Julia Guide

James Byrne

2023-07-04

Table of contents

About

This is a guide to using Julia. Use the sidebars on the left and right to navigate between the pages.

To get started, you can try the demonstration version of Julia on the official Julia website at https://julialang.org/learning/tryjulia/. Click the play button, and you should be greeted with the `julia>` prompt. If you can see that, then you're ready to go!

If you'd like to jump straight to trying some programming, you can skip forward to Chapter 2. Note however, this is limited by being a web-hosted system, and for most uses of Julia it will be insufficient. What's more, it won't allow you to save files, which is something we'll want to do eventually to avoid typing out a long sequence of commands every time we want to do a computation. Instead, you'll eventually want to install Julia for yourself, for which guidance is given in Chapter 1.

Credits

This project was funded by the Cambridge University Press.

This book was rendered by Quarto.

1 Installation and general usage

1.1 Installing Julia

The Julia program can be downloaded from https://julialang.org/downloads/. You'll need to choose the correct version for your system, with different instructions for installation for different systems found at https://julialang.org/downloads/platform/. It will be important to *Add Julia to PATH*, which you will see mentioned in these instructions, as we'll want to make Julia findable for our editor in a moment.

Once the program is installed, you can launch it, and begin to type (for example, some basic sums like `1 + 2`). This input is called the *REPL* or *read-eval-print loop*, because what it does is the following:

- It reads what the user has typed in
- It attempts to evaluate it as Julia code
- It prints the result of the evaluation, whether that is a calculation result, or a message telling the user that something has gone wrong
- It loops back around to the start, putting the `julia>` prompt again and waiting for another user input

The REPL has all the functionality of Julia, and more. Previous inputs can be navigated to by ↑ and ↓, or searched with the search modes (accessible through `Ctrl-S` / `Cmd-S` and `Ctrl-R` / `Cmd-R`). Help can be found by pressing `?` to enter help mode, while package mode comes from typing `;`.

However, it still has one major limitation, which is that nothing is saved. If you close the program and reopen it, you'll notice that it has no memory of any of the computations it has done before. Once we start chaining computations together into fully fledged algorithms, we don't want to lose all of our progress every time we close the program, nor do we want to leave it running the whole time taking up unnecessary memory. For this, we'll need to make use of our own files, saved on our computer, with our code written out in full contained within it. This is entirely possible to do within the REPL, but for most people is needlessly difficult, so we'll make use of a code editor, specifically an *integrated development environment*, or *IDE* for short.

An IDE does many things, with different ones having different attractions. The most common features include:

- Code editor, for writing and editing code files, which can be saved to the computer
- Syntax highlighting, by changing the colour and sometimes other aspects of the font to distinguish keywords, or sections of the code file that do different things
- Automatic indentation, which for Julia is not needed in general, but helps for readability of code to show where blocks of code start and end
- Debugging tools, such as recognising errors in code before it is run
- Integrated terminal/REPL, so code can be run and tested quickly and easily

There are many IDEs in existence, but the most popular for Julia is *VS Code*, which we'll focus on here.

1.2 Installing VSCode

Microsoft's VSCode is an IDE with compatibility with hundreds if not thousands of programming languages, and more beyond that. However, we'll only look at its use for Julia.

To begin with, VSCode will need to be downloaded from https://code.visualstudio.com/download/, where you'll need to choose the right version for your system as you did with Julia. Once it's installed, you'll need to open it, and install the Julia extension, which allows VSCode to work with Julia. This can be done by finding the extensions tab on the left sidebar (which looks like three squares in an L shape with a fourth floating above), and searching for Julia in the search box. Now, if you've installed this, and if you clicked *Add Julia to PATH* earlier, then everything should be set up to start programming. If not, the documentation at https://www.julia-vscode.org/docs/dev/ should help to resolve your issue.

1.3 Using Julia in VSCode

To open an instance of the Julia REPL in VSCode, hold down the `Alt` key, and press `J` followed by `O`. This functions identically to the REPL that appears when we open the Julia program.

In VSCode, we can also open a `.jl` file to be able to write code and save it to be run whenever we wish, as well as to be able to be run from within other programs. This is essential for larger projects, and can be done by going to `File - New File`, and choosing "Julia File". Alternatively, holding down `Alt` and pressing `J` followed by `N` has the same effect. You can save this file as a Julia `.jl` file, and any code written in it can be run by the play button in the top right corner of the editor window.

VSCode is highly customisable, and you are recommended to experimented with its many options to find the layouts and functionalities that work best for you.

1.4 Pluto notebooks

A third alternative working environment is provided by the Pluto notebook, from the `Pluto.jl` package. For this, you'll need to install Julia as before, but you don't need an IDE.

First, launch the Julia program, giving an instance of the REPL. Next, press `]` to enter `Pkg` mode, type `add Pluto`, and press `Enter` which will download and install the package. Once that is done, press `← Backspace` to return to the normal REPL mode, type `using Pluto`, and press `Enter`, which tells Julia to load the package ready for use. Finally, type `Pluto.run()` and press `Enter`, which will launch Pluto in your default web browser.

Pluto works markedly differently from the REPL and editor environments, and is perhaps best explained by the *Getting Started* notebooks on its homepage.

When using Pluto for a second time, you don't need to install it again, just launch the REPL fresh and start at `using Pluto`.

More in depth discussion of using packages in general can be found in Chapter 8.

Part I

The basics of Julia

In this section, we'll examine the basic building blocks of Julia, and of programming in general, and see how we can harness them to maximal effect.

- Chapter 2 begins by looking at the building blocks of Julia programming, in particular *variables*, *functions*, and *types*. We also see how Julia itself can help us if we make a mistake or get stuck
- Chapter 3 dives down into the different ways that numbers can be represented in Julia, their pros and cons, as well as the potential arithmetic errors that we need to be aware of when using them
- Chapter 4 instead shows how text is represented in Julia using the `String` type, and how they can be manipulated
- Chapter 5 discusses the different methods of manipulating code to run in more ways than just line by line down the page, and the possibilities that this brings
- Chapter 6 looks again at *variables*, *functions*, and *types*, and how we can create our own custom structures of these forms
- Chapter 7 analyses the key paradigm of Julia that is *multiple dispatch*, how it works, why we would use it, and how to make use of it ourselves
- Chapter 8 breaks down the packaging system of Julia, which permits the installation and use of specialised code written by others to add functionality
- Chapter 9 investigates ways to store and manipulate several pieces of data in one go. We see how Julia works intelligently with such collections, meet some of the specialised types that take such a form, as well as putting this to use in plotting Julia sets
- Chapter 10 demonstrates Julia's ability to read and write data from or to external files
- Chapter 11 describes how to use the `Plots.jl` package to plot data as a graph
- Chapter 12 introduces ways of testing code

These chapters will refer to each other in places, but may be read somewhat independently of each other. At the start of each chapter, a list of recommended prerequisites is given.

Convention

As with all programming languages, Julia has many conventions of naming, syntax, and style (and these can differ from other languages!), which will be highlighted in boxes like this. These range from the inconsequential mundanities to the genuine good practice stopping you from breaking something important, but beginners should strive to follow them as much as possible nonetheless.

2 Fundamental concepts

! Prerequisites

This chapter has no prerequisites

2.1 Printing and calculating

Now that we've got Julia up and running, let's see what it can do. In time-honoured fashion, we'll begin by getting Julia to say `Hello world!` to us. In the REPL, this is as simple as enclosing the text in double quotation marks `" "` (making it a *string*, which Julia knows not to try and evaluate as code), and pressing `Enter` :

```julia
"Hello world!"
```

```
"Hello world!"
```

To get rid of the quotation marks around the output, we can use the keyword `print`, and enclose our phrase in parentheses `()` immediately after (without a space):

```julia
print("Hello world!")
```

```
Hello world!
```

A relatively unique feature of Julia is the ability to use a wider range of characters in code than just the normal `A-Z`, `a-z`, `0-9` and common punctuation. So let's update `Hello world!` to the 21st century:

```julia
print("  ")
```

These emoji can be copied in from elsewhere, but the Julia REPL as well as environments such as VSCode in a `.jl` file or a Pluto notebook allow *tab-completion* to type such symbols. If you know the right code for the symbol you want (which will always begin with a backslash `\`), typing out the name and pressing `Tab` will give the symbol. For instance, the three emoji used here have codes `\:wave:` for , `\:earth_africa:` for , and `\:exclamation:` for .

What else can we type? We can get Julia to do some basic sums, using the symbols `+`, `-`, `*`, `/`, `^` as add, subtract, multiply, divide, and exponentiate respectively, with parentheses `()` used to prioritise operations as usual, and otherwise the normal order of operations is assumed:

```
1 + 2
```

```
3
```

```
7*8
```

```
56
```

```
5 + (3 - 1/2) * 20
```

```
55.0
```

We also see here the use of a *comment*. Anything in the same line following a hash `#` will be ignored by Julia, so we can write some explanation next to our code to help us remember what it is supposed to do.

Convention

Adding comments isn't particularly useful in the REPL (especially as it is forgotten after the REPL is closed), but in `.jl` script files it is highly recommended, particularly the longer and more complex a program gets.

Julia is far more that a glorified calculator, however. We'll start be seeing how we can save data for later, using *variables*.

2.2 Variables

Variables are a way to give a data a name, which we can use to refer to that data later. This allows us to write code that performs calculations where we don't explicitly tell Julia which numbers to use. A value is assigned to a variable using `=` as follows:

```julia
number1 = 1
```

```
1
```

Names of variables are important to choose carefully. We have a multitude of such characters at our disposal, including non-Latin scripts, and emoji like we've seen before in text form (although we are somewhat limited for the first character, which must be a letter or letter-like symbol, e.g not a number). They must also not conflict with any of Julia's inbuilt keywords, such as `print` we've seen above, or words like `if`, `for`, `end` that we'll see in later chapters. Most punctuation characters (except `_`) are also not allowed, as they have other purposes, such as `" "` for enclosing text and `#` for comments as we've already seen. Moreover, Julia is case-sensitive, so `X` and `x` would refer to different things.

Convention

Names of variables should be in lowercase, with no spaces (as the space marks the end of the variable name) or underscores (unless they are needed to make the name readable).

```julia
    2  = 2
```

```
2
```

We can ask Julia to tell us the value of the variable simply by typing the name of that variable.

```julia
number1
```

```
1
```

We can then use the names in place of the values they represent in sums.

```julia
number1 +     2
```

```
3
```

The value of a variable can be changed by the same means that we gave it a value in the first place.

```
number1 = "one"
```

```
"one"
```

```
number1
```

```
"one"
```

There are many more uses for variables than just saving data for later, but for now we'll move on to look at the reason that Julia already uses many names, which is to refer to its many inbuilt *functions*.

2.3 Functions

A function, at its simplest, is a way of performing a predetermined algorithm or calculation without having to laboriously do each step one at a time yourself. Instead, it is known only by its name, and Julia understands references to its name as requests to perform said algorithm. For example, we can use Julia to calculate square roots using the `sqrt` function, even if we don't know the ins and out of the calculation needed to do it:

```
sqrt(2)
```

```
1.4142135623730951
```

Or, we can find the position of the first letter `t` in a sentence:

```
findfirst('t', "Shall I compare thee to a summer's day?")
```

```
17
```

⚠ Warning

The use of single quotation marks `'t'` instead of double `"t"` is deliberate here, as they mean different things.

The way to input data into a function is to write the function name, and follow it with a set of parentheses containing the function arguments, which can be literals (that is, writing the data in literally), or variables (where the value of the variable will be given to the function). If there are multiple inputs, they will need to be separated by commas, such as in the `min` function which finds the minimum of its inputs:

```julia
min(1, 2, 3, 4, 5)
```

```
1
```

Indeed, we've already met several functions above without even knowing it. `print` is a function that takes an input and prints the result in the REPL. Although they might not look like it, the mathematical operations are also functions, with the special *infix* (putting the function name between two arguments) notation used since that's what we're used to. It is entirely possible to write such calculations in the standard function format, and they work just the same:

```julia
+(1, 2)
```

```
3
```

Not only that, but it's possible to write our own functions to encapsulate bits of code that we want to run again and again, perhaps with different starting values. We'll see how to do this in Chapter 6.

Functions won't always give an answer, though. Often, this is because the input is nonsensical for what the function is supposed to be doing, such as:

```julia
sqrt("cabbage")
```

Running this will give an *exception* (alternatively called an *error*), which far from being a bad thing, can help us to track down and fix problems in our code. In this case, the problem is as simple as the fact that Julia doesn't know how to take the square root of a string of text. This brings us on to looking at the *types* of data that Julia can handle.

2.4 Types

Every piece of data that you give Julia, be it a number, a string of text, a collection of elements, even a block of Julia code, has along with it a label that tells Julia how to interpret it. This is called the type of the data, and we've seen many already:

- Text enclosed in double quotation marks `" "` has type `String`, meant for representing data in the form of text
- Individual symbols enclosed in single quotation marks `' '` have type `Char`, short for character. This is a simpler format than `String`, being more intrinsically understood by the computer, in fact `String`s are little more than sequences of `Char`s
- Numbers have types, although there are many different types for different numbers. Common ones include `Int64` for integers and `Float64` for decimals. For further discussion of this, see Chapter 3
- Functions also have types, although they are slightly more complicated. We can use the `typeof` function to query types, and as we can see, the `sqrt` function has type `typeof(sqrt)`:

```julia
typeof(sqrt)
```

```
typeof(sqrt) (singleton type of function sqrt, subtype of Function)
```

As the bizarre result above suggests, types have many intricacies to them. They are related by the *type graph*, which groups together types in various categories as a tree structure. We'll see examples of this in Chapter 3 and Chapter 9, and dive into the details in Chapter 7. We can also define our own types, as we'll see in Chapter 6. For now, we'll just think of them as a way for Julia to distinguish what sort of data we give it, and how it should be processed and stored.

2.5 Help functionality

Whether you're just getting started, or are very experienced, it can sometimes be difficult to fix problems or remember how certain functions work. For this purpose, Julia has a number of ways to provide guidance to put you back on track.

2.5.1 Tab completion

We've already seen tab-completion to type non-standard characters, like emoji. In fact, tab-completion does more than this, it can complete (hence the name) partially typed keywords or variable names. For example, typing `sq` in the REPL and then pressing `Tab` gives the word `sqrt`, as it is the only keyword that Julia knows beginning with `sqrt`. However, if we define a variable:

```julia
squid = " "
```

```
" "
```

then pressing `Tab` once doesn't do anything, because it doesn't know which one of `sqrt` and `squid` we want. Pressing `Tab` again won't complete it, but it will give us a list of the possibilites that we might mean, so that we can work out which we meant.

If there are multiple options, but they all share some characters, Julia will complete as much as possible, e.g. typing `prin` and pressing `Tab` will give `print`, as the possible options are `print`, `println`, and `printstyled`, all of which have `t` as the next letter.

This can be very useful if you can't remember the exact name of a variable or function, or just if you're exploring what functions exist inbuilt into Julia. Other environments than the REPL like the VSCode editor or Pluto will provide similar tab-completion functionality.

2.5.2 `help?` in the REPL

Pressing `?` in the REPL changes the prompt from the green `julia>` to the yellow `help?>`, entering *help mode*. The functionality of the REPL now changes dramatically, from executing inputs as code, to giving information about the keywords and functions used. For example, if we want to learn more about the function `+`, we can enter help mode and type `+`, or some code including `+` such as `2 + 3`. This gives the following result:

```
help?> 2 + 3
  +(x, y...)

  Addition operator. x+y+z+... calls this function with all
  arguments, i.e. +(x, y, z, ...).

  Examples


  julia> 1 + 20 + 4
  25

  julia> +(1, 20, 4)
  25
```

Help mode can also give similar names that it knows if you've mistyped the input:

```
help?> min
search: min minmax minimum minimum! argmin Main typemin findmin
```

Pasting in a Unicode symbol will give you the code to tab-complete and type the symbol (if a code exists for that symbol):

```
help?>
" " can be typed by \:mushroom:<tab>
```

To exit help mode, you can press `← Backspace` on an empty line.

2.5.3 Error messages

When an exception occurs, Julia provides an error message, which although a little intimidating at first with the red `ERROR:` text, but actually is very useful in diagnosing issues.

Firstly, the type of error is given, which can tell you immediately what the problem is in simple cases. For instance, a `MethodError` may occur when the inputs to a function have the wrong types, or an `UndefVarError` if a word is used that isn't recognised as a variable name. In the REPL, this can often be enough to find the mistake, but in more complex programs, with custom functions and modules, it may not suffice.

Instead, we look to the `Stacktrace` section. It provides a list of all of the functions that have been called, with the error occurring in the first function listed, which itself was called by the second function listed, and so on. Line numbers are also provided, so if unexpected behaviour is happening, it can be traced down exactly to the line or lines of code that caused it.

We'll look more into the ways that exceptions can be used to aid programming in Chapter 12.

3 Different numeric types

! Prerequisites

Before reading this chapter, you are recommended to have read Chapter 2

Numbers are ubiquitous in Julia programming, as well as in programming in general. Almost every programmer has at some early point in their learning set `a` to be `1`, `b` to be `2`, and added them together. Every operation that a computer does can be reduced down to a basic arithmetic operation called a logic gate. Indeed, numbers are so intrinsic to Julia that they are one of the few things to be part of the `Core`.

However, all is not as it may seem on the surface. When calculating with numbers in Julia, it's only a matter of time before you run into some unexpected results, such as:

```
0.1 + 0.2
```

```
0.30000000000000004
```

```
2^100
```

```
0
```

```
cos( ) # Julia can calculate this accurately...
```

```
-1.0
```

```
cos( /2) # ...but not this?
```

```
6.123233995736766e-17
```

These examples (and many others) demonstrate that the way that computers calculate is not quite the same as we are used to. Much of this is to do with the *type* of the numbers in question.

3.1 Overview of types

We've already met types earlier, but here's a quick refresher. All data within Julia, including every variable, every literal (that is, data given as a specific value, like `22`, rather than a variable name referring to a value, like `x`), even every bit of code, has a type, which does several things:

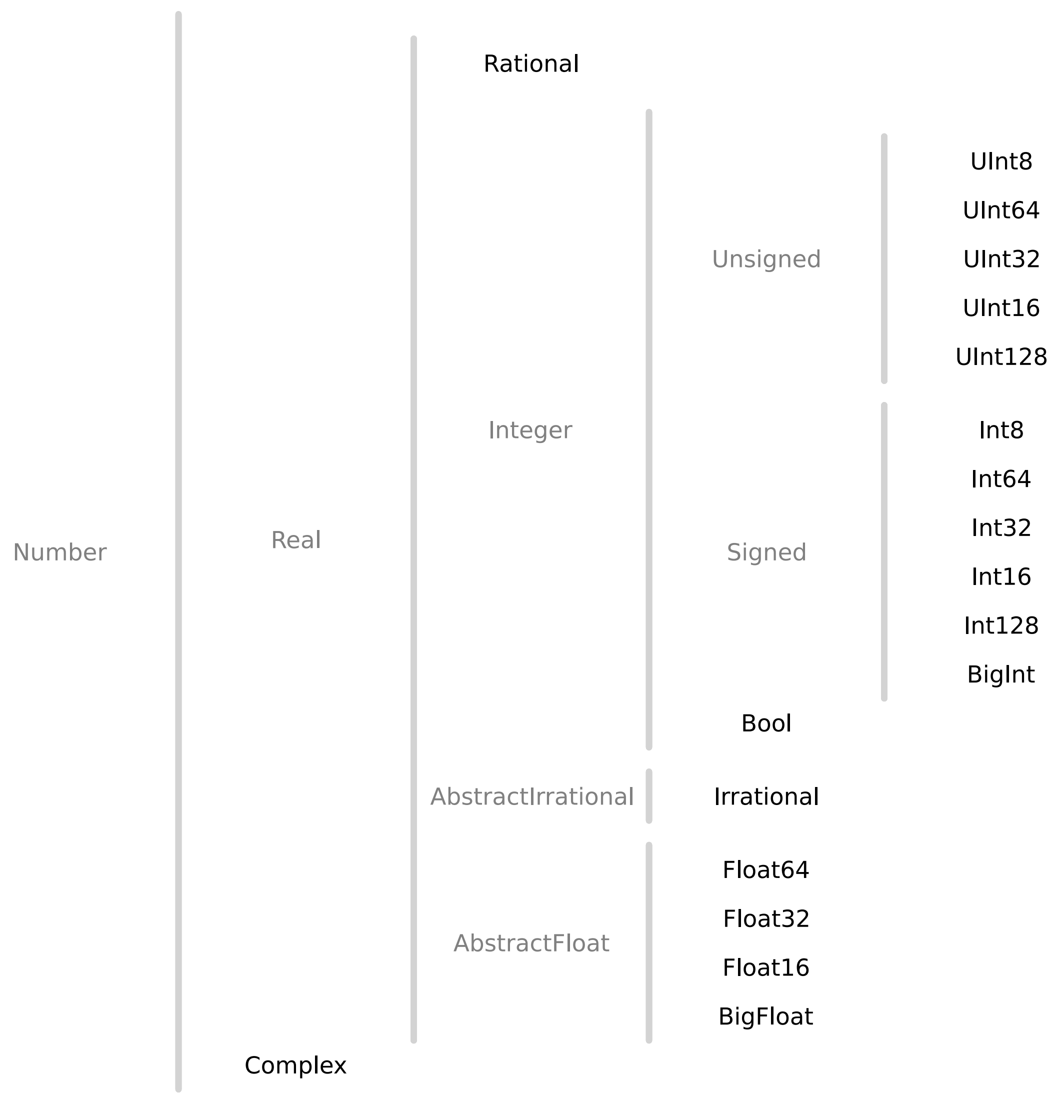
- It tells Julia how to store the data in the computer's memory
- It tells Julia how to read the value from the computer's memory
- It tells Julia how to react when the data is operated on or passed to a function (which we call *multiple dispatch*)

Combined together, Julia is able to work out what to do with data before knowing the exact value of the data (much like you know how to use a pen regardless of what colour it is). This is very flexible for programmers, and very efficient for computation.

Types are arranged into the *type graph*, which is better described as a tree structure, with every type given a *parent type* or *supertype*. Mostly, these parent types are *abstract*, meaning that they can't exist as data themselves, but serve as a label to refer to any and all of the types below them. The opposite of an abstract type is a *concrete* type, which is a type that data can take. (There is a third option, the *parametric* type, which we'll meet later). The presence of abstract types can be very useful for multiple dispatch, allowing a function to be written to apply to several different type inputs at the same time.

To get a bit of a better idea of this structure, we'll look at all of the different types that come under the abstract type `Number` by default in Julia:

```julia
# Useful for convenient broadcasting in the arrange function
import Base.+
+(x::Any,::Nothing) = x

# Arranges the subtype tree of T with positions of each of the subtypes
# Format of output is a vector of tuples of the form:
#     (type, number of nodes below, depth, position from top)
function arrange(T::Type)

    # Deals with small cases, where a type is not defined in Base we ignore it
    isdefined(Base, Symbol(T)) || return Tuple{Type, Int64, Int64, Rational{Int64}}[]
    isabstracttype(T) || return [(T, 1, 1, 1//1)]

    subT = subtypes(T)
    typelist = Tuple{Type, Int64, Int64, Rational{Int64}}[]
    offset = 0
    for S   subT

        # Prevents an infinite loop
        S == Any && continue
        # Recursively gets the arrangement of each subtype
        Stypelist = arrange(S)

        # Alters the arrangement to fit with the new graph
        append!(typelist,
            [st .+ (nothing, nothing, 1, offset) for st   Stypelist]
        )
        # Recalculates the offset to accommodate for the new points added
        isempty(Stypelist) || (offset += (Stypelist[end])[2] + 1)

    end
    # Adds the tuple for T itself to the end of the vector, and returns
    push!(typelist, (T, offset, 1, offset//2))

end

using Plots
# Plots the type tree for the subtypes of T
function typetree(T::Type)

    typelist = arrange(T)

    types::Vector{Type}                 = [st[1] for st   typelist]
    spans::Vector{Int64}                = [st[2] for st   typelist]
    depths::Vector{Int64}               = [st[3] for st   typelist]
    vpositions::Vector{Rational{Int64}} = [st[4] for st   typelist]
    n = length(typelist)

    xgap = 750 ÷ max(depths...)
    ygap = 750 ÷ spans[end]

    plot(
        size = (750, 750),
        grid = false,
        showaxis = false,
        legend = false,
        xlims = (xgap - 50, 750 + 50),
        ylims = (ygap - 20, 750 + 20)
    )

    # Abstract types in grey, concrete types in black
    plot!(
        annotations = [
            (depths[i] * xgap, vpositions[i] * ygap,
            (types[i], 10, isabstracttype(types[i]) ? :grey : :black), "sans-serif")
            for i   1:n
        ]
    )

    lines = [
        ([(depths[i] + 1//2) * xgap, (depths[i] + 1//2) * xgap],
        [(vpositions[i] - spans[i]//2) * ygap, (vpositions[i] + spans[i]//2) * ygap])
        for i   1:n if isabstracttype(types[i])
    ]

    # Lines delineate the set of subtypes for each abstract type
    plot!(
        [Shape(line...) for line   lines],
        fillcolor = :white,
        linecolor = :lightgrey,
        linewidth = 4
    )

end

typetree(Number)
```



Suppose we want to write a function that will work for all types of integers. Integers could come in many forms, such as `UInt8` or `Int64`, but instead of writing a separate method for each of these, we can just write one for the abstract type `Integer` instead, and it will be called regardless of the exact type of the data we input (as long as it is below `Integer` on this diagram).

We'll explore each of these types in more detail in turn, seeing how in some cases they differ from the way we'd expect numbers to behave, and how we can make use of this or avoid the problems it causes.

3.2 The `Int` and `UInt` types

The most common numeric types are in the `Int` family, specifically `Int64`, although the others work similarly, so we'll cover them all together. We'll also look at the `UInt` family, which is much the same as well, but with a minor difference that makes it applicable in different scenarios.

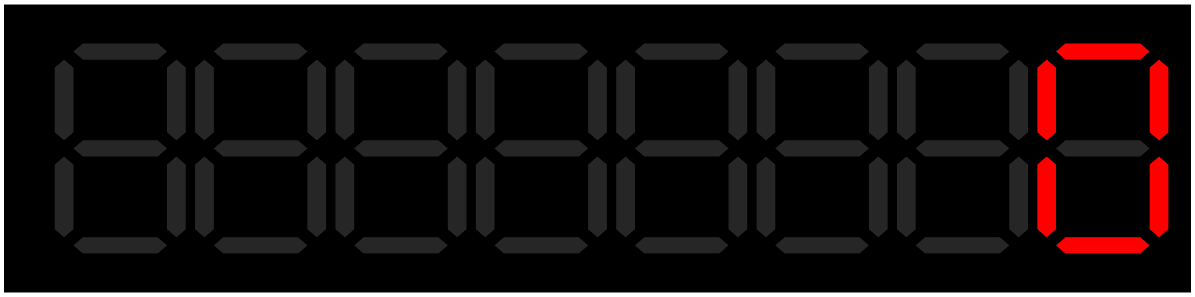
First, why are they called `Int` and `UInt`? `Int` is short for integer, and indeed this describes the numbers that can be stored in these formats: whole numbers. Meanwhile, the `U` in `UInt` stands for `Unsigned` (as we see in the diagram above), which means we can't have a minus sign, so we can only have positive numbers or zero, which may or may not be desirable. If we do need negative numbers, then we need to use a `Signed` type, such as one of the `Int` types, instead. We're not considering `BigInt` here, as it works differently, so we'll cover than in a later section.

3.2.1 `UInt`

Before we look at any of these data types, let's consider a more familiar example. Suppose we have a calculator with an 8 digit display:

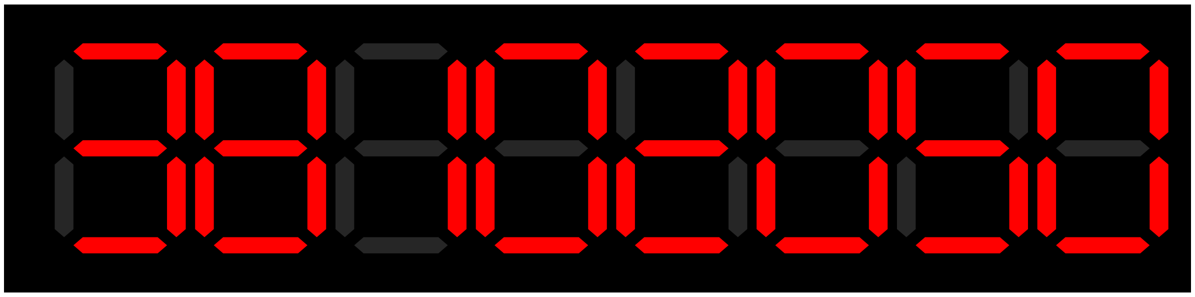
```julia
# Defines which segments light up for each number
# nothing represents an empty space, -1 represents a minus sign
digitionary = Dict(
    nothing => ntuple(_ -> false, 7),
    -1 => (false, false, false, true , false, false, false),
    0  => (true , true , true , false, true , true , true ),
    1  => (false, false, true , false, false, true , false),
    2  => (true , false, true , true , true , false, true ),
    3  => (true , false, true , true , false, true , true ),
    4  => (false, true , true , true , false, true , false),
    5  => (true , true , false, true , false, true , true ),
    6  => (true , true , false, true , true , true , true ),
    7  => (true , false, true , false, false, true , false),
    8  => (true , true , true , true , true , true , true ),
    9  => (true , true , true , true , false, true , true ),
)

# Defines the corners of the seven segments
shapedex = Dict(
    1 => ([-25, -20, 20, 25, 20, -20], [60, 65, 65, 60, 55, 55]),
    2 => ([-30, -35, -35, -30, -25, -25], [5, 10, 50, 55, 50, 10]),
    3 => ([30, 35, 35, 30, 25, 25], [5, 10, 50, 55, 50, 10]),
    4 => ([-25, -20, 20, 25, 20, -20], [0, 5, 5, 0, -5, -5]),
    5 => ([-30, -35, -35, -30, -25, -25], [-5, -10, -50, -55, -50, -10]),
    6 => ([30, 35, 35, 30, 25, 25], [-5, -10, -50, -55, -50, -10]),
    7 => ([-25, -20, 20, 25, 20, -20], [-60, -55, -55, -60, -65, -65])
)

# N is the number to be displayed
# n is the number of digits to display it in
# If negative numbers are allowed, the first digit will be taken up by a minus sign
function sevensegment(N, n, allownegative = false)

    # The number that will actually be displayed
    # Without negatives allowed, functions like UInt
    # With negatives allowed, not exactly like Int, as can't display -10^(n-1)
    N = allownegative ?
        mod(N, -(10^(n-1)-1):(10^(n-1)-1)) :
        mod(N, 10^n)

    # Logic to work out the sequence of digits to be displayed
    digitsN = digits(abs(N))
    truncateddigitsN = reverse(digitsN[1:min(length(digitsN), n - allownegative)])
    displaydigits = [
        allownegative ? (N < 0 ? -1 : nothing) : Nothing[];
        fill(nothing, n - allownegative - length(truncateddigitsN));
        truncateddigitsN
    ]

    # Creates canvas to work on
    p = plot(
        size = (75n + 20, 150),
        grid = false,
        showaxis = false,
        legend = false,
        ticks = false,
        framestyle = :origin,
        xlims = (25, 75n + 35),
        ylims = (-75, 75),
        bg = :black
    )
    # Plots segments with colours according to digits
    for i   1:n
        for j   1:7
            plot!(
                p,
                Shape(broadcast(.+, shapedex[j], (75i, 0))...),
                fillcolor = (digitionary[displaydigits[i]][j] ? :red : :grey15),
                linewidth = 0
            )
        end
    end

    return p

end;
```

```
# Custom function
# First argument is number to be displayed
# Second argument is number of digits
sevensegment(0, 8)
```



This calculator can display the numbers from `0` to `99999999` (it doesn't have a minus sign). What if we try to calculate something that goes beyond this range? For example, let's try multiplying `12345` by `67890`, to which the correct answer would be the nine digit number `838102050`:

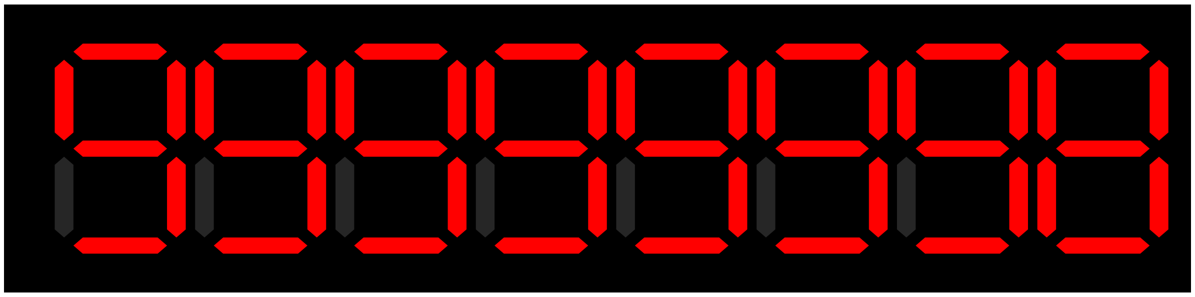
```
sevensegment(12345 * 67890, 8)
```



It didn't have space for the first digit `8`, so it simply cut it off and showed the smaller digits. By analogy to water overflowing a container with lesser volume, this is called *integer overflow*, with the excess digits being simply lost. Of course, most calculators are designed to deal with this sort of issue in a more elegant fashion, but as we'll see, this is exactly what will happen in Julia when we try calculations with answers that are too big.

A related error is *integer underflow*, which happens when we try to calculate a number that is too small to be displayed:

```
sevensegment(3 - 5, 8)
```



Here, we've tried to subtract `5` from `3`, but as we've already discussed, this calculator can't do negative numbers, so instead we end up wrapping back around to the biggest numbers that the calculator can display and counting down from there.

This demonstration shows more or less how the `UInt` types work. Each has a set number of bits, or binary digits (since computers work in binary), to store numeric values in, `8` for `UInt8`, `16` for `UInt16`, etc. If we consider `UInt8`, then this has `8` bits to store numbers, so can only accurately describe the numbers `0` to `255`, since `255 = 2^8 - 1`. If we try to add one more, then:

```
UInt8(255) + UInt8(1)
```

```
0x00
```

```julia
UInt8(0)
```

```
0x00
```

The way that these numbers is displayed may look slightly strange, but this is to distinguish these numbers from `Int` types, which are more commonly used. Specifically it is `0x` (denoting unsigned integers) followed by the hexadecimal representation (that is, base 16), but we needn't worry too much about that. What's important is that when we add `1` to `255` as `UInt8`s, we get `0`, the expected result of integer overflow. We can also demonstrate integer underflow:

```julia
UInt8(0) - UInt8(1)
```

```
0xff
```

```julia
UInt8(255)
```

```
0xff
```

To see a little more clearly how the `UInt8` type works, we're lucky enough to be able to see the exact bits that are stored. This is because `UInt8`, along with many of the other numeric types we'll see here are *primitive* types, meaning that they are stored exactly as the bits that represent them in the computer (the alternative is a *composite* type, which can be thought of as a list of property names, with locations where the corresponding data is stored). Any data which has a primitive type can be input into the inbuilt Julia function `bitstring`, which outputs a string of `0`s and `1`s that are exactly the bits used to represent that number. For example:

```julia
bitstring(UInt8(100))
```

```
"01100100"
```

Indeed, $100 = 64 + 32 + 4$, so it is represented in binary as `1100100`, with the extra `0` in the bit string above to pad it out to 8 binary digits as `UInt8` requires. We can now see why the integer overflow happens, if we try to add two numbers and we end up carrying a `1` beyond the end of the number, then it is lost, which has the effect of wrapping back around to where we started.

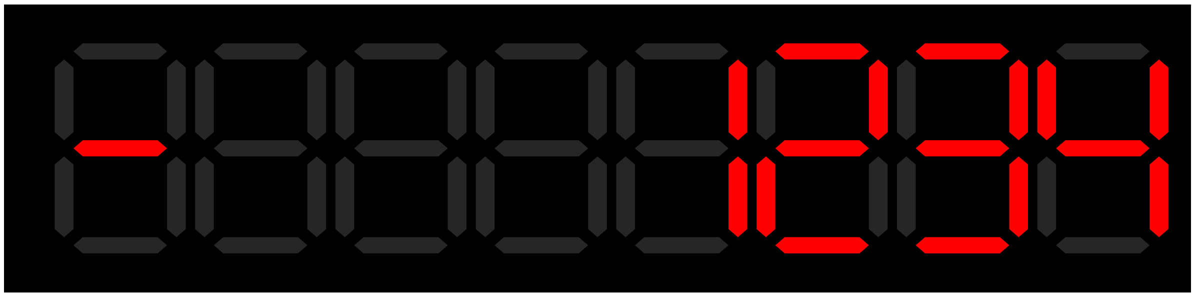
The other `UInt` types work much the same, except they take up a greater number of bits, meaning that they can store bigger numbers at the cost of more space. For example, the largest number that can be stored as a `UInt128` is `340282366920938463463374607431768211455`, i.e. `2^128 - 1`.

Depending on what our data is actually representing, `UInt` types may or may not be appropriate. For example, if we're storing the population of various cities, this will never be negative, and the maximum `UInt32` is `4294967295` (approximately four billion), so `UInt32` may be appropriate as a choice of type. However, if we're storing the change in population from the last year, then this could be negative, so a `UInt` type would no longer suffice. Instead, we need a signed `Int` type.

3.2.2 `Int`

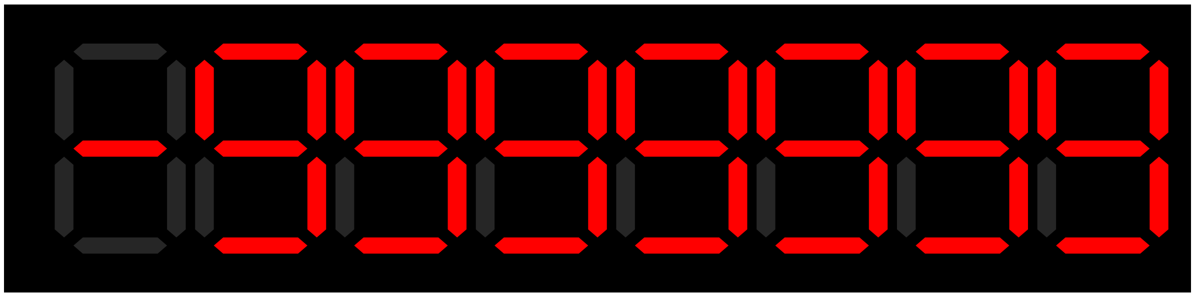
To see how `Int` types work, we'll return again to the calculator, but this time add a third argument. The `true` here sets a variable to allow negative numbers to be shown:

```
sevensegment(-1234, 8, true)
```



Great, so this is an improvement to the calculator, right? It could be, but it has come with a cost, we've lost use of the first digit! So we can do negative numbers, but we need to compromise by lowering the maximum value that we can display accurately. What happens if we overflow this time?

```
sevensegment(9999999 + 1, 8, true)
```



Again, we wrap around, but this time, the smallest number isn't `0`, it's `-9999999`, so that's our answer. Underflows will occur similarly, wrapping around to the largest number `9999999` and counting down as appropriate. This is roughly how the `Int` types work, again as a demonstration we'll consider the smallest one, which is `Int8`.

We've seen how a `UInt8` is simply a binary number with a limited number of digits. For example, if a `UInt8` has bit string `"11010011"`, then we can work out which number this represents by:

128	64	32	16	8	4	2	1
1	1	0	1	0	0	1	1

$$128 + 64 + 16 + 2 + 1 = 211$$

```
bitstring(UInt8(211))
```

```
"11010011"
```

The way that `Int` types such as `Int8` work is to make the first digit subtract instead of add if there is a `1`. We can think of it as a "-128s" place instead of a "128s" place, as follows:

-128	64	32	16	8	4	2	1
1	1	0	1	0	0	1	1

$$-128 + 64 + 16 + 2 + 1 = -45$$

```
bitstring(Int8(-45))
```

```
"11010011"
```

This means that `Int8` can represent any number from `-128` to `127` (as with all the `Int` types, there's one more negative number that it can represent than positive). We can see this as well with bit strings:

```
bitstring(Int8(127))
```

```
"01111111"
```

```
bitstring(Int8(-128))
```

```
"10000000"
```

Similarly, the other `Int` types work much like their `UInt` counterparts, but with the first bit corresponding to a subtracting of a power of 2 rather than the addition of one.

In general, `Int64` is the default whenever you type a number into Julia. An `Int` type is default rather than a `UInt` as it's generally more useful to have the ability to use negative numbers if needed, rather than a load more large positive numbers. As for `64`, modern systems run on 64-bit architecture, i.e. they are built to deal with data in blocks of 64 bits, so it's sensible to store numbers in that format. Some older systems may use 32-bit architecture, in which case Julia will use `Int32` as the default type correspondingly.

However, if you type in a number that's too big (or too small, if it's negative) to be stored in `Int64` format, Julia won't truncate it, instead, it will change the type of the input. First, it will try to store it as an `Int128`, and if that doesn't work, it will store it as a `BigInt`. We can demonstrate this with the `typemax` function, which will tell us that maximum value that the input type can store:

```julia
typemax(Int64)
```

```
9223372036854775807
```

```julia
typeof(9223372036854775807) # Biggest possible Int64
```

```
Int64
```

```julia
typeof(9223372036854775808) # Too big to be an Int64, so is an Int128
```

```
Int128
```

3.3 `BigInt` and `Bool`

We've got two more types under the umbrella of `Integer` that we haven't met yet, the `Signed` type `BigInt`, and the type `Bool` which is neither `Signed` nor `Unsigned`. Let's investigate them further.

3.3.1 `BigInt`

As its name suggests, `BigInt` is used for storing big integers, specifically those that are too big to be stored in any of the normal formats. Since it is `Signed`, we can infer (correctly) that it is also able to store integers that are too small, smaller than the smallest negative number that `Int128` can handle. Indeed, any integer that you can type in can be stored precisely as a `BigInt`:

```julia
googol = 1000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

```
1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

```julia
typeof(googol)
```

```
BigInt
```

The drawback to `BigInt` is that Julia doesn't get much information about the number from the type alone (with `Int64` for example, Julia knows that the number will be a certain size), so calculations cannot be optimised as well. As a result, `BigInt` is recommended to be used only when the precision or size requires it.

3.3.2 `Bool`

The final `Integer` type is `Bool`, which isn't like the other `Signed` and `Unsigned` types. In fact, we often don't think of it as a number at all, rather we think of it in terms of its two possible values, `true` and `false`, which are called the logical values, or Boolean values (hence `Bool`). `Bool` values are returned by various functions like `iseven`, and are used most notably as conditions in `if` statements, or alternative Julia syntax such as short-circuited operations `&&` and `||`, as well as the ternary operator `? :`.

Why are `Bool`s considered to be integers then? `Bool` is a primitive type, so we can use `bitstring` to help find out:

```julia
bitstring(false)
```

```
"00000000"
```

```julia
bitstring(true)
```

```
"00000001"
```

The answer is that they really are integers, `false` is `0` and `true`is `1`! Indeed, some programming languages don't distinguish a special data type for `true` and `false`, instead just using `1` and `0` respectively, which means that some programmers are used to doing things such as multiplying Boolean values together for the logical operation `AND` (where `x AND y` is `true` if both `x` and `y` are `true`, otherwise it is `false`, just as `x * y` is `1` if both `x` and `y` are `1`, otherwise it is `0`). To allow for this, Julia treats `Bool` as a type of `Number`, and defines operations such as:

```julia
true + 7 # 1 + 7 = 8
```

```
8
```

```julia
true * false # 1 * 0 = 0
```

```
false
```

```julia
false ^ false # 0 ^ 0 = 1
```

```
true
```

Convention

Operations like this, although possible, should be avoided, as they are needlessly confusing. More typical operations between `Bool` types are logical operations, like `!`, `&&`, `||` which we'll meet in Chapter 5 when we talk about conditionals.

3.4 The `Float` types

We've seen many ways of representing integers, but from the type diagram we can see that there are many other types of numbers that Julia can represent. Of these, the types that come under the abstract umbrella `AbstractFloat` are the most important, particularly `Float16`, `Float32`, and `Float64`. These are not unique to Julia, indeed the industry standard *IEEE-754* ensures that across all programming languages and all systems, such numbers behave exactly the same, which as we'll see is necessary due to their many quirks.

3.4.1 Floating-point numbers

In this context, `Float` means *floating-point number*, which is a number defined by three parts:

- The *sign* (either positive or negative)
- The significant digits (*mantissa* or *significand*)
- The position of the (binary) decimal point relative to those digits (*exponent*)

The inclusion of this final part is what gives the *floating-point* name, since the decimal point can move around to describe a wider variety of numbers.

This idea may be familiar as scientific notation. For example, the speed of light in a vacuum is $299792458\,\mathrm{ms}^{-1}$ (exactly, as that defines the length of a metre), but this is often represented as $3.00 \times 10^8\,\mathrm{ms}^{-1}$. This is positive, so we don't write the sign, but we understand that the lack of a sign means positive. We choose a level of precision of three significant figures, with the first three significant figures being 3, 0, 0 in this instance, and we write this as a decimal with the decimal point placed after the first digit. Finally, we multiply by an appropriate power of ten to match the size of the exact number we started with, here being 10^8. Assuming that it is understood that we are working with three significant figures, we could write this even more compactly as `+3008`, where the `+` tells us that the number is positive, the first three digits after that are the significant figures or mantissa, and the rest (the `8`) is the exponent. While this format is pretty unreadable to human eyes, it turns out to be rather nice for computers.

Floating-point numbers use the same idea. As their names suggest, `Float16`, `Float32`, and `Float64` (also sometimes known as *half*, *single*, and *double* precision for historical reasons) have 16, 32, and 64 binary digits to use. The first in each case is always the sign bit (`0` for positive, `1` for negative), with the rest needing to be split between the exponent and the mantissa. This is an arbitrary but important choice, with the standard *IEEE-754* decreeing the following split:

Type	Bits for exponent	Bits for mantissa
`Float16`	5	10
`Float32`	8	23
`Float64`	11	52

3.4.2 Bit representation

What does this actually look like in bits? Again, these three types are primitive types, so we can investigate as before with the `bitstring` function. Any decimal typed into Julia will be interpreted as a `Float64` automatically, but we'll convert to `Float16` for a simpler example.

```
bitstring(Float16(0.1))
```

```
"0010111001100110"
```

Referring to the table above, we see that this splits into three parts as:

- Sign bit `0` (so positive)
- Exponent bits `01011`
- Mantissa bits `1001100110`

The sign bit is straightforward enough, but the exponent and mantissa bits are not quite as they seem. With a 5 bit exponent, we could expect to represent any of the numbers from 0 to 31, but this is useless for representing any number between 0 and 1, as they all have negative exponents. Therefore, there is an inbuilt offset of -15, so that instead we represent the exponents -15 to 16. In fact, we do something different (see later) when the exponent bits are all `0` or all `1`, so the normal exponents that we can represent are -14 to 15 for `Float16`s. This works similarly for `Float32`s and `Float64`s, with exponents -126 to 127 and -1022 to 1023 allowed respectively.

As for the mantissa, we could simply store the first 10 significant bits, but that would actually be a little redundant, as in binary, the first bit would (apart from in one very important case, again see later) always be a `1`. As a result, this is `1` is not stored, and instead the 2nd to the 11th significant bits are stored. The same is done for `Float32` and `Float64`, with the 2nd to 24th and 2nd to 53rd significant bits stored respectively.

Returning to 0.1, let's work backwards. The exponent bits is `01011`, which is 11 in decimal, but after the offset it represents -4. The mantissa bits are `1001100110`, so adding the implied `1` back in gives `1.1001100110`. Then, we offset by 4 binary places in the negative direction, so the actual number represented is the binary number `0.00011001100110`, which in decimal is 0.0999755859375.

What's gone wrong here? We tried to represent 0.1, but instead Julia gave us a number that's just under one forty-thousandth less. The reason is that, just as we couldn't represent the speed of light exactly with three significant figures, Julia can't represent 0.1 exactly with a 10 bit mantissa. In fact, 0.1 is `0.0001100110011…` in binary, a recurring decimal, so no matter how many bits we allow for the mantissa, we could never store it exactly. Julia merely rounds to the nearest number it can represent, which happens to be 0.0999755859375.

Confusingly, Julia will display exact decimal values such as `0.1` even if the actual stored values are different. This is a design choice of *IEEE-754*, implemented by the `Base.Ryu` module (which implements the Ryū algorithm), with the output number being the number with the least digits that rounds to the given floating-point number. While this prevents strange behaviour such as typing in `0.1` and Julia echoing back

`0.1000000000000000055511151231257827021181583404541015625` (which is the closest `Float64` representable value to `0.1`, so is actually the true value stored and used for calculating), it does mask some of the strange behaviour of floating-point arithmetic that we've started to uncover. Julia also always displays a decimal point for floating-point numbers, even if they are integers, to clearly distinguish them from `Int` types, and similarly, typing `.0` will turn an integer input into a floating-point input.

```
typeof(12)
```

```
Int64
```

```
typeof(12.0)
```

```
Float64
```

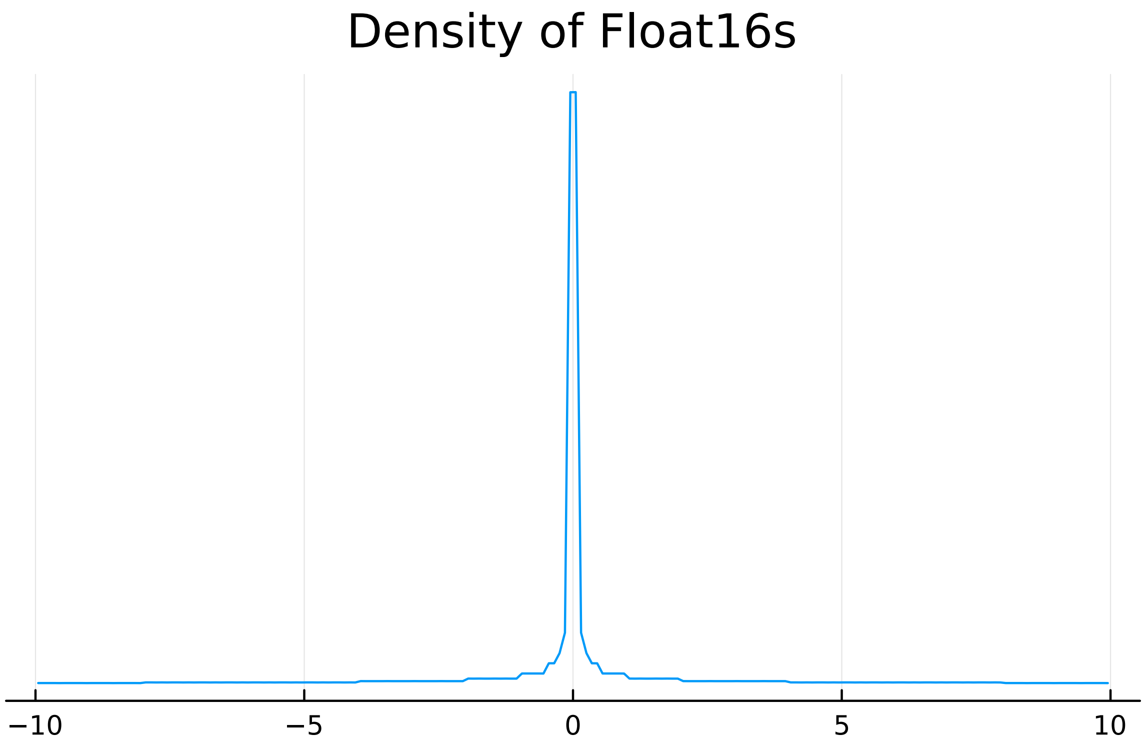
Instead of actual numbers, floating-point numbers can be more accurately thought of as an interval of the number line, containing all of the numbers that round to that specific floating-point representation. For example, the `Float16 0.1` can be thought of as *“all the numbers between* 0.099945068359375 *and* 0.100006103515625*“*, since any number in that range would round to the closest possible representable number, being 0.0999755859375. As the exponent gets smaller, consecutive floating-point representable numbers get closer together, so the intervals are much narrower near 0, with the higher density of representable numbers, and widen towards smaller/larger numbers.

3.4.3 Distribution of floating-point numbers

Since we have only 16 bits to choose, there are a very limited number of possible `Float16` numbers. However, unlike with `Integer` types, these are not evenly spaced, in fact they are very much concentrated around 0, as the diagram below demonstrates:

```
# Generate a vector of all possible Float16s from their bitstrings
float16s = [reinterpret(Float16, n) for n   0x0000:0xffff]

# Parameters for the diagram
totalwidth = 20
intervalwidth = 0.1

# List of endpoints of intervals
intervals = -totalwidth/2:intervalwidth:totalwidth/2
```

```
nbins = length(intervals) - 1
# Counts the number of Float16s found in each interval
frequencies = [count(x -> intervals[i]   x < intervals[i+1], float16s) for i   1:nbins]

# Plot the diagram
plot(
    intervals[1:nbins] .+ intervalwidth/2,
    frequencies ./ intervalwidth,
    legend = false,
    showaxis = :x,
    yticks = false,
    title = "Density of Float16s"
)
```



This is a consequence of the design, since in general numbers closer to 0 can be represented with fewer significant digits, and so more can be represented when we are limited to only 10 mantissa bits. However, this turns out to be an advantage, as it's also useful to distinguish many small numbers, since the difference between them is relatively much larger than between bigger numbers (for example, the difference between 500 g and 501 g is much less important than between 0.5 g and 1.5 g.

Given the format we've met above, we can reason what the largest `Float16` could be. We want to maximise both the mantissa and the exponent, but since the exponent can't be all ones, it must be `"11110"` (representing a shift of the decimal point by 15), while the mantissa is `"1111111111"`. Adding the implied bit, and moving the decimal point, we get the binary number `1111111111100000`, which is 65504 in decimal. Indeed:

```
bitstring(floatmax(Float16)) # floatmax gives the largest normally represented floating-po
```

```
"0111101111111111"
```

Technically, this is the second largest `Float16` value, since there is a positive infinite value to represent anything that needs a larger exponent than 15 to describe.

As for the smallest (positive) value that `Float16` could take, we might guess that it is when the exponent is `"00001"` and the mantissa `"0000000000"`, which is $0.00006103515625 = \frac{1}{16384}$.

```
bitstring(floatmin(Float16)) # floatmin gives the smallest positive normally represented f
```

```
"0000010000000000"
```

However, this is where the exponent `00000` comes in, working differently to other exponents. It turns out that there are a lot of positive floating-point numbers less than this.

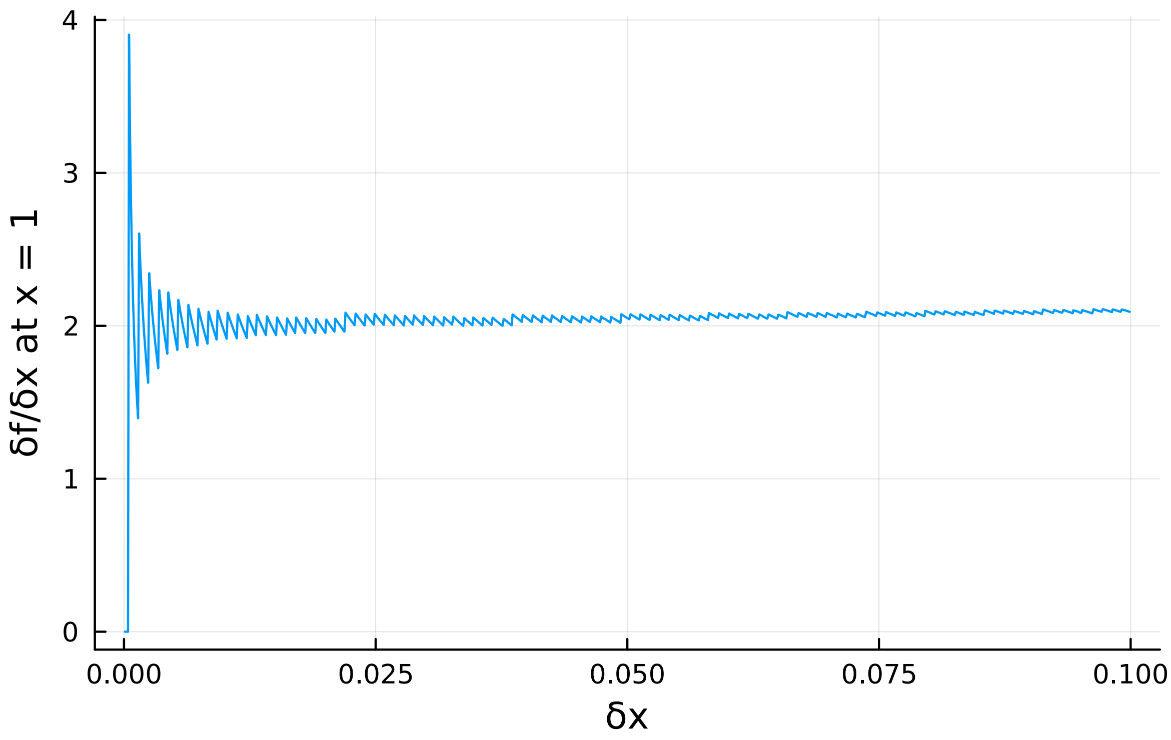
3.4.4 Floating-point arithmetic and errors

If we think of floating-point numbers as a range in which the true value lies, then it's unsurprising that arithmetic won't exactly work as we might expect, particularly when adding numbers of wildly different sizes, or subtracting numbers that are very close to each other. As an example, we'll try to calculate the derivative of $f(x) = x^2$ using the formula:

$$\frac{\delta f}{\delta x}(x, \delta x) = \frac{f(x+\delta x) - f(x)}{\delta x}, \qquad f'(x) = \lim_{\delta x \to 0} \left(\frac{\delta f}{\delta x}(x, \delta x) \right)$$

Since we can't actually plug 0 into this formula (we'd end up dividing 0 by 0), we have to try smaller and smaller values of δx. We'll try it with `Float16` arithmetic to calculate $f'(1)$, starting at `0.1` and making our way down in steps of `0.0001` to `0.0001` (of course, the exact values used in the computation will be the nearest that `Float16` can approximate these by). We'll then graph the result to get an idea of where the values are converging to:

```
f(x) = x^2
 f x(x,  x) = (f(x +  x) - f(x))/ x # The approximate derivative  f/ x at x

plot(
    Float16(0.0001):Float16(0.0001):Float16(0.1),
     x ->  f x(1,  x), # Don't need to specify 1 as Float16 because of multiple dispatch
    legend = false,
    xlabel = " x",
    ylabel = " f/ x at x = 1"
)
```



Looking from the right, we can see that the line, although jagged, is roughly converging in on the correct value of 2. But as we get closer to 0, the line becomes increasingly jagged, and stops converging entirely. What's happened?

The problem comes when calculating `f(1 +  x) - f(1)`, and is twofold. Firstly, let's consider what value `1 +  x` really takes. We can use the `nextfloat` function to tell us what the next representable number after the input is:

```
nextfloat(Float16(1))
```

```
Float16(1.001)
```

```
nextfloat(Float16(1.001))
```

```
Float16(1.002)
```

Herein lies the first issue. While the floating-point representations of `0.0001`, `0.0002`, `0.0003`, etc. are distinct, the density of floating-point numbers is greatly decreased as we go away from `0`, and so the floating-point representations of `1.0001`, `1.0002`, `1.0003`, etc. are not distinct. We can demonstrate by calculating the first twenty of these values:

```
collect((Float16(0.0001):Float16(0.0001):Float16(0.002)) .+ 1)
```

```
20-element Vector{Float16}:
 1.0
 1.0
 1.0
 1.0
 1.001
 1.001
 1.001
 1.001
 1.001
 1.001
 1.001
 1.001
 1.001
 1.001
 1.002
 1.002
 1.002
 1.002
 1.002
 1.002
```

There simply aren't enough possible `Float16` numbers to differentiate these values! This is an example of adding together numbers of different magnitudes, namely adding `1` to numbers between 1000 and 10000 times smaller.

A measure of the required magnitude difference needed for such inaccuracy is given by the *machine epsilon* $\varepsilon_{\text{mach}}$. This is a property of each data type, given by $\varepsilon_{\text{mach}} = 2^{-d}$, where the mantissa is represented by d bits. In the case of `Float16`, $d = 10$, so $\varepsilon_{\text{mach}} = 2^{-10} \approx 0.001$, which agrees with our observations. For `Float64`, $\varepsilon_{\text{mach}} = 2^{-52} \approx 2 \times 10^{-16}$, which for many purposes is too small to care about, but it's large enough that many algorithms using floating-point arithmetic have to be adapted to avoid it.

However, in our case, for the moment, the relative errors remain quite small, as we're still only differing in the fourth significant figure from the intended value. The problem only becomes visible when we square these values (i.e. calculating `f(1 + x)`) and subtract `f(1) = 1`.

`x`	Exact value of `f(1 + x) - f(1)`	`Float16` calculated value
0.0001	0.00020001	`0.0`
0.0002	0.00040004	`0.0`
0.0003	0.00060009	`0.0`
0.0004	0.00080016	`0.0`
0.0005	0.00100025	`0.001953`
0.0006	0.00120036	`0.001953`
0.0007	0.00140049	`0.001953`
0.0008	0.00160064	`0.001953`
0.0009	0.00180081	`0.001953`
0.001	0.002001	`0.001953`

Note that `0.001953` is just an approximation to the real value, which is `0.001953125`. Now, the largest significant figures are cancelled, and what was before a difference in the fourth significant figure is now a difference in the first! What were once errors of a fraction of a percent are now numbers twice as big as they should be (or in some cases, 0 for non-zero numbers).

To avoid this, we need to be aware of the machine epsilon and the limit it places on the accuracy of computations, an inevitability of the way that floating-point numbers are defined. The only way to get around it if you need to do such calculations with floating-point numbers is to increase the precision (e.g. from `Float16` to `Float32` or `Float64`), increasing the length of the mantissa, and decreasing the machine epsilon.

3.4.5 Special exponents `00...0` and `11...1`

There is one more quirk to floating-point numbers that we've eluded to, but not yet mentioned, and that is the numbers with exponent bits all `0` or all `1`. These not only extend the ability of floating-point numbers, but allow deal with the issues of overflow and underflow which we met in the integer case earlier.

A glaring issue in our current implementation of `Float16` is that there is no `0`, since it doesn't really *have* significant figures. It would also seem sensible (although by no means necessary) to have the bit string `"0000000000000000"` represent `0`, but if the exponent `00000` worked like the rest, then this number would be $+1.0000000000_2 \times 2^{-15} = 2^{-15}$, since the implied bit `1` as the first significant figure. Instead, when the exponent is `00000`, we keep the shift of -14 like `00001`, but take the implied bit to be `0` instead of `1`. Therefore, the all zeroes bit string is interpreted as $+0.000000000000_2 \times 2^{-14} = 0$.

Keeping the exponent `00000`, if we change some of the mantissa bits to be non-zero, then we get smaller positive numbers than we could represent before, such as `"0000000101011011"` representing $+0.0101011011_2 \times 2^{-14} \approx 0.0000207$. Such numbers are called *subnormal* numbers, as opposed to the *normal* numbers that we've seen before with other exponents. While it is very useful to have such numbers available for calculation, the initial zero means that they have many fewer significant digits than normal floating-point numbers, and so are even more prone to calculation errors.

Of course, this works the same with sign bit `1`, giving us all the same numbers but with a minus sign. In particular, the bit string `"1000000000000000"` gives the number `-0`, which is actually different from `0`. So now, instead of no `0`, we've got two! If we return to thinking about intervals instead of exact values, then this seems less ridiculous, with `0` encompassing all positive numbers that are too small to represent with even the smallest subnormal number, and similarly `-0` for their negative counterparts.

Meanwhile, the exponent `11111` works in an entirely different way, in fact it doesn't really act as an exponent at all. Rather, it marks out three special values that complete floating-point arithmetic:

- Adding the mantissa `"0000000000"` gives the bit string `"0111110000000000"`, which represents the value `Inf16`, or positive infinity. As a range, it is better thought of as all the numbers too big to represent as a `Float16` (specifically everything that is at least 65520)
- With the sign bit flipped, the bit string `"1111110000000000000"` represents the value `-Inf16`, or negative infinity (or the range of all numbers at most -65520)
- Any other mantissa gives the value `NaN16`, which stands for *"not a number"*. This means that Julia has no idea what answer to give to your calculation

While these are not numbers in the usual sense, they are floating-point values, and we can do arithmetic with them, for example:

```
Float16(1)/Float16(0)
```

```
Inf16
```

```julia
Float16(1)/-Inf16
```

```
Float16(-0.0)
```

```julia
Float16(0)*Inf16
```

```
NaN16
```

More or less any arithmetic you do with `NaN16` will return `NaN16`, because Julia doesn't have a value to calculate with. There are some odd exceptions where the value of the number is irrelevant, such as:

```julia
1^NaN16
```

```
Float16(1.0)
```

The same works for `Float32` and `Float64`, with the exponent of `00...0` being the same as that of `00...01`, but with implied bit `0` instead of `1`. Also, as `Float64` is the default, the infinite and not-a-number values are simply `Inf`, `-Inf` and `NaN` (although you can type in `NaN64` if you like, but Julia will display `NaN`).

3.4.6 `BigFloat`

The other floating-point type is `BigFloat`, which much like `BigInt`, allows for floating-point numbers to be stored with more bits, and therefore more precision. However, unlike `BigInt`, there are decimals that we can write down easily but can't be represented exactly by `BigFloat`. For example, as we've already noted, the decimal 0.1 is represented in binary as `0.0001100110011...`, with infinitely many binary digits. Unless your computer has infinite memory (unlikely), it's clearly impossible to store this exactly as a floating-point number, so instead `BigFloat` simply allows you to increase the level of precision (i.e. the number of mantissa bits, plus one for the implied first bit `1`) as you please, with the exponent stored exactly as an `Int32`.

```julia
BigFloat("0.1", precision = 10)
```

```
0.099976
```

```julia
BigFloat("0.1", precision = 100)
```

```
0.10000000000000000000000000000002
```

```julia
BigFloat("0.1", precision = 1000)
```

```
0.1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Note that we have to input the number `0.1` as a string `"0.1"`, since otherwise Julia will read it as a `Float64` immediately and immediately round it to `53` bit precision.

3.5 Parametric number types

The various `Integer` and `Float` types are by far the most used types, but the others on the diagram are useful in certain circumstances, and for completeness, we'll look at all of them. All three are *parametric* types, which means that they require another type as a parameter before they can become a concrete type and take values.

3.5.1 `Complex`

The `Complex` type is, unsurprisingly, used for dealing with complex numbers. The imaginary constant in Julia is called `im` (since `i` is so commonly used as an index for iteration in loops, see Chapter 5), with complex numbers given by multiplying this by the desired imaginary part and adding the real part:

```julia
z = 1 + 4*im
```

```
1 + 4im
```

```julia
typeof(z)
```

```
Complex{Int64}
```

We see here that the type is not just `Complex`, but `Complex{Int64}`. This means `Complex` where the real and imaginary parts are of type `Int64`. We can do this with any `Real` type:

```julia
Float16(0.4) + Float16(0.6)*im
```

```
Float16(0.4) + Float16(0.6)im
```

We can even mix-and-match, and Julia will choose the best data type to represent both. Here, `1` is of type `Int64`, but `3.5` is of type `Float64`, and Julia knows to pick `Float64` to represent both, as we can see by the addition of `.0` to the end.

```julia
1 + 3.5*im
```

```
1.0 + 3.5im
```

Complex arithmetic is fully implemented with such types, although the same quirks of floating-point arithmetic crop up when the type parameter is one of the `Float` types.

3.5.2 `Rational`

The `Rational` type allows for fractions to be stored and calculated with exact numerators and denominators, avoiding the precision issues of floating-point arithmetic. Again, this is a parametric type, but this type the parameter must be an `Integer` type, as it is the type that the numerator and denominator will take.

Rationals are constructed using the operation `//`, separating the numerator and the denominator:

```julia
12 // 15
```

```
4//5
```

Note that the numerator and denominator are automatically cancelled to the lowest terms. In this case, with `Rational{Int64}`s, this is good, since we want to keep the numerator and denominator as small as possible to avoid issues with overflows and underflows that we saw earlier. Such errors still exist, although they will be identified and give an error message instead of being calculated:

```julia
typemax(Int64)//1 + 1//1
```

```
LoadError: OverflowError: 9223372036854775807 + 1 overflowed for type Int64
```

For maximal precision, we can convert the numerator and denominator to `BigInt` form, which can (inefficiently) store arbitrarily large rational numbers with no error. This can be preferable to using floating-point numbers in instances where precision is more important than speed.

3.5.3 `Irrational`

The final numeric type on the diagram is `Irrational`, and it's special, because while it's a parametric type, the parameter is not the name of another type. Instead, it's a `Symbol`, which is a type that represents variable names. The most familiar example is the constant , which is represented by the `Symbol :` (the colon denotes that the following word should be treated as a `Symbol` and not a variable name), and so has type `Irrational{: }`:

```
 = 3.1415926535897...
```

```julia
typeof( )
```

```
Irrational{: }
```

The main purpose of `Irrational` types is for multiple dispatch. Irrational numbers such as   can't be represented exactly by floating-point types, and if the floating-point approximation is used instead of the exact value, the answer of calculations may be wrong. For example, `sin( )` should exactly equal `0`, but if we were to first approximate   by a floating-point number, we don't get `0`:

```julia
sin(Float64( ))
```

```
1.2246467991473532e-16
```

The `Irrational` type fixes this by defining a method for an input of type `Irrational{: }`, with output exactly `0` (or more specifically, the `Float64` value `0.0`):

```julia
Base.sin(::Irrational{: }) = 0.0 # From https://github.com/JuliaLang/julia/blob/master/bas
```

This overrides the usual method for computing `sin` of a `Float64`, and we get:

```julia
sin( )
```

```
0.0
```

For any other calculation where an exact answer is not specified, the irrational number will be treated just like its `Float64` approximation as normal.

3.6 Resolving problems

It's all very well knowing why these arithmetic errors are happening, but what can we do to avoid them? The first thing you should ask yourself is whether it is worth the effort to avoid them. Sometimes, errors will exist, but are too small to care about, or won't realistically happen, and the easy solution of simply ignoring them is the best way to go.

However, as we've seen, sometimes such errors become a genuine issue, and it's worth knowing about some of the features that Julia has resolve them. We'll return to the examples we saw at the start, and see that they were carefully picked to be able to be solved neatly and concisely.

To begin with, we tried adding `0.1` and `0.2`. Because these don't have exact floating-point representations, and neither does their sum, we got `0.30000000000000004` instead of `0.3`. If we use `Rationals` instead (specifically `Rational{Int64}`s), then the addition will be done exactly.

```
1//10 + 1//5
```

```
3//10
```

If we wish to convert to a `Float64` afterwards, we can do, and the answer will be correct:

```
Float64(1//10 + 1//5)
```

```
0.3
```

Note that we can't go back in the other direction, trying to convert the `Float64` `0.1` to a `Rational` gives:

```
Rational(0.1)
```

```
3602879701896397//36028797018963968
```

This is the exact floating-point value that is stored as a best approximation to `0.1`, but is clearly not equal to `0.1`, so some error would remain if we tried to calculate with this.

How about raising `2` to the power `100`? This time, the error was an integer overflow error, because `2` is an `Int64`, but raising it to the power `100` gives more than `typemax(Int64)`, which is $2^{63} - 1 = 9223372036854775807$. Instead, it overflows, and wraps back around to 0 every 2^{64}. Due to the repeated squaring algorithm that Julia uses to calculate such exponentials, it ends up calculating that$2^{100} = 2^{64} \times 2^{32} \times 2^4 = 0 \times 4294967296 \times 16 = 0$. In this case, `Int128` will be enough to handle this big a number (although `BigInt` would also work), and we only need to convert the `2` to be an `Int128`, as raising this to the power `100` will automatically be understood as a operation on a `Int128`, and will return a `Int128`:

```
Int128(2)^100
```

```
1267650600228229401496703205376
```

Finally, why can Julia correctly calculate `cos( )`, but not `cos( /2)`? The `cos` function has a special method to specifically calculate `cos( )` correctly, just as `sin` did, since   has its own type `Irrational{: }`. However, `/2` does not have such a type, in fact it's not even `Irrational` type – it is the output of the `/` function of types `Irrational{: }` and `Int64`, which is `Float64` as that is the best approximation that exists. This means that its value is not exact, and this floating-point error is carried through when taking the cosine. We can demonstrate this by making use of the `@which` macro, which tells us which method Julia is using to calculate the final answer:

```
@which cos( )
```

```
cos(::Irrational{: })
     @ Base.MathConstants mathconstants.jl:127
```

```
@which cos( /2)
```

```
cos(x::T) where T<:Union{Float32, Float64}
     @ Base.Math special\trig.jl:98
```

To remedy this, Julia provides us with a function specifically for calculating the cosine of multiples of , called `cospi` (similar functions such as `sinpi`, `sincospi`, and `cispi` also exist). We can see that this works as expected:

```
cospi(0.5)
```

```
0.0
```

4 Representing text

! Prerequisites

Before reading this chapter, you are recommended to have read Chapter 2.

4.1 `Char`s and `String`s

4.2 Special characters

4.3 The `*` and `^` operators

4.4 String interpolation

4.5 Indexing `String`s

5 Control flow

! Prerequisites

Before reading this chapter, you are recommended to have read Chapter 2.

In most books, the *flow* is very simple, you simply read page by page in order from front to back. However, this is not always the case. A detective novel may have you flicking back and forth referring to clues cleverly hidden in earlier dialogue, an encyclopedia may consist of several sections which can be read in any order, and a choose-your-own-adventure book will explicitly direct you across its pages following your chosen path. These are all an analogy to the programming concept of *control flow*, that is mechanisms to make your program run in a way other than straight down the page line by line.

While a simple enough idea, the power of this is immense. With a few simple words, we can add branching paths, repeat code, or automatically detect problems and raise exceptions. Not only does this broaden the computations we are able to do, in fact it can be argued that simply the addition of loops and conditionals to a programming language give all the computational power we need, by making the language *Turing-complete*. Everything else we add to the language are just shortcuts to save us time. With that in mind, let's begin, with `begin`.

5.1 `begin-end`

In Julia, the line break is simple enough syntax, that you may even ignore its importance. Indeed, in the REPL, pressing `Enter` doesn't just give a new line, it executes the previous as code too. In a script file, each line is executed one at a time. However, sometimes, there are calculations that are written as several lines of code, but we really want to run as a block. For example, suppose that we run the following two lines of code:

```julia
a = 1
```

```
1
```

```julia
a = a + 1
```

```
2
```

This initialises the variable `a` as `1`, then changes the value of `a` by adding `1` to the previous value of `a`, i.e. `1 + 1 = 2`. If we want to run this program again, we need to make sure that the individual lines are evaluated in the right order. If not (for example if we use ↑ and ↓ in the REPL to rerun previous lines), we could get a different result:

```julia
a = a + 1
```

```
3
```

```julia
a = 1
```

```
1
```

We now add `1` to `a` as before, but the current value of `a` is `2`, so we get `3`. Then, this is overwritten with the initial value `1`, so we end up with `1` as a final answer.

This seems a pretty trivial example, but it demonstrates that the order of lines of code is important, and that there may be reason to group lines of code together in a fixed order that cannot be changed.

Convention

Lines of code grouped together in such a way should be indented (using `Tab` or several spaces) relative to the code around them, to emphasise them as a collection. This is done for all of the *blocks* that we use here, as demonstrated in the example code.

The `begin-end` block is the simplest example of such a block, and also the simplest example of control flow in Julia. The syntax is equally straightforward: we start with the keyword `begin`, on a new line start writing the lines of code we want to enclose, and finish with `end` on a final line:

```julia
begin
    a = 1
    a = a + 1
end
```

```
2
```

By enclosing this code in a block, it is treated as one line of code, so we can't possibly run it in the wrong order. In this sense, it is like a set of parentheses in algebra, grouping together calculations that would normally be done seperately. Much like a set of parentheses, we can also nest many blocks inside each other (indeed, we will want to do this in certain cases, although sometimes it can get a little messy), so we need to make sure that each block has a corresponding `end` codeword.

5.2 Conditionals

Probably the most widely applicable of all of the structures we look at here, conditionals provide the means to change the behaviour based on whether a condition is met or not. First, we'll examine what we mean by a condition in Julia.

5.2.1 Examples of conditions

Conditions come in many forms, but in general they will be a function (including infix operators such as "<") that return a `Bool` value. For example, we have the infix operators:

- Equality of two values can be checked by `==`. This is a *smart* equality, in that it can return `true` even if the same value is represented in two different ways (such as the `Int64 0` and the `Float64 0.0`)

```
5 == 6
```

```
false
```

- Identicality is checked by `===`, i.e. is the data on either side stored identically as bits. `0 === 0.0` returns `false`, since although the values are mathematically the same, they are represented in two different formats

```
0 == 0.0
```

```
true
```

```
0 === 0.0
```

```
false
```

- Inequalities `<` and `>` work exactly as you'd expect for numeric values, with `<=` and `>=` (or indeed   and) including the equality case as well. They have different behaviours on other types, for example `String`s are compared against alphabetical order

```
3 > 4
```

```
false
```

```
"apple"   "banana"
```

```
true
```

- Conditions can be combined together with logical operators, such as `&&` for `AND` and `||` for `OR` (these aren't technically functions, but `Core` Julia syntax)

```
3 > 4 && "apple"   "banana"
```

```
false
```

```
3 > 4 || "apple"   "banana"
```

```
true
```

- When evaluated on two types, the `<:` operator checks if the first type is inherits from the second type (i.e. below it in the type graph, see Chapter 7 for more details on this)

```
Float64 <: Number
```

```
true
```

```
Float64 <: Integer
```

```
false
```

- The function `isa` can be used as an infix operator, checking if the value on the left can be interpreted as the type on the right (`x isa T` can be thought of as `typeof(x) <: T`)

```julia
5 isa Number
```

```
true
```

```julia
isa(5, Number) # Equivalently, in more usual function syntax
```

```
true
```

Like `isa`, other functions can behave as conditions, in fact any function that returns a `Bool` value will do, of which Julia provides many (often with names beginning with `is`). Some examples are:

```julia
isinteger(5)
```

```
true
```

```julia
islowercase(' ')
```

```
true
```

```julia
isabstracttype(Real)
```

```
true
```

```julia
!(false) # Boolean NOT, sends true to false and false to true
```

```
true
```

We'll now see how we can put these conditions to use.

5.2.2 `if`-statements

An `if` statement begins with the keyword `if`, followed by a condition. If the value of the condition is `true`, then the block of code following `if` will be run, and if the value is `false`, the code won't be run. As with `begin-end`, the block of code following `if` starts on the next line, and can be stopped with the keyword `end`.

We can also give some alternative code to be run if the condition is not met, using the keyword `else`. This serves a dual purpose, marking the end of the code block after `if` just like `end`, but also marking the start of a new code block. This can be seen in the example below.

```julia
x = 5
```

```
5
```

```julia
if iseven(x)
    print("Even!")
else
    print("Odd!")
end
```

```
Odd!
```

You may also want to add further conditions, which can be done with `elseif`:

```julia
if x < 0
    print("Negative!")
elseif x > 0
    print("Positive!")
else
    print("Zero!")
end
```

```
Positive!
```

5.2.3 Short-circuited `&&` and `||`

In Julia, the `AND` (`&&`) and `OR` (`||`) operators have an optimisation called *short-circuiting*, allowing for more efficient code:

- Consider the statement `x && y`. If `x` is `false`, then there is no need to evaluate `y`, since the overall result will always be `false`, so Julia short-circuits by ignoring it and simply returning `false`
- Similarly, in the case `x || y` with `x` true, the overall result will be true without checking `y`

This is why `&&` and `||` aren't functions, because functions don't allow this behaviour, and it is a decent optimisation in certain cases. However, Julia allows us to do even more, as `y` doesn't need to be a condition, in fact it can be any code at all. It will then only get executed if the `&&` or `||` is not short-circuited, as we see below:

```julia
3 > 4 && print("Secret message")
```

```
false
```

```julia
3 < 4 && print("Secret message")
```

```
Secret message
```

This could instead be written as a very short `if` statement:

```julia
if 3 > 4
    print("Secret message")
end
```

```julia
if 3 < 4
    print("Secret message")
end
```

```
Secret message
```

5.2.4 The ternary operator `? :`

As well as the above, another common short `if` statements is to choose between two values by a condition. For example, the following is an equivalent way to write `y = max(0,x)`:

```julia
y = if x < 0
    0
```

```
else
    x
end
```

```
5
```

The ternary operator is a way to reduce this to one line. We follow the condition by a question mark `?`, and then give the value if the condition holds and the value if it doesn't, seperating them with a colon `:`.

```
y = (x < 0) ? 0 : x
```

```
5
```

Of course, in this example, we already had a way of doing the same thing in one line (`y = max(0,x)`), but in more complex cases this can be very valuable.

You may wonder what was wrong with the longer `if`-statements, and why we would need to replace them. The answer is that there isn't really anything wrong, but it's additional syntax that we can use to more quickly and easily do common tasks. The other extreme would be that we'd have very little syntax in the language, which would be easy to learn, but more difficult to master, as anything more than the simplest task would require strong familiarity with the few tools at your disposal to get the most out of them. This isn't necessarily a bad thing, but it's not the way that Julia works.

Convention

Short-circuiting and the ternary operator should both be used in preference to `if` where possible for short conditionals. Longer conditional statements (for example where multiple lines of code need to be run depending on the outcome of the condition) should use `if`, as it can get messy with the compact forms.

We now have the means of changing the behaviour of a program depending on whether a condition is met. The other thing that we'd like to be able to do for Turing completeness (and just in general) is repeat the same code multiple times, such that we don't have to write it out again and again, which is achieved by *loops*.

5.3 Loops

5.3.1 `while`

Julia offers two options for loops, the first of which is the `while` loop. This executes the same block of code again and again, but at the start of each iteration, it checks a condition that is put the loop. If the condition is `true`, the loop continues, and if the condition is `false`, the loop is broken, and the program continues past the `while` block. This is obviously very desirable, we most likely don't want to be stuck repeating the same code indefinitely. Also, we'll want to make use of variables whose values we can change, allowing the condition to be true for some time until we choose it to be false and the loop to stop.

A `while` block starts with `while`, followed by a condition, much like the `if`-statements we saw before. The block of code to be repeated then begins on the next line, and lasts until the `end` codeword just like any other block. An example is shown below:

```julia
z = 0
c = 0.3
iterations = 0

# Keeps going if z has absolute value less than 2, and iterations is less than 100
while (abs(z) < 2) && (iterations < 100)
    z = z^2 + c
    # Shorthand, means "add 1 to the variable iterations"
    # Equivalent to "iterations = iterations + 1"
    iterations += 1
end
```

5.3.2 `for`

The other type of loop in Julia is a `for` loop, which runs the same code for each element in some kind of collection. Collections come in many forms, for instance `String`s are a collection of `Char`s, or ranges of numbers such as `1:10` meaning the numbers from `1` to `10`. We investigate more of these in Chapter 9.

To iterate through the collection, we need to give a variable name to represent the element we choose from the array, such as `n` in the example below. Then, we can write the `for` loop, by starting with the keyword `for`, followed in turn by the chosen variable name, the keyword `in` (or the symbol , written by tab-completing `\in`, or simply `=`), then the collection.

```
total = 0

for n in 1:10
    total += n
end

total
```

```
55
```

This can be thought of as a special case of the `while` loop, indeed in many cases, we can write the same thing with `while`:

```
total = 0

n = 1
while n   10
    total += n
    n += 1
end

total
```

```
55
```

Warning

The `for` loop introduces a problem we haven't had with any of the blocks introduced so far: it also defines a variable at the same time, namely `n` in the above. This variable is *local* to the loop, meaning that it doesn't exist outside of the loop. There are some further intricacies with `for` and `while` loops and the variables that can and can't be defined in them, which we'll discuss more in Chapter 6.

5.4 Recursion

There's another way to create a loop in Julia, which is *recursion*. For this, we need to be able to write a custom function, as we learn in Chapter 6.

A recursive function is one that, as part of calculating its answer, calls itself with different inputs (if you call it with the same inputs, you'll get an infinite loop!), with a conditional used to stop the recursion eventually at some base value. On the surface, this might seem useless, or at best no better than `while` and `for` that we've seen before, but let's use it for one of its most classic uses: the Fibonacci sequence.

The Fibonacci sequence is defined as follows:

$$F_0 = 0, \qquad F_1 = 1, \qquad F_n = F_{n-1} + F_{n-2} \quad \text{for } n \geqslant 2$$

and goes:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987$$

We can encapsulate this mathematical definition in code as follows:

```
function fibonacci(n)
    n == 0 && return 0
    n == 1 && return 1
    return fibonacci(n-1) + fibonacci(n-2)
end
```

```
fibonacci (generic function with 1 method)
```

Indeed, this will work:

```
fibonacci(16)
```

```
987
```

However, you might notice that it's quite inefficient. Every time the function recurses, the function is called twice, so the number of function calls in total is increasing exponentially in the input. Depending on the computer that is running it, it will be able to comfortably calculate up to about the 40th Fibonacci number in under a second, but past that it gets debilitatingly slow very quickly. This is a common issue with recursion, where the program can very easily get out of control. Luckily, we can fix this problem rather easily, by simply computing two numbers at a time:

```
# Calculates F   and F
function fibonaccipair(n)
    n == 0 && return 1, 0
    F , F   = fibonaccipair(n-1)
    return F  + F  , F
end
```

```
fibonaccipair (generic function with 1 method)
```

```
function fastfibonacci(n)
    n == 0 && return 0
    F , F   = fibonaccipair(n-1)
    return F
end
```

```
fastfibonacci (generic function with 1 method)
```

This recursive function can quickly compute any Fibonacci number up to F_{92}, when the limitations of Julia's `Int64` number format kick in. `while` and `for` loops could achieve the same thing, but in this case, recursion is a very natural tool for the job.

5.4.1 Stopping loops

When loops are involved, the bane of any programmer is the infinite loop, such as the condition of a `while` loop never becoming `false`, or recursion where the base case is never triggered. Although `for` loops don't have the issue of running infinitely, as they iterate over a finite number of things, they can still take longer than expected and need cutting short.

For `while` loops, the simplest thing to do is to add a failsafe, where after a certain number of iterations, the loop stops anyway (possibly with an error, see the later section on Exceptions). We can see this in the example `while` loop from before; the `iterations` variable starts at `0`, and increments by `1` each cycle, with the loop automatically cutting off if it reaches `100`.

Additionally, a loop can be stopped while it's running by pressing `Ctrl-C` / `Cmd-C` (in fact, this can stop any code from running, not just a loop). This should be seen as a last resort, as terminating a computation halfway through leaves no guarantee that the values that variables take are at all meaningful, but it is very useful if for whatever reason your program gets out of hand.

Recursive functions have their own exception built into Julia to stop them when they get out of hand. This is called `StackOverflowError`, which means that too many nested functions have been called in one go. This limit is actually from your operating system, not Julia, but is big enough that realistically the only way to reach it is with out of control recursion (for instance, the `fastfibonacci` function above can calculate up to F_{52204}, if incorrectly due to `Int64` limitations). Nonetheless, this does provide a hard limit to recursion.

There might also be algorithmic reasons that we would want to cut a loop short, such as if the calculation has finished early, and we don't want to waste time continuing going, or potentially ruining the result. For this purpose, we can use the keyword `break`, telling Julia to stop repeating the current block and move on. As an example, the following loop would otherwise get all the way up to 9 before stopping, but instead it is broken at 5.

```
i = 1
while i < 10
    println(i)
    # Using short-circuiting, "break" is evaluated only when i is 5
    i == 5 && break
    i += 1
end
```

```
1
2
3
4
5
```

Alternatively, we may simply want to stop what we're doing in the current iteration, and move on to the next. This can be done by the keyword `continue`, used in the same way that `break` would be used:

```
for i = 1:3
    i == 2 && continue
    println(i)
end
```

```
1
3
```

5.5 Exceptions

We've already met some exceptions from erroneous code that we've written (and if you're following along yourself by writing some of your own examples, inevitably you'll have seen more). However, we may want to introduce our own error messages, for diagnostic purposes within our own programs.

The simplest way of doing this is the `error` function, which takes a `String` as argument that will become an error message:

```
error("an error has occurred.")
```

```
LoadError: an error has occurred.
```

There are also various `Exception` types, allowing special errors to be constructed with more information, much of which are used in the common errors that we see. The function `throw` serves to cause the error to occur, stopping the program and giving the message.

```
e = BoundsError("message", 10)
```

```
BoundsError("message", 10)
```

```
throw(e)
```

```
LoadError: BoundsError: attempt to access 7-codeunit String at index [10]
```

This is most useful when combined with conditionals, so we can detect if an error has occurred, and raise an exception if needs be.

```
x = -1
x > 0 || throw(DomainError(x, "x must be positive"))
```

```
LoadError: DomainError with -1:
x must be positive
```

6 Creating custom structures

! Prerequisites

Before reading this chapter, you are recommended to have read Chapter 2. Also valuable will be Chapter 5, in particular for the discussion of scope.

In Chapter 2, we already met *variables*, *functions*, and *types* at a basic level. We'll now look at them in a little more detail, in particular focusing on creating our own custom functions and types to morph Julia's behaviour to our needs. We'll also endeavour to better understand variables, primarily their *scope*.

6.1 `Symbol`

First, however, let's meet an important new type: `Symbol`. On the surface, the `Symbol` type is much like a `String`, indeed to write a `Symbol`, we start with a colon `:`, and then write our word:

```
s = :symbol
```

```
:symbol
```

```
typeof(s)
```

```
Symbol
```

`Symbols` work with the same sort of names that work as variable names. If we try to create `Symbols` with invalid names, such as using numeric or `String` literals, we can see that Julia interprets it just as the literal we've written:

```
:6
```

```
6
```

```
typeof(:6)
```

```
Int64
```

```
:"six"
```

```
"six"
```

```
typeof(:"six")
```

```
String
```

This is because a `Symbol` represents what Julia reads when we give it text. If we type `print`, Julia reads `:print`, and recognises this `Symbol` as corresponding to a function. If we type `1.4`, Julia reads this as it is, as the `Float64` literal `1.4`, and deals with it as such.

`Symbol`s are not only used as an internal technicality in Julia though. They are often used at a human level in Julia as names, in place of `String`s, for instance the `Colors` package recognises colour names such as `:red`, `:blue`, `:violet` in `Symbol` form. This is a sensible because `Symbols` are words, whereas `String`s are sequences of characters, and when referring to items by their names, we don't care about the individual characters, we care about the name as a whole.

6.2 Custom functions

6.2.1 Defining a new function

Functions allow preset algorithms to written and referred to by a name, much like variables allow for a value to be referred to by name instead of remembering it. Julia has plenty of inbuilt functions, for all sorts of task, and the ecosystem of packages (see Chapter 8) only adds to that. However, these functions are written to serve as the ingredients, not the end product, and most of the time they aren't specialised enough to do what we want to do in one simple call. Instead, we need to build up these ingredients into a new function of our own.

Note

Technically, what we're writing here are not functions, but *methods* for functions. Functions are the object with the name, and are what are accessed when we call them, but each function can have many methods which dictate the code that will actually run.

Methods don't have names as such, instead they're distinguished by different patterns of inputs, and choosing which method to run when is the point of *multiple dispatch*, which we'll meet in Chapter 7. For instance, `range` is a function, with four methods:

```
methods(range)
```

```
# 6 methods for generic function "range" from Base:
 [1] range(; start, stop, length, step)
     @ range.jl:147
 [2] range(start::T; stop, length) where T<:ColorTypes.Colorant
     @ Colors C:\Users\James\.julia\packages\Colors\mIuXl\src\utilities.jl:230
 [3] range(start::T, stop::T; kwargs...) where T<:ColorTypes.Colorant
     @ Colors C:\Users\James\.julia\packages\Colors\mIuXl\src\utilities.jl:235
 [4] range(start; stop, length, step)
     @ range.jl:142
 [5] range(start, stop; length, step)
     @ range.jl:144
 [6] range(start, stop, length::Integer)
     @ range.jl:145
```

For now, we'll use the words *function* and *method* mostly interchangeably.

First, we need to give the function a name (the same as we would a variable), as well as give variable names to the inputs to the function that we expect. To tell Julia that we want a function, we use the keyword `function`, followed by the name that we want to give it. Then, we list out the number of inputs (also called *arguments*) that we want, giving each of them a name as well to be referred to later. These need to be separated by commas, just as multiple inputs to a function do when we call it.

```
function spherevolume(radius)
```

Convention

In Julia, function names are usually written in lowercase with no spaces. They should also be named to not conflict with existing functions, so as not to overwrite them. We can, however, add onto existing functions, as we'll see in Chapter 7.

Next comes the code, doing whatever calculations we need it to do on the inputs. This works much the same as code anywhere else, so we won't really focus on it here. Indeed, for some later examples, you may not recognise all the syntax being used, but you can feel free to ignore it if so. All that's important is that something fills in this gap.

```
    volume = 4/3 *   * radius^3
```

Convention

This code is indented once compared to the level of the `function` keyword, for readability purposes to clearly show which code belongs to the function.

Finally, we want to return our answer, which will be what we get back when the function is run. Using the `return` keyword before a value tells the function to return that as the answer (this is particularly useful when combined with conditionals from Chapter 5), but if `return` is never specified, the function will default to returning the last value it calculated. We then finish off the block with `end`.

```
end # In this case, the volume is the last thing calculated, so we don't need `return`
```

Putting this all together, we have a function:

```
function spherevolume(radius)
    volume = 4/3 *   * radius^3
    return volume # Using `return` here is redundant, but for clarity we show it here
end
```

```
spherevolume (generic function with 1 method)
```

We can use this just as we would any other function:

```
spherevolume(1)
```

```
4.1887902047863905
```

```
spherevolume(0.781592641796772)
```

```
1.9999999999999996
```

Alternatively, functions can be written using more algebraic syntax. We do away with the `function-end` block, instead writing in one line how to calculate the output from the inputs

```
cubevolume(length) = length^3
```

```
cubevolume (generic function with 1 method)
```

```
cubevolume(1)
```

```
1
```

```
cubevolume(1.2599210498948732)
```

```
2.0
```

Convention

For longer functions, combining this syntax with a `begin-end` block can give the same effect as using `function-end`. However, the latter is preferable in most cases; the algebraic-style syntax is intended as convenient shorthand for the normal `function-end` syntax for quick functions.

It's possible to make functions that return multiple arguments, simply by following `return` with a list of comma-separated values. For example, we can mimic the inbuilt function `minmax`, which given two inputs, returns the smallest followed by the largest:

```
function minmaxagain(x, y)
    if x > y
        return y, x
    else
        return x, y
    end
end
```

```
minmaxagain (generic function with 1 method)
```

```
minmaxagain(4,5)
```

```
(4, 5)
```

```
minmaxagain(5,4)
```

```
(4, 5)
```

If we provide one variable as an output, it will take the value of the `Tuple` containing all outputs:

```
x = minmaxagain(5,4)
```

```
(4, 5)
```

```
x
```

```
(4, 5)
```

However, if we provide two, the first will get the first value, and the second the second:

```
y, z = minmaxagain(5,4)
```

```
(4, 5)
```

```
y
```

```
4
```

```
z
```

```
5
```

Using certain symbols as function names (particularly those in the *Symbol, Math* category in Unicode) allows them to be used automatically as infix operations, as we are used to with the likes of `+`, `-`, `*`, etc.:

```
±(x, y) = (x + y, x - y)
```

```
± (generic function with 1 method)
```

```
0.8 ± 0.03
```

```
(0.8300000000000001, 0.77)
```

6.2.2 Prescribing the inputs

Not all functions should accept all inputs, in fact, there are very few that should! What's more, we may want the function to do different things depending on the inputs, via multiple dispatch. To remedy this at least partially, Julia provides type declarations, allowing us to prescribe what types are allowed for specific inputs.

To declare the type of a specific argument, we follow the variable name we've given it by two colons `::` and the type name:

```
function spherevolume(radius::Float64)
```

Often, there may be many types which would work as the input. If there is a natural supertype that encompasses all of these, we can use that, even if it is an abstract type:

```
function spherevolume(radius::Real)
```

More unusual possibilites can be realised with the `Union` type, which takes other types as parameters, and acts as a supertype for all of them:

```
# Can't use this for `Bool`, as `true + one(true)` gives `2`, which isn't a `Bool`
function nextinteger(x::Union{Signed,Unsigned})
    x + one(x)
end
```

```
nextinteger (generic function with 1 method)
```

Here, the use of `Union` is justified, since we do the same thing for `Signed` and `Unsigned` inputs, and we can't use `Integer` because we want to exclude `Bool`. However, if the algorithm of your function changes significantly depending on the exact types it gets as inputs, then multiple methods for the function with each of those cases covered should be written. To see this, and more complicated type declarations, see Chapter 7.

What we can't do in this way is restrict to only certain values within a type. For example, we can't declare that the `radius` in `spherevolume` needs to be positive, because multiple dispatch sees only types, not values. This will require a check in the body of the function, for example:

```julia
radius < 0 && throw(DomainError(radius, "sphere must have positive radius."))
```

Another way we may wish to prescribe the inputs is by giving them default values, making specifying them optional. The following function cuts a `String` `s` after the `n`th character, if it is long enough. Here, it's combined with type specification, but it needen't be.

```julia
function cutstring(s::String, n::Int64 = 1)
    length(s) < n ? s : s[1:n]
end
```

```
cutstring (generic function with 2 methods)
```

```julia
cutstring("hydrogen", 6)
```

```
"hydrog"
```

```julia
cutstring("boron", 6)
```

```
"boron"
```

```julia
cutstring("gold")
```

```
"g"
```

Note

When we defined `cutstring`, we can see that it defined a function with 2 methods, instead of 1 method like the other functions we've defined. This is because, in the background, it's defined a second method:

```julia
cutstring(s::String) = cutstring(s, 1)
```

with only one input, corresponding to what happens if you don't give `n` a value.

Functions with lots of inputs can be clunky to call, as you need to remember exactly what order the inputs need to go in. Instead, we can write functions with keyword arguments, which differ from normal arguments in that you specify them by name, not by order.

The volume of a cone of radius r and height h is given by $\frac{1}{3}\pi r^2 h$. However, writing a function for this, it's not clear what order `radius` and `height` should go. We could make a choice, but instead, let's make them keyword arguments, so that someone using the function can't get them the wrong way around. Arguments are usually separated by commas, but after a semi-colon, all arguments become keyword arguments:

```julia
conevolume(; radius, height) = 1/3 *   * radius^2 * height
```

```
conevolume (generic function with 1 method)
```

If we try to call the function as normal, we'll get an error:

```julia
conevolume(1, 1)
```

```
LoadError: MethodError: no method matching conevolume(::Int64, ::Int64)
```

Instead, we need to specify which argument is which by naming them:

```julia
conevolume(radius = 1, height = 1)
```

```
1.0471975511965976
```

Since they have names, the order is now irrelevant:

```julia
conevolume(height = 1, radius = 1)
```

```
1.0471975511965976
```

Just as with normal arguments, keyword arguments can have types declared or default values given. Function can also have a combination of normal arguments and keyword arguments:

```julia
function friedeggs(eggs; people = 2)
    println("This recipe serves $(people) people.")
    println("You will need $(eggs * people) eggs.")
    println("To make fried eggs, simply crack the eggs into a pan and wait.")
end
```

```
friedeggs (generic function with 1 method)
```

```julia
friedeggs(5)
```

```
This recipe serves 2 people.
You will need 10 eggs.
To make fried eggs, simply crack the eggs into a pan and wait.
```

```julia
friedeggs(2; people = 12)
```

```
This recipe serves 12 people.
You will need 24 eggs.
To make fried eggs, simply crack the eggs into a pan and wait.
```

6.2.3 Anonymous functions

We've already seen two different syntaxes for defining functions, but in fact, there's a third:

```julia
f = x -> x^2 - 3x + 2
```

```
#15 (generic function with 1 method)
```

This function behaves just as others do:

```julia
f(1)
```

```
0
```

```julia
f(10)
```

```
72
```

However, it's a little different. This is no longer a function called `f`, it's a variable called `f` whose value is a function. Using the `methods` functon to list the methods, we get:

```julia
methods(f)
```

```
# 1 method for anonymous function "#15":
 [1] (::var"#15#16")(x)
     @ In[37]:1
```

As we can see here, `f` is what is called an anonymous function, which is a fitting description as they don't have a name like normal functions do. Anonymous functions don't participate in multiple dispatch, and so can only have one method, but can be used more easily as a variable. Their main use is for functions that themselves take functions as arguments, such as `minimum`, which finds the minimum value that a function takes on a given set of inputs:

```julia
minimum(f, 0:0.01:3)
```

```
-0.25
```

Indeed, we don't even need to give an anonymous function a variable name, and we can enter it as a literal:

```julia
minimum(x -> x^2 + 4x - 3, -3:0.01:-1)
```

```
-7.0
```

6.2.4 Piping and composing functions

A common occurrence in programming with functions is the need to apply several functions one after the other to the same value. The problem is, this can lead to a mess of parentheses. Let's say we have a variable called `capital`:

```julia
capital = "Antananarivo"
```

```
"Antananarivo"
```

We want to do two calculations on it. First, we want to count the number of unique letters (ignoring upper and lower case), which we can do as follows:

```julia
length(unique!(sort!(collect(lowercase(capital)))))
```

```
7
```

Also, we want to find if the last appearance of the letter `'a'` is an even number of letters from the end:

```
iseven(findfirst('a', reverse(lowercase(capital))))
```

```
false
```

Both of these are a bit of an eyesore. Julia provides two ways of helping with this, each with their own benefits. The first is *piping*, which uses the `|>` operator to successively apply functions to the output of the previous step:

```
capital |> lowercase |> collect |> sort! |> unique! |> length
```

```
7
```

This can also include anonymous functions as arguments, which can be useful when we need to add an input to one of the functions along the way:

```
capital |> lowercase |> reverse |> (x -> findfirst('a', x)) |> iseven
```

```
false
```

This is much cleaner, as it better separates the many function applications into a readable format. The second option we have is *composition*, which is done with the   operator (typed by tab-completing `\circ`):

```
(length   unique!   sort!   collect   lowercase)(capital)
```

```
7
```

Under the hood,   creates a special type of function called a `ComposedFunction`, which works just like a normal function, but calling each of its component parts one at a time in the prescribed order. This means we can give it a variable name, and use it as we would a normal function:

```
contrivedfunction = (iseven   (x -> findfirst('a', x))   reverse   lowercase)
```

```
iseven   var"#21#22"()   reverse   lowercase
```

```julia
typeof(contrivedfunction)
```

```
ComposedFunction{ComposedFunction{ComposedFunction{typeof(iseven), var"#21#22"}, typeof(reve
```

```julia
contrivedfunction(capital)
```

```
false
```

Note

While we aren't thinking of them as such here, all functions are actually just a special type of variable. Specifically, that special type is a subtype of `Function`, named `typeof([function-name])`, with the property that preceding a `Tuple` with a function's name starts multiple dispatch on the methods stored under that name. They are also `const` values (see later), so cannot be redefined as any other type.

6.3 Custom types

6.3.1 Types of types

Types are all important in letting Julia know how to deal with the data we give it. As we've seen above, they are invaluable in determining what values a method of a function can accept, and Chapter 3 gives an idea of why we might choose a particular format to store our data in (e.g. `Float64` for speed, `Rational{BigInt}` for accuracy) Much as we can add to Julia's many functions with our own, we can do the same with types. First though, let's cover the different flavours of types which Julia affords us.

The most common is the *composite type*. Such a type has various *fields*, each with a given value. We'll see what this actually means when we come to define one, but an example for now is `ErrorException`, which is the type you get when using the function `error`.

```julia
e = ErrorException("an error")
```

```
ErrorException("an error")
```

```julia
throw(e)
```

```
LoadError: an error
```

It has a single field `msg` of type `AbstractString`, which stores the error message to be displayed.

```
fieldnames(ErrorException) # The function `fieldnames` lists the fields of a type
```

```
(:msg,)
```

```
ErrorException.types # The property `types` of a type lists the types of the fields
```

```
svec(AbstractString)
```

Related to the composite type is the *parametric type*, which behaves much the same, but it also requires one or more other types in curly braces after its name to specify exactly what data it represents. An example of this from Chapter 3 is `Rational`, which requires a type parameter to tell us what format the numerator and denominator are in:

```
typeof(4//5)
```

```
Rational{Int64}
```

```
typeof(BigInt(4)//BigInt(5))
```

```
Rational{BigInt}
```

A *mutable type* can be either composite or parametric, but it has the property that its fields can be edited after it is instantiated. The data corresponding to a variable with a mutable type is stored not as values, but as *pointers* which serve as addresses to where the values are stored, and therefore these values can be changed without changing any of the data defining the variable. The prototypical example of a mutable type is `Array` (which is also parametric), as described in Chapter 9.

If a composite type or a parametric type has no fields, and is immutable, then there is no way to distinguish any one instance of that type from any other. This makes it a *singleton type*, examples of which include function types such as `typeof(sin)`, and `Nothing` which represents the absence of a value.

```julia
typeof(Nothing())
```

```
Nothing
```

```julia
fieldnames(Nothing)
```

```
()
```

A *primitive type* is one where the data is stored primitively, meaning it is just zeroes and ones in memory (of course, everything is just zeroes and ones eventually, but most types have fields as an intermediate step). For instance, `Int64` is a primitive type, where any data in this format is 64 consecutive bits of memory that exactly correspond to the value we mean by it.

FInally, an *abstract type* cannot take a value, but instead serves as a label collecting together many related types, allowing them to be referred to as a collective (such as for the purposes of multiple dispatch). Again, we return to Chapter 3 and numeric types for an example of this, with `Number` encompassing all possible numeric types, `Real` excluding complex numbers, `Integer` excluding fractional numbers, and so on.

Some inbuilt types don't fit nicely into these categories, particularly those more fundamental to the inner workings of Julia. For example, `String` isn't primitive, but still doesn't have any fields; it's mutable, but you can't change its characters. However, any types that we define will follow the usual rules.

6.3.2 Defining a new type

To define a new composite type, we use the keyword `Struct`, followed by the name we want to give the type. Then, on

```julia
struct Animal
    name::String
    symbol::Char
    legs::Int64
end
```

Convention

Types are named in upper camel-case, meaning every word is capitalised, with no spaces separating words. For examples, consider types that we have already met, like `String`, `BigInt`, and `Function`.

We can instantiate a variable of this type by using the name of the type like a function, with the arguments being the values we want to give to the fields in the same order that we defined them:

```julia
elephant = Animal("Elephant", ' ', 4)
```

```
Animal("Elephant", ' ', 4)
```

```julia
flamingo = Animal("Flamingo", ' ', 2)
```

```
Animal("Flamingo", ' ', 2)
```

We use dot syntax to query the value of one of the fields, following the variable with a `.`, and then the name of the field we want to know:

```julia
elephant.symbol
```

```
' ': Unicode U+1F418 (category So: Symbol, other)
```

If these were mutable types, we could use the same syntax followed by an `=` to change their values. However, `Animal` is immutable, so we get an error:

```julia
flamingo.legs = 1
```

```
LoadError: setfield!: immutable struct of type Animal cannot be changed
```

It is also possible to declare where our new type belongs in the type graph, by giving it an abstract type as a parent node. This can be done by following the type name with `<:`, and then the abstract type:

```julia
struct Prime <: Integer
    p::BigInt
end
```

```julia
p = Prime(BigInt(5))
```

```
Prime(5)
```

```
p isa Integer
```

```
true
```

If no parent is specified, it defaults to `Any`, which is the abstract type at the top of the tree.

We can also define our own parametric types, by following our type name with curly braces, inside which we can give variable names to our list of parameters.

```
struct Doublet{T}
    first::T
    second::T
end
```

```
Doublet{Int64}(10, 20)
```

```
Doublet{Int64}(10, 20)
```

```
Doublet{Symbol}(:ten, :twenty)
```

```
Doublet{Symbol}(:ten, :twenty)
```

We can restrict these parameters if needs be, for example if we needed the parameter `T` to be numeric, we could have written:

```
struct Doublet{T <: Number}
```

Parameters are usually other types, but don't need to be, for example `Array` (see Chapter 9) has two parameters `T` and `N`, which define the type of the elements `T`, and the number of dimensions `N`. We'll see an example of this below when considering inner constructors, which are required to deal with parametric types with values as parameters.

Note

The other varieties of types can also be defined:

- If you want to create an abstract type to add to the type graph, replace `struct` with `abstract type` (note that you can't specify any fields, since an abstract type can never take a value)

- If you want to make your type mutable (see Chapter 9), replace `struct` with `mutable struct`
- If you (for some reason) want to create your own primitive type, replace `struct` with `primitive type`, and add the number of bits you want your type to take up in place of fields. However, be warned that most problems are made more complicated by using primitive types instead of composite types

6.3.3 Inner constructors

For some composite and parametric types we create, it may suffice to simply specify the fields and their types. However, we may want to have more flexibility, or further restrictions, in defining instances of our new type. To do this, we'll want to make use of *constructors*.

Constructors looks very much like a method of a function, and can be thought of as such, behaving the same way when to calling it with a `Tuple` of inputs, and participating in multiple dispatch. What's special about them, however, is that the function's name is the type which they construct, for instance calling a constructor for the type `Rational{Int64}` might look like `Rational{Int64}(4, 5)`. In fact, we've already used constructors unwittingly in Chapter 3, to convert between the various numeric types in Julia, and in Chapter 5, to create exceptions.

The first type of constructor we'll look at is the *inner constructor*. These are defined inside the `struct` block (hence *inner*), after the fields are listed, and their primary purpose is to impose further restrictions on the fields than simply their types. To create an inner constructor, we use the same syntax as we did when creating a function (either the `function-end` block or the algebraic `f(x) =` syntax will work, but not the anonymous function syntax as we don't want to create an anonymous function).

Convention

The arguments of this function should be the fields, in the same order as listed before, with the same names and type declarations. While this function will be called just like any other, with the variable names strictly local to the function, it's pointlessly more difficult to read if the field names are changed or reordered.

The body of the function will consist of whatever checks we need to do, with either the fields corrected if possible, or `Exceptions` thrown when the values are irretrievably wrong. However, since there doesn't exist a function to create a new instance of our type to return (that's what we're writing with the inner constructor), we need to use the special function `new`. The `new` function is exclusive to inner constructors, and simply instantiates a variable of the type in question with fields as listed in the arguments. For example, a type with two fields that we wanted to have values `2` and `"two"` would be created by `new(2, "two")`. Mostly, this will be the value you want the function to return, so we can finish the function with a call to `new`, creating and returning our new object back to us.

For a familiar example, let's create a type called `Password`, in which we'll store a single field of type `String` to represent the password. This will be a parametric type, with a single parameter `N` which will be an `Integer`. Using values as parameters is another task for inner constructors, as we can't do it with type declarations. We also want to impose some restrictions on the password:

- It must contain at least `N` characters in total (so if `N` is zero or negative, then any length will be allowed)
- It must contain at least two letters and two numbers
- It must contain at least one character than isn't a letter or a number

The tools from Chapter 5 suffice to make these checks, so we'll use them to build our inner constructor:

```
struct Password{N}
    word::String

    function Password{N}(word::String) where N
        # Checks that N is an integer
        N isa Integer || throw(TypeError("parameter for `Password` must be an `Integer`.")

        # Checks length
        length(word) < N && throw(ArgumentError("password must contain at least $N charact

        # Counts the number of each character type
        letters = numbers = others = 0
        for char   word
            isletter(char) ? (letters += 1) : (isnumeric(char) ? (numbers += 1) : (others
        end

        # Raises errors if any character types are not represented enough
        letters < 2 && throw(ArgumentError("password must contain at least 2 letters."))
        numbers < 2 && throw(ArgumentError("password must contain at least 2 numbers."))
        others < 1 && throw(ArgumentError("password must contain at least 1 character othe

        # If no error, creates the new `Password`
        new{N}(word)
    end
end
```

Note

The `where` keyword used here is required when defining constructors of parametric types. In this instance, we've only needed to use it to tell Julia to define `N` locally as the name of a parameter, rather than trying to substitute in the value of some variable called `N`. More uses of `where` are demonstrated in Chapter 7.

Let's try this out:

```julia
Password{8}("great")
```

```
LoadError: ArgumentError: password must contain at least 8 characters.
```

```julia
Password{8}("fantastic")
```

```
LoadError: ArgumentError: password must contain at least 2 numbers.
```

```julia
Password{8}("fanta5t1c")
```

```
LoadError: ArgumentError: password must contain at least 1 character other than letters and 
```

```julia
Password{8}("fanta5t1(")
```

```
Password{8}("fanta5t1(")
```

There is more that can be done with inner constructors, including writing functions with different names to change how the type can be constructed, or leaving mutable types with some fields uninitialised, to be added in later, but we'll leave it there for now.

6.3.4 Outer constructors

The other type of constructor is the *outer constructor*, so called because they are defined outside of the `struct` block. These each take a different pattern of inputs, so that multiple dispatch knows which to call, and will use another constructor (possibly initially a sequence of outer constructors, but ultimately the inner constructor has to be called eventually) to return an instance of the desired type.

We'll use this to add onto our `Password` type. Instead of coming up with our own password, let's say that giving an `Integer` instead of a `String` as an argument to the `Password{N}` constructor will generate a password of that length for us:

```julia
const LETTERS = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklnopqrstuvwxyz"
const NUMBERS = "0123456789"
const SYMBOLS = "!@#\$%^&*"

using Random # for the function `randperm`

function Password{N}(n::Integer) where N
    n < 5 && throw(ArgumentError("cannot have a password of length $n."))
    word = *(
        rand(LETTERS, 2)...,
        rand(NUMBERS, 2)...,
        rand(SYMBOLS, 1)...,
        rand(LETTERS*NUMBERS*SYMBOLS, n-5)...
    )
    # This use of `randperm` shuffles the characters
    # Otherwise, we'd always start with two letters, two numbers, and one symbol
    Password{N}(word[randperm(n)])
end
```

Note

Nowhere in this constructor have we:

- Said that `N` needs to be an `Integer`
- Said that `n` needs to be at least `N`

These are already handled by the inner constructor, which is what we call at the end when we give `Password{N}` a single `String` argument, so we don't need to do these checks here. We've also built the random password to ensure that it contains the requisite character, meaning that it will pass those tests. The only check we have done, checking that `n` is at least 5, is to ensure that the code within the outer constructor doesn't break, specifically when we try to choose `n-5` random items later.

Note

The `using` keyword we have used here is to be able to make use of a *package*, which is an add-on to Julia with extra functionality, in this case `Random`. More discussion of packages can be found in Chapter 8.

We can now generate random passwords:

```julia
Password{8}(10)
```

```
Password{8}("&jc6I891S3")
```

```julia
Password{8}(15)
```

```
Password{8}("k2sLw9t!OS#2JBb")
```

```julia
Password{8}(6)
```

```
LoadError: ArgumentError: password must contain at least 8 characters.
```

Outer constructors need not refer directly to inner constructors, instead we can chain them together. For example, let's suppose that we don't specify any arguments to `Password{N}`, and we want that to generate a password of length `N` (the minimum allowed by `Password{N}`). We can do this simply by calling our first outer constructor:

```julia
Password{N}() where N = Password{N}(N)
```

```julia
Password{10}()
```

```
Password{10}("C6Knx5k1#g")
```

```julia
Password{15}()
```

```
Password{15}("dxoy7\$fn^BI%%8*")
```

Convention

Inner constructors should be used minimally, only where an outer constructor could not perform the same task.

Note

Why are there two different types of constructors anyway? As similar as they might seem, the two types of constructors perform quite different roles.
Inner constructors are intended as intrinsic parts of the type, perhaps enforcing conditions upon the parameters and fields that can't be done by type declarations as we've seen, although they do have some other uses particularly with mutable types. Once the type has been defined, they are set in stone, and cannot be altered or amended.
Outer constructors, meanwhile, are just like methods are to functions. They can be changed, overwritten, and new ones added at will (although doing this to inbuilt types might cause problems!) They are more limited than inner constructors, but their greater flexibility means that anything possible with an outer constructor should be done with an outer constructor.

6.3.5 Adding a display style

When the value of a new type is displayed, the default appearance looks very much like the inner constructor we use to create it, as we can see from the `Password` outputs earlier on. This is deliberate, automatically allowing for string interpolation, interpretation, and evaluation by Julia if needed. However, it's not always the most useful way of showing the value, and in certain self-referential cases, can break entirely. Therefore, it's valuable to be able to customise this look, which we can do by overloading (i.e. writing a new method for) the inbuilt function `show`.

Before we write a new method, we need to `import` the old ones, so that we don't redefine `show` accidentally and then nothing but our new type can be displayed by Julia. `show` comes from the module `Base`, so we use:

```
import Base.show
```

In Chapter 7, we'll make further use of `import`, while in Chapter 8 we'll understand it a little better.

Now, we can write our own method. The best way to do this is to write a method with two inputs, one of type `IO`, and one of our new type that we want to display. The reason for this is that the `IO` argument determines where the output will go, allowing us to display our value wherever it needs to be displayed.

```
function show(io::IO, x::[type])
```

When outputting the type, there are two function that you'll likely want to use: `print` and `println`. Both take the `IO` argument first, followed by a `String` that you want printed.

Both output this as text to wherever `IO` tells them to. However, `println` also adds a newline character (the equivalent of pressing `Enter` on your keyboard) after this message, while `print` doesn't, allowing you to keep adding to the same line. You can use these in combination with other string manipulations (see Chapter 4).

Unlike other functions, we're not interested in a value being returned by `show`. Indeed, the last thing your method will likely do is call one of `print` or `println`, which have no output (or more accurately, their output is `nothing`, the value of the singleton type `Nothing`), so the same will be true of `show`. This is fine, however, as the output that we need is the printed text, which happens in the middle of the function anyway regardless of what is returned at the end.

For a quick demonstration of this, let's change the output of our `Password` type. Of course, `Password`s should be secret, so we don't want to show their value to the world whenever they enquire! Instead, we'll output `"•"` in place of each of the characters:

```
show(io::IO, x::Password) = print(io, "•"^length(x.word))
```

```
show (generic function with 380 methods)
```

```
Password{10}()
```

```
••••••••••
```

```
Password{15}()
```

```
•••••••••••••••
```

Note

Much like functions, types are really just another type of variable. Their type is `DataType`, and preceding a `Tuple` with their name calls a constructor. Similar to functions, once defined, they cannot be redefined as anything else.

6.4 Scope of variables

Suppose, somewhere deep in the Julia codebase, someone has defined the variable `x` (this isn't much of a supposition, it happens countless times!). When Julia initialises, and this code is run, the name `x` is, at some point, used to refer to some value. Wouldn't it be really annoying if, because of this, no-one was allowed to use the name `x` ever again? Or, every time that data is given a name somewhere in the code, its value is stored in perpetuity, waiting to be overwritten? This sounds ridiculous, but if it weren't for the system of variable *scope*, it would be a reality.

The scope of a variable defines where and when its name may be used to reference its value. Outside of scope, it may refer to a different defined value (acting as a different variable, but happens to have the same name in a different context), or give an error saying that it's not defined.

The scope always starts where the variable in question is first defined, but pinning down where it ends is trickier. To illustrate this, consider the following examples:

```
x = 0

for i   1:2
    x = 1
    i = 1
end

x
```

```
1
```

We start by setting `x = 0`, and then run a `for` loop. First, it runs with `i = 1`, setting the value of `x` to `1`, and then setting `i` to be `1`. Then, it runs with `i = 2`, again setting the values of `x` and `i` to `1`. The loop concludes, and we display the value of `x`. Unsurprisingly, perhaps, its value is `1`. What about `i`?

```
i
```

```
LoadError: UndefVarError: `i` not defined
```

`i` doesn't have a value at the end of this section of code, even though the loop finished with `i = 1`. This is because the variable `i` is defined inside the `for` loop, so its scope stops at the keyword `end` that marks the end of the loop.

Now let's tweak the code slightly, changing the iterated variable from `i` to `x`:

```julia
x = 0

for x   1:2
    x = 1
    i = 1
end

x
```

```
0
```

This starts again by setting `x = 0`, but then the loop runs differently. This time, we begin the loop with `x = 1`, set the values of `x` and `i` to `1`, and do the same with `x = 2`. So why isn't the final value of `x` `2`? By defining `x` as the iterating variable in the `for` loop, we've unwittingly created a new variable called `x` that belongs only within the loop (a variable that exists only within a certain block of code like this is *local* to that block). Therefore, the original `x` is entirely unaffected. Just as before, `i` is only defined within the `for` loop, so Julia won't recognise it outside of that:

```julia
i
```

```
LoadError: UndefVarError: `i` not defined
```

6.4.1 Code blocks

Note

The diagrams in this section are generated in Julia, and the code used is given in the functions hidden below. To read them, all you need to know is:

- Each dot marks where a variable is defined or overwritten
- Solid lines show where a variable is in scope
- Dotted lines show where a variable is temporarily out of scope, with the variable name having been reused to define a local variable in a new block

```julia
"""
WARNING
This code is deliberately scrappy and unoptimised.
Its only purpose is to generate the diagrams below.
```

```
Known issues
- Code must be correct Julia code (in particular, no missing or unmatched `end`s)
- Only blocks `begin`, `if`, `elseif`, `else`, `while`, `for`, `function`, `struct` can be
- Functions can't be defined short-form (e.g. `f(x) = x + 2`), they must use `function` to
- Variables given as inputs to functions are only detected if they are the first input
- Doesn't deal with branched code (e.g. `if`-statements) properly
- Ignores `local` and `global`

"""

using Plots

# Given a list of lines of code, finds what the indentation level of each should be,
# as well as a list of blocks in which said line is nested
function findstructure(code::Vector{String})

    n = length(code)
    indents = zeros(Int64, n)
    blocks = fill(String[], n+1)
    # The keywords that mark an indentation. Not comprehensive, e.g. do and let are not in
    INDENTORS = ["begin", "if ", "while ", "for ", "function ", "struct "]

    # Keeps track of the current indent level
    rollingindent = 0
    # Goes line by line, increasing/decreasing indent and updating blocks as it goes
    for i   1:n
        if code[i] == "end"
            rollingindent -= 1
            indents[i] = rollingindent
            pop!(blocks[i])
        elseif code[i] == "else"
            indents[i] = rollingindent - 1
        else
            indents[i] = rollingindent
            for indentor   INDENTORS
                if startswith(code[i], indentor)
                    rollingindent += 1
                    append!(blocks[i], [indentor])
                end
            end
        end
        blocks[i+1] = copy(blocks[i])
    end

    return indents, blocks[1:n]
end

# Returns a list of `Symbol`s detailing the scope of a given variable `var` line by line i
# `var` is assumed to be defined on line `linedefined`
# `indents` and `blocks` are as from `findstructure`
#
# Returned `Symbol`s are:
# - :out means out of scope
# - :in means in scope
# - :def means that the variable is defined (or redefined) on this line
# - :paused means that the variable is currently out of scope due to local redefinition
function findscope(
    var::Symbol,
    linedefined::Int64,
    code::Vector{String},
    indents::Vector{Int64},
    blocks::Vector{Vector{String}}
)

    n = length(code)
    scope = Vector{Symbol}(undef, n)

    @assert 1   linedefined   n

    # The keywords that begin a new scope
    BEGIN_SCOPERS = ["while ", "for ", "function ", "struct "]
    # The keywords that begin a new scope in which the scope of `var` may be paused
    PAUSE_SCOPERS = ["for " * string(var), "function ", "struct "]

    # Set the scope before and at initial definition
    scope[1:(linedefined-1)] .= :out
    scope[linedefined] = :def

    # Finds the indent of the scope in which the variable lies
    scopeindent = findlast([b   BEGIN_SCOPERS for b   blocks[linedefined]])
    isnothing(scopeindent) && (scopeindent = 0)
```

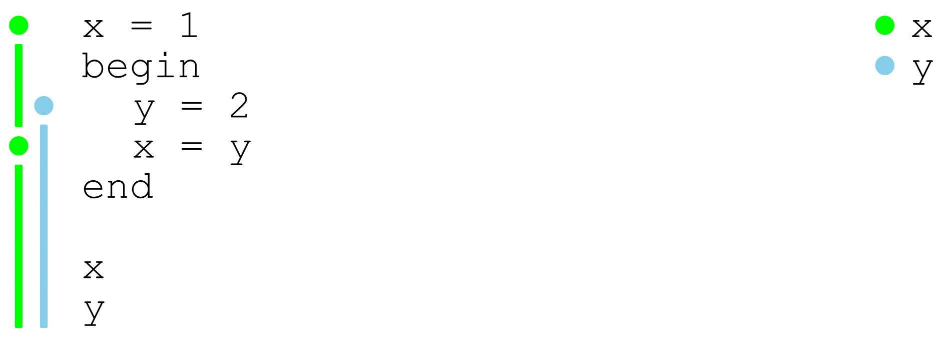
```
# Lists the indentation levels where new scopes have begun
pausableindents = Int64[]
pauseindent = 0

# Goes line by line, finding in what state of scope the variable lies at the end of ea
for i   (linedefined+1):n
    # If scope is currently paused, check if it has ended yet
    if scope[i-1] == :paused
        if indents[i] == pauseindent
            scope[i] = :in
        else
            scope[i] = :paused
        end
    # If scope is not paused
    else
        # If we exit the block in which the variable is defined, it is out of scope fo
        if indents[i] < scopeindent
            scope[i:end] .= :out
            break
        # If we start a new scope, we need to track it by adding its indentation level
        elseif any(startswith.(code[i], PAUSE_SCOPERS))
            append!(pausableindents, indents[i])
            # If our variable isn't defined in a function's arguments, it remains in s
            if startswith(code[i], "function ") &&
                !startswith(code[i], Regex("function .+\\(" * string(var) * "[,)]"))
                scope[i] = :in
            # Otherwise, the scope is paused
            else
                scope[i] = :paused
                pauseindent = pop!(pausableindents)
            end
        # If our variable is redefined
        elseif startswith(code[i], string(var) * " =")
            # If this happens in a new scope, then it is locally redefined
            if !isempty(pausableindents)
                scope[i] = :paused
                pauseindent = pop!(pausableindents)
            # Otherwise, the value is overwritten
            else
                scope[i] = :def
            end
            # Otherwise, we remain in scope
            else
                # If an `end` keyword ends a new scope, we need to delete it from `pausabl
                if code[i] == "end" &&
                    !isempty(pausableindents) &&
                    indents[i] == pausableindents[end]

                    pop!(pausableindents)
                end
                scope[i] = :in
            end
        end
    end

    return scope
end

# Creates a diagram detailing the scope of variables in `vars` defined on lines `linesdefi
function scopediagram(
    vars::Vector{Symbol},
    linesdefined::Vector{Int64},
    code::Vector{String}
)

    n = length(code)
    nvars = length(vars)
    nvars == length(linesdefined) || error("mismatched number of variables and lines where
    nvars > 4 && error("can draw scope of at most 4 variables at a time")

    p = plot(
        framestyle = :origin,
        size = (400, 18n),
        grid = false,
        legend = false,
        showaxis = false,
        ticks = false,
        xlims = (0, 40),
        ylims = (-n, 0)
    )

    indents, blocks = findstructure(code)
```

```
# Writes the code with correct indentations
plot!(
    annotations = [
        (1 + nvars + 2indents[i], 0.5-i, (code[i], 8, :black, :left), "courier")
    for i   1:n]
)

cs = [:lime, :skyblue, :orange, :pink]

# Draws the scope of each of the variables
for j   1:nvars
    scope = findscope(vars[j], linesdefined[j], code, indents, blocks)

    # Dots for (re)definitions
    scatter!(
        [(j-0.5, 0.5-i)
            for i   1:n if scope[i] == :def],
        markercolor = cs[j],
        markerstrokecolor = cs[j]
    )

    # Filled line for in scope
    plot!(
        [Shape([j-0.6, j-0.4, j-0.4, j-0.6], [1-i, 1-i, -i, -i])
            for i   1:n if scope[i] == :in],
        fillcolor = cs[j],
        linecolor = cs[j]
    )

    # Dotted line for paused scope
    plot!(
        [Shape([j-0.6, j-0.4, j-0.4, j-0.6], [1-i, 1-i, -i, -i])
            for i   1:n if scope[i] == :paused],
        fillcolor = cs[j],
        linecolor = :white,
        fillstyle = :x
    )

    # Key
    scatter!(
        (35, 0.5-j),
                markercolor = cs[j],
                markerstrokecolor = cs[j],
                annotations = (36, 0.5-j, (string(vars[j]), 8, :black, :left), "courier")
            )
    end

    return p
end;
```

By default, every code block (that is, one of the pieces of code that begins with a codeword such as `if` or `function`, and ends with `end`) has its own behaviour with respect to the scope of variables defined outside/inside of it. We'll examine the ones we've met so far, as well as a couple that we'll meet later.

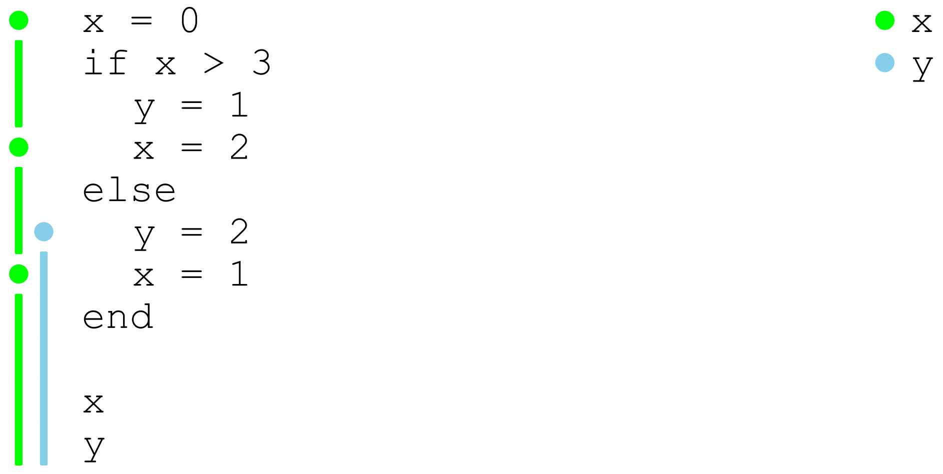
`begin-end` is the simplest code block, and has perhaps the simplest scope behaviour, in that it has no effect on scope. Anything defined inside it can be accessed on the outside, and vice versa.

```
scopediagram(
    [:x, :y], [1, 3],
    [
        "x = 1",
        "begin",
        "y = 2",
        "x = y",
        "end",
        "",
        "x",
        "y"
    ]
)
```



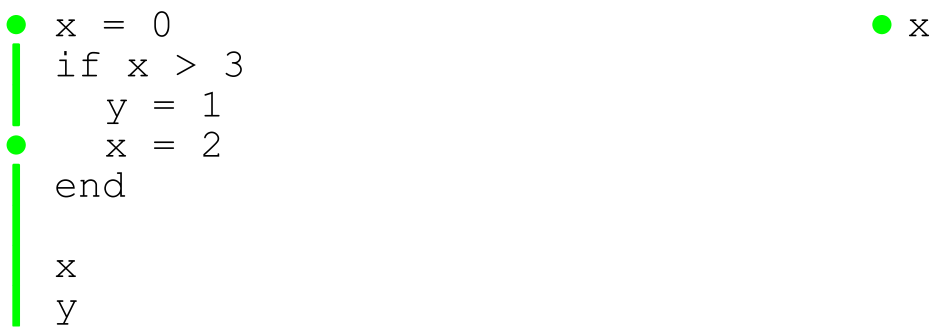
Similarly, `if`-statements have no special effect on scope. The same is true for its replacements `"? :"`, `"&&"`, and `"||"`, although these aren't really code blocks in and of themselves.

```
scopediagram(
    [:x,:y], [1,6],
    [
        "x = 0",
        "if x > 3",
        "y = 1",
        "x = 2",
        "else",
        "y = 2",
        "x = 1",
        "end",
        "",
        "x",
        "y"
    ]
)
```



However, since `if`-statements give branching paths, it's possible to miss the definition of a variable, like `y` in the example below, so the query of the value of `y` later will cause an `UndefVarError`.

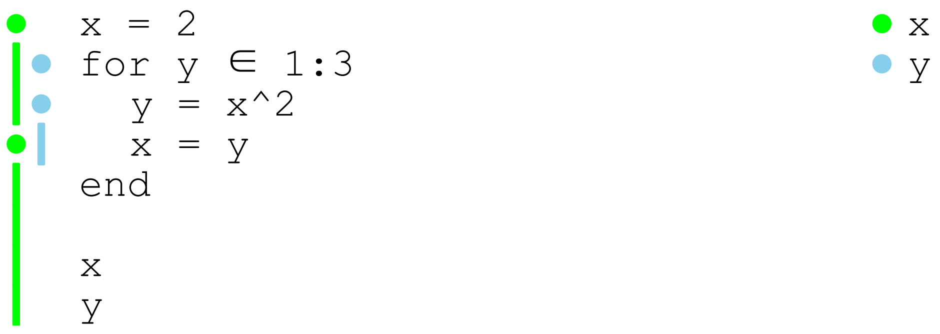
```
scopediagram(
    [:x], [1],
    [
        "x = 0",
        "if x > 3",
        "y = 1",
        "x = 2",
        "end",
        "",
        "x",
        "y"
    ]
)
```



```
x = 0
if x > 3
   y = 1
   x = 2
end

x
y
```

`while` and `for` blocks can reference and update any variable that was defined before they began. Anything defined for the first time within them, however, is local to the loop, and is lost as soon as the loop ends.

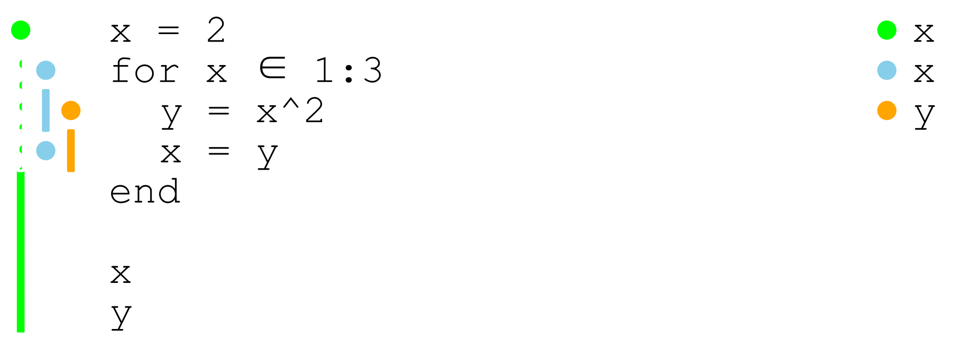
```
scopediagram(
    [:x, :y], [1, 2],
    [
        "x = 2",
        "for y   1:3",
        "y = x^2",
        "x = y",
        "end",
        "",
        "x",
        "y"
    ]
)
```



```
x = 2
for y ∈ 1:3
  y = x^2
  x = y
end

x
y
```

A special case exists for the variable or variables that are iterated through many values by the `for` loop. These are considered new local variables regardless of whether they have been defined before or not.

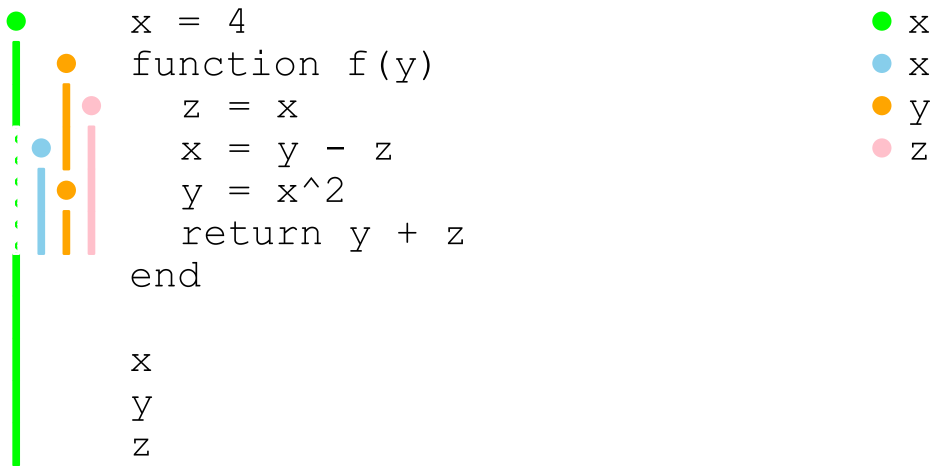
```
scopediagram(
    [:x, :x, :y], [1, 2, 3],
    [
        "x = 2",
        "for x   1:3",
        "y = x^2",
        "x = y",
        "end",
        "",
        "x",
        "y"
    ]
)
```



Note here that the period when the original `x` is temporarily out of scope exactly corresponds to when the other `x` is in scope (perhaps unsurprisingly).

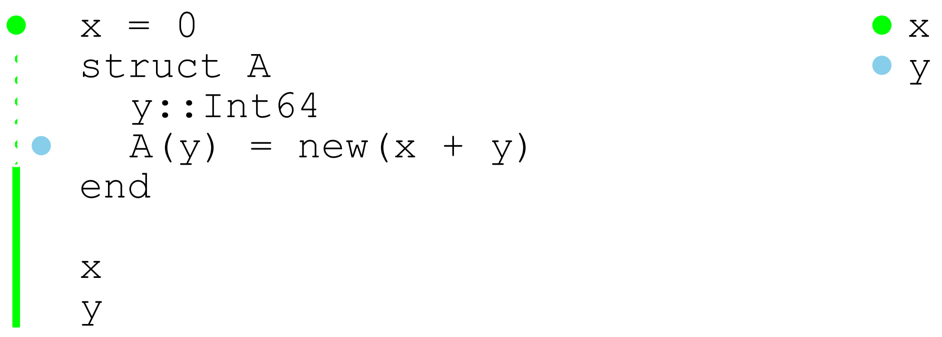
For a `function`, any variables defined within the function are local to the function, but simply referencing a variable from outside the function works fine because it remains in scope.

```
scopediagram(
    [:x, :x, :y, :z], [1, 4, 2, 3],
    [
        "x = 4",
        "function f(y)",
            "z = x",
            "x = y - z",
            "y = x^2",
            "return y + z",
        "end",
        "",
        "x",
        "y",
        "z"
    ]
)
```



Meanwhile, for a `struct`, variables defined outside can't be referenced inside, and no variables defined inside can be referenced outside.

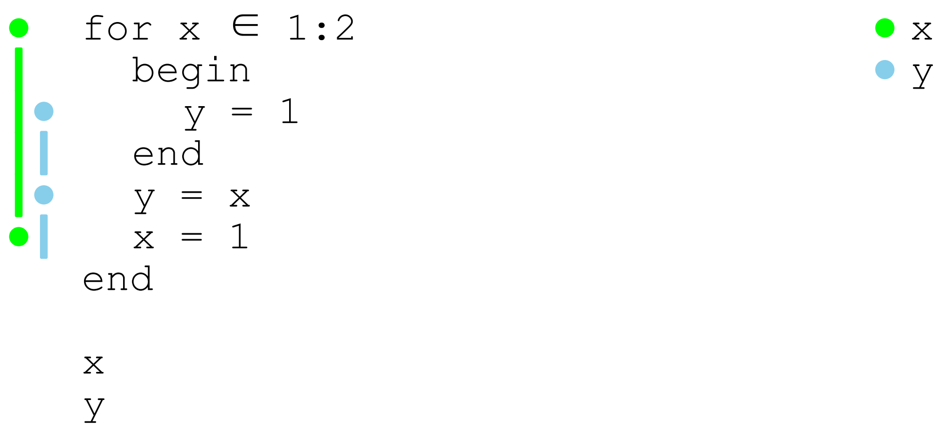
```
scopediagram(
    [:x, :y], [1, 4],
    [
        "x = 0",
        "struct A",
        "y::Int64",
        "A(y) = new(x + y)",
        "end",
        "",
        "x",
        "y"
    ]
)
```



```
x = 0
struct A
  y::Int64
  A(y) = new(x + y)
end

x
y
```

Scope works in a nested fashion, for example a variable defined inside a `begin-end` that itself is inside a `for` loop will not be accessible outside the `for` loop, like `y` below.

```
scopediagram(
    [:x, :y], [1, 3],
    [
        "for x   1:2",
        "begin",
        "y = 1",
        "end",
        "y = x",
        "x = 1",
        "end",
        "",
        "x",
        "y"
    ]
)
```



```
for x ∈ 1:2
  begin
    y = 1
  end
  y = x
  x = 1
end

x
y
```

6.4.2 `local` and `global`

For more control over the scope in which your variables exist, you may wish to use the keywords `local` and `global`. These keywords go before a definition or reassignment of a variable, marking its scope explicitly:

- Using `local x` creates a new local variable with the name `x`, with scope restricted to the block it is defined in. This is how the iterating variable of a `for` loop is defined (even though we didn't type `local`), and so is why it behaves differently to any other variable in a `for` loop
- Using `global x` declares that this variable name is a *global* variable, which means that whenever the name is used, it refers to the same variable (except where an explicitly `local` variable called `x` exists). If, in a large file, `global x` is written anywhere, it applies to the whole file, not just the references further down the page

Convention

Both of these are situationally useful, but particularly the use of `global` should be avoided most of the time, as it can have unexpected effects on other bits of supposedly unrelated code. Instead, the value can be passed around between functions as an additional argument, or for unchanging values, a constant (`const`) variable can be used.

There are some differences in behaviour between the REPL and code run from `.jl` files when it comes to `local` and `global` variables. In general, the REPL allows more flexibility, while `.jl` code will produce warnings about clumsy use of `global` variables. If code is mostly contained within functions in `.jl` files (as is advised), no variable will be automatically `global`, so this isn't a major worry.

Note

Technically, global variables are not truly global, instead their scope is the *module* in which they lie. Modules are a way to group related code together under a single name, and can be convenient for sharing code.
If no module is ever declared, code will run in `Main`, so variables will be functionally global. However, `global` variables from other modules (such as from installed packages, see Chapter 8) would not be accessible, and any `global` variables you declare won't affect identically named variables in these other modules.

6.4.3 Constants

Constants are a special type of variable, intended to have a single unchanging value which can be accessed from anywhere (i.e. a global scope). For example, you could be running a business that want to make a 4.3% profit on anything it sells, so you might declare:

```
const PROFIT_MARGIN = 1.043
```

```
1.043
```

Then, anywhere else in your program, you could refer to `PROFIT_MARGIN`, instead of having to remember what its value is (provided that the variable name `PROFIT_MARGIN` isn't taken by another value).

Convention

`const` variables are named differently from normal variables, instead using capital letters with words separated by underscores. One good reason to do this is ensure it keeps its global scope, for instance, if we called our constant `p`, then it wouldn't be accessible inside a function that had an input called `p`.

To demonstrate how constants differ from normal variables, consider the following example:

```
N = 3
```

```
3
```

```
addN(x) = x + N
```

```
addN (generic function with 1 method)
```

```julia
addN(2)
```

```
5
```

```julia
N = 4
```

```
4
```

```julia
addN(2)
```

```
6
```

With the non-constant value `N`, the function `addN` looks up the value of `N` in order to add it each time it is needed. However, if we use constants:

```julia
const M = 3
```

```
3
```

```julia
addM(x) = x + M
```

```
addM (generic function with 1 method)
```

```julia
addM(2)
```

```
5
```

```julia
const M = 4
```

```
WARNING: redefinition of constant M. This may fail, cause incorrect answers, or produce othe
```

```
4
```

```julia
addM(2)
```

```
5
```

Now, the `const` value `M` is included in the function verbatim, and since we never redefined the function after that, it's still expecting `M` to be the same as it was originally. Notice that we were warned of exactly this issue when we changed the value of the constant.

In fact, most of the time, trying to change the value of a `const` won't give a warning, it will just result in an error message, with the value not being changed. Earlier, we noted that functions and types are actually just `const` variables of a form, and you'll note that you won't be able to change their values:

```julia
Int64 = 12
```

```
LoadError: cannot assign a value to imported variable Core.Int64 from module Main
```

There are good reasons you may want to use `const`s, for values that you want to define programmatically but never change; some examples built into Julia are the mathematical constants , , etc. As we've seen though, it's necessary to ensure that these constants are never redefined, they really should be constant!

6.5 Example: Unit conversion

A common problem to come across is the need to convert some quantity between units. There are many online tools that do this, but let's put some of our new knowledge to the test and create our own crude tool to do the same. There are many ways to approach this, but we'll be creating a type to represent a unit, as well as a function to convert between them.

We'll start with creating a `Unit` type to represent the units we want to convert between.

```julia
struct Unit
end
```

What fields do we need? We need a conversion factor to be able to convert between units, which will be relative to some standard unit, such as the SI units, and this will be some sort of number. Since we don't know exactly what type it will be, and we don't particularly mind, we can use the abstract type `Real` as an umbrella term. Also, we need to know what quantity the unit measures, as we can't convert between a unit of length and a unit of mass, for example! This could come in various forms, but the simplest will be just to store this as a `String`. For the purposes of this example, we won't need any more fields, but for more functionality you may wish to add others.

```
struct Unit
    factor::Real
    quantity::String
end
```

We haven't used an inner constructor here, but we will add some outer constructors to allow for easier construction of new units. First, we've mentioned the idea of a base unit, to which all the factors are relative. We would represent this as a `Unit` with a `factor` of `1`, so let's add a constructor where if the `factor` isn't specified, it's assumed to be `1`, giving the base unit:

```
Unit(quantity::String) = Unit(1, quantity)
```

```
Unit
```

When we think of the way that units are usually defined to us, it's generally in terms of another unit that measures the same thing (i.e. 1 kilometre is 1000 metres). We can add this as a constructor too, using the `factor` of an old `Unit` to calculate the new `factor`:

```
# Creates the new unit corresponding to x lots of u
Unit(x::Real, u::Unit) = Unit(x * u.factor, u.quantity)
```

```
Unit
```

Now that our type is defined, we can create some variables of this type. For example, here are some units of length:

```
metre = Unit("length")
kilometre = Unit(1000, metre)
centimetre = Unit(1//100, metre)
inch = Unit(2.54, centimetre)
foot = Unit(12, inch)
yard = Unit(3, foot)
mile = Unit(1760, yard)
```

```
Unit(1609.3440000000003, "length")
```

some of mass:

```
kilogram = Unit("mass")
gram = Unit(1//1000, kilogram)
pound = Unit(453.59237, gram)
ounce = Unit(1//16, pound)
shortton = Unit(2000, pound)
longton = Unit(2240, pound)
metricton = Unit(1000, kilogram)
```

```
Unit(1000, "mass")
```

some of time:

```
second = Unit("time")
minute = Unit(60, second)
hour = Unit(60, minute)
day = Unit(24, hour)
julianyear = Unit(365.25, day)
gregorianyear = Unit(365.2425, day)
tropicalyear = Unit(365.24219, day)
```

```
Unit(3.1556925216e7, "time")
```

and some of angles:

```
radian = Unit("angle")
fullcircle = Unit(2 , radian)
degree = Unit(1//360, fullcircle)
arcminute = Unit(1//60, degree)
arcsecond = Unit(1//60, arcminute)
```

```
Unit(4.84813681109536e-6, "angle")
```

As is often done with `Units`, we may wish to combine them together to make new ones, such as combining `metre` and `second` to get `metrepersecond` measuring `"speed"`. Although our implementation doesn't allow for this automatically (we'd have to tell it that dividing a unit of `"length"` by a unit of `"time"` gives a unit of `"speed"`, etc.), we can do this manually:

```julia
metrepersecond = Unit("speed")
# Speed of light in a vacuum
const C = Unit(299792458, metrepersecond)

# One lightyear is the distance travelled by light in a vacuum in one Julian year
# Calculated by distance = speed * time
lightyear = Unit(C.factor * julianyear.factor, metre)
```

```
Unit(9.4607304725808e15, "length")
```

```julia
# One astronomical unit (au) is approximately the average distance between the Earth and t
au = Unit(149597870700, metre)

# One parsec is approximately the distance to an object of parallax angle 1 arcsecond (1//
parsec = Unit(1/arcsecond.factor, au)
```

```
Unit(3.085677581491367e16, "length")
```

Now let's write a function to convert between units. We need three inputs, the amount to convert, the `Unit` that this amount is in, and the `Unit` to convert it into.

```julia
function convertunits(x::Real, u ::Unit, u ::Unit)
end
```

Warning

The natural choice of function name here would be `convert`, but this is a crucial function used by Julia to convert between types, so we don't want to overwrite that. Theoretically, we could use multiple dispatch (Chapter 7) to write our own method, but that would be bad practice, as the `convert` function is specifically meant for converting between types and nothing else.

First, we need to check that the `Unit`s entered are compatible, namely that they measure the same quantity. This can be done by an `if`-statement, or even simpler, short-circuited, with an error displayed if the quantities do not match:

```julia
u .quantity == u .quantity || throw(ArgumentError("units measure different quantities."))
```

Now we just need to do the conversion. A little thinking (or experimenting) tells us that the correct formula for this is to multiply `x` by the `factor` of `u` , and then divide by the `factor` of `u` :

```
x * u .factor / u .factor
```

This is the value we want, so we return it, finishing the function.

```
function convertunits(x::Real, u ::Unit, u ::Unit)
    u .quantity == u .quantity || throw(ArgumentError("units measure different quantities."
    x * u .factor / u .factor
end
```

```
convertunits (generic function with 1 method)
```

Now we can put this function to the test:

```
convertunits(110, kilometre, mile)
```

```
68.35083114610673
```

```
convertunits(45, degree, radian)
```

```
0.7853981633974483
```

```
convertunits(28, day, minute)
```

```
40320.0
```

```
convertunits(1, parsec, lightyear)
```

```
3.2615637771674333
```

```
convertunits(1, metricton, longton)
```

```
0.9842065276110605
```

```julia
convertunits(12, inch, pound)
```

```
LoadError: ArgumentError: units measure different quantities.
```

As mentioned, there are other ways of approaching this problem, and ways of improving this method further. Some ideas for improvement that you might want to try yourself are:

- The `quantity` field of `Unit` is used to check if two `Units` measure the same thing, but perhaps could do with an inner constructor to constrain the values that we're allowed to put in it (e.g. only `"length"`, `"mass"`, `"time"`, etc)
- Since this type only uses multiplicative factors, it won't work for unit conversion between degrees Celsius and degrees Fahrenheit. You could alter the `Unit` type to account for such an offset
- Using multiple dispatch (see Chapter 7), we can write our own methods for inbuilt Julia functions. For example, a method for `show` can make the displayed output nicer, a method for `*` would allow clean syntax like `inch = 2.54 * centimetre`, and a method for `+` could allow quantities like 2 years and 5 months
- Instead of using types, we could use a different structure to represent units. If you've read Chapter 9, you may want to consider using a `Dict` for a similar purpose

7 Multiple dispatch

! Prerequisites

Before reading this chapter, you are recommended to have read Chapter 2 and Chapter 6

- Type graph
- Types/functions recap
- Methods (with input type specification)
- Promotion
- Adding methods to inbuilt functions

8 Using packages

❗ Prerequisites

Before reading this chapter, you are recommended to have read Chapter 2

9 Collections of values

❗ Prerequisites

Before reading this chapter, you are recommended to have read Chapter 2, Chapter 5, and Chapter 6.

While we've seen how to perform many computations so far, the amount of time and effort it takes to get them done is often more than if we'd just sat down and worked them out by hand. This is because computers (and Julia in particular) are not very well suited to doing a single complicated calculation, as the instructions we'd have to give it would be as long as the calculation itself. What they're much better at is anything that boils down to doing the same simple thing repeatedly, which we can express in much shorter terms.

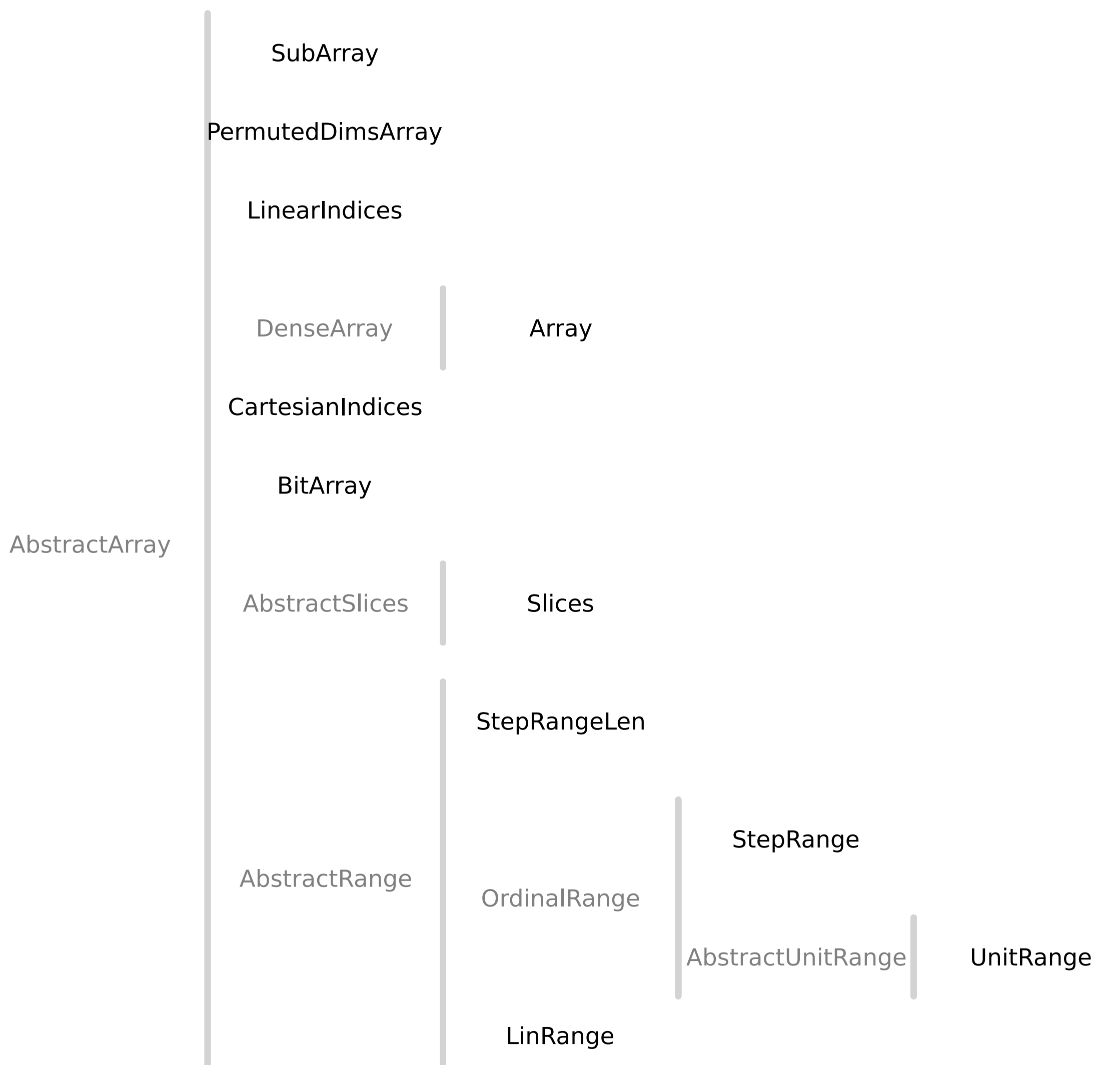
ℹ Note

This is an example of *Kolmogorov complexity*, which defines the complexity of an algorithm as the length of the shortest possible program that could implement it. In general, we'd consider shorter to be better, as it should take us less time to program.

This may sound quite limiting, but in fact it isn't really, in fact it often serves as more of a benefit for streamlining some or all of the problem we're trying to solve rather than a detriment where such repetition is infeasible. We've already met some of the tools needed to achieve this, namely loops from Chapter 5 and custom functions from Chapter 6. We'll now look at how we can store this data, into several different types that we'll term *collections*, and some tricks to operating on our data that are better optimised than the loops and functions that we may write. In particular, we'll look at some of the subtypes of `AbstractArray`, all of which perform this purpose.

```julia
# Useful for convenient broadcasting in the arrange function
import Base.+
+(x::Any,::Nothing) = x

# Arranges the subtype tree of T with positions of each of the subtypes
# Format of output is a vector of tuples of the form:
#     (type, number of nodes below, depth, position from top)
```

```
function arrange(T::Type)

    # Deals with small cases, where a type is not defined in Base we ignore it
    isdefined(Base, Symbol(T)) || return Tuple{Type, Int64, Int64, Rational{Int64}}[]
    isabstracttype(T) || return [(T, 1, 1, 1//1)]

    subT = subtypes(T)

    typelist = Tuple{Type, Int64, Int64, Rational{Int64}}[]
    offset = 0
    for S   subT

        # Prevents an infinite loop
        S == Any && continue
        # Recursively gets the arrangement of each subtype
        Stypelist = arrange(S)

        # Alters the arrangement to fit with the new graph
        append!(typelist,
            [st .+ (nothing, nothing, 1, offset) for st   Stypelist]
        )
        # Recalculates the offset to accommodate for the new points added
        isempty(Stypelist) || (offset += (Stypelist[end])[2] + 1)

    end
    # Adds the tuple for T itself to the end of the vector, and returns
    push!(typelist, (T, offset, 1, offset//2))

end

using Plots
# Plots the type tree for the subtypes of T
function typetree(T::Type)

    typelist = arrange(T)

    types::Vector{Type}                  = [st[1] for st   typelist]
    spans::Vector{Int64}                 = [st[2] for st   typelist]
    depths::Vector{Int64}                = [st[3] for st   typelist]
    vpositions::Vector{Rational{Int64}} = [st[4] for st   typelist]
    n = length(typelist)
    xgap = 750 ÷ max(depths...)
    ygap = 750 ÷ spans[end]

    plot(
        size = (750, 750),
        grid = false,
        showaxis = false,
        legend = false,
        xlims = (xgap - 50, 750 + 50),
        ylims = (ygap - 20, 750 + 20)
    )

    # Abstract types in grey, concrete types in black
    plot!(
        annotations = [
            (depths[i] * xgap, vpositions[i] * ygap,
            (types[i], 10, isabstracttype(types[i]) ? :grey : :black), "sans-serif")
            for i   1:n
        ]
    )

    lines = [
        ([(depths[i] + 1//2) * xgap, (depths[i] + 1//2) * xgap],
        [(vpositions[i] - spans[i]//2) * ygap, (vpositions[i] + spans[i]//2) * ygap])
        for i   1:n if isabstracttype(types[i])
    ]

    # Lines delineate the set of subtypes for each abstract type
    plot!(
        [Shape(line...) for line   lines],
        fillcolor = :white,
        linecolor = :lightgrey,
        linewidth = 4
    )

end

typetree(AbstractArray)
```



We'll also demonstrate our first example of computing something that would be slow and difficult to do by hand, by looking at Julia sets.

9.1 `Tuples`

The first type we'll look at doesn't actually appear on this diagram due to the way it is implemented, although it behaves similarly to many that do, and this is the `Tuple`. A `Tuple` brings several values together under one variable name, each of which can be individually retrieved if needed. We do this by separating the values with commas, and enclosing the whole collection in parentheses:

```julia
sthings = ("sassy", 6, Char(128013), typeof("s"))
```

```
("sassy", 6, ' ', String)
```

```julia
tthings = ("tatty", 2, Char(128047), typeof(sthings))
```

```
("tatty", 2, ' ', Tuple{String, Int64, Char, DataType})
```

We query a particular element of a `Tuple` by following the variable name (or indeed a `Tuple` literal) with square brackets enclosing the position of the element we want as a number (the *index* of the element):

```julia
(2, 4, 6, 8)[3]
```

```
6
```

Caution

In many programming languages, tuples or similar structures are indexed starting at `0` (so the equivalent of `x[0]` would return the first element of `x`). Julia takes the other approach, and starts indexing from `1` (so `x[1]` gives the first element), which tends to be more natural for beginners anyway.

Multiple elements can be returned at once, by indexing with some of the types we'll meet later, such as using the `Vector` of `Int64`s `[1, 4]` to get the first and fourth elements, or the `UnitRange` `1:4` to get the first four elements.

```julia
sthings[[2,4]]
```

```
(6, String)
```

```julia
tthings[1:3]
```

```
("tatty", 2, ' ')
```

The index `:` returns all elements, while `end` returns the last one (and `end-1` the second last, etc.).

```julia
sthings[:]
```

```
("sassy", 6, ' ', String)
```

```julia
tthings[end-2]
```

```
2
```

Also, there are functions that will return elements of a `Tuple`, such as `first` and `last`, which do exactly as you'd expect:

```julia
first(sthings)
```

```
"sassy"
```

```julia
last(sthings)
```

```
String
```

The function `length` gives the number of elements in a `Tuple`:

```julia
length(sthings)
```

```
4
```

Additionally, we can search through a `Tuple` with functions like `findall`, `findmin`, `findnext`. We'll demonstrate `findall`, which applys a function to each element of the `Tuple`, and returns a `Vector` of indices where the function returns `true`:

```julia
findall(x -> x isa String, tthings)
```

```
1-element Vector{Int64}:
 1
```

If the elements all have the same type, other function may be used. For example, a `Tuple` consisting solely of `Bool`s can be used with the functions `any` and `all`, which act much like `||` and `&&` respectively:

```julia
any((4 isa Integer, "4" isa Integer, :four isa Integer))
```

```
true
```

```julia
all((4 isa Integer, "4" isa Integer, :four isa Integer))
```

```
false
```

Many other such functions exist, and all of these functions, as well as the means of indexing, will work on any other ordered collection such as those we describe below.

9.2 `Arrays`

An `Array` is a collection of many values of a given type (or subtypes thereof). These can be arranged all in a line (like a `Tuple`), or indexed in multiple dimensions, such as a two-dimensional table, or a three-dimensional set of coordinates in 3D space. `Array` is a parametric type, with parameters of the type of the elements, and the number of dimensions. For example, an `Array` could exist with type `Array{Int64, 2}`, which would mean that it consists of `Int64`s arranged in two dimensions.

Two "types" that don't appear in the diagram above are `Vector` and `Matrix`, which are actually just aliases for varieties of `Array`: `Vector{T}` is a one-dimensional `Array` of type `T`, and `Matrix{T}` is a two-dimensional `Array` of type `T`. Most `Arrays` that we define will end up being one of these, unless more dimensions are specifically necessary.

9.2.1 Defining and indexing `Arrays`

To define a `Vector` (i.e. a one-dimensional `Array`), we seperate the elements we want with either commas or semi-colons:

```julia
numbers = [1, 5, 3, 8, -6]
```

```
5-element Vector{Int64}:
  1
  5
  3
  8
 -6
```

```
countries = ["India"; "Sweden"; "Brazil"; "Ghana"]
```

```
4-element Vector{String}:
 "India"
 "Sweden"
 "Brazil"
 "Ghana"
```

The type of a `Vector` can be specified by preceding the square brackets with the type name. This can construct `Vectors` with zero length, such as:

```
v = Int64[]
```

```
Int64[]
```

`Vectors` can be indexed just like `Tuples`, but using the same syntax, we can also change a particular element, treating it as we would a variable:

```
numbers[5] = 6
numbers
```

```
5-element Vector{Int64}:
 1
 5
 3
 8
 6
```

Note

The reason that we can do this is that, unlike `Tuples`, `Arrays` are *mutable*, allowing part of the data to be changed without having to rewrite the whole lot. For reasons of efficiency, this is not true for most types in Julia. Mutable types have some other different behaviour from normal immutable types, but we won't discuss it here.

To define a `Matrix`, we can write its columns out as we would `Vector`s, and then concatenate them horizontally with spaces inside a new set of square brackets:

```
morenumbers = [[1, 2, 3] [-1, -2, -3]]
```

```
3×2 Matrix{Int64}:
 1  -1
 2  -2
 3  -3
```

Alternatively, we can use double semi-colons to mark a new column:

```
morecountries = ["Peru"; "Colombia";; "Egypt"; "Zambia"]
```

```
2×2 Matrix{String}:
 "Peru"      "Egypt"
 "Colombia"  "Zambia"
```

Finally, we may use a combination of spaces and line breaks to construct a matrix, with spaces separating elements in the same row, and line breaks separating rows from each other:

```
I = [ 1 0 0
      0 1 0
      0 0 1 ]
```

```
3×3 Matrix{Int64}:
 1  0  0
 0  1  0
 0  0  1
```

To index elements of a `Matrix`, we use two numbers, the first for the row number, the second for the column number, separated by a comma:

```
morecountries[2,1]
```

```
"Colombia"
```

It is also possible to use single numbers as indices for a `Matrix`, the order being to go down the columns first (so-called *column-major* ordering). This can be seen through the `LinearIndices` type, which gives each the numerical index of each element of the `Matrix` used in its construction:

```julia
LinearIndices(I)
```

```
3×3 LinearIndices{2, Tuple{Base.OneTo{Int64}, Base.OneTo{Int64}}}:
 1  4  7
 2  5  8
 3  6  9
```

Similar principles apply to higher dimensions, in particular the use of semi-colons to denote dimension number to concatenate along, and the means of indexing the elements.

An `Array` specific relative of `length` is the `size` function, which returns a `Tuple` of the number of elements along each dimension of the `Array` (column-major ordering meaning that the number of elements in each column comes first):

```julia
size(morenumbers)
```

```
(3, 2)
```

9.2.2 Construction of arrays by functions

Another way we can construct an `Array` is by the use of a function which returns an `Array`. To get a `Vector` of type `T` and length `n`, we can use the `Vector{T}` constructor, with arguments `undef` (which initialises an `Array` with arbitrary values that we can replace later) and the desired length `n`:

```julia
Vector{String}(undef, 3)
```

```
3-element Vector{String}:
 #undef
 #undef
 #undef
```

We can do the same with a `Matrix`, specifying two dimensions, or a general `Array`, specifying as many dimensions as we need:

```julia
Matrix{ComplexF64}(undef, 2, 4)
```

```
2×4 Matrix{ComplexF64}:
 1.02183e-311+1.02183e-311im  …  0.0+0.0im  0.0+0.0im
 1.02183e-311+1.02183e-311im     0.0+0.0im  0.0+0.0im
```

```julia
Array{Bool}(undef, 2, 2, 2)
```

```
2×2×2 Array{Bool, 3}:
[:, :, 1] =
 0  1
 1  0

[:, :, 2] =
 1  0
 1  0
```

We can also initialise `Array`s with specified values, with functions such as `fill`, `zeros`, `ones`:

```julia
fill("ostrich", 2, 2)
```

```
2×2 Matrix{String}:
 "ostrich"  "ostrich"
 "ostrich"  "ostrich"
```

```julia
zeros(Rational{Int64}, 3)
```

```
3-element Vector{Rational{Int64}}:
 0//1
 0//1
 0//1
```

The `rand` function fills an `Array` of the given dimensions with (as best it can) uniformly random `Float64`s between `0` and `1`:

```julia
rand(4,3)
```

```
4×3 Matrix{Float64}:
 0.108065  0.868455  0.995519
 0.165468  0.078252  0.0696596
 0.753003  0.62981   0.065973
 0.655447  0.258828  0.325023
```

It can also randomly choose from a collection:

```
rand(1:100, 6)
```

```
6-element Vector{Int64}:
 53
 27
 51
 75
 62
 35
```

The `collect` function allows an `Array` to be created with the values of another sort of collection. We can see this demonstrated with a `UnitRange` and a `String`, which we discuss more later:

```
collect(1:5)
```

```
5-element Vector{Int64}:
 1
 2
 3
 4
 5
```

```
collect("mongoose")
```

```
8-element Vector{Char}:
 'm': ASCII/Unicode U+006D (category Ll: Letter, lowercase)
 'o': ASCII/Unicode U+006F (category Ll: Letter, lowercase)
 'n': ASCII/Unicode U+006E (category Ll: Letter, lowercase)
 'g': ASCII/Unicode U+0067 (category Ll: Letter, lowercase)
 'o': ASCII/Unicode U+006F (category Ll: Letter, lowercase)
 'o': ASCII/Unicode U+006F (category Ll: Letter, lowercase)
 's': ASCII/Unicode U+0073 (category Ll: Letter, lowercase)
 'e': ASCII/Unicode U+0065 (category Ll: Letter, lowercase)
```

9.2.3 Adding and removing elements from a `Vector`

The mutability of `Vector`s allows another operation that `Tuple`s can't do, which is to add onto and take away elements from `Vector`s that we've already got. Let's start with an empty `Vector{Int64}`;

```julia
v = Int64[]
```

```
Int64[]
```

The `append!` function add a given element or elements to the end of a `Vector`:

```julia
append!(v, 6)
```

```
1-element Vector{Int64}:
 6
```

```julia
append!(v, [4, 2])
```

```
3-element Vector{Int64}:
 6
 4
 2
```

The `prepend!` function does the same at the start:

```julia
prepend!(v, 3)
```

```
4-element Vector{Int64}:
 3
 6
 4
 2
```

The functions `push!` and `pushfirst!` work similarly to `append!` and `prepend!` respectively. To insert an element at a specific index, we can use `insert!`:

```julia
insert!(v, 4, -1) # First number is the index, second is the value to insert
```

```
5-element Vector{Int64}:
  3
  6
  4
 -1
  2
```

Convention

Functions on mutable inputs which mutate one of the inputs (usually the first) are named with an `!` at the end. For example, the functions above mutate the input `v`, changing the value of the variable and not just calculating an output.

To remove elements, we can use `pop!`. This gets returns the last element of the `Vector`, but also gets rid of it:

```
pop!(v)
```

```
2
```

```
v
```

```
4-element Vector{Int64}:
  3
  6
  4
 -1
```

Other elements can be removed by `popfirst!` and `popat!`:

```
popat!(v, 3)
```

```
4
```

```
v
```

```
3-element Vector{Int64}:
  3
  6
 -1
```

The function `splice!` combines `popat!` and `insert!`, allowing us to remove an element and replace it with one or more new ones at the same time:

```
splice!(v, 2, [2, 1, 0])
```

```
6
```

```
v
```

```
5-element Vector{Int64}:
  3
  2
  1
  0
 -1
```

For higher dimensional `Array`s, such as a `Matrix`, this is not as simple (for example you can't just remove the last element). In general, for reasons of cleanness of code and efficiency, `Array`s should be initialised as the size that they need to be to begin with, having their values filled in later, but sometimes you won't be able to do this, in which case the mutating functions are your best option.

9.2.4 Array comprehension

There's a third way of filling `Array`s with even more control over the initial values, which is *array comprehension*. This acts much like a `for` loop, indeed it uses much of the same syntax, filling an `Array` with some function of the elements in a collection. Instead of writing the elements of the `Array` between the square brackets, we write a rule for generating them:

```
[i^2 for i   1:10]
```

```
10-element Vector{Int64}:
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

```
[first(c) for c   countries]
```

```
4-element Vector{Char}:
 'I': ASCII/Unicode U+0049 (category Lu: Letter, uppercase)
 'S': ASCII/Unicode U+0053 (category Lu: Letter, uppercase)
 'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)
 'G': ASCII/Unicode U+0047 (category Lu: Letter, uppercase)
```

These give `Vectors`, as the `for` loop iterates over one-dimensional collections. We can get a two-dimensional `Matrix` either by iterating over two one-dimensional collections:

```
[i*j for i   1:10, j   1:10]
```

```
10×10 Matrix{Int64}:
  1   2   3   4   5   6   7   8   9   10
  2   4   6   8  10  12  14  16  18   20
  3   6   9  12  15  18  21  24  27   30
  4   8  12  16  20  24  28  32  36   40
  5  10  15  20  25  30  35  40  45   50
  6  12  18  24  30  36  42  48  54   60
  7  14  21  28  35  42  49  56  63   70
  8  16  24  32  40  48  56  64  72   80
  9  18  27  36  45  54  63  72  81   90
 10  20  30  40  50  60  70  80  90  100
```

or by iterating over one two-dimensional collection, such as another matrix:

```
[first(c) for c   morecountries]
```

```
2×2 Matrix{Char}:
 'P'  'E'
 'C'  'Z'
```

Following this with `if` inside the square brackets allows us to skip elements which don't meet a condition:

```
[i for i   1:10 if iseven(i) && i < 7]
```

```
3-element Vector{Int64}:
 2
 4
 6
```

9.3 Other ordered collection types

9.3.1 Ranges

We've already met a type of range in Chapter 5, which used the syntax `1:10` to represent the numbers from `1` to `10` (or similar). This notation extends further:

- `a:b` means the numbers `a`, `a+1`, `a+2`, continuing to add `1` until we would exceed `b` by adding `1` more
- `a:d:b` means the same, but the gap between successive numbers is now `d` instead of `1`

It is also possible to specify ranges using the `range` function, which allows four keyword parameters to be set, `start`, `stop`, `step`, and `length`. Only three of these are necessary (indeed you get an error if you try to specify all four), and they allow for options of evenly spaced ranges, including using `Complex` numbers:

```
range(start = 1, step = 1im, length = 5)
```

```
1 + 0im:0 + 1im:1 + 4im
```

Warning

Despite how this is displayed, `1 + 0im:0 + 1im:1 + 4im` will not work if you try to type it out, because it tries to call `range` with the `start`, `stop`, and `step` keywords. For `Complex` values, where a notion of `<` doesn't exist, having a stopping point is difficult to work with, so `length` should be specified instead of `stop`.

The exact type of the range returned depends on the values you start with, but any parametric subtypes of `UnitRange`, `StepRange`, or `StepRangeLen` are possible.

```
typeof(1:6)
```

```
UnitRange{Int64}
```

```julia
typeof(1:2:7)
```

```
StepRange{Int64, Int64}
```

```julia
typeof(range(start = 1, step = 1im, length = 5))
```

```
StepRangeLen{Complex{Int64}, Int64, Complex{Int64}, Int64}
```

The remaining subtype of `AbstractRange` is `LinRange`, which is a performance optimised but error prone alternative to specifying a range with `start`, `stop`, and `length`.

9.3.2 `CartesianIndices`

A `CartesianIndex` is a type that acts much like a `Tuple`, but is specifically meant as an index for an `Array`.

```julia
ci = CartesianIndex(1,2)
```

```
CartesianIndex(1, 2)
```

Indeed, we can use a `CartesianIndex` as an index:

```julia
A = [ 1 3
      2 4 ]

A[ci]
```

```
3
```

```julia
A[1,2]
```

```
3
```

Despite being `Tuple`-like, `CartesianIndex` is not considered a collection of items (there's a specific error if you try to index one of them). However, a collection of all possible `CartesianIndex` indices of an `Array` does exist as its own type, `CartesianIndices`, which is a subtype of `AbstractArray`, and so does have the expected indexing behaviour. We can either give it an `Array`:

```
indicesA = CartesianIndices(A)
```

```
CartesianIndices((2, 2))
```

```
collect(indicesA)
```

```
2×2 Matrix{CartesianIndex{2}}:
 CartesianIndex(1, 1)  CartesianIndex(1, 2)
 CartesianIndex(2, 1)  CartesianIndex(2, 2)
```

or a `Tuple` of ranges, to form the grid of possible indices:

```
rangeindices = CartesianIndices((1:2:5, 3:-1:0))
```

```
CartesianIndices((1:2:5, 3:-1:0))
```

```
collect(rangeindices)
```

```
3×4 Matrix{CartesianIndex{2}}:
 CartesianIndex(1, 3)  CartesianIndex(1, 2)  …  CartesianIndex(1, 0)
 CartesianIndex(3, 3)  CartesianIndex(3, 2)     CartesianIndex(3, 0)
 CartesianIndex(5, 3)  CartesianIndex(5, 2)     CartesianIndex(5, 0)
```

9.3.3 `Strings`

Strings are again not a subtype of `AbstractArray`, and for good reason, they perform a wildly different function most of the time (see Chapter 4), and don't work the same with functions as the other types here do. However, it is worth mentioning them, as they exhibit similar indexing properties as we've seen before, and in this way can sometimes behave as if they were a `Tuple` consisting of `Chars`:

```julia
"daffodil"[3:4]
```

```
"ff"
```

```julia
length("daffodil")
```

```
8
```

9.4 Unordered collection types

All of the above types we've encountered are ordered, in the sense that it's meaningful to ask what the first element is, what the second is, etc. However, sometimes we don't even need this information, and Julia provides types that ignore this for the sake of efficiency.

9.4.1 `Set`

A set is a collection of values, much like a `Vector` or a `Tuple`, but without an order, and without duplicates. We can construct a set by giving our collection of values in one of the other formats we've seen above:

```julia
numbersset = Set(1:6)
```

```
Set{Int64} with 6 elements:
  5
  4
  6
  2
  3
  1
```

```julia
magicset = Set("abracadabra")
```

```
Set{Char} with 5 elements:
  'a'
  'c'
  'd'
  'r'
  'b'
```

Although these have an order of sorts when displayed, this is merely for the purpose of displaying them, and indexing doesn't work.

```
numbersset[1]
```

```
LoadError: MethodError: no method matching getindex(::Set{Int64}, ::Int64)
```

This doesn't make them useless, however. Instead, they serve a different purpose, being optimised for searching and set operations. Searching can be done with the `in` function, returning `true` or `false` depending on whether the queried value is found in the `Set`:

```
in(2, numbersset)
```

```
true
```

The `in` function can also be written inline, or replaced by its alias   (written by tab-completing `\in`):

```
7 in numbersset
```

```
false
```

```
'b'   magicset
```

```
true
```

Set operations are also defined, such as `union`/ (`A   B` is the `Set` containing all elements that appear in `A` or `B`, or both), `intersect`/ (`A   B` is the `Set` containing all elements that appear in `A` and in `B`), and `setdiff` (`setdiff(A,B)` is the `Set` containing all elements of `A` that don't appear in `B`).

```
numbersset   Set([10, 11, 12])
```

```
Set{Int64} with 9 elements:
  5
  4
  6
  2
  11
  10
  12
  3
  1
```

```
magicset   Set("abcdef")
```

```
Set{Char} with 4 elements:
  'a'
  'c'
  'd'
  'b'
```

```
setdiff(magicset, "abcdef")
```

```
Set{Char} with 1 element:
  'r'
```

These functions also work for ordered collections such as `Tuple`s or `Vector`s, but on larger scales are quicker with `Set`s.

9.4.2 `Dict`

A `Dict` (short for dictionary) is a little like a `Tuple` or a `Vector`, but the indices can be whatever we choose. This again means that there isn't necessarily any order to a `Dict`, since there isn't necessarily an obvious order to the indices. There are several ways to initialise a `Dict`, but we'll start with an empty one, `Dict()`. Before we run this, we may wish to choose the types that will be used for the *keys* (that is, the indices), and the *values* (the data which we store for each key). These go in curly brackets like a parametric type (indeed, that's what they are!), for example below, we have chosen `Char`s as keys, and `Int64`s as values:

```
frequencies = Dict{Char, Int64}()
```

```
Dict{Char, Int64}()
```

Alternatively, we can give a list of initial values, as a `Vector` of `Tuple`s, with each tuple containing a key followed by the desired value:

```
capitals = Dict([("France", "Paris"), ("Nigeria", "Abuja"), ("Chile", "Santiago")])
```

```
Dict{String, String} with 3 entries:
  "Nigeria" => "Abuja"
  "Chile"   => "Santiago"
  "France"  => "Paris"
```

Finally, we can do the same with a sequence of `Pairs`. `Pair` is a type storing exactly two values, written with `=>` between them, and they can be used to construct a `Dict` by putting them as inputs into the `Dict` constructor, each following the format `key => value`:

```
shades = Dict(
    "red" => ["scarlet", "blood", "ruby"],
    "green" => ["lime", "forest", "british racing"],
    "blue" => ["sky", "ocean", "navy"]
)
```

```
Dict{String, Vector{String}} with 3 entries:
  "blue"  => ["sky", "ocean", "navy"]
  "green" => ["lime", "forest", "british racing"]
  "red"   => ["scarlet", "blood", "ruby"]
```

To add values to a `Dict`, we can use the indexing syntax that we've seen before. If the key doesn't exist, we'll get an error when trying to query a value, but setting a value works for any key, and will either add or overwrite that value into the `Dict`.

```
lipsum = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor i

for c   lipsum
    frequencies[c] = haskey(frequencies, c) ? frequencies[c] + 1 : 1
end

frequencies
```

```
Dict{Char, Int64} with 28 entries:
  'n' => 24
  'f' => 3
  'd' => 18
  'E' => 1
  'e' => 37
  'o' => 29
  'D' => 1
  'h' => 1
  's' => 18
  'i' => 42
  'r' => 22
  't' => 32
  ',' => 4
  'q' => 5
  ' ' => 68
  'a' => 29
  'c' => 16
  'p' => 11
  'm' => 17
  '.' => 4
  'U' => 1
  'L' => 1
  'g' => 3
  'v' => 3
  'u' => 28
     =>
```

We can list out the keys and the values, although as before, the order will be meaningless:

```
keys(capitals)
```

```
KeySet for a Dict{String, String} with 3 entries. Keys:
  "Nigeria"
  "Chile"
  "France"
```

```
values(capitals)
```

```
ValueIterator for a Dict{String, String} with 3 entries. Values:
  "Abuja"
  "Santiago"
  "Paris"
```

We can also delete keys (and the values that go with them) with `pop!` or `delete!`. These have the same effect on the `Dict`, but produce different outputs, `pop!` returns the value corresponding to the deleted key, while `delete!` returns the new `Dict`:

```
pop!(shades, "green")
```

```
3-element Vector{String}:
 "lime"
 "forest"
 "british racing"
```

```
delete!(shades, "blue")
```

```
Dict{String, Vector{String}} with 1 entry:
  "red" => ["scarlet", "blood", "ruby"]
```

`Dict`s are an efficient way to store values under meaningful names without having to define a large number of variables. For this reason, one of their most common usages is as a lookup table, where a common calculation can be precomputed for the possible values that it could take, and then its answer looked up instead of doing the calculation all over again every time. For example, instead of counting the number of times a particular character appears in the text `lipsum`, we can look up the count in `frequencies`:

```
frequencies['s']
```

```
18
```

```
count(==('s'), lipsum) # ==('s') returns the anonymous function x -> x == 's'
```

```
18
```

9.5 Interaction with functions

These types are already quite useful in storing data in their own different ways, but where they really come in handy is using them in combination with functions. One example of this is that the standard linear algebra rules for multiplying matrices by scalars, vectors, or other matrices, work seamlessly with the usual operations like `*`, `/`, `^`, and the `Vector` and `Matrix` types:

```
A = [ 2 3
      5 1 ]

2A
```

```
2×2 Matrix{Int64}:
  4  6
 10  2
```

```
x = [-1,4]

A*x
```

```
2-element Vector{Int64}:
 10
 -1
```

More widely applicable are the `.`, `...`, and `...` operations (those last two are different!) that allow for functions to act on collections, and collections to be input into functions. These are named broadcasting, splatting, and slurping.

9.5.1 Broadcasting

When introducing the many different types above, we've seen some examples of functions which take collections as an input. However, often we don't want to apply a function to the collection as a whole, instead we want to apply it to each element individually. This is achieved by *broadcasting* a function.

The `broadcast` function provides the first way of doing this. Its first argument is the function that we want to broadcast, and the rest of the inputs will serve as the arguments to that function. For example, compare the following:

```
A * A
```

```
2×2 Matrix{Int64}:
 19   9
 15  16
```

```
B = broadcast(*, A, A) # B[i,j] = A[i,j] * A[i,j]
```

```
2×2 Matrix{Int64}:
  4  9
 25  1
```

Broadcasting will break if the sizes of the inputs don't match. However, if the sizes do match, but some dimensions are missing, Julia will smartly account for the missing dimensions and index the inputs accordingly:

```
C = broadcast(*, A, x) # C[i,j] = A[i,j] * x[i]
```

```
2×2 Matrix{Int64}:
 -2  -3
 20   4
```

```
D = broadcast(^, A, 2) # D[i,j] = A[i,j] ^ 2
```

```
2×2 Matrix{Int64}:
  4  9
 25  1
```

There are two further ways of broadcasting, with shorter syntax. If a function has a name (i.e. isn't anonymous), following its name with a dot `.` has the effect of broadcasting:

```
sinpi.(0:0.2:2)
```

```
11-element Vector{Float64}:
  0.0
  0.5877852522924731
  0.9510565162951536
  0.9510565162951536
  0.587785252292473
  0.0
 -0.587785252292473
 -0.9510565162951535
 -0.9510565162951535
 -0.587785252292473
  0.0
```

Most infix operators support a similar syntax, with the `.` coming before the operator this time:

```
A .* A
```

```
2×2 Matrix{Int64}:
  4  9
 25  1
```

Also, we can start the line with `@.` (which calls the macro `@.`), and this broadcasts all of the functions in that line, without us having to specify each individually:

```
@. sin(2^(-0.5:0.2:0.5) + 0.4)
```

```
6-element Vector{Float64}:
 0.8944084355923738
 0.9364087671697013
 0.9718672058520753
 0.99510124105
 0.9981796067983775
 0.9705200191672874
```

Convention

Note that while array comprehension can achieve the same outcome, broadcasting tends to be more efficient. For example, the `@.` broadcast above could be equally calculated as the following:

```
[sin(2^t + 0.4) for t   -0.5:0.2:0.5]
```

```
6-element Vector{Float64}:
 0.8944084355923738
 0.9364087671697013
 0.9718672058520753
 0.99510124105
 0.9981796067983775
 0.9705200191672874
```

but this is noticeably slower for operations on larger collections.

9.5.2 Splatting and slurping

Another way we may want to apply a function to an collection is to have the elements serve as individual arguments to the same function call. We do this by *splatting*, effectively emptying the elements in the order that they are indexed as arguments.

Using the vector `x = [-1, 4]` above, we can splat this into the `-` function to subtract the second value from the first.

```
-(x...)
```

```
-5
```

This isn't particularly useful at the moment, although it could be situationally handy. Splatting tends to be more useful for certain functions that take an arbitrary number of inputs, such as `+`:

```
+(A...)
```

```
11
```

How do we write a function that has arbitrarily many inputs then? The answer is the visually identical but conceptually distinct syntax of *slurping*. If the last argument in the function definition is followed by `...`, then it will collect together any further inputs into the function into a `Tuple` with that variable name.

```
inputcounter(args...) = length(args)
```

```
inputcounter (generic function with 1 method)
```

```
inputcounter(4, 12, "Friday", :cos)
```

```
4
```

We can see how splatting and slurping can work together, first emptying the values into the function call, and then collecting them back up into a `Tuple`:

```
inputcounter(A)
```

```
1
```

```julia
inputcounter(A...)
```

```
4
```

9.6 Example: Julia sets

Since we're working in Julia, it would be remiss not to use it to generate its namesake mathematical objects: Julia sets (technically filled Julia sets). These are fractal sets, composed of the points where iterating some function again and again does not lead you off towards infinity. They are related to the more famous Mandelbrot set, but their name happens to be shared with a programming language instead of a type of biscuit. They will also serve as a useful way to demonstrate some of the `Array` operations seen above. Before we start programming, we'll look at the definition, so we know what to draw.

9.6.1 Mathematical definition of the Julia set

First, we need a function to iterate. We will use the same functions as used by the Mandelbrot set, which have one complex parameter c, and are defined by:

$$F_c(z) = z^2 + c$$

The (filled) Julia set with parameter c is then the set of points in the complex plane where repeated application of F_c is bounded (i.e. the magnitude of the answers doesn't go off to infinity):

$$\mathcal{J}(c) = \left\{ z \in \mathbb{C} : |(F_c)^n(z)| \not\to \infty \text{ as } n \to \infty \right\}$$

Here, the power of n on the function F_c means applying that function repeatedly n times (so $F_c(F_c(\ldots(F_c(z)))))$.

As an example, we'll consider $\mathcal{J}(0)$. The function F_0 is simply $F_0(z) = z^2$, so:

$$(F_0)^n(z) = (((z^2)^2)\ldots)^2 = z^{2^n}$$

Therefore, $\mathcal{J}(0)$ is the set of points where z^{2^n} is bounded as $n \to \infty$, which is the unit disc, so $\mathcal{J}(0) = \{z \in \mathbb{C} : |z| \leqslant 1\}$.

To demonstrate how close this is to the Mandelbrot set, here is its definition:

$$\mathcal{M}(c) = \left\{ c \in \mathbb{C} : |(F_c)^n(0)| \not\to \infty \text{ as } n \to \infty \right\}$$

or equivalently:

$$\mathcal{M}(c) = \{c \in \mathbb{C} : 0 \in \mathcal{J}(c)\}$$

9.6.2 Drawing Julia sets

It's all well and good to have this definition, but as we saw in Chapter 3, translating mathematics to code is not straightforward, and in this case, we run into the same issues of having to represent infinite concepts finitely. In particular, there are two approximations that we have to make:

- We can't take a limit as $n \to \infty$ as we would like. Instead, we'll set a limit of the number of times we iterate, and assume that if $(F_c)^n(z)$ doesn't get too large in that many iterations, it never will. If this limit on n is large enough, this will be a decent approximation
- We can't check whether every point in the complex plane is or isn't in the Julia set (or even a finite area within it). Instead, we'll choose ranges of x and y values, which will give us a lattice of points $z = x + yi$ that we can check individually

For this purpose, we'll define two parameters to balance accuracy with speed of computation. `n` will be the maximum number of applications of our function before we consider it not to have diverged, and `h` will be the distance between successive `x` or `y`-values, and so controls how fine the lattice will be.

```
n = 1000
h = 0.005
```

```
0.005
```

Why do we define these values outside of our functions, instead of just hard-coding the values where they are needed in the function code? Because it's much easier to tweak and change them to get the outputs that we're looking for, all we need to do is redefine a variable, or change the inputs to a function call, rather than rewrite and recompile a function.

The iteration function is as simple as a `for` loop, using `z = z^2 + c` to have the effect of applying the function repeatedly. We stop the loop and return a result early if the absolute value of `z` is ever 2 or more (we can prove that when `abs(c) < 2`, if `abs(z)` is ever greater than 2, it will never be less than 2 again and will diverge to infinity). The returned value of the function could just be `true` or `false`, but we're actually returning the number of the iteration if it breaks early, which we'll use to make a more interesting visualisation.

```
function iterateF(z, c, n)
    for i   1:n
        z = z^2 + c
        abs(z) < 2 || return i
    end
    return 0
end
```

```
iterateF (generic function with 1 method)
```

Here, we've included `n` as an input, instead of using the variable `n` that we've defined, even though the value of `n` will end up being given as that input. This is good practice, as it's easier to change the inputs to a function than redefine a variable and then rerun a function. Also, if this function was in a different package or module, the variable `n` may not be in scope, so it wouldn't work anyway.

Convention

As a rule of thumb, the only variables that should be referred to within a function are the inputs, those defined in the function, and any `const` values that will be in scope.

Naturally, we want a visual output for the Julia set, and there are multiple ways of doing this, although all will require the use of one or more packages. The most elegant solution uses the `Colors` package, which includes the `RGB` type to represent colours in the familiar RGB format. The package displays `RGB` types not with text, but as a small image of that colour, and moreover a `Matrix` (i.e. a 2-dimensional `Array`) of `RGB`s is displayed as an image with pixels of those colours arranged as they are in the `Matrix`. As with all packages (see Chapter 8), this will first need to be installed using `Pkg`, then added to our toolkit with `using`.

```
using Colors
```

Now all we need is a function which can output an image. We begin by setting up a `Matrix` of values for `z ` (the initial values for `z`), which we'll call `z s`. The `Matrix` is constructed by array comprehension, using a range of `x` and `y` values between `-2` and `2` (as that is where the Julia set lies) graduated by the parameter `h` as described above. The ordering of the indexing sets needs to be carefully chosen so that the top left corner is `-2.0 + 2.0im`, the bottom left is `-2.0 - 2.0im` etc.

```
z s = [x  + y *im for y   2:-h:-2, x   -2:h:2]
```

We then want to run the function `iterateF` on each of these, which is simple enough to do by broadcasting using the `.` syntax. Since the other inputs `c` and `n` are not `Array`s, they will be the same for every time the function is run with a different value of `z`.

```
iterations = iterateF.(z s, c, n)
```

This gives us a new `Matrix` of values, which we want to interpret and turn into colours. The usual way of doing this for Julia sets is to colour the inside of the set in black, and colour the outside of the set as a gradient, with the colour depending on how long it took to get to `abs(z) > 2` and for the iteration to stop. Hence, we chose the output to `iterateF` that we did – the more iterations needed, the higher the output value, but if we run out of iterations, we return `0` to delineate a sharp boundary.

To define the gradient, we know the smallest value in the `Matrix iterations` will be `0` (which will be the black of the inside of the Julia set), but we need to know the largest value to pin the other end of the gradient to. The `max` function will do this, but it takes an arbitrary number of individual numeric inputs, not a single `Array` as we currently have, so we need to splat `iterations` to empty its values out into the `max` function. Since it is conceptually similar to the variable `n`, we'll overwrite that and call this new value `n` as well (although there's no real need to).

```
n = max(iterations...)
```

The `RGB` constructor we'll use takes three floating-point inputs between `0.0` and `1.0` as proportions of the maximum amount of red, green, and blue present in the colour. By dividing each entry of `iterations` by `n`, we'll get such a number, and raising these to various powers will change the scale on which the intermediate values lie, allowing us to give the gradient a coloured tint. For instance, sending the entry `i` to `RGB((i/n)^0.5, (i/n)^1, (i/n)^0.5)` gives a purple hue, as we'll see in the output below. To apply this to the whole `Matrix`, we'll using broadcasting again, this time with the `broadcast` function and an anonymous function to encode the operation.

```
broadcast(i -> RGB((i/n)^0.5, (i/n)^1, (i/n)^0.5), iterations)
```

Putting this all together, we have the function `juliaset`:

```
function juliaset(c, n, h)

    z s = [x  + y *im for y   2:-h:-2, x   -2:h:2]

    iterations = iterateF.(z s, c, n)

    n = max(iterations...)
    return broadcast(i -> RGB((i/n)^0.5, (i/n)^1, (i/n)^0.5), iterations)

end
```

```
juliaset (generic function with 1 method)
```

ℹ Note

Since the function `juliaset` always returns the same calculated value, the `return` keyword is redundant here, as functions will always return the last calculated value if no `return` is specified. It's included only for clarity.

Before we run this, we'll need another package, called `ImageShow`. While `Colors` allows for a `Matrix` of `RGB`s to be displayed as an image, it is limited to smaller matrices than we're using (our value of `h` being `H = 0.005` results in an `801` by `801` `Matrix`). If you were to run this, it suggests installing and using the `ImageShow` package to help with this, which is what we will do.

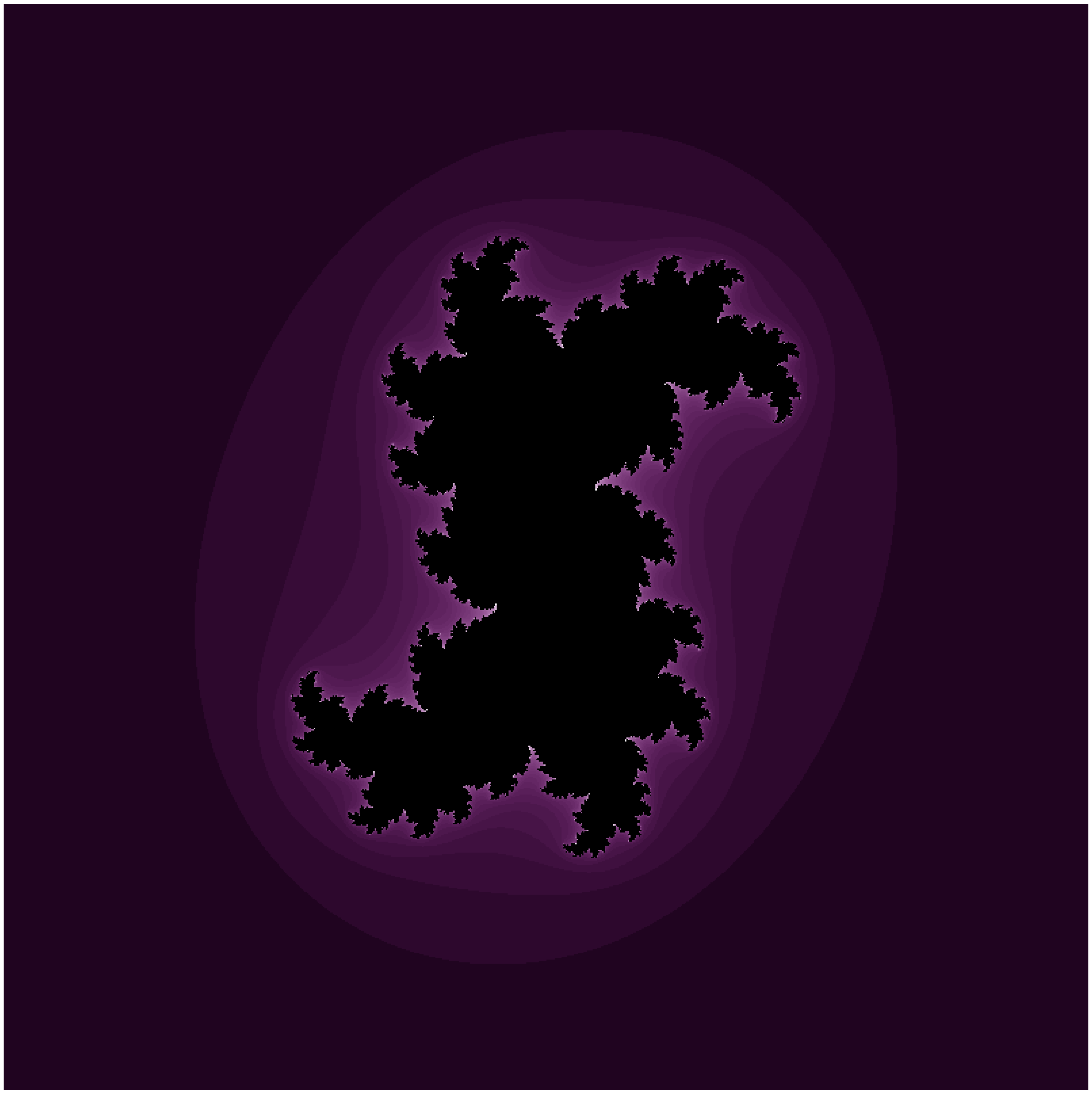
```
using ImageShow
```

⚠ Warning

In some environments (such as Pluto), the `ImageIO` package may be required as well. This can be added simply by installing through `Pkg` as usual, and then running `using ImageIO`.

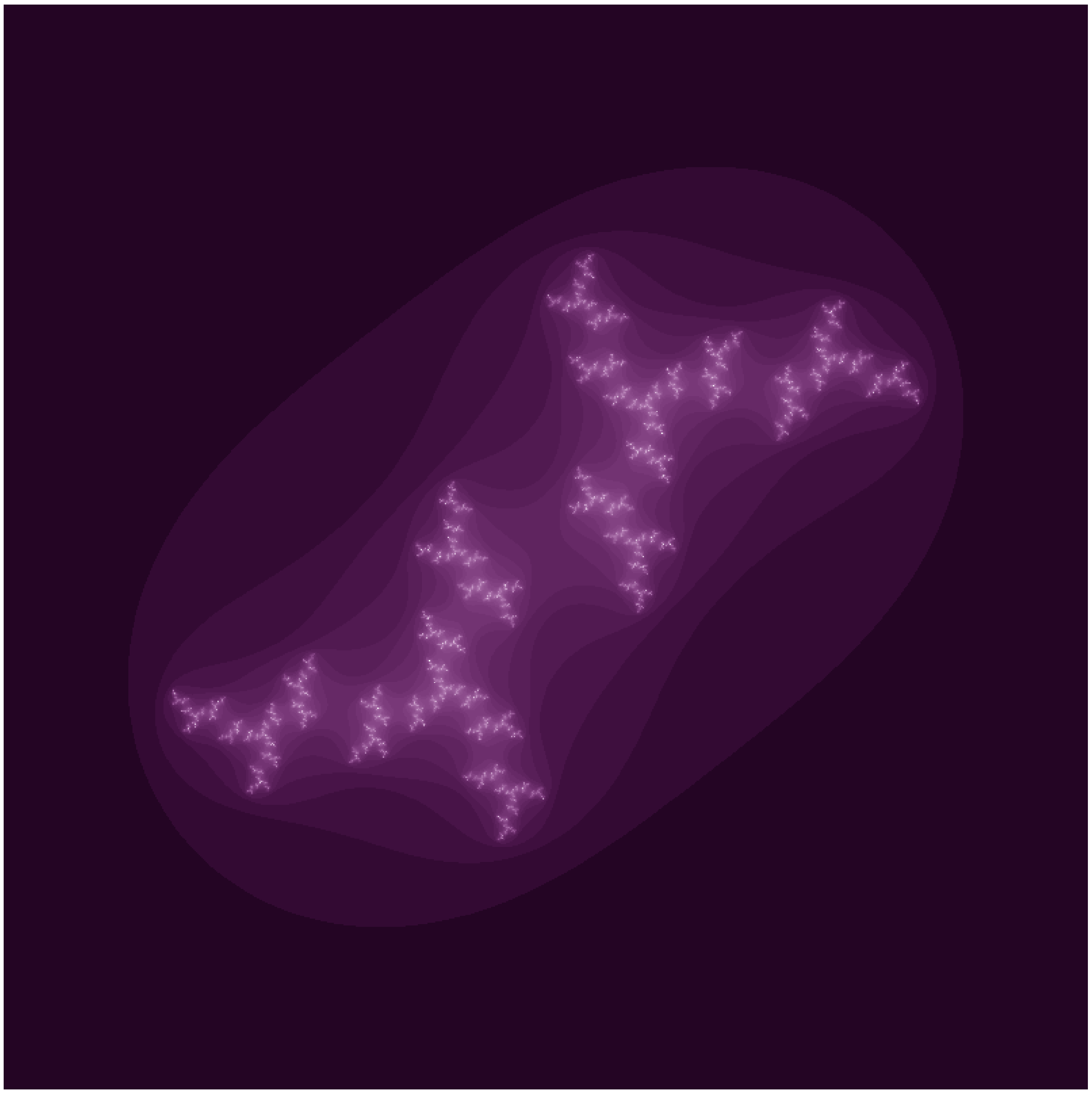
Now, we can run the function, and see what the Julia set looks like for some different values of `c`:

```
juliaset(0.3531 - 0.3165im, n, h)
```



```julia
juliaset(-0.2252 + 0.7049im, n, h)
```

```
juliaset(-0.2359 - 0.9173im, n, h)
```



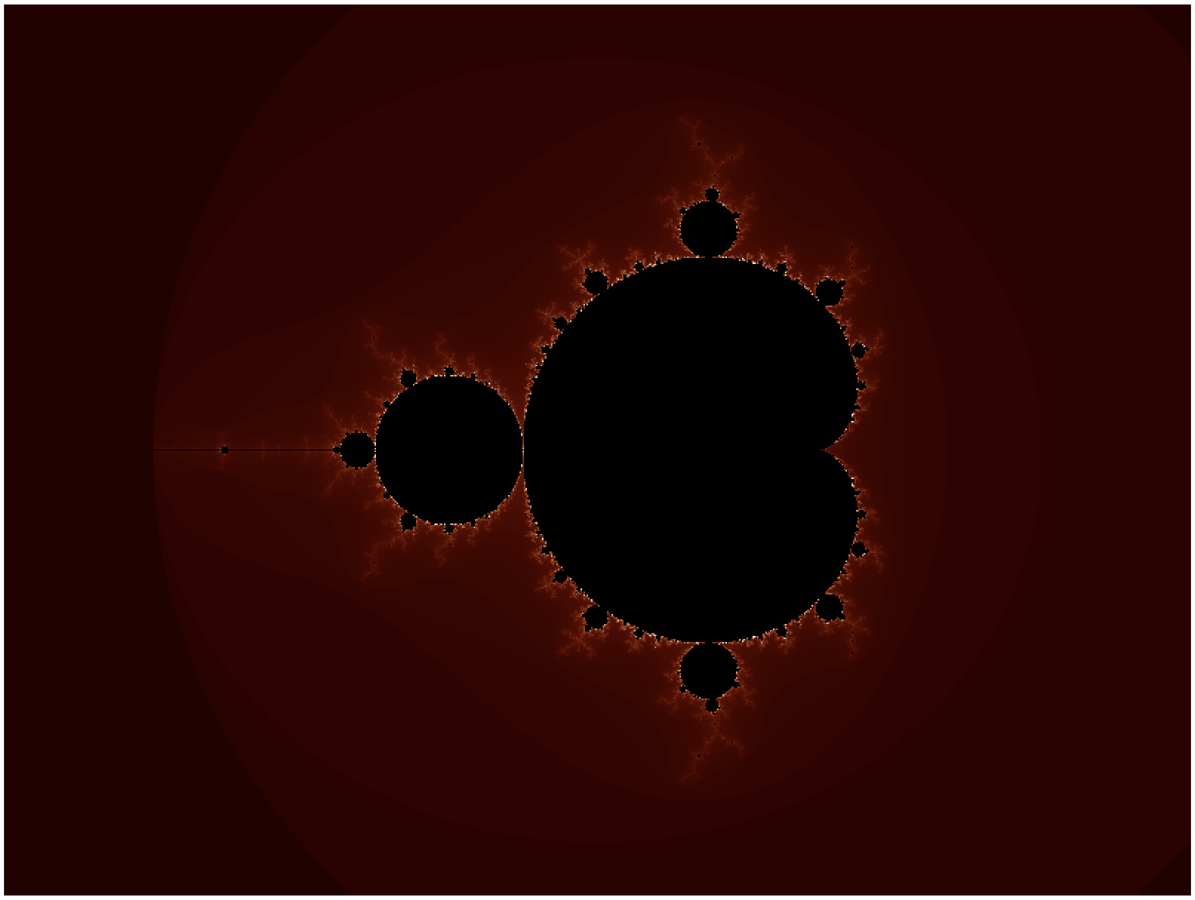
9.6.3 Drawing the Mandelbrot set

We noted earlier how similar the mathematical definitions for the filled Julia set and the Mandelbrot set are, and indeed the same is true to generate images of them. The only change we need to make is to start with a `Matrix` of values of `c`, not `z`, and change the inputs of `iterateF` accordingly. We also change the ranges of `x` and `y` values to better frame the Mandelbrot set, and tweak the colour function for a different shade:

```
function mandelbrotset(n, h)

    cs = [x + y*im for y   1.5:-h:-1.5, x   -2.5:h:1.5]

    iterations = iterateF.(0, cs, n)

    n = max(iterations...)
    colours = broadcast(i -> RGB((i/n)^0.3, (i/n)^0.7, (i/n)^1), iterations)

end
```

```
mandelbrotset (generic function with 1 method)
```

```
mandelbrotset(n, h)
```



You could, of course, go further with this. Here are some ideas that you may wish to try:

- The Julia set has two-fold symmetry, since z and $-z$ have the same result whenever they are input (as the first operation is squaring). Hence, to make the code (approximately) twice as efficient, you need only compute half of the diagram, and rotate it to get the other half.
- Both the Julia set and the Mandelbrot set are known for their fractal nature, where zooming in to an area of their boundary gives similar complexity. Therefore, you could change the functions to accept different ranges of `x` and `y` values so that you could examine parts of the sets in closer detail.
- You could change the colouring to have a more colourful gradient than we've done here, as can be seen in many images online. All that would be required for this would be a more complicated anonymous function (or perhaps you would prefer to write an actual function for this purpose).

10 Data from and to files

! Prerequisites

Before reading this chapter, you are recommended to have read Chapter 2

- stdin and stdout examples
- Opening file from path
- CSV.jl?

11 Plotting outputs with `Plots.jl`

❗ Prerequisites

Before reading this chapter, you are recommended to have read Chapter 2, Chapter 8

12 Testing, debugging, and fixing code

❗ Prerequisites

Before reading this chapter, you are recommended to have read Chapter 2

References

docs.julialang.org github.com/julialang/julia