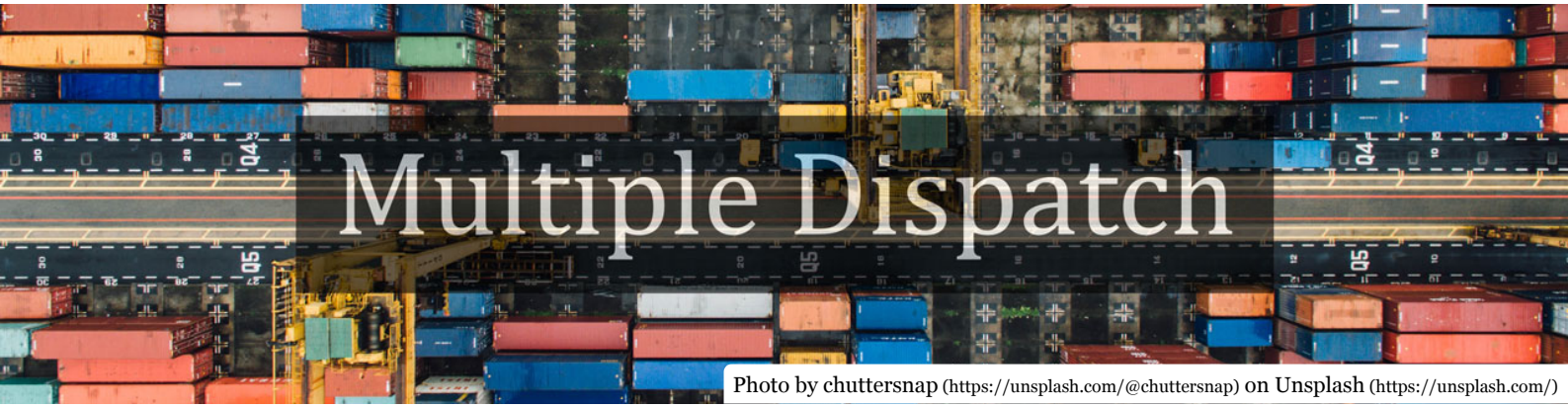

Photo by chuttersnap (https://unsplash.com/@chuttersnap) on Unsplash (https://unsplash.com/)

# Multiple Dispatch

6 minute read

In this lesson we will talk about **type annotations** and **multiple dispatch**. We will learn how it is possible to add instructions to our functions to make them behave differently according to the number and type of parameters given.

# Type annotations

Please refer to this (https://techytok.com/lesson-types) lesson for a quick recap on types. The type, as the name suggests, specifies how a variable should be handled: a `Number` has a certain behaviour when summed or subtracted, a `Matrix` behaves differently when multiplied (we have the matrix multiplication operation) and `Strings` cannot be summed, but can be truncated, concatenated and handled in many different ways.

Julia is based on many different types and even if you don't explicitly invoke them, **the REPL is in charge of inferring the type of each end every variable**, each return value, **and compile optimised machine code** like a C compiler would. This is also why it is important to have type stable code (see this (https://techytok.com/code-optimisation-in-julia/#type-stability) lesson): if the type of a variable or the returned value of a function can not be determined, Julia won't be able to compile efficient machine code and your program will run extremely slowly.

Julia is an **optionally typed** language, which means that it is possible to specify the type of a parameter given to a function, but this is **not mandatory** and in general it is **not advised unless you need to**.

# What are type annotations?

Let's consider a simple function:

```
function f1(x)
    return 42 + x
end
```

We can call `f1(x)` using any `x` for which the addition with a number is defined.

If we try to call `f1("TechyTok")` an error is returned since there is no way to add a `String` to a `Number`. To explicitly state that `x` must be a `Number`, we use the `::` operator:

```
function f2(x::Number)
    return 42 + x
end
```

Let's try it in the REPL:

```
>>>f2(3)
45

>>>f2(3.14)
45.14

>>>f2("TechyTok")
ERROR: MethodError: no method matching f2(::String)
```

As we can see, an error is explicitly thrown when we try to pass a string to `f2`. The typing is optional, in the sense that even if we call `f1("TechyTok")` we get a pretty similar error:

```
>>>f1("TechyTok")
ERROR: MethodError: no method matching +(::Int64, ::String)
```

There is no performance gain in adding type annotations as long as a function is type stable (https://techytok.com/code-optimisation-in-julia/#type-stability). In my opinion, they may make debugging easier and the code may become more readable if you specify that a variable must be a `Number` rather than a `String`, a `Dict` or something else.

# Abstract vs concrete types

While adding type annotations, you should try to use abstract types whenever you can, in this way your code will be more general. There is no need to specify whether a `Number` is a `Float64` or an `Int`, Julia will figure it out by itself. Furthermore, if an abstract type is used, your function will work with any properly defined child type. Since users can implement their own types and sub-types, there is no need to limit our code and restrict the types on which our function can work unless there is a good reason to do so. At best the code will work without any further tuning, at worst the user will receive an error and will know that something must be done.

So, as a rule of thumb, **don't add type annotations, unless they are general or needed for a specific reason**.

# Multiple dispatch

Multiple dispatch is the practice of having a function behave differently according to the number and the type of the parameters which it receives.

Multiple dispatch is one of the cases where type annotations are needed. With the addition of type annotations, it is possible to write multiple definitions for a single function which will behave differently.

In Julia the "name" of a function is "the function", while an implementation of said function is called a method of the function. A single function may have different methods: for example the `+` (plus) function has a method to deal with each concrete type: it will behave differently when you sum two integers or two floating point numbers.

We have already seen an example of multiple dispatch in the lesson about types (https://techytok.com/lesson-types). We will now explore another example.

Let's write a new function which will behave differently when we pass a `Number` or a `String`:

```
function f3(x::Number)
    return x^2 + 42
end

function f3(x::String)
    println("Hello! I see you like $x")
end
```

Now I can call `f3`:

```
>>>f3(1)
43

>>>f3("TechyTok")
Hello! I see you like TechyTok
```

If I try to call `f3` with an `x` which is neither a `Number` or a `String`, an error will be returned:

```
>>>f3([1,2,3])
ERROR: MethodError: no method matching f3(::Array{Int64,1})
```

Notice that we can always use the broadcasting syntax to compute `f3` element-wise:

```
>>>f3.([1,2,3])
3-element Array{Int64,1}:
 43
 46
 51

>>>f3.(["apples", "oranges", "hugs"])
Hello! I see you like apples
Hello! I see you like oranges
Hello! I see you like hugs
```

Let's define a generic method which will work on everything which is not a `Number` or a `String`:

```
function f3(x)
    println("type of x: $(typeof(x))")
end
```

```
>>>f3([1,2,3])
type of x: Array{Int64,1}
```

We could also define a new method which uses a different number of parameters:

```
function f3(x, y)
        return x + 3*y
end
```

```
>>>f3(1,3.14)
10.42
```

In this case I've decide not to add type annotations and this function will work with any type for which addition and multiplication by a scalar is defined. For example, it will work for two arrays with the same length:

```
>>>f3([1,2,3],[4,5,6])
3-element Array{Int64,1}:
 13
 17
 21
```

Multiple dispatch is a simple but powerful concept. Understanding and mastering it is one of the keys to write better code in Julia. Since the methods for a single function can be extremely different, it enables us to keep names concise and simple while differentiating the implementation for each use case.

To list all the methods available for a function you can use `methods(fname)`:

```
>>>methods(f3)
# 4 methods for generic function "f3":
[1] f3(x::String) in Main at REPL[2]:2
[2] f3(x::Number) in Main at REPL[1]:2
[3] f3(x) in Main at REPL[3]:2
[4] f3(x, y) in Main at REPL[4]:2
```

You can check which method will be used by a function call using the `@which` macro:

```
>>>@which f3(2)
f3(x::Number)
```

# Adding new methods to existing functions

While adding new methods to functions we have defined ourselves is useful, multiple dispatch is not limited to user defined functions. We can also define new methods for existing functions defined in other libraries.

It is possible, for example, to extend the `+` function to concatenate `Strings`. In order to extend a function first we need to `import` it, then we can extend it adding a new method:

```
import Base.+
+(x::String, y::String) = "$x$y"

>>>"Techy"+"Tok"
"TechyTok"
```

This extension is not particularly useful, since there is a function explicitly designed to concatenate strings, but it is nonetheless possible. For the sake of completeness, the same result can be obtained using `join` or `*`

```
>>>join(["Techy","Tok"])
"TechyTok"

>>> "Techy"*"Tok"
"TechyTok"
```

Furthermore, we can add a new method to an existing function inside a module. When we import that module, we will be able to use the “old” function with the new method too.

```
module TestMe
import Base.+

+(x::String, y::String) = "$x$y"

end
```

```
using .TestMe

>>>"Techy"+"Tok"
"TechyTok"
```

# Conclusions

In this lesson we have learned how to add **type annotations** to our functions and how it is possible to **add new methods to existing functions**, which is called **multiple dispatch**. We have also seen how you can add a new method to an existing function inside a module and how to seamlessly use the new method once the module is imported.

If you liked this lesson and you would like to receive further updates on what is being published on this website, I encourage you to subscribe to the **newsletter** (https://techytok.com/newsletter/)! If you have any **question** or **suggestion**, please post them in the **discussion below**!

Thank you for reading this lesson and see you soon on TechyTok!

**Tags:** Julia

**Categories:** Lessons Tutorial

**Updated:** May 25, 2020

**LEAVE A COMMENT**

ALSO ON **TECHYTOK**

### Variables and Types

8 months ago • 1 comment

From zero to Julia Lesson 1. An introduction to variables and Types

### Control Flow

8 months ago • 5 comments

From zero to Julia Lesson 4. Control Flow

### Parallel computin

7 months ago • 4 comm

From zero to Julia Le 15. Parallel computin



/