



# Functions

Photo by Filip Varga (<https://unsplash.com/@vargafilep123>) on Unsplash (<https://unsplash.com>)

## Functions

🕒 6 minute read

Functions are the main building blocks in Julia. Every operation on variables and other elements is performed through functions, even the mathematical operators that we have seen in the chapter before are functions in an infix form.

If we want to simplify what a function is, we can picture it as a box which can optionally take an input, performs some operations and returns an output of some sort.

## Defining functions

A typical function looks like this:

```
1 function plus_two(x)
2     #perform some operations
3     return x + 2
4 end
```

A `function` starts with the word `function` and ends with `end`.

Please notice the **indentation** of the body of the function (lines 2 and 3): the body of a function (and many other constructs in Julia) must be indented. Although the number of indentation spaces is not strictly fixed, it is a conventional rule to **indent by four spaces**.

In Julia the indentation rules are not as strict as in Python, nonetheless it is recommended that you don't mix tabs and spaces while indenting you code, and tabs are usually discouraged as they may lead to problems when switching from one operating system to another.

Please notice that every “block” of code ends with an `end`, this way you can always know when something starts and ends.

Although this is the most common way to write functions, it is sometimes convenient to use the inline version:

```
1 | plus_two(x) = x+2
```

It is also possible to create anonymous functions (like lambdas in Python) using the following structure:

```
1 | plus_two = x -> x+2
```

It is **not recommended** to use anonymous functions unless they are really simple. It is generally better to write functions in the first or second way unless you need to write a small wrapper around another function and pass it to a third function.

The following example is a bit more advanced and it includes topics which we have not yet studied, like integrals and how to install and use an external package. You can always come back later to read this example as it is not fundamental at this point of the course but you may find it useful one day. The take home message is **never use anonymous functions unless you know what you are doing**.

## More on anonymous functions

For example, let's consider a function  $f$  of three variables  $x$ ,  $y$  and  $z$ . Let's suppose we want to fix two variables ( $y$  and  $z$ ) and integrate  $f$  over  $x$ , we could do it with:

```
1 | using Pkg
2 | Pkg.add("QuadGK")
3 | using QuadGK
4 |
5 | f(x,y,z) = (x^2 + 2y)*z
6 |
7 | quadgk(x->f(x,42,4), 3, 4)
```

but we could have also written:

```
1 | using Pkg
2 | Pkg.add("QuadGK")
3 | using QuadGK
4 |
5 | f(x,y,z) = (x^2 + 2y)*z
6 | arg(x) = f(x,42,4)
7 |
8 | quadgk(arg, 3, 4)
```

and the result would have been the same, furthermore, in my opinion, the latter is a cleaner way to write integrals, especially if we wrap everything inside another function:

```

1  f(x,y,z) = (x^2 + 2y)*z
2
3  function integral_of_f(y,z)
4      arg(x) = f(x,y,z)
5      result = quadgk(arg, 3, 4)
6      return result
7  end

```

Functions can be defined inside other functions, which is really useful to make the code more readable. Avoid writing long functions inside other functions, but use small lightweight functions if needed (think of them as if they were aliases to make the code more readable and simple to maintain). When we define a function inside another function, we can use simpler and shorter names (like `arg` or `f`) without the risk of having overlapping functions and unexpected behaviour.

## Void functions

Functions may also take no arguments and return no value, if needed, for example we can create a function which prints a string:

```

1  function say_hi()
2      println("Hello from TechyTok!")
3      return
4  end

```

If the function returns no value the `return` can be omitted. In general it is also possible to omit the `return` statement even in regular functions and Julia will return the last computed value. However, in my opinion, in this case it is better to return explicitly a value, for clarity's sake.

## Optional positional arguments

Sometimes a parameter may have a default value which can be specified so that the user doesn't need to always type it. For example let's write a function which converts our "weight" as measured on Earth (in kg) to the one measured on another planet.

```

1  function myWeight(weightOnEarth, g=9.81)
2      return weightOnEarth*g/9.81
3  end

```

If the user types `myWeight(60)` and doesn't specify `g`, he or she will get his/her weight as measured on Earth:

```

1  >>>myWeight(60)
2  60

```

If I specify the gravitational acceleration `g` on another planet (let's say Mars)

```
1 | >>>myWeight(60, 3.72) # https://en.wikipedia.org/wiki/Gravity\_of\_Mars
2 | .https://en.wikipedia.org/wiki/Gravity\_of\_Mars.
   | 22.75
```

I get my weight on Mars! Pretty light, isn't it?

As the name suggests **positional arguments** must be used in the right order, we cannot specify `g` before `weightOnEarth` and, as opposed to other languages like Python, in Julia we cannot change the order of the arguments even if we specify the name of the parameter. If we want optional arguments with no fixed position, we need to use **keyword arguments**.

## Keyword arguments

Let's say we have a function which takes many optional parameters, it might become cumbersome for the user to remember the right order. That's the case where keyword arguments come in handy!

Keyword arguments are separated from positional arguments by a semicolon `;` and must always be addressed by their name, although their order is irrelevant. They can be either optional and not, but usually we use keyword arguments for optional parameters.

```
1 | function my_long_function(a, b=2; c, d=3)
2 |     return a + b + c + d
3 | end
```

Here `a` and `b` are positional arguments, while `c` and `d` are keyword arguments.

```
1 | >>>my_long_function(1, c=3)
2 | 9
3 |
4 | >>>my_long_function(1, 2, c=3)
5 | 9
6 |
7 | >>>my_long_function(1, 2, d=5, c=3)
8 | 11
9 |
10 | >>>my_long_function(1, 2, d=5)
11 | ERROR: UndefKeywordError: keyword argument c not assigned
```

As you can see, even if `c` is a keyword argument, it must always be specified!

## Performance tip

It is preferable to use positional arguments whenever high performance is required (for example when writing functions which need to be called many times), it is always possible to write a wrapper to a function (or a method of the function) which uses keyword arguments, if needed, as we will see later on when we deal with **multiple dispatch**.

# Function documentation

It is possible to get information on a function by typing in the REPL `? functionName`, this will fetch the documentation for that function, if available, or it will print some information which the compiler is able to infer.

For example if we query the documentation of the `sin` function with `? sin` we get this output:

```
help?> sin
search: sin sinh sind sinc sinpi sincos sincosd asin using isinf asinh asind isinteger

sin(x)

Compute sine of x, where x is in radians.

-----

sin(A::AbstractMatrix)

Compute the matrix sine of a square matrix A.

If A is symmetric or Hermitian, its eigendecomposition (eigen) is used to compute
the sine. Otherwise, the sine is determined by calling exp.

Examples
=====

julia> sin(fill(1.0, (2,2)))
2x2 Array{Float64,2}:
 0.454649  0.454649
 0.454649  0.454649
```

techytok.com

It is also possible to write the documentation of your functions in the following way:

```
1  """
2  Description of the function
3  """
4  function foo(x)
5      #... function implementation
6  end
```


You can find more information on how to write the code documentation in the [lesson on modules](https://techytok.com/lesson-modules/#code-documentation) (<https://techytok.com/lesson-modules/#code-documentation>) and at the [official documentation](https://docs.julialang.org/en/v1/manual/documentation/index.html) (<https://docs.julialang.org/en/v1/manual/documentation/index.html>).

# Conclusions


In this lesson we learned how to define functions and use positional as well as keyword arguments.

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:** November 3, 2019

---

LEAVE A COMMENT

## Control Flow

8 months ago • 5 comments

From zero to Julia Lesson 4.  
Control Flow

## Variable Scope

7 months ago • 4 comments

From zero to Julia Lesson 6.  
Variable scope

## Variables and Typ

8 months ago • 1 comm

From zero to Julia Le  
An introduction to var  
and Types

2 Comments

techytok

 Privacy Policy

 1 Login ▾

 Recommend

 Tweet

 Share

Sort by Best ▾

 Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 



joe • 6 months ago • edited

Could you further explain this line, please?

```
quadgk(x->f(x,42,4), 3, 4)
```

What is going to be the value of x here?

^ | ▾ • Reply • Share ›



aureamerio Mod ➔ joe • 6 months ago

OK, there is a small misunderstanding about anonymous functions.  
For more information on anonymous functions, I advise you to read  
the [official documentation](#)

`x->f(x,42,4)` is an anonymous function declaration. It means  
"create a function of x and return `f(x,42,4)`", so basically we pass as  
a first argument to `quadgk` a function.