to say an iterator is an object that containts a list of items that can be iterated through. The iterator will iterate over a list of `RegexMatch` objects, so if we want the results themselves, we will need to call the `.match` method on each of them:

```
for i in eachmatch(r"That .*?,", declaration)
    println("A matching search result is: $(i.match)")
end
```

The result is quite similar to that returned by `matchall()`:

```
A matching search result is: That to secure these rights,
A matching search result is: That whenever any Form of Government becomes destructive of thes
```

ismatch()

`ismatch()` returns a boolean value depending on whether the search text contains a match for the regex provided.

```
julia> ismatch(r"truth(s)?", declaration)
true

julia> ismatch(r"sausage(s)?", declaration)
false
```

Replacing substrings

Julia can replace substrings using the `replace()` syntax… let's try putting some sausages into the Declaration of Independence!

```
julia> replace(declaration, "truth", "sausage")
"We hold these sausages to be self-evident, that all men are created equal,..."
```

An interesting feature of `replace()` is that the replacement does not need to be a string. In fact, it is possible to pass a function as the third argument (as always, without the parentheses `()` that signify a function call). Julia will interpret this as 'replace the substring with the result of passing the substring to this function':

```
julia> replace(declaration, "truth", uppercase)
"We hold these TRUTHs to be self-evident, that all men are created equal,..."
```

Much more dignified than self-evident sausages, I'd say! At risk of repeating myself, it is important to note that since strings are immutable, `replace()` merely returns a copy of the string with the search string replaced by the replacement string or the result of the replacement function, and the original string itself will remain unaffected.

Where the substring is not found, the result will be, unsurprisingly, an unaltered string.

Regex flags

A little-known feature of Julia regexes is the ability for a regex to be appended one or more flags. These, like most of Julia's regex capability, derive from Perl's regex module `perlre`.

Flag	Function
i	Case-insensitive pattern matching
m	Treats string as a multiline string, so that `^` and `$` will refer to the start or end of any line withi
s	Treats line as a single line. This will result in `.` accepting a newline as well. When used togethe
x	Ignore non-backslashed, non-classed whitespace.

Flags are appended to the end of each regex, which might strike users more familiar with e.g. the Pythonic way of modifying the regex search object itself, as somewhat unusual:

```
multiline = r"^We"m
```

In this case, the regex `r"^We"` was augmented by the multiline flag, appended at its end.

String transformation and testing

Case transformations

Case transformations are functions that act on `String`s and transform character case. Let's examine the effect of these transformations in turn.

Function	Effect	Result
`uppercase()`	Converts the entire string to upper-case characters	`WE HOLD THESE TRUTHS TO BE SELF-EVI`
`lowercase()`	Converts the entire string to lower-case characters	`we hold these truths to be self-evi`
`ucfirst()`	Converts the first character of the string to upper-case	`We hold these truths to be self-evi`
`lcfirst()`	Converts the first character of the string ot lower-case	`we hold these truths to be self-evi`

Testing and attributes

Chapter 6: Control flow

So far, we've looked at variables, types, strings and collections – in short, the things programs operate on. Control flow is where we start to delve into *how* to engineer programs that do what we need them to.

We will be encountering two new things in this chapter. First, we will be coming across a number of *keywords*. Functional languages are generally known for having a minimum of keywords, but they usually still need some. A keyword is a word that has a particular role in the syntax of Julia. Keywords are *not functions*, and therefore are *not called* (so no need for parentheses at the end!).

The second new concept is that of *blocks*. Blocks are chunks of expressions that are 'set aside' because they are, for instance, executed if a certain criterion is met. Blocks start with a starting expression, are followed by indented lines and end on an unindented line with the `end` keyword. A typical block would be:

```
if variable1 > variable2
    println("Variable 1 is larger than Variable 2.")
end
```

In Julia, blocks are normally indented as a matter of convention, but as long as they're validly terminated with the `end` keyword, it's all good. They are not surrounded by any special characters (such as curly braces `{}` that are common in Java or Javascript) or terminators after the condition (such as the colon `:` in Python). It is up to you whether you put the condition in parentheses `()`, but this is not required and is not an unambiguously useful feature at any rate. `if` and `elseif` are keywords, and as such they are not 'called' but invoked, so using the parentheses would not be technically appropriate.

Conditions, truth values and comparisons

Much of control flow depends on the evaluation of particular conditions. Thus, for instance, an `if/else` construct may perform one action if a condition is true and a different one if it is false. Therefore, it is important to understand the role comparisons play in control flow and their effective (and idiomatic) use.

Comparison operators

Julia has six comparison operators. Each of these acts differently depending on what types it is used on, therefore we'll take the effect of each of these operators in turn depending on their types. Unlike a number of programming languages, Julia's Unicode support means it can use a wider range of symbols as operators, which makes for prettier code but might be inadvisable – the difference between `>=` and `>` is less ambiguous than the difference between and `>`, especially at small resolutions.

Operator	Name
`==` or `isequal(x,y)`	equality
`!=` or	inequality
`<`	less than
`<=` or	less or equal
`>`	greater than
`>=` or	greater or equal

Numeric comparison

When comparing numeric types, the comparison operators act as one would expect them, with some conventions that derive from the IEEE 754 standard for floating point values.

Numeric comparison can accommodate three 'special' values: `-Inf`, `Inf` and `NaN`, which might be familiar from `R` or other programming languages. The paradigms for these three are that

- `NaN` is not equal to, larger than or less than anything, including itself (`NaN == NaN` yields `false`),
- `-Inf` and `Inf` are each equal to themselves but not the other (`-Inf == -Inf` yields `true` but `-Inf == Inf` is false),
- `Inf` is greater than everything except `NaN`,
- `-Inf` is smaller than everything except `NaN`.
- `-0` is equal to `0` or `+0`, but not smaller.

Julia also provides a function called `isequal()`, which at first sight appears to mirror the `==` operator, but has a few peculiarities:

- `NaN != NaN`, but `isequal(NaN, NaN)` yields true,
- `-0 == 0`, but `isequal(-0, 0)` yields true.

`Char` comparisons

`Char` comparison is based on the *integer value* of every `Char`, which is its position in the code table (which you can obtain by using `int()`: `int('a') = 97`).

As a result, the following will hold:

- A `Char` of a lowercase letter will be larger than the `Char` of the same letter in uppercase: `'A' > 'a'` yields false,
- A `Char` plus or minus an integer yields a `Char`, which is the initial character offset by the integer: `'A' + 4` yields `'E'`.
- A `Char` plus or minus another `Char` yields an `Int`, which is the offset between the two characters: `'E' - 'A'` yields `4`.

A lot of this is a little counter-intuitive, but what you need to remember is that `Char`s are merely numerical references to where the character is located, and this is used to do comparisons.

`String` comparisons

For `AbstractString` descendants, comparison will be based on *lexicographical comparison* – or, to put it in human terms, where they would relatively be in a lexicon.

```
julia> "truths" > "sausages"
true
```

For words starting with the same characters in different cases, lowercase characters come after (i.e. are larger than) uppercase characters, much like in `Char` comparisons:

```
julia> "Truths" < "truths"
true
```

Chaining comparisons

Julia allows you to execute multiple comparisons – this is, indeed, encouraged, as it allows you to reflect mathematical relationships better and clearer in code. Comparison chaining associates from right to left:

```
julia> 3 <   < 4
true

julia> 5    < 12 > 3*e
true
```

Combining comparisons

Comparisons can be combined using the boolean operators `&&` (`and`) and `||` (`or`). The truth table of these operators is

Expression	a	b	result
`a && b`	true	true	true
	false	true	false
	true	false	false
	false	false	false
`a || b`	true	true	true
	false	true	true
	true	false	true
	false	false	false

Therefore,

```
julia>   > 2 &&   < 3
false

julia>   > 2 ||   < 3
true
```

Truthiness

No, it's not the Colbert version. Truthiness refers to whether a variable that is not a boolean variable (`true` or `false`) is evaluated as true or false.

Definition truthiness

Julia is fairly strict with existence truthiness. A non-existent variable being tested for doesn't yield `false`, it yields an error.

```
julia> if seahawks
           println("We know the Seahawks' lineup.")
       else
           println("We don't know the Seahawks' lineup.")
       end
ERROR: seahawks not defined
```

Now let's create an empty array named `seahawks` which will, one day, accommodate their lineup:

```
julia> seahawks = Array{ASCIIString}
Array{ASCIIString,N}
```

Will existence of the array, empty though it may be, yield a truthy result? Not in Julia. It will yield, again, an error:

```
julia> if seahawks
           println("We know the Seahawks' lineup.")
       else
           println("We don't know the Seahawks' lineup.")
       end
ERROR: type: non-boolean (DataType) used in boolean context
```

Value truthiness

Nor will common values yield the usual truthiness results. 0, which in some languages would yield a `false`, will again result in an error:

```
julia> seahawks = 0
0

julia> if seahawks
           println("We know the Seahawks' lineup.")
       else
           println("We don't know the Seahawks' lineup.")
       end
ERROR: type: non-boolean (Int64) used in boolean context
```

The same goes for any other value. The bottom line, for you as a programmer, is that anything but `true` or `false` as the result of evaluating a condition yields an `error` and if you wish to make use of what you would solve with truthiness in another language, you might need to explicitly test for existence or value by using a function in your conditional expression that tests for existence or value, respectively.

Implementing definition truthiness

To implement definition truthiness, Julia helpfully provides the `isdefined()` function, which yields true if a symbol is defined. Be sure to pass the symbol, not the value, to the function by prefacing it with a colon `:`, as in this case:

```
julia> if isdefined(:seahawks)
           println("We know the Seahawks' lineup.")
       else
           println("We don't know the Seahawks' lineup.")
       end
We know the Seahawks' lineup.
```

`if/elseif/else`, `?/:` and boolean switching

Conditional evaluation refers to the programming practice of evaluating a block of code if, and only if, a particular *condition* is met (i.e. if the expression used as the condition returns `true`). In Julia, this is implemented using the `if/elseif/else` syntax, the *ternary operator*, `?:` or boolean switching:

`if/elseif/else` syntax

A typical conditional evaluation block consists of an `if` statement and a condition.

```
if "A" == "A"
    println("Aristotle was right.")
end
```

Julia further allows you to include as many further cases as you wish, using `elseif`, and a catch-all case that would be called `else` and have no condition attached to it:

```
if weather == "rainy"
    println("Bring your umbrella.")
elseif weather == "windy"
    println("Dress up warm!")
elseif weather == "sunny"
    println("Don't forget sunscreen!")
else
    println("Check the darn weather yourself, I have no idea.")
```

For `weather = "rainy"`, this predictably yields

```
Bring your umbrella.
```

while for `weather = "warm"`, we get

```
Check the darn weather yourself, I have no idea.
```

Ternary operator ?/:

The ternary operator is a useful way to put a reasonably simple conditional expression into a single line of code. The ternary operator *does not create a block*, therefore there is no need to suffix it with the `end` keyword. Rather, you enter the condition, then separate it with a `?` from the result of the true and the false outcomes, each in turn separated by a colon `:`, as in this case:

```
julia> x = 0.3
0.3

julia> x < /2 ? sin(x) : cos(x)
0.29552020666133955
```

Ternary operators are great for writing simple conditionals quickly, but can make code unduly confusing. A number of style guides generally recommend using them sparingly. Some programmers, especially those coming from Java, like using a multiline notation, which makes it somewhat more legible while still not requiring a block to be created:

```
julia> x < /2 ?
          sin(x) :
          cos(x)
0.29552020666133955
```

Boolean switching || and &&

Boolean switching, which the Julia documentation refers to as *short-circuit evaluation*, allows you quickly evaluate using boolean operators. It uses two operators:

Operator	Meaning
`a || b`	Execute `b` if `a` evaluates to `false`
`a && b`	Execute `b` if `a` evaluates to true

These derive from the boolean use of `&&` and `||`, where `&&` means `and` and `||` means `or` - the logic being that for a `&&` operator, if `a` is false, `a && b` will be false, `b`'s value regardless. `||`, on the other hand, will necessarily be true if `a` is true. It might be a bit difficult to get one's head around it, so what you need to remember is the following equivalence:

- `if <condition> <statement>` is equivalent to `condition && statement`, and
- `if !<condition> <statement>` is equivalent to `condition || statement`.

Thus, let us consider the following:

```
julia> isprime(is_this_a_prime) && println("Yes, it is.")
Yes, it is.

julia> isprime(is_this_a_prime) || println("No, it isn't.")
true

julia> is_this_a_prime = 8
8

julia> isprime(is_this_a_prime) || println("No, it isn't.")
No, it isn't.
```

It is rather counter-intuitive, but the alternative to it executing the code after the boolean operator is returning `true` or `false`. This shoud not necessarily trouble us, since our focus on getting our instructions executed. Outside the REPL, what each of these functions return is irrelevant.

while

With `while`, we're entering the world of repeated (or *iterative*) evaluation. The idea of `while` is quite simple: as long as a condition is met, execution will

continue. The famed words of the pirate captain whose name has never been submitted to history can be rendered in a `while` clause thus:

```
while morale != improved
    continue_flogging()
end
```

`while`, along with `for`, is a *loop operator*, meaning - unsurprisingly - that it creates a loop that is executed over and over again until the conditional variable changes. While `for` loops generally need a limited range in which they are allowed to run, `while` loops can sometimes become infinite and turn into *runaway loops*. This is best avoided, not the least because it tends to crash systems or at the very least take up capacity for no good reason.

Breaking a `while` loop: `break`

A `while` loop can be terminated by the `break` keyword prematurely (that is, before it has reached its condition). Thus, the following loop would only be executed five times, not ten:

```
    while i < 10


    i += 1
    println("The value of i is ", i)
    if i == 5
        break
    end
end
```

The result is:

```
The value of i is 1
The value of i is 2
The value of i is 3
The value of i is 4
The value of i is 5
```

This is because after five evaluations, the conditional would have evaluated to true and led to the `break` keyword breaking the loop.

Skipping over results: `continue`

Let's assume we have suffered a grievous blow to our heads and forgotten all we knew about `for` loops and negation. Through some odd twist of fate, we absolutely *need* to list all non-primes from one to ten, and we need to use a `while` loop for this.

The `continue` keyword instructs a loop to finish evaluating the current value of the iterator and go to the next. As such, you would want to put it as early as possible - there is no use instructing Julia to stop looking at the current iterator when

`for`

Like `while`, `for` is a loop operator. Unlike `while`, it operates not *as long as* a condition is met but rather until it has burned through an *iterable*. As we have seen, a number of objects consist of smaller chunks and are capable of being iterated over this, one by one. The archetypical iterable is a `Range` object. In the following, we will use a `Range` literal (created by a colon `:`) from one to 10 in steps of 2, and get Julia to tell us which numbers in that range are primes:

```
julia> for i in 1:2:10
           println(isprime(i))
       end
false
true
true
true
false
```

Iterating over indexable collections

Other iterables include indexable collections:

```
julia> for j in ['r', 'o', 'y', 'g', 'b', 'i', 'v']
           println(j)
       end
r
o
y
g
b
i
v
```

This includes, incidentally, multidimensional arrays – but don't forget the direction of iteration (column by column, top-down):

```
ulia> md_array = [1 1 2 3 5; 8 13 21 34 55; 89 144 233 377 610]
3x5 Array{Int64,2}:
  1    1    2    3    5
  8   13   21   34   55
 89  144  233  377  610

julia> for each in md_array
           println(each)
       end
1
8
89
...
55
610
```

Iterating over dicts

As we have seen, dicts are non-indexable. Nevertheless, Julia can iterate over a dict. There are two ways to accomplish this.

Tuple iteration

In tuple iteration, each key-value pair is seen as a tuple and returned as such:

```
julia> for statistician in statisticians
           println("$(statistician[1]) lived $(statistician[2]).")
       end
Galton lived 1822-1911.
Pearson lived 1857-1936.
Gosset lived 1876-1937.
```

While this does the job, it is not particularly graceful. It is, however, useful if we need to have the key and the value in the same object, such as in the case of conversion scripts often encountered in 'data munging'.

Key-value (`k,v`) iteration

A better way to iterate over a dict assigns two variables, rather than one, as iterators, one each for the key and the value. In this syntax, the above could be re-written as:

```
julia> for (name,years) in statisticians
           println("$name lived $years.")
       end
```

Iteration over strings

Iteration over a string results in iteration over each character. The individual characters are interpreted as objects of type `Char`:

```
julia> for each in "Sausage"
           println("$each is of the type $(typeof(each))")
       end
S is of the type Char
a is of the type Char
u is of the type Char
s is of the type Char
a is of the type Char
g is of the type Char
e is of the type Char
```

Compound expressions: `begin`/`end` and `;`

A *compound expression* is somewhat similar to a function in that it is a predefined sequence of functions that is executed one by one. Compound expressions allow you to execute small and concise blocks of code in sequence and return the result of the last calculation. There are two ways to create compound expressions, using a `begin`/`end` syntax creating a block or surrounding the compound expression with parentheses `()` and delimiting each instruction with `;`.

`begin`/`end` blocks

A `begin`/`end` structure creates a block and returns the result of the last line evaluated.

```
julia> circumference = begin
           r = 3
           2*r*
       end
18.84955592153876
```

`;` syntax

One of the benefits of compound expressions is the ability to put a lot into a small space. This is where the `;` syntax shines. Somewhat similar to anonymous functions or `lambda`s in other languages, such as Python, you can simplify the calculation above to

```
julia> circumference = (r = 3; 2*r* )
18.84955592153876
```

Chapter 13

Chapter 7: Functions and methods

Julia is what is described as a *functional programming language*, meaning that functions are the principal building blocks of a Julia program (as opposed to objects and their instances in OOP). Introducing functions is the last part we are missing before we can start building fully-fledged applications to solve real world problems. Let's get cracking!

Syntax and arguments

General syntax and invocation

There are two general ways to define a function. The first way is usually suited for simple, single-expression functions, while the second way is more suitable for longer functions that include multiple expressions.

Single expression functions

Single expression functions are written very similarly to their mathematical form:

```
julia> geom_average(a,b) = sqrt(a^2 + b^2)
geom_average (generic function with 1 method)

julia> geom_average(3,4)
5.0
```

Multiple expression functions

If your function is more complex, and needs to evaluate multiple functions, this syntax is no longer suitable. The syntax to use in such cases uses a block, introduced by `function` and terminated by `end`, to describe the function:

```
julia> function breakfast(pancakes, coffee)
           println("$coffee cups of coffee and $pancakes pancakes, please.")
       end
breakfast (generic function with 1 method)

julia> breakfast(2,4)
4 cups of coffee and 2 pancakes, please.
```

Return values

In general, Julia returns the last value to come from the last calculation within the block:

```
julia> function dinner(sausages, mash)
           cost_of_sausages = sausages * 0.85
           cost_of_mash = (mash == true ? 0.60 : 0.00)
           cost_of_sausages + cost_of_mash
       end
dinner (generic function with 1 method)

julia> dinner(2, true)
2.3
```

While we haven't told Julia what we exactly want the function to return, it infers that it would probably be the result of the last calculation (`cost_of_sausages + cost_of_mash`).

Now imagine that the fictitious canteen, who are so keen on calculating the cost of sausages and mash for dinner, get back to you and want the function to be changed. They are, it turns out, only interested in the cost of sausages. You could simply put `cost_of_sausages` to the very end of the function, before the `end` keyword, or you could use the `return` keyword, which will tell the function what to give back. Let's redefine `dinner(sausages, mash)` to fit the canteen's expectations using the `return` keyword:

```
julia> function dinner(sausages, mash)
           cost_of_sausages = sausages * 0.85
           cost_of_mash = (mash == true ? 0.60 : 0.00)
```

```
            return cost_of_sausages
        end
dinner (generic function with 1 method)

julia> dinner(2, true)
1.7
```

As a matter of style, `return` is a good idea to use, even if the function would return the right value. Whoever ends up debugging the script will be grateful you told them what exactly a function ends up returning.

Variable numbers of positional arguments: ... ('splats')

The above simple function had a definite number of arguments that had to be in a particular order. Arguments where the identity of the particular argument is determined by its position among the arguments are called *positional arguments* – so in the example above, Julia knew the argument `2` related to `sausages`, not `mash, because that's the position in which it was defined. But what if you don't know how many inputs you are likely to get for a particular function? Let us imagine a function, called`shout()', that shouts the patrons' orders back to the short-order cook. Some customers want a long list of items, others just one or two.

One way to implement this is to expect an array argument:

```
function shout(food_array)
    food_items = join(food_array, ", ", " and ")
    println("Get this guy $food_items\!")
end
```

Invoking this with two arguments, we get

```
julia> shout(["some pancakes", "sausages with gravy"])
Get this guy some pancakes and sausages with gravy!
```

These are returned to the function as a tuple listing each element covered by the splats.

What, however, if someone has more of an appetite? Our `shout()` function can accommodate it - the array just needs to get larger. This approach is perfectly viable, but regarded as a bit clumsy. What if I forget the array square brackets, for instance?

```
julia> shout("pancakes")
Get this guy p, a, n, c, a, k, e and s!
```

Well, that's not quite what I wanted! Fortunately, Julia allows us to have not merely multiple arguments but indeed an indefinite number. We effect this by suffixing the variable we wish to hold the positional arguments with three full stops `...`, also known as a 'splat':

```
function shout_mi(foods...)
    food_items = join(foods, ", ", " and ")
    println("Get this guy some $food_items\!")
end
```

Now our function performs perfectly, whether our customer is ravenous or he just wants some pancakes:

```
julia> shout_mi("pancakes")
Get this guy some pancakes!

julia> shout_mi("pancakes", "sausages", "gravy", "a milkshake")
Get this guy some pancakes, sausages, gravy and a milkshake!
```

What, however, if our customer does not seem to say anything? We would expect this to raise an error… but it doesn't:

```
julia> shout_mi()
Get this guy some !
```

Therefore, we need be mindful of using a splatted positional argument right in the beginning, since it will accept the input of, well, no input! A common way to fix this is to require one positional argument, then add a splatted second argument. This way, if the function is called with no arguments at all, it will raise an error. A better way, perhaps, is to simply test for it ourselves:

```
function bulletproof_shout(foods...)
    if length(foods) > 0
        println("Get this guy some $(join(foods, ", ", " and "))\!")
    else
        error("The customer needs to order something!")
    end
end
```

As you recall, the `foods` variable is passed on to us as a tuple. We can use the `length()` function on this tuple (although do note, we do test explicitly for `length(foods) > 0`: a result of zero would not be 'falsey', so testing simply for `if length(foods)` would not cut it!), which will indicate how many elements the tuple has and raise an error if it is zero:

```
julia> bulletproof_shout("sausages", "pancakes", "gravy")
Get this guy some sausages, pancakes and gravy!

julia> bulletproof_shout("sausages")
Get this guy some sausages!

julia> bulletproof_shout()
ERROR: The customer needs to order something!
 in bulletproof_shout at none:5
```

Finally, the function works. The last marginal case that you might want to deal with is when the customer's order consists of an empty string `""` or is the wrong type. These are further marginal cases and will not be explored here (although we will be looking at user input in quite a bit of detail in the second part of the book). The take-away is this - a good function (one you would let your grandmother use) needs to cater for a range of marginal cases and inputs. Splats are, however, somewhat performance-consuming and are best avoided in code that needs to run fast. In such situations, usability and performance need to be weighed and balanced.

Optional positional arguments

Positional arguments may be 'optional'. This does not mean they are not used - they are optional only from the user's p] erspective, who will not be required to enter them. A perhaps better way to put this is that these arguments have *default values* that take effect if they are not provided at invocation. Consider the following function, which accepts 2D as well as 3D coordinates, and sets 2D coordinates, by default, on the `z = 0` plane:

```
function coords(x, y, z = 0)
    return(x,y,z)
end
```

The result:

```
julia> coords(1, 6, 7)
(1,6,7)

julia> coords(3, 1)
(3,1,0)
```

Setting defaults allows you to prevent the inevitable error that would be triggered if `z = 0` were not provided for. Consider, for instance, what would happen if the value for `y`, for which no default value has been set, were to be missing:

```
julia> coords(3)
ERROR: `coords` has no method matching coords(::Int64)
```

Julia is telling us, in its somewhat odd grammar, that the function `coords()` is not defined for a single input.

Keyword arguments

The drawback of positional arguments is that getting the order right can be an inconvenience. Wouldn't it be much easier, not the least from a documentation perspective, if we were allowed to give arguments names and use these names at invocation? With Julia, you can do so at your heart's content, as long as you put them at the end of your variables when defining the function and delimit keyword arguments from non-keyword arguments with a semicolon `;`, as in this snippet:

```
function buzzphrase(verb, adjective; subject="defence", goal="world peace")
    println("$(verb)ing $adjective $subject for $goal.")
end
```

In this function, `verb` and `adjective` are necessary positional arguments. However, you can use the keyword argument syntax for `subject` and `goal`. As you can see, both have defined default values – this is necessary for keyword arguments in Julia. Thus,

```
buzzphrase("leverag", "effective", subject="best practices", goal="increased margi
```

is equivalent to

```
buzzphrase("leverag", "effective", goal="increased margins", subject="best practic
```

and yield the same results.

Stabby lambda functions: `->`

Sometimes, you're in a hurry and need a throwaway function. Whether it's for `map`ping an Array or comparing values in a sort, sometimes you don't want to define a function. A number of languages refer to these as *anonymous functions*, because they do not have a defined name, or reserve a `lambda` keyword for this, harkening back to Alonzo Church's 'lambda calculus' well before the advent of modern computers. Julia has a stylised arrow `->`, leading to the name *stabby lambda* for such functions.

Assume you want to `map` the array of all primes under 10 `[2,3,5,7]` to a function `f` so that `f(x) = 2x^3 + x^2 - 2x + 4`. In case you're unfamiliar with `map()` functions, here's the elevator pitch: map functions take a function and an iterable and return an iterable of equal length, each element of which will be the result of feeding an element of the original iterable into the function. In Julia, `map()` takes two arguments - a function and an iterable. For the former, you can use a function defined in advance or use the stabby lambda notation that is the subject of this section.

The mapping function would be written in the stabby lambda notation as

```
x -> 2x^3 + x^2 - 2x + 4
```

somewhat similar to the maplet notation in mathematics. Thus, we would use the `map` function as follows:

```
julia> map(x -> 2x^3 + x^2 - 2x + 4, [2, 3, 5, 7])
4-element Array{Int64,1}:
  20
  61
 269
 725
```

The stabby lambda is a little controversial, being even discouraged where it serves as a mere wrapper by the official Julia Style Guide, for the reason that such functions are impossible to unit test and can make code confusing. In general, the advice that is often given to, and by, Python programmers about `lambda`s in Python holds for their stabby Julia equivalents: a stabby lambda should be *obviously and unambiguously true*, that is, it should be evident at first glance

- what it does,
- how it does what it does, and
- that it does what it's supposed to do correctly.

In other words, consider a stabby lambda a sort of 'special pleading' - you're arguing that the function is so trivially true, defining it in a long and extensive way would benefit the code less than what is gained by the brevity of the stabby lambda syntax.

`do` blocks

`do` blocks are another form of anonymous functions. Similarly to stabby lambdas, they introduce a functional process that doesn't need to be defined by name. Let's consider the stabby lambda in the previous example and try to rewrite it as a `do` block:

```
map([2, 3, 5, 7]) do x
    2x^3 + x^2 - 2x + 4
end
```

The `do` block is a bit of syntactic sugar that helps us avoid unduly long stabby lambdas, as well as do slightly more complex things that the stabby lambda's restricted format might not allow for, such as more complex testing than a stabby lambda coupled with a ternary operator would allow:

```
map([2, 3, 5, 7]) do x
    if mod(x, 3) == 0
        x^2 + 2x - 4
    elseif mod(x, 3) == 1
        2x^3 + x^2 - 2x + 4
    else
        2x-4
    end
end
```

Returning multiple values

A function needs to return a single object, but that object may take the shape of a collection containing multiple values. If your function does return multiple values from within the function, they will be returned as a tuple:

```
julia> function squares(x, y)
           return x^2, y^2
       end
squares (generic function with 1 method)

julia> squares(2,5)
(4,25)

julia> typeof(squares(2,5))
(Int64,Int64)
```

As we can see, the function returned two values of type `Int64`, in a tuple. For various reasons, you may prefer defining your own type to return, such as a composite type - this is up to you and Julia gives you considerable freedom in doing so.

Scope in function evaluation

Scope in function evaluation refers to the availability of variables within or outside a function. Much of what has been said about scope in blocks in general

applies here, but function evaluation has some peculiar quirks that are worth mentioning.

Lexical scoping

Julia implements *lexical scoping*, that is, the scope of a function is inherited not from its caller but its definition. Consider the following:

```
function foo()
    println(x)
end

function bar()
    x = 2
    foo()
end

julia> bar()
ERROR: x not defined
 in bar at none:3
```

This is not unexpected, since the assignment of `x` to 2 is 'not visible' to the function `foo` when it's called. In other words, the assignment of `x` is *outside the scope* of the function. Therefore, it does not `see` the variable's definition and this yields an undefined variable error.

Global variables

A variable defined in the global scope is available to all functions:

```
julia> x = 2
2

julia> foo()
2
```

While this is very helpful, global variables incur an immense performance penalty. Their use is generally discouraged unless absolutely necessary.

Higher order functions

In general, the idea of a *higher order function* serves to distinguish functions that accept a function as an argument from other functions, sometimes referred

to as *first-order functions*. In functional programming, higher order functions are much more important than in OOP or other paradigms, and indeed even if you return to your OOP roots, an understanding of higher order functions will help you enormously in dealing with the implementations of higher order functions in your language of choice: since most higher-order functions are so useful for munging data, most programming languages do have implementations of `map()`, `sort()` and other archetypal higher order functions.

Functions that accept functions as arguments

We have already introduced `map()`, a typical higher-order function, above. While higher-order functions appear to be somewhat complex, they are actually easier than they seem. A function is an object like any other, and so can be fed into another function as an argument. You will not, generally, need to do anything special for your function to accept a function as an argument, except make sure you are calling the function provided to you.

```
function greet(x)
    str = x()
    println("Hello, $str\!")
end

function tell_me_where_I_live()
    return("world")
end

julia> greet(tell_me_where_I_live)
Hello, world!
```

Quite importantly, when you are passing a function to another function as an argument, *you are not passing a call, you're passing the function object* - so don't forget to skip the parentheses `()`!

Operators and higher-order functions

Operators, such as `+`, are just clever aliases for functions. Thus, there is no reason why they couldn't be passed into a function:

```
function oper(x, y, z)
    return x(y, z)
end

julia> oper(+,  , e)
5.859874482048838
```

In this case, the operator `+` was fed into our function (which did nothing but execute the operator fed in as `x` on `y` and `z`).

Functions that return functions

Just as accepting functions is perfectly permissible, a function can return a function as a result. Consider a function that returns an exponential function based on your input as the exponent.

```
function create_exponential_function(exponent)
    exp_func = function(x)
        return x^exponent
    end
    return exp_func
end

julia> power_of_five = create_exponential_function(5)
(anonymous function)

julia> power_of_five(5)
3125
```

The function above can be written more concisely with the stabby lambda syntax we encountered earlier:

```
function create_exponential_function(exponent)
    y ->  y^exponent
end
```

Currying

Some languages, including some functional languages, support a feature called *currying*, named not after the Indian spice but after logician Haskell Curry (namesake of the `Haskell` language). A curried function is one that has multiple arguments. If it is provided with values for all of them, it returns a value. If it is provided with only part of them, it returns a function that takes the missing values as arguments.

Currying was proposed for Julia in 2012, but voted down, not least because it would have been difficult to accommodate within multiple dispatch.

Methods and multiple dispatch

Understanding multiple dispatch

When you call a function on a number of arguments, Julia needs to decide how exactly that function makes sense for those arguments. In this sense, functions are not so much names for individual functions but for bunches of conceptually similar functions, with Julia deciding which particular one to call. Consider the * operator (which, like all operators, is a function):

```
julia>   * e
8.539734222673566

julia> "sausages " * "mash"
"sausages mash"
```

As the example above shows, the * function can take various types, and it has various actions defined for each - for numeric types, this involves multiplication, while for strings, * means concatenation. The feature of Julia that allows the call of the right implementation of a function based on arguments is called *multiple dispatch*, and the implementations are referred to as *methods*. Each function may have a number of methods defined for various data types, and it may have no methods at all defined for some. Finally, the error message we get when we use the 'wrong' type of input starts to make sense:

```
julia> 2 * "sausage"
ERROR: `*` has no method matching *(::Int64, ::ASCIIString)
Closest candidates are:
  *(::Number, ::Bool)
  *(::Int64, ::Int64)
  *(::Real, ::Complex{T<:Real})
  ...
```

What Julia is referring to in this instance is that * is not defined for one `Int64` and one `ASCIIString` operator. In other words, the function * has no method defined that would take these two particular kinds, after which it then recommends various options (some fairly unexpected, for instance, `::Number` * `::Bool` is perfectly valid – it multiplies the `::Number` by 1 if the `::Bool` is `true` and 0 if it is `false`).

Building methods

To construct a method, you can simply declare the function for a particular data type. Let's consider a function that adds numbers and concatenates strings (for

now, only two of each - the function can be expanded using the splat `...` syntax easily).

```
function merge_together(a::Number, b::Number)
    a + b
end
```

This is great. It does a great job at adding up numbers:

```
julia> merge_together(2, )
5.141592653589793
```

It's less adept at doing the string concatenation part we need it to do:

```
julia> merge_together("Sausages with", " mash")
ERROR: `merge_together` has no method matching merge_together(::ASCIIString, ::ASCIIString)
```

Therefore, we will need to define a method for `merge_together()` that will accept `ASCIIString` arguments. When Julia tells us a method is missing, it will give us the concrete data type of the argument we have entered. This is useful, but try to resist the temptation to define `merge_together` for `::ASCIIString`. In general, if your use case relates not to the concrete type but to the broader, *abstract* type (such as ours, where our use case is really *all* strings, not just `ASCIIString`), it's good practice to use the broadest abstract type that will include only the data types that you need. In this case, it is not `ASCIIString` but its abstract ancestor, `AbstractString` (in case you forgot your handy inheritance dendrogram, you can look at the supertype of any type by using `super(ASCIIString)` with the name of the type you're interested in). Let's define `merge_together` for two `::AbstractString` objects:

```
function merge_together(a::AbstractString, b::AbstractString)
    a * b
end
```

That's it, folks! Julia helpfully tells us that `merge_together` now has two methods. Using `methods(merge_together)`, we can list these:

```
julia> methods(merge_together)
# 2 methods for generic function "merge_together":
merge_together(a::Number,b::Number) at none:2
merge_together(a::AbstractString,b::AbstractString) at none:2
```

Let's give the second one, for strings, a try:

```
julia> merge_together("Sausages with", " mash")
"Sausages with mash"
```

It works! In general, when creating a function, you need to be circumspect as to what you want to use it for and what it needs to be able to deal with. There is no need for a function to have methods for all data types. So far, we have generally not defined the data types of arguments. This is a bad practice, and when you are building functions, you should always think of yourself as building methods at the same time, and define the types you want your function to accept.

Call order and method ambiguities

Consider the following function `f`:

```
function f(x)
    return x
end

function f(x::Int)
    return x^2
end
```

The first definition, lacking a type restriction, is deemed by Julia to accept inputs of type `Any` - that is, any type. The second method, however, only takes inputs of type `Int`. As such, it is more specific (or, if you please, 'further downstream on the type dendrogram'). The result is that when you call `f(2)`, the second, more specific method will be called, even if technically, the argument `2` would be acceptable for both. This is a sensible approach, since the broader the type, the more likely that the method is intended to be a 'catch-all' to mop up cases that have not been caught by any of the subtypes.

However, for functions with multiple arguments, it is possible that there is no unique method that is more unambiguous than the others. Consider the following:

```
function g(x::Int, y)
    return 2x^2 - 2y
end

function g(x, y::Int)
    return 2x - 2y^2
end
```

Which of these functions is 'more definite' when called as, say, `g(6, 8)`? The answer is 'neither', and Julia says so much when declaring the second method:

```
Warning: New definition
    g(Any,Int64) at none:2
is ambiguous with:
    g(Int64,Any) at none:2.
To fix, define
    g(Int64,Int64)
before the new definition.
g (generic function with 2 methods)
```

A method `g(x::Int64, y::Int64)` will be more specific than either of the previously defined methods, and as such capable of dealing with the indefinite middle.

```
function g(x::Int, y::Int)
    return x^2 - y^2
end
```

Parametric methods

A parametric method, similar to parametric types, is one in which a logical relationship is asserted between types, rather than an actual type name. You may think of parameters as 'variables' for type assertions. The parameter - by convention, but not by necessity, `T` for type - is enclosed in curly braces `{}` and interposed between the function name and its arguments:

```
function identical_types{T}(x::T, y::T)
    ...
end
```

This function would accept arguments of the same type, regardless of what that type is. You can restrict the possible values `T` might take based on type hierarchy:

```
function identical_numbers{T<:Number}(x::T, y::T)
    ...
end
```

This function allows for any inputs that are both identical *and* descendants of the `Number` supertype. Contrast that with

```
function divergent_numbers(x::Number, y::Number)
    ...
end
```

which accepts inputs that are descendants of the `Number` supertype, regardless of whether their type matches or not.

Inspecting methods

Entering a function object into the REPL, but not calling the function object, will indicate the number of methods under the function:

```
julia> +
+ (generic function with 150 methods)
```

You can inspect methods available under a function by using the `method()` command and passing the function or operator as argument:

```
julia> methods(+)
# 150 methods for generic function "+":
+(x::Bool) at bool.jl:34
+(x::Bool,y::Bool) at bool.jl:37
...
+(a,b,c) at operators.jl:83
+(a,b,c,xs...) at operators.jl:84
```

Chapter 14

Chapter 8: Handling errors

Error (or exception) handling is an essential feature of writing all but trivial programs. Let's face it – s**t happens, and sometimes the best-written programs encounter errors. Well-written code handles errors gracefully and as early as possible.

Over the years, two main 'approaches' to error handling have emerged. Those advocating the **LBYL** approach (**L**ook **b**efore **y**ou **l**eap) support validating every of data well before they are used and only use data that has passed the test. LBYL code is lengthy, and looks very solid. In recent years, an approach known as **EAFP** has emerged, asserting the old Marine Corps motto that it is **e**asier to **a**sk **f**orgiveness than **p**ermission. EAFP code relies heavily on exception handling and `try/catch` constructs to deal with the occasional consequences of having leapt before looking. While EAFP is generally regarded with more favour in recent years than LBYL, especially in the Python community, which all but adopted it as its official mantra, both approaches have merits (and serious drawbacks). Julia is particularly suited to an amalgam of the two methods, so whichever of them suits you, your coding style and your use case more, you will find Julia remarkably accommodating.

Creating and raising exceptions

Julia has a number of built-in exception types, each of which can be thrown when unexpected conditions occur.

Note that these are exception types, rather than particular exceptions, therefore despite their un-function-like appearance, they will need to be called, using parentheses.

Throwing exceptions

The `throw` function allows you to raise an exception:

```
if circumference > 0
    circumference/2
elseif circumference == 0
    throw(DivideError())
else
    throw(DomainError())
end
```

As noted above, exception types need to be called to get an `Exception` object. Hence, `throw(DomainError)` would be incorrect.

In addition, some exceptions take arguments that elucidate upon the error at hand. Thus, for instance, `UndefVarError` takes a symbol as an argument, referring to the symbol invoked without being defined:

```
julia> throw(UndefVarError(thisvariabledoesnotexist))
ERROR: thisvariabledoesnotexist not defined
```

Throwing a generic `ErrorException`

The `error` function throws a generic `ErrorException`. This will interrupt execution of the function or block immediately. Consider the following example, courtesy of Julia's official documentation. First, we define a function `fussy_sqrt` that raises an `ErrorException` using the function `error` if `x < 0`:

```
julia> fussy_sqrt(x) = x >= 0 ? sqrt(x) : error("negative x not allowed")
```

Then, the following verbose wrapper is created:

```
julia> function verbose_fussy_sqrt(x)
         println("before fussy_sqrt")
         r = fussy_sqrt(x)
         println("after fussy_sqrt")
         return r
       end
verbose_fussy_sqrt (generic function with 1 method)
```

Now, if `fussy_sqrt` encounters an argument `x < 0`, an error is raised and execution is aborted. In that case, the second message (`after fussy_sqrt`) would never come to be displayed:

```
julia> verbose_fussy_sqrt(2)
before fussy_sqrt
after fussy_sqrt
1.4142135623730951

julia> verbose_fussy_sqrt(-1)
before fussy_sqrt
ERROR: negative x not allowed
 in verbose_fussy_sqrt at none:3
```

Creating your own exceptions

You can create your own custom exception that inherits from the superclass `Exception` by

```
type MyException <: Exception
end
```

If you wish your exception to take arguments, which can be useful in returning a useful error message, you will need to amend the above data type to include fields for the the arguments, then create a method under `Base.showerror` that implements the error message:

```
type MyExceptionTree <: Exception
    var::String
end

Base.showerror(io::IO, e::MyExceptionTree) = print(io, "Something is wrong with ", e.var, "!")

julia> throw(MyException("this code"))
ERROR: Something is wrong with this code.
```

Handling exceptions

The `try/catch` structure

Using the keywords `try` and `catch`, you can handle exceptions, both generally and dependent on a variable. The general structure of `try/catch` is as follows:

1. **`try` block**: This is where you would normally introduce the main body of your function. Julia will attempt to execute the code within this section.

2. `catch`: The `catch` keyword, on its own, catches all errors. It is helpful to instead use it with a variable, to which the exception will be assigned, e.g. `catch err`.
3. If the exception was assigned to a variable, **testing for the exception**: using `if/elseif/else` structures, you can test for the exception and provide ways to handle it. Usually, type assertions for errors will use `isa(err, ErrorType)`, which will return true if `err` is an instance of the error type `ErrorType` (i.e. if it has been called by `ErrorType()`).
4. `end` all blocks.

This structure is demonstrated by the following function, creating a resilient, non-fussy `sqrt()` implementation that returns the complex square root of negative inputs using the `catch` syntax:

```
function resilient_square_root(x::Number)
    try
        sqrt(x)
    catch err
        if isa(err, DomainError)
            sqrt(complex(x))
        end
    end
end
```

There is no need to specify a variable to hold the error instance. Similarly to not testing for the identity of the error, such a clause would result in a catch-all sequence. This is not necessarily a bad thing, but good code is responsive to the nature of errors, rather than their mere existence, and good programmers would always be interested in *why* their code doesn't work, not merely in the fact that it failed to execute. Therefore, good code would check for the types of exceptions and only use catch-alls sparingly.

One-line `try/catch`

If you are an aficionado of brevity, you should be careful when trying to put a `try/catch` expression. Consider the following code:

```
try sqrt(x) catch y end
```

To Julia, this means `try sqrt(x)`, and if an exception is raised, pass it onto the variable `y`, when what you probably meant is `return y`. For that, you would need to separate `y` from the `catch` keyword using a semicolon:

```
try sqrt(x) catch; y end
```

`finally` clauses

Once the `try`/`catch` loops have finished, Julia allows you to execute code that has to be executed whether the operation has succeeded or not. `finally` executes whether there was an exception or not. This is important for 'teardown' tasks, gracefully closing files and dealing with other stateful elements and resources that need to be closed whether there was an exception or not.

Consider the following example from the Julia documentation, which involves opening a file, something we have not dealt with yet explicitly. `open("file")` opens a file in path `file`, and assigns it to an object, `f`. It then tries to operate on `f`. Whether those operations are successful or not, the file will need to be closed. `finally` allows for the execution of `close(f)`, closing down the file, regardless of whether an exception was raised in the code in the `try` section:

```
f = open("file")
try
    # operate on file f
finally
    close(f)
end
```

It's good practice to ensure that teardown operations are executed regardless of whether the actual main operation has been successful, and `finally` is a great way to achieve this end.

Advanced error handling

`info` and `warn`

We have seen that calling `error` will interrupt execution. What, however, if we just want to display a warning or an informational message without interrupting execution, as is common in debugging code? Julia provides the `info` and `warn` functions, which allow for the display of notifications without raising an interrupt:

```
julia> info("This code is looking pretty good.")
INFO: This code is looking pretty good.

julia> warn("You're not looking too good. Best check yourself.")
WARNING: You're not looking too good. Best check yourself.
```

`rethrow`, `backtrace` and `catch_backtrace`

Julia provides three functions that allow you to delve deeper into the errors raised by an operation.

- `rethrow`, as the name suggests, raises the last raised error again,
- `backtrace` executes a stack trace at the current point, and
- `catch_backtrace` gives you a stack trace of the last caught error.

Consider our resilient square root function from the listing above. Using `rethrow()`, we can see exceptions that have been handled by the function itself:

```
julia> resilient_square_root(-2.345)
0.0 + 1.5313392831113555im

julia> rethrow()
ERROR: DomainError
 in resilient_square_root at none:3
```

As it's evident from this example, `rethrow()` does not require the error to be actually one that is `thrown` - if the error itself is handled, it will still be retrieved by `rethrow()`.

`backtrace` and `catch_backtrace` are functions that return stack traces at the time of call and at the last caught exception, respectively:

```
julia> resilient_square_root(-4)
0.0 + 2.0im

julia> x^2 - 2x + 3
11

julia> backtrace()
13-element Array{Ptr{Void},1}:
 Ptr{Void} @0x00000001013cbfae
 Ptr{Void} @0x000000010349ec30
 Ptr{Void} @0x000000010349ebb0
 Ptr{Void} @0x00000001013776e8
 Ptr{Void} @0x00000001013c6982
 Ptr{Void} @0x00000001013c5203
 Ptr{Void} @0x00000001013d4abd
 Ptr{Void} @0x000000010137cdfd
 Ptr{Void} @0x0000000103455c41
 Ptr{Void} @0x0000000103455747
```

```
 Ptr{Void} @0x00000001013776e8
 Ptr{Void} @0x0000000103451cca
 Ptr{Void} @0x00000001013cccc8

julia> catch_backtrace()
14-element Array{Ptr{Void},1}:
 Ptr{Void} @0x00000001013cc506
 Ptr{Void} @0x00000001013cc5a9
 Ptr{Void} @0x00000001034a58e7
 Ptr{Void} @0x00000001034a56a7
 Ptr{Void} @0x00000001013776e8
 Ptr{Void} @0x00000001013c6982
 Ptr{Void} @0x00000001013c5203
 Ptr{Void} @0x00000001013d4abd
 Ptr{Void} @0x000000010137cdfd
 Ptr{Void} @0x0000000103455c41
 Ptr{Void} @0x0000000103455747
 Ptr{Void} @0x00000001013776e8
 Ptr{Void} @0x0000000103451cca
 Ptr{Void} @0x00000001013cccc8
```

The first backtrace block shows the stack trace for the time after the function `x^2 - 2x + 3` has been executed. The second stacktrace, invoked by the `catch_backtrace()` call, shows the call stack as it was at the time of the `catch` in the `resilient_square_root` function. # Chapter 9: I/O

Interacting with real world data is where programming gets interesting. Mastering I/O opens up a world of information for your applications and is quite fun, too! With that, let's delve into working with various file types.

Text files

Opening and closing files

In general, whenever you open a file, you will eventually need to close it. It's quite crucial to be circumspect with these housekeeping duties, lest you end up with corrupted data!

Julia's way of dealing with files resembles that of Python and a number of other languages. First, using the `open(path)` function, you open a file. The function returns an object that represents the file within Julia, known sometimes as a *file handle*. Commonly, the variable file handles are assigned to is `f`, but this does not have to be the case.

```
f = open("textfile.txt")
```

When you're done with your connection, use the `close()` function to close down the file handle:

```
close(f)
```

Reading files

Creating a file handle does not actually read the file - it merely checks where it is and figures out how to deal with it. As such, creating your file handle is usually pretty quick and memory-inexpensive, even if the file itself is very large.

However, we want to get something out of the text we imported. For the rest of this chapter, I will be using Virgil's Aeneid to demonstrate text functions, which you can obtain for yourself here courtesy of Project Gutenberg.

Once you have called the `open` function and assigned the file handle to `f`, you can start reading the file. Julia helpfully offers multiple ways to accomplish this, and it's useful to remember the ability to read a large file line-by-line once we enter the realm of handling large data sets.

You might notice that using the read functions 'uses up' the file. This is true - once you have 'read' all lines, the file will be empty. This is handy for keeping track of how much has been already read and ensure that where a read process has been interrupted, you will know where to pick up the thread.

readall

The `readall` function allows you to read the entire file, which it contains in a massive string, with line breaks represented as newlines (`\n`):

```
julia> readall(f)
```

```
julia> typeof(readall(f))
ASCIIString
```

readlines

Unlike `readall`, `readlines` creates an array of strings, each representing a line:

```
julia> readlines(f)
14656-element Array{Union(UTF8String,ASCIIString),1}:
"\ufeff  Arms, and the man I sing, who, forc'd by fate,\r\n"
"  And haughty Juno's unrelenting hate,\r\n"
"  Expell'd and exil'd, left the Trojan shore.\r\n"
```

```
"  Long labors, both by sea and land, he bore,\r\n"
"  And in the doubtful war, before he won\r\n"
"  The Latian realm, and built the destin'd town;\r\n"
"  His banish'd gods restor'd to rites divine,\r\n"
"  And settled sure succession in his line,\r\n"
"  From whence the race of Alban fathers come,\r\n"
"  And the long glories of majestic Rome.\r\n"
```

You can technically iterate through it line by line, which is a useful function as it allows you to perform functions on smaller chunks of data.

```
i = 1
for line in readlines(f)
    println("$i \t $line")
    i += 1
end
```

```
1       Arms, and the man I sing, who, forc'd by fate,

2       And haughty Juno's unrelenting hate,

3       Expell'd and exil'd, left the Trojan shore.
```

However, Julia has something far better to accomplish that, as we'll see in the next subsection.

`eachline`

`eachline` creates an iterator that you can use to go through and apply linewise functions. Let's see how many characters are in each line:

```
for line in eachline(f)
    print("$(length(line)) \t $line")
end
```

```
46      Now, in clos'd field, each other from afar
46      They view; and, rushing on, begin the war.
58      They launch their spears; then hand to hand they meet;
51      The trembling soil resounds beneath their feet:
56      Their bucklers clash; thick blows descend from high,
51      And flakes of fire from their hard helmets fly.
```

As we can see, the numbers are quite a bit off. This is ok - it's due to the indentation and the rather archaic spelling. What matters is that we can execute a function on each line of the function.

`enumerate`

`enumerate` is like a bonus function – it takes an iterable and creates an enumerator that also keeps track of

Chapter 15

Chapter 7: Writing good Julia functions

Writing good functions is not an art - it's something you can learn with reasonably little practice. A good function, by our definition, is one that is

- *performant*: it consumes as few resources as needed,
- *type-stable*: it always returns the same type of object, and
- *legible*: Julia is a fairly easy to read language, and so should your code be.

These desiderata aren't always compatible with each other, and it is your job as a programmer to figure out how to balance them, with their application in mind. The following therefore are not strict requirements, they are ways to accomplish each of the individual objectives.

Performant code

Generally, the more complex an operation, the more the impact of performance optimisations is. The `O` ('big O') notation expresses this aptly. Consider two functions, one running at linear time $O(n)$ and one running at exponential time $O(2^n)$. For a sufficiently small number of elements, the difference might not be visible. However, as the number of elements grows, the performance gaps are going to be huge, and well-written functions are the difference between a calculation taking minutes versus hours or even days.

Define arguments' types, whenever you can and however precisely you can

When it comes to arguments, have a good think about what your function is supposed to do and what types it can, and what types it cannot, ingest. Planning first (or, according to some, even documenting first) is a good idea - it will help you have an understanding of the needs of your code.

A functional programming language with multiple dispatch challenges you to *think in methods, not functions*. Create small, narrow methods that do something for particular and narrowly defined types, rather than broad functions. Especially to programmers coming from OOP thinking, the idea that functions are not atomic and actually break down into methods is difficult to digest. However, it's worth the try - functions that Julia doesn't have to 'guess' about evaluate much faster. Defining types precisely is also a sieve for incorrect input types. Programmers from other paradigms might be used to having to test whether inputs are the right format – in Julia, this is a 'built-in' feature: as long as you define your code precisely, it *will* accept only the right kind of data and raise an error for any other call. I call this a win-win!

```
function add_integers_badly(x, y)
    x = int(x)
    y = int(y)
    x + y
end
```

The function above is problematic because it says one thing and does another. On its face, it accepts all kinds of values. In reality, however, not only is the function meant to only work with numeric types (since `x + y` has a very specific meaning for other types), it is in fact meant to enforce a particular type (`Int` types) by using a type conversion command (`int()`). If so, it would be easier to simply limit the function to accepting the right kind of input type. Not only would this prevent the overhead penalty of converting data types, it would provide for type-stable and performant code that tells readers (and automated documentation generators) what the function takes in.

Time your functions, time your changes

Julia includes a very convenient macro, `@time`, that helps you keep track of the time and memory allocation of your functions. It pays to check the execution time of your functions, in particular when you have made changes that you think will affect performance. It's easiest to use the REPL or IJulia to time functions. Define your function first, then invoke it following the macro `@time`. In the following, we will define a Fibonacci function `fib(n) = n < 2 ? n : fib(n`

`- 1) + fib(n - 2)`, and look at its speed in detecting the 32nd Fibonacci number:

```
julia> @time fib(32)
elapsed time: 0.028219738 seconds (33112 bytes allocated)
2178309
```

You can also use `@elapsed`, `@time`'s younger sibling, which returns only the elapsed seconds:

```
julia> @elapsed fib(32)
0.028076393
```

However, `@time` is vastly superior. Runaway memory is the first sign that something is not going well with your application.

Write short, concise functions

The core ideology, and key success factor, of *NIX systems was to conceive of a complex system as a sum of small applications that did one thing, and did it well. Many, such as `grep`, have been in use for decades with much success. The same idea applies for Julia. Not only will it make your code better (it's easier to forget something in the middle of a big, complicated function), it will also make your code faster. This is because Julia's compiler can benefit from type-specialising code at function boundaries. Julia's own documentation has a great documentation of this feature:

```
function strange_twos(n)
    a = Array(randbool() ? Int64 : Float64, n)
    for i = 1:n
        a[i] = 2
    end
    return a
end
```

When Julia's compiler is handed this code, it does not know at the time of executing the loop what the value of `a` is - only that it can be one of `Array{Int64}` or `Array{Float64}`. As such, it will have to provide for both outcomes. A better pair of functions would separate the inner and outer loops:

```
function fill_twos!(a)
    for i=1:length(a)
        a[i] = 2
```

```
    end
end

function strange_twos(n)
    a = Array(randbool() ? Int64 : Float64, n)
    fill_twos!(a)
    return a
end
```

The result is that when calling `strange_twos(n)`, Julia's interpreter will know whether to compile `fill_twos()` for `Int64` or `Float64`. This will yield a performance benefit:

```
julia> @time strange_twos(16)
elapsed time: 0.000767759 seconds (7120 bytes allocated)

julia> @time fast_strange_twos(16)
elapsed time: 0.009221679 seconds (179592 bytes allocated)
```

Wait a second! That's actually 12 times slower! What happened? Let's see if it's just an accident.

```
julia> @time fast_strange_twos(16)
elapsed time: 9.286e-6 seconds (256 bytes allocated)
```

That looks much closer to what we expected. The reason is that when we called `fast_strange_twos()` the first time, the time included the JIT compiler's copmilation time. The lesson is to run every function once before testing the time it takes, and where multiple functions can result in multiple types, we might need to understand that unless the JIT compiler has encountered each type, there might be anomalous results. At second execution, we were lucky - the random number generator had the program retrieve the same type of number to fill the array. Then we ran it again:

```
julia> @time fast_strange_twos(16)
elapsed time: 0.002645026 seconds (6600 bytes allocated)
```

This time, the numbers to fill the array were different. As a result, it took a little longer. However, from this point on, until I close down the REPL, the `fast_strange_twos()` function will evaluate at the same time and at the same memory cost.

Danger zone

Julia allows you something called *performance annotations*, one of which is `@inbounds`. This speeds up array processing by circumventing bounds checking. Thus, for instance, a function to calculate the dot product of two arrays, might make use of this function:

```
function dotproduct(x::Array, y::Array)
    result = zero(eltype(x))
    for i = 1:length(x)
        @inbounds result += x[i] * y[i]
    end
    return result
end
```

In this case, `@inbounds` is safe because we know that due to the way `i` is defined (`1:length(x)`), there will never be an `i` that exceeds the length of `x` that would result in an out-of-bounds subscript. As such, bounds checking is more or less superfluous. To see the effect this has on execution time, let's compare it with the same function without the bounds checking disabled, called `slow_dotproduct`:

```
julia> @time(dotproduct([1234, 5747, 2243243, 535345, 76345, 2346], [23468, 4563, 2457, 12455
elapsed time: 1.4494e-5 seconds (288 bytes allocated)
74500427955

julia> @time(slow_dotproduct([1234, 5747, 2243243, 535345, 76345, 2346], [23468, 4563, 2457,
elapsed time: 0.004523183 seconds (105816 bytes allocated)
74500427955
```

Not only did the bounds-checking dot product require over 350 times the allocated memory of our non-bounds-checking product, but it also took over 300 times the execution time of the faster process. This might be inperceptible, since even the slower function took only 4.5ms to execute, but at a scale, these differences extrapolate to massive performance gains through good but circumspect programming. Just be sure you eliminate bounds checking only where you have a good reason to assume you will not need it because your own script provides for it.

Type-stable code

Type-stable is functional programmer speak for code in which variables don't change their type. The reason why this is important is that type conversion

represents an overhead that, at scale, slows down the process dramatically. Consider the following example:

```
function badly_written(n::Integer)
    result = 0
    for i in 1:n
        result += cos(rand())
    end
    return result
end
```

This function adds the cosine of a pseudorandom number (`rand()` generates a pseudorandom number using Julia's Mersenne twister implementation) to a result variable that is initialised as zero, which will be interpreted by Julia as an `Int64`. What we do know about cosines, however, is that they generally do not tend to yield integers, and indeed what we do know about Julia is that the cosine function will yield even results expressible as integers as floating-point results (try `cos(2)`). Therefore, we know that before it can do anything, Julia will need to convert `result`, an `Int64`, to a `Float` type it can add another `Float` to. This, effectively, is wasted time.

A better function, thus, would be

```
function well_written(n::Integer)
    result = 0.0
    for i in 1:n
        result += cos(rand())
    end
    return result
end
```

Just how significant is the performance difference? After doing a dry run of running each function once, which allows the JIT compiler to compile the function so that we avoid the issues we discussed above, we get the following:

```
julia> @time badly_written(100000)
elapsed time: 0.006668156 seconds (3200048 bytes allocated)

julia> @time well_written(100000)
elapsed time: 0.001911581 seconds (64 bytes allocated)
```

Not only did type conversion (just once!) make our code about 3.5 times slower, it also made it consume 50,000 (!) times the memory. Type stability makes for faster and more performant code.

Legible code

Julia is a *high-level language*, meaning that it is closer to natural languages than low-level languages are. In fact, it could, like Python, be described as 'executable pseudocode', with a minimum of syntax, eschewing unnecessary braces and parentheses and instead using a clear, legible indented structure. However, it's not difficult to write illegible code in Julia – indeed, the Stack Exchange Code Golf board, where users specialise in cramming their solution into the shortest possible sequence, has quite a lot of it. Unless you're code golfing, your code should be beautiful and legible.

Unfortunately, legible means different things to different people, and that's how style guides came to be. Julia, being a young language, does not have as many and as thoroughly debated style guides as, say, Python's `PEP8` or various Javascript style guides. The following seeks to point out a few of the most salient points of writing idiomatic Julia, as observed from core packages and prominent Julia packages, as well as the official Style Guide and John Myles White's Style.jl. Both are definitely worth reading - the former is more a high-level overview while the latter is very specific. However, coding is a matter of judgment and as a programmer, one of the things you are paid for is your sense of judgment, both in resolving the occasional inconsistencies within and among style guides by balancing countervailing objectives and in deciding whether to follow particular rules. Sure, the style guide says to stick to 80 characters a line, but should I break an 81 character line? The conventions suggest to eschew `lowercase_separated_names` and `CamelCase`, but what if the name is too long and cannot be sensibly abbreviated (a problem people encounter in the educational context quite often, as the author did while writing this book and trying to balance adherence to writing more understandable code)?

Chapter 16

Next steps