Unit 5: More language features for software architecture

Table of contents

1 Unit 5: More language features for software architecture

In the previous three units we explored basics of programming and computation (Unit 2), algorithms and data structures (Unit 3), and data files and numerics (Unit 4). In this unit we take a deeper and more thorough approach at basic Julia language features.

Programming languages, including Julia, are designed mostly with these aims in mind:

1. **Execution speed** — programs should run fast enough to solve the problem at hand.
2. **Coding speed** — writing programs to solve problems should be quick, easy and enjoyable for the coder.
3. **Scalable engineering** — creating and maintaining large programs with many contributors should be no harder than writing small programs yourself.

Different programming languages have different goals and target audiences, and make different trade-offs and affordances for the above. The trickiest one to get right is the third one — as codebases grow it is *typical* to get stuck in a tarpit of complexity, where it gets harder and harder to change and improve the code, and progress grinds to a halt. A corollary of that is that code in large pieces of software is *read* far more often than it is *written*, so it becomes crucial to make code that is written easy to understand.

Julia is a high-performance language for solving large mathematical problems, and compiled code needs to run "close to the metal". It makes many affordances to solve mathematical problems quickly and efficiently. But most crucially, it tries to solve these in such a way that is "scalable" — where generic code can be reused and repurposed, and pieces of a program can be connected together elegantly like lego bricks.

The goal of this unit is to show you the most important language features of Julia and how to use them to best effect. The most important language features that we explore are the syntax, type system, user-defined types, and multiple-dispatch. At the end we'll consider how to use these to construct your code with an emphasis on types and functions (data and transformations / nouns and verbs). Choosing good *nouns* and *verbs* for your problem domain is the core of good program design. The notes below often refer to the Julia documentation.

1.1 Values and their types

One way Julia allows for easy and fast mathematical code is to emphasise *values* (or *objects*), and transformations of values (via functions).

A few ways that is apparent is:

1. All valid Julia code is an *expression* that evaluates to a value (no statements, no `void`).
2. Every value has a specific (or "concrete") *type* (like `String` or `Int64`).
3. Functions are values, which can be stored in variables or passed to other functions.

Many modern languages feature the above, but Julia takes it even further than that:

4. Types are also values that can be passed around and manipulated.
5. Julia code is a value (you can use and create macros, access generated native code, etc).

With these last two, you can do some pretty powerful things with Julia. In fact, most of Julia is itself written in Julia. However, we won't focus on those here.

1.1.1 Why is it so?

Remember, all this is designed to make it easy for mathematicians to write code that fast to run and easy to scale.

In mathematics, we are interested in values or objects, as well as operations and transformations on such objects. Basing the language around values (objects), and allowing functions to behave like values (objects), makes our code resemble mathematics and have a similar interpretation. It does this without "getting in your way" or stopping you writing imperative code.

From a historical perspective, Julia is strongly influenced by lisp (scheme) which is a functional language where everything is an expression.

The fact that values have *types* is for two reasons:

1. Types let us specify our data and define the *behaviour* of functions that operator on them (using multiple dispatch).
2. Types let the compiler know the *representation* of the value in memory (e.g. how many bytes it is) so it can generate fast code.

Types have two purposes: help *humans* organise their data and define behaviour, and the help *the computer* to execute the program quickly.

1.1.2 All valid code is an *expression*

An expression is a piece of code that evaluates to a value. (Some languages, like Python or C, also have other pieces of syntax which are not expressions, typically known as "statements" - a statement tells to computer or compiler to "do something" but does not produce a value you can use).

Here are some expressions in Julia:

```
1 + 1
["a", "b", "c"]
println(pi)
```

One nice aspects of expressions is you can copy them and paste them anywhere that expects an expression, and it is guaranteed to by syntactically valid. Each of these expressions evaluate to a value, which you can for example bind to a variable:

```
x = 1 + 1
y = ["a", "b", "c"]
z = println(pi)
```

1.1.3 Every value has a concrete type

The types of the first value is straightforward:

```
typeof(1+1)
```

```
Int64
```

This is just a 64-bit signed integer, which requires 64 bits (or 8 bytes) of memory to represent at runtime.

The second is a 1D array, or vector:

```
typeof(["a", "b", "c"])
```

Vectors can be arbitrarily long and use an unknown amount of memory at runtime. We'll talk about type parameters (the `{String}` part) and aliases in more detail later, but first let's look at the third:

```
typeof(println(pi))
```

```
Nothing
```

The `println` function takes input and prints it to the console (followed by a new line). It doesn't really have any "value" to return. But I said earlier that all expressions in Julia return *some* value. Julia has an value defined called `nothing` for this purpose, which is of type `Nothing`. There is nothing special about this type, except that it has no associated data (meaning it needs 0 bytes of data to represent at runtime).

1.1.4 What is a concrete type?

A type can be thought of as a set of values.

Every value is a concrete type, and Julia has just these concrete types:

1. `primitive type`
2. `struct` and `mutable struct`
3. `Memory{T}` (a 1D non-resizable container of `T`, used to store the elements of `Array{N, T}`)
4. `String`
5. `Tuple`

1.1.5 Primitive types

Primitive types is where we begin describing data at the bits-and-bytes level. Here is the definition of `UInt8`:

```
primitive type UInt8
   8
end
```

There are 256 possible `UInt8` values: `0x00, 0x01, ..., 0xfe, 0xff`. You could say `UInt8` is a set with 256 elements.

Here are the definitions of `Int64` and `Float64`:

```
primitive type Int64
   64
end

primitive type Float64
   64
end
```

Each of these are sets containing `2^64` (about `10^18`) possible values. However, for every value in your program Julia keeps track of the type of the value, as well as the data. You can consider `Int64` and `Float64` to be *disjoint* sets. The values `0` (an integer) and `0.0` (a float) are represented by exactly the same bits (all zero), yet they are not "the same":

```
reinterpret(Float64, 0)
```

```
0.0
```

```
reinterpret(Int64, 0.0)
```

```
0
```

```
0 === 0.0
```

```
false
```

(Type tracking typically occurs at compile time, before the program runs, so Julia doesn't need to store extra data describing whether every value is an integer or a float or something else.)

The advantage of distinct types is we can describe different behaviour for these types. For example, there are separate methods for addition representing the different ways the CPU handles integer and floating-point arithmetic, `+(x::Int64, y::Int64)` and `+(x::Float64, y::Float64)`.

(Note that all primitive types are immutable values. You cannot "mutate" `0` to become `1` - you construct a new value instead.)

1.1.6 Structs

Structs allow us to organise different pieces of data into a "structure". In general, structs represent composition of data: e.g. a complex number consists of a real component and an imaginary component:

```
struct Complex64
   re::Float64
   im::Float64
end
```

The compiler knows that a `Complex64` needs 128 bits of data to represent - 64 bits each for `re` and `im`. This type is a distinct set with `2^128` possible values.

You can construct a struct with the "constructor" - where the type doubles as a kind of function that lets you specify the fields:

```
z = Complex64(1.0, 2.0)
```

```
Complex64(1.0, 2.0)
```

```
sizeof(z)
```

```
16
```

You can get a field from the struct with the `.` operator, for example

```
z.re
```

```
1.0
```

```
z.im
```

```
2.0
```

1.1.6.1 Empty structs

A struct can have any number of fields, including zero:

```
struct Nothing
end

const nothing = Nothing()
```

This is exactly how `nothing` is defined in Julia! It takes zero bytes to represent nothing, and you could say `Nothing` is like a set containing `2^0 = 1` possible value - `nothing`.

1.1.6.1.1 Mutable structs

By default structs are immutable - once constructed you cannot edit the fields. It is possible to create structs that you can modify with the `mutable` keyword:

```
mutable struct MutableComplex64
   re::Float64
   im::Float64
end

z = MutableComplex64(1.0, 2.0)

z.im = 0.0  # this would throw an error if z were immutable
```

1.1.6.2 Generic structs

Finally, one more thing you can do is create *generic* structs. In Julia the type for complex numbers is more like:

```
struct Complex{T}
   re::T
   im::T
end
```

Given some type `T`, `Complex{T}` is a struct with fields `re` and `im` of type `T`.

This useful because now we don't need a separate `struct` definition for complex numbers with `Float32` components, or `Int64` components, or whatever. Just one definition suffices for them all. This lets us write quite generic code that is oblivious to the exact type `T` but uses properties common to all complex numbers. For example, the absolute value squared can be written:

```
function abs2(z::Complex)
   return z.re*z.re + z.im*z.im
end
```

```
abs2 (generic function with 1 method)
```

and this single definition is sufficient for all possible `T`.

Note that the type `Complex` is an *abstract* type (not a *concrete* type). We'll talk more about abstract types and multiple dispatch later. For now, note that Julia lets you write both generic data types and generic functions.

1.1.6.3 Type aliases

Earlier we saw an example of a type "alias". You can create them like so:

```
const Complex64 = Complex{Float64}
```

Type aliases are useful to make your code shorter or more readable.

1.1.6.4 Tuples

Tuples in Julia are a kind of "lightweight" struct with no type or field names that you need to define in advance. You construct tuples with brackets like `(1, 2, 3)`. Note the types don't have to be uniform, `t = (1, true, "abc")` is equally valid. You access the fields with the index, such as `t[3] === "abc"`.

Note the syntax of tuples is much like the arguments to a function - `(x, y)` is a tuple and `f(x, y)` calls function `f` with inputs `(x, y)`. The correspondence is intentional.

Julia also has "named" tuples, where you can name the fields, like `(a = 1, b = true, c = "abc")`. (These look a bit like keyword arguments to functions).

1.1.7 Arrays

Structs are useful for collecting data, but only if you know how many elements you want at compile time.

Arrays are dynamically-sized collections, that can contain zero-or-more elements. We use arrays a lot in Julia.

Arrays in Julia are multi-dimensional and have the type `Array{T, N}`. Each array value knows the type of its contents `T` as well as how many dimensions `N` it has. However, since we don't know how many elements might be in an array, the computer needs to allocate memory to store the array at runtime. In terms of sets, arrays are a type contain an unbounded number of possible values.

Arrays are mutable collections - elements can be edited, inserted or deleted.

Remember that vectors and matrices are just aliases for 1D and 2D arrays.

```
const Vector{T} = Array{1, T}
const Matrix{T} = Array{2, T}
```

Underneath, `Array{N,T}` is now (as of Julia 1.11) implemented in terms of the low level `Memory{T}` - a one dimensional array of elements of type `T` with a length which is fixed at the time of construction. However, `Vector{T}` is more convenient to work with and you'll generally see that used in most Julia code.

1.1.8 Strings

Strings are built into the Julia language, for convenience. They are much like a `Vector{UInt8}` containing UTF-8 encoded string data, but are immutable.

(If a future version of Julia had immutable arrays, `String` could theoretically become a "normal" Julia type, but for now this is a pragmatic tradeoff).

...and, that's it!

Julia provides other important types, like `Set`s and `Dict`s, but these are implemented in terms of the above. *Every* value is one of:

1. A primitive type
2. A struct (immutable, mutable, tuple or named tuple)
3. A `Memory`
4. A `String`

1.2 Logic and syntax

Now that we've covered all the possible types of values in Julia, we'll take a look at how data is manipulated via logic and functions.

Eventually we'll come back to abstract types and multiple dispatch, but first we'll cover some fundamentals of Julia code syntax and functions so we understand *what* we are looking at later. A lot of this might be intuitive to some of you, but since we are mathematicians, we'd like to spell it out more formally :)

1.2.1 Everything is an expression, everything has a value, and everything has a type

We saw earlier that an expression is a piece of code that, when evaluated, produces a value. For example:

```
1 + 1
```

```
2
```

One thing you should understand about Julia is that it uses type inference to predict the output type of an expression.

```
using InteractiveUtils

@code_typed 1 + 1
```

```
CodeInfo(
1   %1 = intrinsic Base.add_int(x, y)::Int64
        return %1
) => Int64
```

The compiler essentially contains a "proof engine" that can formally prove constraints on the outputs of expressions. In this case it knows for sure that the output will be an `Int64`. Sometimes no proof can be found - in Julia this just makes the code slower, while in some other programming languages that could be a compilation error. This choice makes Julia feel more "dynamic" and makes prototype code easier to write.

1.2.2 `if-else`

In Julia, *every* piece of syntactically valid code is an expression — which means at run time it generates a value which you can assign to a variable. Even things like `if` statements!

```
x = 42

is_x_even = if x % 2 == 0
   "$x is even"
else
   "$x is odd"
end
```

```
"42 is even"
```

Like many languages, Julia has a shorthand "ternary operator" (i.e. operator taking 3 inputs) for *if-else* using `?` and `:`

```
is_x_even = x % 2 == 0 ? "$x is even" : "$x is odd"
```

```
"42 is even"
```

It's worth noting these are 100% identical. We say that `x ? y : z` is "syntax sugar" for the `if` statement — it's a sweetener to make the programmer's life easier, but not an entirely new feature since you can always achieve the same thing with more characters of code. Another piece of syntax sugar is that `x + y` has exactly the same meaning as `+(x, y)`.

1.2.2.1 Our first abstract type

What if the types of the `true` branch and the `false` branch of `if-else` don't match?

```
is_even(x) = x % 2 == 0 ? true : "false"
```

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

Q: what is the output type of `is_even`?

```
is_even(2)
```

```
true
```

```
is_even(3)
```

```
"false"
```

Sometimes it's a `Bool` and sometimes it's a `String`.

```
@code_typed is_even(42)
```

```
CodeInfo(
1   %1 = intrinsic Base.checked_srem_int(x, 2)::Int64
    %2 =   builtin (%1 === 0)::Bool
         goto #3 if not %2
2        return true
3        return "false"
) => Union{Bool, String}
```

If `Bool` represents the set containing `true` and `false`, and `String` represents the set of all possible strings, then the type `Union{Bool, String}` represents the union of these two sets.

`Union` is the first **abstract type** we'll cover, and is relatively straightforward. `Union` itself is not a concrete type; there are no values of type `Union`. It mostly exists in Julia to aid with type inference for cases like `if-else` above – to abstractly analyse the outputs and intermediate values of programs during compilation (where more precise inference leads to more efficient code generation). In cases like this Julia can keep track which of the two possibilities the data is in relatively efficiently (e.g. with a single byte for a tag).

There are two more ways abstract types can show up Julia that we'll see later.

1.2.2.2 Short cutting logic

Julia has operators for Boolean arithmetic, including `&` for *AND* (logical conjunction) and `|` for *OR* (logical disjunction).

```
@show false & false
@show false & true
@show true & false
@show true & true
```

```
false & false = false
false & true = false
true & false = false
true & true = true
```

```
true
```

```
@show false | false
@show false | true
@show true | false
@show true | true
```

```
false | false = false
false | true = true
true | false = true
true | true = true
```

```
true
```

The operation `x & y` is equivalent to `&(x, y)` and will evaluate `x`, then evaluate `y` and then determine the result.

The operation `x && y` is similar but it sometimes takes a shortcut. If `x` is false, it doesn't bother compute a value for `y` (as the answer is `false` no matter the value for `y`).

This matters when `y` is particularly difficult/slow to calculate, or when it may be something you want to avoid doing when `x` is false.

Note that `x && y` is 100% equivalent to:

```
if x
   y
else
   false
end
```

Note that the type of `y` doesn't even need to be `Bool`, since it is never compared to anything. Sometimes people use this in a cute way:

```
x < 0 && println("x is negative")
```

People tend to use that because it takes less space to write than:

```
if x < 0
   println("x is negative")
end
```

Don't feel you need to use "cute" code like this, but if you see it somewhere I don't want you to feel confused.

Q: What does `(x < 0) & println("x is negative")` do?

There is an equivalent shortcut operator for `|`:

```
x || y
```

which is equivalent to

```
if x
   true
else
   y
end
```

One "cute" way to people use this is to perform assertions:

```
x >= 0 || error("x is negative")
```

1.2.3 Blocks of code

Above, we saw blocks of code containing values. They didn't really *do* anything, like assign values to variables.

The rule in Julia is that the last expression in a block of code is the value returned from that block of code. If for some reason you want to *explicitly* create a code block (sometimes useful) you can use `begin` like so:

```
begin
   1
   2
   3
end
```

```
3
```

We see blocks of code everywhere in Julia — inside `if` and `else` statements, within `for` and `while` loops, within `function` bodies, etc. Blocks of code can be nested — we can have `if` statements inside `while` loops inside `function` bodies, and we tend to represent this visually with indentation:

```
function f()
   x = 0
   while x < 10
   if x % 2 == 0
   println("$x is even")
```

```
   else
   println("$x is odd")
   end
   x = x + 1
   end
   return x
end

f()
```

```
0 is even
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
```

```
10
```

Note: `for` and `while` loops always produce `nothing`!

If you prefer, you can separate multiple expressions in a code block with `;` instead of on separate lines. Sometimes you'd even use parentheses to clarify where the code block begins and ends.

```
x = (1; 2; 3)
```

```
3
```

1.2.4 Empty blocks of code return `nothing`

So what happens when the block of code is empty?

```
begin
end
```

Since every expression has a value in Julia, it must return *something*! That something is called `nothing`. It is a special builtin value that contains 0 bytes of data and has the builtin type `Nothing`.

```
typeof(nothing)
```

```
Nothing
```

While the above is contrived, there are some more likely places to see this.

```
half_of_x = if x % 2 == 0
   x ÷ 2
end
```

This is *implicitly* the same as

```
half_of_x = if x % 2 == 0
   x ÷ 2
else
end
```

which is *implicitly* the same as

```
half_of_x = if x % 2 == 0
   x ÷ 2
else
   nothing
end
```

So if `x` is odd, then `half_of_x` is `nothing`.

We also use `nothing` as the return value for functions that don't have a value to return — e.g. when the purpose of the function is to perform some action rather that compute some value.

```
function hello()
   println("Hello class")
   return nothing
end

x = hello()
@show x;
```

```
Hello class
x = nothing
```

Q: Does anyone know what the equivalent thing in `C` is?

1.2.5 Why expressions? Metaprogramming!

Having the syntax rule that "all valid syntax is an expression" makes it easier to analyse and manipulate code than it would be otherwise. It lets you rearrange code much like you can manipulate a mathematical expression with algebra (via substitution, etc), resulting in code that is correct and still compiles.

For human coders, this means it is easier to copy-paste code from one place to another, or refactor it into a function.

A "metaprogram" is a program that writes a program. Julia supports metaprogramming through macros and other advanced techniques. We won't be teaching how to create a macro, but using a macro is pretty easy. A good example is the `@show` macro:

```
x = 10
@show x + 1;
```

```
x + 1 = 11
```

Effectively the macro takes the "expression" `x + 1`, prints the expression, prints equals, and prints the evaluated value. While Julia is busy replacing the builtin syntax sugar (like operators, `x + 1`) with more fundamental expressions (like function calls, `+(x, 1)`), it also "expands" the macro, resulting in code that is roughly:

```
x = 10
println("x + 1 = ", x + 1)
```

```
x + 1 = 11
```

Each macro can be thought of as like user-defined syntax sugar. We can extend the language with our own ideas of how we can write programs.

1.2.6 An even better way to metaprogram

Another form of metaprogramming in Julia is multiple dispatch. Multiple dispatch allows us to construct entire classes of programs that depend on the *types* of values — and Julia will construct on-demand a particular program *specialized* on the inputs you actually provide.

Next we'll learn more about Julia functions and how multiple dispatch works. Afterwards we'll learn about abstract types and how to use this metaprogramming approach.

1.3 Functions in Julia

Functions are at the heart of Julia, and there's lots of ways of expressing different functions. The Julia documentation of functions provides a rich description of all of the details. We now overview a few special features that were perhaps not evident from Units 1-3. There are also links to the documentation for it.

1.3.1 Function syntaxes

In Julia there are multiple ways of creating a function. There is the "long form":

```
function f1(x)
   return x^2
end
```

```
f1 (generic function with 1 method)
```

The `return` is optional in this case — why?

There is a "short form":

```
f2(x) = x^2
```

```
f2 (generic function with 1 method)
```

And there are "anonymous functions" (or "arrow functions"):

```
const f3 = x -> x^2
```

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

In this form `x -> x^2` is an expression that constructs a function which is "anonymous" — it has no name. In this case we have bound it to the name `f3` so it can be referred to later, but anonymous functions are most useful to write simple functions inline within some other code. For example to pass to `map`:

```
map(x -> x^2, [1,2,4])
```

```
3-element Vector{Int64}:
  1
  4
 16
```

1.3.1.1 What type is a function?

You can think of the above functions as being a struct with zero fields. Each function has a unique type:

```
typeof(f1)
```

```
typeof(f1) (singleton type of function f1, subtype of Function)
```

```
typeof(f2)
```

```
typeof(f2) (singleton type of function f2, subtype of Function)
```

```
typeof(f1) == typeof(f2)
```

```
false
```

And like `nothing`, there is no data associated with these:

```
sizeof(f1)
```

```
0
```

These types have only one possible value, which is why Julia describes them as "singleton types". Singleton types have a one-to-one correspondence between type and value.

1.3.1.2 Function methods

Note that in Julia you can attach *function methods* to any type (primitive, struct, etc), not just those defined like above.

```
methods(f1)
```

```
# 1 method for generic function "f1" from Main.Notebook:
 [1] f1(x)
     @ ~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qm
```

```
methods(nothing)
```

```
# 0 methods for callable object
```

A "method" encodes the logic performed when the function executes. Types without methods are not callable. Types can have more than one method. We'll talk more about multiple methods when we discuss multiple dispatch.

For now, just know you can attach methods to *any* type with special syntax like this:

```
(x::Bool)() = "this is $x"

function (x::Bool)()
   return "this is $x"
end
```

For "singleton" types you can use the singleton value instead of the type, so these are equivalent:

```
nothing() = "Nothing else matters"
(::Nothing)() = "Nothing else matters"
```

If this seems a bit crazy or abstract - don't worry! Just remember that *anything* can be made callable - functions are not special.

1.3.2 Nested functions: "closures"

In Julia you can define functions within functions.

```
function adder(x)
   return y -> y + x
end
```

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

This function returns another function! We can use the output to add things.

```
add_2 = adder(2)
add_2(3)
```

```
5
```

Such functions are called "closures" because they "close over" (or "capture") the values of variables outside their scope — in this case it captured the value of `x`, `2`, and implicitly stores it inside `add_2`.

```
add_2.x
```

```
2
```

```
sizeof(add_2)
```

```
8
```

We can see the full internal structure using the `dump()` function:

```
dump(add_2)
```

```
#adder##0 (function of type var"#adder##0#adder##1"{Int64})
  x: Int64 2
```

Note again it doesn't matter which form of function syntax we use. It's just different syntax for the same thing. We could have equally written:

```
function adder(x)
   return function (y)
   return y + x
   end
end
```

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

In Julia, closures are given their own types and methods. The closure is just an automatically generated `struct` datatype with a method attached.

```
typeof(add_2)
```

```
var"#adder##0#adder##1"{Int64}
```

```
methods(add_2)
```

```
# 1 method for anonymous function "#adder##0":
 [1] (::Main.Notebook.var"#adder##0#adder##1")(y)
     @ ~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qm
```

Because closures are "just" a convenient way to do things you can already do in Julia with structs and methods, you can think of them as an advanced type of syntax sugar. The above is equivalent to this:

```
struct Adder{T}
   x::T
end

function (a::Adder)(y)
   return a.x + y
end

function adder(x)
   return Adder(x)
end

add_2 = adder(2)
```

```
Adder{Int64}(2)
```

```
add_2(3)
```

```
5
```

Note that *every* function in Julia has its own type — e.g. `sin` and `cos` have different types. We'll come back to types and methods later.

1.3.3 Argument types

To constrain the type of a function argument we use the `::` operator: When written in the argument list, `x::T` means the argument named `x` must be of type `T` for this method to match.

We have seen Julia methods from the start, e.g.

```
f(x::Int) = x^2
```

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

or,

```
f(x::Number) = x^2
```

```
f (generic function with 3 methods)
```

or simply,

```
f(x) = x^2
```

```
f (generic function with 4 methods)
```

Remember that `x` and `x::Any` are the same thing. The final method will accept *any* inputs. Whether or not `x^2` works or throws an error depends on `x`.

In a sense, the third method isn't *safe* since someone could provide an input like `(1, 2, 3)` that can't be squared. What happens if someone provides a string? Is this expected? On the other hand, the third method provides maximum flexibility — someone can introduce a new type that works with `f` at any point in time, without having to edit the definition of `f` to make it work.

Note that there are two other uses of the `::` operator outside of function arguments - type assertion and typed variable declarations - but these aren't used as commonly and we won't cover them here.

1.3.4 Dispatch

"Dispatch" is the process of determining which method of a function is called.

```
example_f(x::Int) = "input is an integer"
example_f(x::String) = "input is a string"

@show example_f(1)
@show example_f("abc")

methods(example_f)
```

```
example_f(1) = "input is an integer"
example_f("abc") = "input is a string"
```

```
# 2 methods for generic function "example_f" from Main.Notebook:
 [1] example_f(x::String)
     @ ~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qm
 [2] example_f(x::Int64)
     @ ~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qm
```

You can use it to tailor the logic of a function to its inputs. Try type `methods(+)` at the REPL!

Argument types can be abstract types, like `Any` or `Union`.

```
example_f2(x::Int) = "input is an integer"
example_f2(x::Any) = "input is not an integer"

@show example_f2(1)
@show example_f2("abc")

methods(example_f2)
```

```
example_f2(1) = "input is an integer"
example_f2("abc") = "input is not an integer"
```

```
# 2 methods for generic function "example_f2" from Main.Notebook:
 [1] example_f2(x::Int64)
     @ ~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qm
 [2] example_f2(x)
     @ ~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qm
```

Note that `Int <: Any` and the two methods overlap somewhat. In this case Julia will pick the most "specific" method. In terms of "types as sets of values" it matches to the "smallest" set.

The term "multiple dispatch" means that dispatch can depend on the types of multiple arguments:

```
example_f3(x::Int, y::Int) = "both input are integers"
example_f3(x::String, y::Int) = "the inputs have different types"
example_f3(x::Int, y::String) = "the inputs have different types"
example_f3(x::String, y::String) = "both input are strings"

@show example_f3(1, 2);
@show example_f3(1, "abc");

methods(example_f3)
```

```
example_f3(1, 2) = "both input are integers"
example_f3(1, "abc") = "the inputs have different types"
```

```
# 4 methods for generic function "example_f3" from Main.Notebook:
 [1] example_f3(x::String, y::String)
     @ ~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qm
 [2] example_f3(x::Int64, y::String)
     @ ~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qm
 [3] example_f3(x::String, y::Int64)
     @ ~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qm
 [4] example_f3(x::Int64, y::Int64)
     @ ~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qm
```

In object-oriented languages (like Python, Javascript, C#, Java or C++) dynamic dispatch occurs only on the type of first object (the class method being called). Multiple dispatch is a lot more flexible and changes the way we design software.

1.3.5 Varargs functions

Sometimes functions have a variable number of arguments — sometimes called "varargs" (or "variadic functions").

A simple thing to do might be to add up all the inputs:

```
function add_all(inputs...)
   out = 0
   for x in inputs
   out += x
```

```
   end
   return out
end

add_all(1, 2, 3, 4, 5)
```

```
15
```

```
function polynomialGenerator(a...)
   n = length(a) - 1
   poly = function(x)
   return sum(a[i+1] * x^i for i in 0:n)
   end
   return poly
end

polynomial = polynomialGenerator(1, 3, -10)

[polynomial(-1), polynomial(0), polynomial(1)]
```

```
3-element Vector{Int64}:
 -12
   1
  -6
```

You could use then use `Roots` package to automatically find out where all the inputs produce zero output:

```
using Roots

zero_vals = find_zeros(polynomial, -10, 10)
println("Zeros of the function f(x): ", zero_vals)
```

```
Zeros of the function f(x): [-0.19999999999999998, 0.5]
```

Here you start to see how Julia features like varargs functions, closures, and packages let you solve a wide range of problems without writing very much code.

1.3.6 Optional arguments

You can have optional arguments (or default values):

```
using Distributions

function my_density(x::Float64, ::Float64 = 0.0, ::Float64 = 1.0)
   return exp(-(x- )^2 / (2 ^2)) / ( *√(2 ))
end

x = 1.5
@show pdf(Normal(), x) my_density(x)
@show pdf(Normal(0.5), x) my_density(x, 0.5);
```

```
pdf(Normal(), x) = 0.12951759566589174
my_density(x) = 0.12951759566589174
pdf(Normal(0.5), x) = 0.24197072451914337
my_density(x, 0.5) = 0.24197072451914337
```

1.3.7 Keyword arguments

Arguments following the ; character in the function definition or the function call are called **keyword arguments**. These are named as they are used.

```
function my_density(x::Float64;  ::Float64 = 0.0,  ::Float64 = 1.0)
   return exp(-(x- )^2/(2 ^2) ) / ( *√(2 ))
end

@show pdf(Normal(0.0,2.5),x) my_density(x,  =2.5);
```

```
pdf(Normal(0.0, 2.5), x) = 0.13328984115671988
my_density(x,   = 2.5) = 0.13328984115671988
```

We can even make a function that takes arbitrary numbers of positional and keyword arguments:

```
function very_flexible_function(args...; kwargs...)
   @info "All the args" args kwargs
end

very_flexible_function(2.5, false, a=1, b="two", c=:three)
```

```
┌ Info: All the args
│   args = (2.5, false)
│   kwargs =
│    pairs(::NamedTuple) with 3 entries:
│      :a => 1
│      :b => "two"
└      :c => :three
```

The standard non-keyword arguments are called "positional" arguments since they are identified by their position not their name.

Note that when *calling* a function you may either put a `;` or a `,` before the keyword arguments. In a function definition there's an ambiguity between optional arguments and keyword

1.3.8 Destructing

In Julia you can *destructure* tuples (and other iterables) in the reverse way you *construct* them:

```
my_tuple = (42, "abc")  # construct
(a, b) = my_tuple       # destruct

@show a
@show b;
```

```
a = 42
b = "abc"
```

A really common pattern here is swapping two variables:

```
x = 1
y = 2
(x, y) = (y, x)
@show x
@show y;
```

```
x = 2
y = 1
```

You can also destructure fields of structs and named tuples in the reverse way you create a named tuple:

```
function f(x::Complex)
   (; re, im) = x
   println("Real part: $re")
```

```
   println("Imaginary part: $im")
end

f(10 + 42im)
```

```
-1664 + 840im
```

(But note that the `x.re` and `x.im` fields are internal - you should use the `real()` and `imag()` functions to work with `Complex` values!)

1.3.9 Function composition and piping

Just a little bit of syntax sugar for "piping" values into function calls:

```
 /4 |> cos |> acos |> x -> 4x
```

```
3.141592653589793
```

Here is a function just like `identity`:

```
const ii = cos  acos #\circ + [TAB]
(ii( /4),  /4)
```

```
(0.7853981633974483, 0.7853981633974483)
```

1.3.10 Dot syntax for broadcasting functions

You already know the broadcast operator. It is syntax sugar for the `broadcast` function.

```
x_range = 0:0.5:
cos.(x_range)
```

```
7-element Vector{Float64}:
  1.0
  0.8775825618903728
  0.5403023058681398
  0.0707372016677029
 -0.4161468365471424
 -0.8011436155469337
 -0.9899924966004454
```

See the docs for a discussion of performance of broadcasting.

You can also use the macro `@.`

```
x_range = 0:0.5:
@. cos(x_range + 2)^2
```

```
7-element Vector{Float64}:
 0.17317818956819406
 0.6418310927316131
 0.9800851433251829
 0.8769511271716524
 0.4272499830956933
 0.0444348690576615
 0.08046423546177377
```

1.4 More items from control flow

You are already very familiar with conditional statements (`if`, `elseif`, `else`), with loops (`for` and `while`), with short circuit evaluation, and with many other variants. For example you can use `continue` and `break` in loops, and Julia even supports `@goto` and `@label` for when that isn't enough.

You can find more control flow details here: control flow in Julia docs. We'll just dive into `for` loops and error handling below.

1.4.1 Iteration and `for` loops

One impressive piece of syntax sugar is the `for` loop. It is in fact a special case of a `while` loop, like so:

```
# this loop...
for x in iterable
   # <CODE>
end

# ...is equivalent to
tmp = iterate(iterable)
while tmp !== nothing
   (x, state) = tmp

   # <CODE>

   tmp = iterate(iterable, state)
end
```

In the above, `tmp` is either `nothing` or it's a tuple of length two — containing the next value and whatever metadata is needed to iterate to the next element.

You can make any of your data types iterable in a `for` loop by implementing a method on the `iterate` function. We won't be doing this in this course, but it is an excellent example of an *interface* in Julia. We'll talk more about interfaces later.

1.4.2 Errors and exceptions

One additional thing to know about is exception handling.

```
function my_2_by_2_inv(A::Matrix{Float64})
   if size(A) != (2,2)
   error("This function only works for 2x2 matrices")
   end

   d = A[1,1]*A[2,2] - A[2,1]*A[1,2]

   if d   0
   throw(ArgumentError("matrix is singular or near singular")) #\approx + [TAB]
   end

   return [A[2,2] -A[1,2]; -A[2,1] A[1,1]] ./ d
end

my_2_by_2_inv(rand(3,3))
```

```
ErrorException: ErrorException("This function only works for 2x2 matrices")
This function only works for 2x2 matrices
Stacktrace:
 [1] error(s::String)
   @ Base ./error.jl:44
 [2] my_2_by_2_inv(A::Matrix{Float64})
```

```
   @ Main.Notebook
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qmd:1032
 [3] top-level scope
   @
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qmd:1044
```

```
my_2_by_2_inv(ones(2,2))
```

```
ArgumentError: ArgumentError("matrix is singular or near singular")
ArgumentError: matrix is singular or near singular
Stacktrace:
 [1] my_2_by_2_inv(A::Matrix{Float64})
   @ Main.Notebook
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qmd:1038
 [2] top-level scope
   @
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qmd:1048
```

```
using LinearAlgebra, Random

Random.seed!(0)
A = rand(2, 2)
A_inv = my_2_by_2_inv(A)
@assert A_inv*A   I
```

```
Random.seed!(0)
for _   1:10 #\in + [TAB]
   A = float.(rand(1:5,2,2))
   try
   my_2_by_2_inv(A)
   catch e
   println(e)
   end
end
```

An exception may be caught way down the call stack:

```
f(mat) = 10*my_2_by_2_inv(mat)
g(mat) = f(mat .+ 3)
h(mat) = 2g(mat)
A = ones(2,2)
h(A)
```

```
ArgumentError: ArgumentError("matrix is singular or near singular")
ArgumentError: matrix is singular or near singular
Stacktrace:
 [1] my_2_by_2_inv(A::Matrix{Float64})
   @ Main.Notebook
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qmd:1038
 [2] f
   @
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qmd:1075
[inlined]
 [3] g(mat::Matrix{Float64})
   @ Main.Notebook
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qmd:1076
 [4] h(mat::Matrix{Float64})
   @ Main.Notebook
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qmd:1077
 [5] top-level scope
```

```
   @
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qmd:1079

ERROR: ArgumentError: matrix is singular or near singular
Stacktrace:
 [1] my_2_by_2_inv(A::Matrix{Float64})
   @ Main ~/git/mine/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/markdown/lecture-
 [2] f(mat::Matrix{Float64})
   @ Main ./REPL[33]:1
 [3] g(mat::Matrix{Float64})
   @ Main ./REPL[34]:1
 [4] h(mat::Matrix{Float64})
   @ Main ./REPL[35]:1
 [5] top-level scope
   @ REPL[36]:1
```

1.4.2.1 Try, catch and finally

It's possible to "catch" and handle errors that happened using `try`, `catch` and `finally`.

```
try
   h(A)
catch e
   @info "Handling exception" println(e)
end
```

```
ArgumentError("matrix is singular or near singular")
  Info: Handling exception
    println(e) = nothing
```

```
try
   0 ÷ 0
catch e
   @show e
finally
   println("done")
end;
```

```
e = DivideError()
done
```

1.4.2.2 Resource cleanup

One place closures, and `do` syntax in particular, gets used is when you want resources to be cleaned up automatically. Here's how one method of `open` is defined in Julia:

```
function open(f, filename::String)
   io = open(filename)
   try
   f(io)
   finally
   close(io)
   end
end
```

You can call the `open` function with an anonymous function like so:

```
# Read the first 128 bytes from the file
open(io -> read(io, 128), filename)
```

This way even when `read` might error out (e.g. too few bytes), the file is *always* closed in the `finally` block above. Using this pattern avoids mistakes and resource leakage.

When you want to use this pattern it is common to use `do` syntax like so:

```
open(filename) do (io)
   # Read the first 128 bytes from the file
   return read(io, 128) # might throw an exception
end
```

The `do` syntax is *yet another* way of making a function. It creates a new function and injects it into the first argument.

1.4.3 Variable scope

See variables and scoping in Julia docs.

Julia actually has two separate scopes — global or "toplevel" scope, and local scope.

Local scope is the simplest, and we'll look at that first. By default bindings inside a function body create *new* variables that take precedence (or "shadows") any variables of the same name outside the function body.

```
x = 0

function f()
   x = 1 # create a new local variable
   return 2 * x
end

f()
```

```
2
```

You can use the `local` keyword to create new local bindings. The above is 100% equivalent to:

```
function f()
   local x = 1 # create a new local variable
   return 2 * x
end
```

If you want to modify the global variable, there is a `global` keyword for that:

```
x = 0

function f()
   global x = 1 # modify the global variable
   return 2 * x
end

@show x
@show f()
@show x;
```

```
x = 0
f() = 2
x = 1
```

Inside other code blocks such as `if`, `while` and `for`, the default is to overwrite existing local variables if they exist.

Q: What is the difference between these two functions?

```
function f1()
   x = 0
   for i = 1:10
   x = i
   end
   return x
end

function f2()
   x = 0
   for i = 1:10
   local x = i
   end
   return x
end
```

```
f2 (generic function with 2 methods)
```

By default variables created *inside* a loop no longer exist after the loop:

```
function f()
   for i = 1:3
   x = i
   end
   return x
end
f()
```

```
1
```

To fix that you can define that variable in advance using `local`:

```
function f()
   local x
   for i = 1:3
   x = i
   end
   return x
end
f()
```

```
3
```

Occasionally you might want to use the value of what is being iterated, where you can use `outer`:

```
function f()
   x = 0
   for outer x = 1:3
   # empty
   end
   return x
end
f()
```

```
3
```

That can be useful in combination to `break`, where you can see where the loop terminated:

```
function f()
   x = 0
   for outer x = 1:3
   if x > 1.5
```

```
   break
   end
   end
   return x
end
f()
```

2

Working with variables in different scopes can a little confusing and error prone. My advice:

- Don't use non-const global variables unless there is no alternative. Prefer passing variables to functions, including using closures to capture state (see also ScopedValue in Julia 1.11).
- Use distinct names for variables as much as possible
- The larger the scope of a variable, the longer and more descriptive its name should be

Remember you want your programs to be *simple* and *easy to read*. Variable names are cheap (though naming things well is hard!)

1.5 More on mutation

As we saw earlier, a type can be **mutable** or not (**immutable**). Values/objects of the immutable types are typically stored on the stack. Values/objects of mutable types are typically allocated and stored on the heap.

```
@show ismutable(7)
@show ismutable([7]);
```

```
ismutable(7) = false
ismutable([7]) = true
```

When you pass a mutable object to a function, the function can change the data contained within.

```
function mutating_function(z::Array{Int})
   z[1] = 0
end

x = [1]
println("Before call: ", x)
mutating_function(x)
println("After call: ", x)
```

```
Before call: [1]
After call: [0]
```

Programmers tend to need to keep track of which objects are being mutated and when. You'll see functions that mutate things often end in `!`, e.g. `push!`, `pop!`, `setindex!`, etc. This is just a convention – we could have called the above `mutating_function!`.

Note that `collection[index]` is shorthand for `getindex(collection, index)` and `collection[index] = value` is shorthand for `setindex!(collection, index, value)`.

For immutable values, there is no way to modify their contents. They are never modified by functions.

1.5.0.1 Variable bindings

Variable bindings are not modified inside functions. The function below introduces a *new* binding `z` every time the function is called, and `x` is unaffected.

```
function rebinding_function(z::Array{Int})
   z = [0]
end

x = [1]
println("Before call: ", x)
mutating_function(x)
println("After call: ", x)
```

```
Before call: [1]
After call: [0]
```

So, the rule is that you can mutate the *inside* of a mutable value (`Array` or `mutable struct`) and *other* variables and/or data structures with references to that mutable value will be able to see the effect of the mutation.

However variables are *always* bound to *new* values with `=` (e.g. `z = 0`) and the usual lexical scoping rules apply.

To help with preserving data that might get modified, we often make a copy with `copy` and `deepcopy`:

```
println("Immutable:")
a = 10
b = a
b = 20
@show a;
```

```
Immutable:
a = 10
```

```
println("\nNo copy:")
a = [10]
b = a
b[1] = 20
@show a;
```

```
No copy:
a = [20]
```

```
println("\nCopy:")
a = [10]
b = copy(a)
b[1] = 20
@show a;
```

```
Copy:
a = [10]
```

```
println("\nShallow copy:")
a = [[10]]
b = copy(a)
b[1][1] = 20
@show a;
```

```
Shallow copy:
a = [[20]]
```

```
println("\nDeep copy:")
a = [[10]]
b = deepcopy(a)
b[1][1] = 20
@show a;
```

```
Deep copy:
a = [[10]]
```

1.6 Abstract type tree

We saw the `Union` abstract type earlier.

Julia also has an abstract type hierarchy (a tree). At the top of the tree is the type `Any`, which encompasses every possible value in Julia. All types have a supertype (the supertype of `Any` is `Any`). Types that are not leaves of the tree have subtypes. Some types are **abstract** while others are **concrete**. One particularly distinctive feature of Julia's type system is that concrete types may not subtype each other: all concrete types are final and may only have abstract types as their supertypes.

```
x = 2.3
@show typeof(x)
@show supertype(Float64)
@show supertype(AbstractFloat)
@show supertype(Real)
@show supertype(Number)
@show supertype(Any);
```

```
typeof(x) = Float64
supertype(Float64) = AbstractFloat
supertype(AbstractFloat) = Real
supertype(Real) = Number
supertype(Number) = Any
supertype(Any) = Any
```

There is an **is a** relationship:

```
isa(2.3, Number)
```

```
true
```

```
isa(2.3, String)
```

```
false
```

```
2.3 isa Float64
```

```
true
```

Note that `x isa T` is the same as `typeof(x) <: T`, where we say `<:` as "is a subtype of".

```
@show Float64 <: Number
@show String <: Number;
```

```
Float64 <: Number = true
String <: Number = false
```

We can ask whether a given type is abstract or concrete.

```
@show isabstracttype(Float64)
@show isconcretetype(Float64);
```

```
isabstracttype(Float64) = false
isconcretetype(Float64) = true
```

```
@show isabstracttype(Real)
@show isconcretetype(Real);
```

```
isabstracttype(Real) = true
isconcretetype(Real) = false
```

Structs with undefined type parameters are not concrete:

```
@show isconcretetype(Complex);
```

```
isconcretetype(Complex) = false
```

Once we provide the type parameters we do get a concrete type:

```
@show isconcretetype(Complex{Float64});
```

```
isconcretetype(Complex{Float64}) = true
```

As mentioned, Julia has a type tree. Let's walk down from `Number`:

```
using InteractiveUtils: subtypes

function type_and_children(type, prefix = "", child_prefix = "")
   if isconcretetype(type)
   @assert isempty(subtypes(type))

   println(prefix, type, ": concrete")
   else
   println(prefix, type, isabstracttype(type) ? ": abstract" : ": parameterised")

   children = subtypes(type)
   for (i, c) in enumerate(children)
   if i == length(children)
   type_and_children(c, "$(child_prefix)   ", "$(child_prefix)    ")
   else
   type_and_children(c, "$(child_prefix)   ", "$(child_prefix)    ")
   end
   end
   end
end

type_and_children(Number)
```

```
Number: abstract
   Base.MultiplicativeInverses.MultiplicativeInverse: abstract
       Base.MultiplicativeInverses.SignedMultiplicativeInverse: parameterised
       Base.MultiplicativeInverses.UnsignedMultiplicativeInverse: parameterised
   Base.Complex: parameterised
   Real: abstract
       AbstractFloat: abstract
           BigFloat: concrete
           Core.BFloat16: concrete
           Float16: concrete
           Float32: concrete
```

```
        Float64: concrete
    AbstractIrrational: abstract
        Irrational: parameterised
        IrrationalConstants.IrrationalConstant: abstract
            IrrationalConstants.Fourinv : concrete
            IrrationalConstants.Four : concrete
            IrrationalConstants.Half : concrete
            IrrationalConstants.Inv2 : concrete
            IrrationalConstants.Inv4 : concrete
            IrrationalConstants.Invsqrt2: concrete
            IrrationalConstants.Invsqrt2 : concrete
            IrrationalConstants.Invsqrt : concrete
            IrrationalConstants.Inv : concrete
            IrrationalConstants.Log2 : concrete
            IrrationalConstants.Log4 : concrete
            IrrationalConstants.Loghalf: concrete
            IrrationalConstants.Logten: concrete
            IrrationalConstants.Logtwo: concrete
            IrrationalConstants.Log : concrete
            IrrationalConstants.Quart : concrete
            IrrationalConstants.Sqrt2: concrete
            IrrationalConstants.Sqrt2 : concrete
            IrrationalConstants.Sqrt3: concrete
            IrrationalConstants.Sqrt4 : concrete
            IrrationalConstants.Sqrthalf : concrete
            IrrationalConstants.Sqrt : concrete
            IrrationalConstants.Twoinv : concrete
            IrrationalConstants.Two : concrete
            StatsBase.Mad_constant: concrete
    Integer: abstract
        Bool: concrete
        Signed: abstract
            BigInt: concrete
            Int128: concrete
            Int16: concrete
            Int32: concrete
            Int64: concrete
            Int8: concrete
        Unsigned: abstract
            UInt128: concrete
            UInt16: concrete
            UInt32: concrete
            UInt64: concrete
            UInt8: concrete
    Rational: parameterised
    StatsBase.PValue: concrete
    StatsBase.TestStat: concrete
  Unitful.AbstractQuantity: abstract
    Unitful.Quantity: parameterised
  Unitful.LogScaled: abstract
     Unitful.Gain{L} where L<:Unitful.LogInfo: parameterised
     Unitful.Level{L} where L<:Unitful.LogInfo: parameterised
```

In Julia, you can define abstract types with the `abstract type` keywords:

```
abstract type Number
end

abstract type Real <: Number
end

abstract type AbstractFloat <: Real
end

primitive type Float64 <: AbstractFloat
   64
end
```

1.6.1 Parameterized / generic types

We've actually now seen all three types of abstract types in Julia – the `abstract type`s that make up the type tree, the `Union` type, and the parameterized types (abstract `Complex` vs concrete `Complex{Float64}`).

Actually `Complex` is a shorthand. It's full type is written like this:

```
Complex{T} where T <: Real
```

This object is of type `UnionAll`:

```
typeof(Complex)
```

```
UnionAll
```

You can read this like "The abstract type which is the union of the concrete types `Complex{T}` for all possible `T <: Real`" – hence the shorthand `UnionAll`. Parameterised types can have bounds, like the components of complex numbers being real numbers.

```
@show Complex{Float64} <: Complex
@show isconcretetype(Complex)
@show isconcretetype(Complex{Float64});
```

```
Complex{Float64} <: Complex = true
isconcretetype(Complex) = false
isconcretetype(Complex{Float64}) = true
```

Sometimes it's useful to **use values which aren't types as type parameters ("Value types")**. One standard example is the **`N` in `Array{T,N}`** which is just an integer signifying the dimension of the array. Doing this helps the compiler know some parts of the "problem size" at compile time based on the type and can lead to much more efficient code or data storage (see example `RegularPolygon` below).

Julia is pretty capable at figuring out the subtype (or subset) relationships:

```
(Complex{T} where T <: AbstractFloat) <: Complex
```

```
true
```

which follow from `AbstractFloat <: Real`.

You've seen other `UnionAll` types like `Vector`, `Matrix`, `Set`, `Dict`, etc.

Don't worry about this seeming complex – consider this background material!

1.6.2 Union types

We saw earlier that Julia has a third abstract type called `Union` which let's you reason about a finite set of (abstract or concrete) types.

```
42::Union{Int, Float64}
```

42

```
3.14::Union{Int, Float64}
```

3.14

```
"abc"::Union{Int, Float64}
```

```
TypeError: TypeError(:typeassert, "", Union{Float64, Int64}, "abc")
TypeError: in typeassert, expected Union{Float64, Int64}, got a value of type String
Stacktrace:
 [1] top-level scope
   @
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qmd:1534
```

`Union` can handle an arbitrary number of types, `Union{T1, T2, T3, ...}`.

As a special case `Union{T}` is just the same as `T`. We also have `Union{T, T} == T`, etc.

The union of no types at all, `Union{}`, is a special builtin type which is the opposite of `Any`. No value can exist with type `Union{}`! Sometimes `Any` is called the "top" type and `Union{}` is called the "bottom" type. It's used internally by the compiler to rule out impossible situations, but it's not something for you to worry about.

You've now seen *every possible concrete type* and *every possible abstract type* in all of Julia. You've also looked a functions, methods and multiple dispatch.

Don't worry if that was whirlwind. You don't need to memorise every detail that I've covered!

Now we will practice using these tools to build larger programs, so you are ready for Projects 1, 2 and 3.

1.7 Crafting programs in Julia

1.7.1 Making types for your programs - dimensions and units

You can define your own types. Any "serious" programming task would almost always merit that you do that.

1.7.1.1 Dimensions and units

One thing types let you do is ensure you do things "safely" without making mistakes.

When solving physics problems its always useful to perform dimensional analysis to ensure your answer makes sense, and to make sure you are using a consistent set of units in your numeric computations.

- **Dimensions** - length, time, mass, area (length squared), etc
- **Units** - metres, kilometres, centimetres, feet, inches, miles, etc

1.7.1.2 Our first unit - `Metre`

Julia lets us encode dimensions and units. There is a nice package called Unitful.jl that you can download and use. Lets see how a package like this works.

We can start simple with a single type:

```
struct Metre{T <: Real}
   value::T
end
```

That's all very good, but we can't actually *do* anything with it. When you introduce new data types you'll generally need some functions or methods to interact with the data. Lets define some basic algebra:

```
Base.:(+)(x::Metre, y::Metre) = Metre(x.value + y.value)
Base.:(-)(x::Metre, y::Metre) = Metre(x.value - y.value)
Base.:(-)(x::Metre) = Metre(-x.value)
Base.:(*)(x::Metre, y::Real) = Metre(x.value * y)
Base.:(*)(x::Real, y::Metre) = Metre(x * y.value)
Base.:(/)(x::Metre, y::Metre) = x.value / y.value
Base.:(/)(x::Metre, y::Real) = Metre(x.value / y)
```

Here we have added new methods to existing functions (from `Base`). What we are doing here is opting-in to an existing interface. If it makes sense to use existing interfaces - do so! Later we'll create our own interface, but baby-steps first.

(In an object-oriented language, you would probably add a class with some data and some function methods. It is very similar except for where the methods live!)

We can make this a little "nicer" to use with one more definition.

```
const m = Metre(1)
```

This way we can write the code rather naturally:

```
3.5m
```

(Remember that's just short for `3.5 * m`).

We can also make it print something similar to the terminal:

```
Base.show(io::IO, x::Metre) = print(io, x.value, "m")
```

1.7.1.2.1 How does this make things safe?

If I try and do something invalid like `1 + 2m`, Julia will throw an error. This means algebra with incorrect dimensions will not compute in Julia. You'll get an error message and have the opportunity to fix your code.

1.7.1.2.2 Isn't all of this slow?

Surprisingly, in Julia this doesn't slow anything down. The `Metre{Float64}` object uses the same 64 bits to represent as a bare `Float64`. The compiler is smart enough to know it can use the same representation for either type interchangeably, and your program won't spend any effort "constructing structs" or "getting fields" at run time.

1.7.1.3 Other length units

How would you deal with other kinds of units of length?

- kilometres
- centimetres
- inches
- miles

Q: Does anyone have any other favourite unit of length?

As you can see, there might be a lot of different units of length. And units of time, units of mass, units of temperature, etc. And the variety grows further when you consider compound dimensions (like area, or velocity). And then we need to consider *binary* operators like `+`, where the number of possible combinations is quadratic in the number of units being considered.

How should we deal with all these combinations? How will the system know how to add centimetres and yards? What is `3yd + 68cm`?

1. Write every single method by hand?
2. Have a "blessed" unit (like SI units - kilograms, metres, seconds, Kelvin, etc)
3. Have some set of clever rules (e.g. if both inputs are imperial, stay imperial)

Option #1 is not practical. Here we are going to look at option #2. Julia actually uses option #3 for numeric conversion (what is `0x20 + 3.14`?), but we'll keep it a bit simpler.

This is a good place to introduce some abstract types. Let's focus on just length units.

```
abstract type AbstractLength
end;
```

This will be the "supertype" for our length types. (Remember "super" is Latin for "above" – your supervisor is your overseer, etc).

```
struct Metre{T} <: AbstractLength
   value::T
end

struct Kilometre{T} <: AbstractLength
   value::T
end

struct Centimetre{T} <: AbstractLength
   value::T
end

struct Inch{T} <: AbstractLength
   value::T
end

struct Mile{T} <: AbstractLength
   value::T
end;
```

1.7.1.3.1 Conversion to and from metres

For each of these, we might want a way to convert these to metres. We can use the `Metre` type, and attach methods to that. Such methods are known as "constructors".

```
Metre(x::Kilometre) = Metre(1000 * x.value)
Metre(x::Centimetre) = Metre(0.01 * x.value)
Metre(x::Inch) = Metre(0.0254 * x.value)
Metre(x::Mile) = Metre(1760 * 3 * 12 * 0.0254 * x.value);
```

In the above, I know an inch is 2.54 centimetres, so I used that. I don't recall how many metres per mile, but I can look up how many yards per mile, and I recall there are 3 feet per yard and 12 inches per foot.

There is one more we should add, just to make sure things are never ambiguous for Julia:

```
Metre(x::Metre) = x;
```

We'll also find it useful to go the other direction:

```
Kilometre(x::Metre) = Kilometre(0.001 * x.value)
Centimetre(x::Metre) = Centimetre(100 * x.value)
Inch(x::Metre) = Inch(x.value / 0.0254)
Mile(x::Metre) = Mile(x.value / (1760 * 3 * 12 * 0.0254));
```

1.7.1.3.2 Conversion between any two types

If we are converting between two length units neither of which is `Metre`, we might want to simply convert the input to metres first. Doing this we end up with:

```
Kilometre(x::AbstractLength) = Kilometre(Metre(x))
Kilometre(x::Kilometre) = x

Centimetre(x::AbstractLength) = Centimetre(Metre(x))
Centimetre(x::Centimetre) = x

Inch(x::AbstractLength) = Inch(Metre(x))
Inch(x::Inch) = x

Mile(x::AbstractLength) = Mile(Metre(x))
Mile(x::Mile) = x;
```

Note that if we have `N` different length types, we'd only need to define `2N` different conversions (to and from metres) instead of `N^2` conversions (between every two possibilities). If `N` is large, this is hugely beneficial!

Another fun fact is two different programmers can define different `AbstractLength` units without talking to each other. A third programmer can use the code from the first two, and those units will seamlessly work together with zero effort. This is great for sharing working in teams, between teams, or between entirely separate organizations (e.g. open-source code).

Note that in general, we can get Julia to automatically do this step for every `AbstractLength` type. The syntax for this one is a bit crazy:

```
(::Type{T})(x::AbstractLength) where {T <: AbstractLength} = T(Metre(x));
(::Type{T})(x::T) where {T <: AbstractLength} = x;
```

(Don't worry – we won't expect you to create something like that in this course!)

1.7.1.3.3 Algebra with metres

We have seen these already:

```
Base.:(+)(x::Metre, y::Metre) = Metre(x.value + y.value)
Base.:(-)(x::Metre, y::Metre) = Metre(x.value - y.value)
Base.:(-)(x::Metre) = Metre(-x.value)
Base.:(*)(x::Metre, y::Real) = Metre(x.value * y)
Base.:(*)(x::Real, y::Metre) = Metre(x * y.value)
Base.:(/)(x::Metre, y::Metre) = x.value / y.value
Base.:(/)(x::Metre, y::Real) = Metre(x.value / y)
```

1.7.1.3.4 Algebra with abstract types

Now we have enough to perform algebra between all these types. Let's look at addition, subtraction and division:

```
Base.:(+)(x::AbstractLength, y::AbstractLength) = Metre(x) + Metre(y)
Base.:(-)(x::AbstractLength, y::AbstractLength) = Metre(x) - Metre(y)
Base.:(/)(x::AbstractLength, y::AbstractLength) = Metre(x) / Metre(y);
```

So to add, we just convert to metres first and then keep going.

For the other operations we can "keep" the existing type, but we need to actually know the "unit" separately to its type parameter (e.g. `Centimetre` vs `Centimetre{Int64}` or `Centimetre{Float64}`). We define a brand-new function to describe this:

```
unit(::Metre) = Metre
unit(::Kilometre) = Kilometre
unit(::Centimetre) = Centimetre
unit(::Inch) = Inch
unit(::Mile) = Mile
```

```
unit (generic function with 5 methods)
```

Using this we can define "generic" methods for these functions:

```
Base.:(-)(x::AbstractLength) = unit(x)(-x.value)
Base.:(*)(x::AbstractLength, y::Real) = unit(x)(x.value * y)
Base.:(*)(x::Real, y::AbstractLength) = unit(y)(x * y.value)
Base.:(/)(x::AbstractLength, y::Real) = unit(x)(x.value / y);
```

If we want to be a bit more fancy, we can get Julia to skip the conversion for addition, subtraction and division when it is unnecessary. There are different ways to do this like fancy dispatch rules, but you can just work with types directly *inside* the functions:

```
function Base.:(+)(x::AbstractLength, y::AbstractLength)
   if unit(x) == unit(y)
   return unit(x)(x.value + y.value)
   else
   return Metre(x) + Metre(y)
   end
end

function Base.:(-)(x::AbstractLength, y::AbstractLength)
   if unit(x) == unit(y)
   return unit(x)(x.value - y.value)
   else
   return Metre(x) - Metre(y)
   end
end

function Base.:(/)(x::AbstractLength, y::AbstractLength)
   if unit(x) == unit(y)
   return x.value / y.value
   else
   return Metre(x) / Metre(y)
   end
end;
```

(Note that the `if` part will be dealt with at compile time.)

1.7.1.3.5 Niceties

```
const m = Metre(1) # done above
const km = Kilometre(1)
const cm = Centimetre(1)
const inch = Inch(1)
const mile = Mile(1);
```

```
Base.show(io::IO, x::Metre) = print(io, x.value, "m")
Base.show(io::IO, x::Kilometre) = print(io, x.value, "km")
Base.show(io::IO, x::Centimetre) = print(io, x.value, "cm")
Base.show(io::IO, x::Inch) = print(io, x.value, "inch")
Base.show(io::IO, x::Mile) = print(io, x.value, "mile");
```

1.7.1.3.6 Some results

Q: How many kilometres in a mile?

```
mile / km
```

```
1.6093439999999999
```

Q: Jenny is 5'9" (5 feet, 9 inches) tall, and has a ladder 210cm long. Can she reach a ceiling 4m high to change a light bulb?

```
(5*12 + 9)inch + 210cm
```

```
3.8526m
```

A: Her head won't quite hit the ceiling, but Jenny will be able to reach the light with her arms.

1.7.1.3.7 New interfaces

So far we have mostly overloaded existing functions built into `Base` Julia. There is nothing stopping you defining your own functions – in fact you'll likely define more new functions that overload existing ones. Here we also added the `unit` function to help out.

But how did we know to overload the `Base` functions? Well, these functions are *interfaces* used throughout Julia. We can define new interfaces for our new *abstract* types. (Typically, a function we write that accepts some concrete type is not a *generic interface* – it is just a one-off function with a single method).

Now we have our `AbstractLength` interface, we can document how people *implement* it so people could add new types, like `Nanometres` and `Yard`. To *implement the `AbstractLength` interface* you'd need to implement:

- Conversion to and from `Metre`
- Define a method for `unit`
- Optionally, define a `const` unit value and overload `show`

After this, everything will "just work"™. This is the power of interfaces!

1.7.1.4 Other dimensions

We could do the same for time, mass, temperature, etc. But then we need to deal with compound dimensions like area, velocity, energy, etc. Again, we need a clever interface to deal with all this complexity. This (and more) is implemented in the Unitful.jl package.

1.7.2 Making types and functions together – shapes

This time we'll practice making new types together with functions.

For this example we'll look at geometric shapes, starting with two-dimensional shapes.

1.7.2.1 Abstract type and interface

We can start with an abstract type for two-dimensional shapes.

```
abstract type Shape2D
end;
```

Even *before* we talk about what kinds of 2D shapes there are, we want to think about what you can *do* with 2D shapes. For our purposes let's look at finding a couple of properties like perimeter and area.

```
"""
   perimeter(shape::2DShape)

Return the perimeter of `shape`.
```

```
"""
function perimeter
end

"""
   area(shape::2DShape)

Return the area of `shape`.
"""
function area
end;
```

Here we've defined two functions, each with zero methods but some documentation explaining their purpose (interfaces in Julia are informal, and are specified through documentation). You can find the documentation with `?area` at the REPL.

1.7.2.2 Types of 2D shape

There are various kinds of 2D shapes. Let's introduce a few here, and organize them by the number of sides. Here is some triangles:

```
abstract type Triangle <: Shape2D
end

struct EquilateralTriangle{T} <: Triangle
   side::T
end

struct RightAngledTriangle{T} <: Triangle
   height::T
   width::T
end;
```

And here are some quadrilaterals:

```
abstract type Quadrilateral <: Shape2D
end

struct Square{T} <: Quadrilateral
   side::T
end

struct Rectangle{T} <: Quadrilateral
   height::T
   width::T
end;
```

And here is a circle, a direct subtype of `Shape2D`:

```
struct Circle{T} <: Shape2D
   radius::T
end;
```

Sometimes it's useful to **use values which aren't types as type parameters ("Value types")**. One standard example is the `N` in `Array{T,N}` which is just an integer signifying the dimension of the array. Doing this helps the compiler know some parts of the "problem size" at compile time based on the type and can lead to much more efficient code or data storage. It can also let us cover several different sub-cases with one type definition. In our `Shape2D` example we can use it to define a regular polygon with `N` sides as follows:

```
struct RegularPolygon{N, T} <: Shape2D
    side_length::T
end

# and a constructor for convenience
function RegularPolygon{N}(side) where N
   RegularPolygon{N,typeof(side)}(side)
end

# const Square = RegularPolygon{4}  # We already defined squares above
const Pentagon = RegularPolygon{5}
const Hexagon = RegularPolygon{6}
```

`RegularPolygon{6}`

(One limitation of this is that we we can't have `RegularPolgyon{4} <: Quadrilateral` in this generic definition so if `Quadrilateral` was important to our design we may have some thinking to do here.)

1.7.2.3 Implementing the `Shape2D` interface

For each of these shapes we can determine the perimeter:

```
function perimeter(shape::EquilateralTriangle)
   return 3 * shape.side
end

function perimeter(shape::RightAngledTriangle)
   angle = atan(shape.height / shape.width)
   return shape.width + shape.height + shape.width / cos(angle)
end

function perimeter(shape::Square)
   return 4 * shape.side
end

function perimeter(shape::Rectangle)
   return 2 * shape.width + 2 * shape.height
end

function perimeter(shape::Circle)
   return 2  * shape.radius
end

function perimeter(shape::RegularPolygon{N, T}) where {N, T}
    return N * shape.side_length
end;
```

And the area:

```
function area(shape::EquilateralTriangle)
   return 0.5 * sind(60) * shape.side * shape.side
end

function area(shape::RightAngledTriangle)
   return 0.5 * shape.width + shape.height
end

function area(shape::Square)
   return shape.side * shape.side
end
```

```
function area(shape::Rectangle)
   return shape.width * shape.height
end

function area(shape::Circle)
   return   * shape.radius * shape.radius
end

function area(shape::RegularPolygon{N, T}) where {N, T}
    return  (N * shape.side_length^2) / (4 * tand(180/N))
end;
```

We can use these pretty straightforwardly.

```
@show perimeter(Rectangle(3.5, 2.0))
@show perimeter(Hexagon(1.5))
@show area(RightAngledTriangle(10, 10));
```

```
perimeter(Rectangle(3.5, 2.0)) = 11.0
perimeter(Hexagon(1.5)) = 9.0
area(RightAngledTriangle(10, 10)) = 15.0
```

At any point in time we can introduce new subtypes of `Shape2D`, `Triangle` or `Rectangle` so long as we define their perimeter and area. The person who wrote `Shape2d` does not even need to be aware.

1.7.3 Combining interfaces

Today we have introduced two new abstract types with their own interfaces. One for units of length, and one for shapes.

We have said Julia is good for quickly writing mathematical code, and good for writing code that executes fast. Here we can see how Julia "scales" well when combining code together – everything will "just work".

Take this:

```
perimeter(Square(4cm))
```

```
16cm
```

Here, the code that handles `perimeter` doesn't need to know anything about `AbstractLength`. The only thing it relies on is that the lengths in question work with `+`, `*`, `/`, etc.

```
perimeter(Circle(10.0cm))
```

```
62.83185307179586cm
```

However, in our units implementation we can't (yet) multiply lengths to get units of area (dimensions of length squared).

```
area(Circle(10.0cm))
```

```
MethodError: MethodError(*, (31.41592653589793cm, 10.0cm), 0x00000000000099ba)
MethodError: no method matching *(::Centimetre{Float64}, ::Centimetre{Float64})
The function `*` exists, but no method is defined for this combination of argument types.
Closest candidates are:
  *(::Any, ::Any, ::Any, ::Any...)
   @ Base operators.jl:642
  *(::Real, ::AbstractLength)
   @ Main.Notebook
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qmd:1778
```

```
  *(::SpecialFunctions.SimplePoly, ::Any)
   @ SpecialFunctions ~/.julia/packages/SpecialFunctions/pIm7V/src/expint.jl:8
  ...
Stacktrace:
 [1] *
   @ ./operators.jl:642 [inlined]
 [2] area(shape::Circle{Centimetre{Float64}})
   @ Main.Notebook
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qmd:2023
 [3] top-level scope
   @
~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lecture-unit-5.qmd:2062
```

If instead we used a more thorough implementation of units, like Unitful.jl, this will work fine.

```
using Unitful

area(Circle(10.0u"cm"))
```

```
314.1592653589793 cm^2
```

1.8 Going forward

We are now done with the material for Unit 5. We have learned:

1. What are the "concrete" types represents sets of values in Julia?
   - `primitive type`
   - `struct`, `Tuple`, `NamedTuple`
   - `Array`, `String`
2. What are the "abstract" types that represent sets of types in Julia?
   - `abstract type`
   - Generic type parameters, `UnionAll`
   - `Union` of types
3. How do functions work in Julia?
   - functions and closures
   - methods
   - multiple dispatch
4. How do we create types and interfaces in Julia?
   - length units
   - 2D shapes
   - how interfaces compose naturally

This will place you in good stead for Units 6-8, and in particular Projects 1-3 where you will need to craft non-trivial programs to create a solution.

1.9 Bonus material

Consider the following material optional. Have a look through if you are curious, have extra time, or would like extra exposure and practice to defining types and interfaces.

1.9.1 Online statistics

Lets consider an example where we want to collect some quick running statistics from some data coming in. E.g.:

```
using Statistics

# A function that returns a data point - could be streamed from disk or the internet
fetch_new_data() = 100*rand()
```

```
function print_running_stats(n::Integer, running_stats_storage)
   for i in 1:n
   # Get some data
   data_point = fetch_new_data()

   # Collect data for statistics
   push!(running_stats_storage, data_point)

   # Periodically we look at summary statistics
   if i % 20 == 0
   println("-------")
   println("Count: ", length(running_stats_storage))
   println("Mean: ", mean(running_stats_storage))
   println("Max: ", maximum(running_stats_storage))
   end
   end
end

Random.seed!(0)
running_stats_storage = Float64[] # Some place to hold the running data
print_running_stats(100, running_stats_storage)
```

```
-------
Count: 20
Mean: 43.111155613892045
Max: 90.47275767596541
-------
Count: 40
Mean: 44.657640026434834
Max: 90.47275767596541
-------
Count: 60
Mean: 47.48326993372904
Max: 98.13388392678519
-------
Count: 80
Mean: 46.553989816155244
Max: 98.13388392678519
-------
Count: 100
Mean: 45.13778124972068
Max: 98.13388392678519
```

Consider the scenario where you have a lot of data to stream and `n` gets very big… so you don't want to recompute the mean and max every time. In fact — you might not even want to store the a copy of the data in RAM at all!

Here's an approach to do "online" statistics:

```
mutable struct RunningStats
   count::Int
   sum::Float64
   max::Float64

   RunningStats() = new(0, 0.0, -Inf)
end
```

```
running_stats_storage = RunningStats()
```

```
RunningStats(0, 0.0, -Inf)
```

We can now make specific methods for `push!`, `length`, `sum`, `mean` and `maximum` for this new type:

```
import Base: push!, length, sum, maximum
import Statistics: mean

length(rsd::RunningStats) = rsd.count
sum(rsd::RunningStats) = rsd.sum
mean(rsd::RunningStats) = rsd.sum / rsd.count
maximum(rsd::RunningStats) = rsd.max

function push!(rsd::RunningStats, data_point)
   # Update the count
   rsd.count += 1

   # Update the sum
   rsd.sum += data_point

   # Update the maximum
   if rsd.max < data_point
   rsd.max = data_point
   end
end

Random.seed!(0)
running_stats_storage = RunningStats()
print_running_stats(100, running_stats_storage)
```

```
-------
Count: 20
Mean: 43.111155613892045
Max: 90.47275767596541
-------
Count: 40
Mean: 44.65764002643482
Max: 90.47275767596541
-------
Count: 60
Mean: 47.483269933729034
Max: 98.13388392678519
-------
Count: 80
Mean: 46.553989816155244
Max: 98.13388392678519
-------
Count: 100
Mean: 45.137781249720675
Max: 98.13388392678519
```

1.9.1.1 Running statistics of other types

Going a bit more generic we could have also had,

```
mutable struct FlexRunningStats{T <: Number}
   data::Vector{T}
```

```
    count::Int
    sum::T
    max::T

    FlexRunningStats{T}() where {T} = new{T}(T[], 0, zero(T), typemin(T))
end

length(rsd::FlexRunningStats) = rsd.count
sum(rsd::FlexRunningStats) = rsd.sum
mean(rsd::FlexRunningStats) = rsd.sum / rsd.count
maximum(rsd::FlexRunningStats) = rsd.max

function push!(rsd::FlexRunningStats, data_point)
    # Insert the new datapoint
    push!(rsd.data, data_point)

    # Update the count
    rsd.count += 1

    # Update the sum
    rsd.sum += data_point

    # Update the maximum
    if rsd.max < data_point
    rsd.max = data_point
    end
end

fetch_new_data() = rand(0:10^4) # override this to return integers

Random.seed!(0)
running_stats_storage = FlexRunningStats{Int}()
print_running_stats(100, running_stats_storage)
```

```
-------
Count: 20
Mean: 4310.95
Max: 9048
-------
Count: 40
Mean: 4465.675
Max: 9048
-------
Count: 60
Mean: 4748.3
Max: 9814
-------
Count: 80
Mean: 4655.375
Max: 9814
-------
Count: 100
Mean: 4513.76
Max: 9814
```

```
@which mean(running_stats_storage)
```

```
mean(rsd::Main.Notebook.FlexRunningStats)
     @ Main.Notebook ~/src/ProgrammingCourse-with-Julia-SimulationAnalysisAndLearningSystems/quarto/lec
```

But there is a problem with the above. What if `T` was `Complex`? Should we have done `<: Number` or `<: Real`?

In generic code its good to consider what *interfaces* we want to use, so we can figure out which type constraints are appropriate. Julia will go ahead and assemble the pieces of your program together into a coherent whole.

1.9.2 Interfaces

So above we used a few interface methods to implement our online-statistics accumulator.

- `push!` - add an element to a collection (mutates the input)
- `length` - the number of element in a collection
- `sum` - add up all the element in a collection
- `maximum` - the maximum number of elements in a collection
- `mean` - the mean of all the elements in a collection

Let's have a look at some common interfaces

1.9.2.1 Numbers

- Common arithmetic operations `+`, `-`, `*`, `/`, etc.
- `promote(x, y)` - take two numbers and return two numbers of the same type, e.g `promote_type(3.14, 1) = (3.14, 1.0)`
- `promote_rule(T1, T2)` - e.g. `promote_rule(Float64, Int64) = Float64`.

Some subtypes of `Number` like `AbstractFloat` have more interface methods.

1.9.2.2 Iterables

- `iterate(iter)` - used in `for` loops
- `length(iter)` - the number of elements to iterate
- `eltype(iter)` - the type of the elements to iterate

Some things iterate but we might not have known length or element type, so there you can define some *traits* to describe these facts:

- `IteratorSize(iter)` - return whether `length` is known
- `IteratorEltype(iter)` - return whether `eltype` is known

1.9.2.3 Arrays

Arrays are iterable and satisfy the the above. They are also indexable

- `getindex(a, i)` - the function behind `a[i]`
- `setindex!(a, v, i)` - the function behind `a[i] = v`
- `size(a)` - gives a tuple of sizes, e.g. `size([1,2,3]) = (3,)`

In fact you can create a custom array in Julia with just those methods above! Things like ranges `1:10` are subtypes of `AbstractArray`.

Some types like `Vector` are resizable and for these types there are functions like `push!`, `pop!`, `insert!`, `deleteat!`, `append!`, `empty!`, etc to manipulate them as arbitrary-sized lists.

Arrays are also defined as arithmetic objects according to linear algebra, and therefore have `+`, `-`, `*`, `/`, etc defined. The `LinearAlgebra` standard library package has lots of functions to invert or decompose matrices, find eigenvalues, etc.

1.9.2.4 Sets

Julia has a `Set{T}` type and an `AbstractSet{T}` supertype. Sets are iterable and support things like:

- `in`
- `union`
- `intersect`
- `setdiff`
- `symdiff`

Here's where interfaces get interesting. Things like `Array` also support `in`, but it is slow (must check every element). With `Set` lookup is fast (e.g. O(1) instead of O(N)) - so the operations above are fast.

1.9.2.5 Dictionaries

Julia has a `Dict{K, T}` type and an `AbstractDict{K, T}` supertype. Like arrays, dictionaries are iterable and indexable. You can imagine they are like a `Set` but have associate values (in fact their implementation is related).

- `keys(dict)` - return the set of keys
- `haskey(dict, key)` - check if a key exists in the dictionary
- `getindex(dict, key)` - get a value associated with a key
- `setindex!(dict, value, key)` - insert or update a key
- `delete!(dict, key)` - remove a key
- `keytype(dict)` - the type of the dictionary keys
- `valtype(dict)` - the type of the dictionary values

You generally iterate a dictionary by specifying if you want to iterate the keys, values, or both:

- `pairs(dict)` - an iterable of `key => value` pairs (the default)
- `keys(dict)` - an iterable of the dictionary keys only
- `values(dict)` - an iterable of the dictionary values only

1.9.3 Using structs for complex, nested data structures

You can make interesting data structures by referencing other instances of yourself, like this tree structure below:

```
Random.seed!(0)

struct Node
   id::UInt16
   friends::Vector{Node}
   Node() = new(rand(UInt16), [])
   Node(friend::Node) = new(rand(UInt16),[friend])
end

"""
Makes 'n` children to node, each with a single friend
"""
function make_children(node::Node, n::Int, friend::Node)
   for _ in 1:n
   new_node = Node(friend)
   push!(node.friends, new_node)
   end
end

root = Node()
make_children(root, 3, root)
for node in root.friends
```

```
    make_children(node, 2,root)
end
root
```

```
Node(0x67db, Node[Node(0x118c, Node[Node(#= circular reference @-4 =#), Node(0xa95f, Node[Node(#= circul
```

1.9.4 More to be covered as part of Unit 6:

See constructors in Julia docs. More on this in Unit 6.

See conversion and promotion in Julia docs

See interfaces in Julia docs