Functions

A function was the very first thing you learned when starting with Julia. You have learned how to use arithmetic functions and functions that manipulate your data or create a set. They are useful and sometimes complicated. Julia would be nothing without its functions. So I think it's time you learned how to make them.

User defined functions

Before you learn the syntax of writing a function, I should cover what a function actually is. A function is a bit of software that you can call upon, more than once, to perform some sort of action. They don't necessarily have an output. Their purpose is to split up long sections of code and be called from anywhere in your code.

A function can have all sorts of code inside of it, like loops or other functions. A function can even call itself, which is called a recursive function.

When writing a function that mutates the input values, make sure to use a bang after the function name, just like sort! or pop!.

Syntax of a function

A function begins with the keyword function. Then it is followed by the function name, and any input values in parentheses. Indent once, write the body of the function, and finish with the keyword end.

```
julia> function basicfunction(x, y)
           println("The user input " * string(x) * "and " * string(y))
           return y
       end
basicfunction (generic function with 1 method)
```

Here is my very simple function. It's even called "basic function". It takes two input values and assigns them the name x and y. x and y are both local variables and only used inside the function. Then, the function would print a bit of text and return only the second value.

The example above doesn't actually do anything with the function; none of the code is executed. It only creates it, to be called later. This makes it really easy to have a bug in your function, which needs to be solved with testing.

```
julia> basicfunction(5, "loser")
The user input 5and loser
"loser"
```

When I call the function with the input values of 5 and “loser”, the output is a println and the second input value. That was exactly what I expected, so my code doesn’t have any bugs. But wait, when I look a bit more carefully, the printed string is a little janky. It says “The user input **5and** loser” when the 5 and the word “and” should probably be separate. This isn’t a world-ending problem, but your code looks a lot better without it. It’s a quick fix and I call my function a second time.

```
julia> basicfunction(2.67, true)
The user input 2.67 and true
true
```

This time I used a float and a boolean, to make sure those work in my function too.

Inline functions

Functions can also be written in one line where some keywords can be omitted. These are called inline functions and are used to write simpler functions in less time.

The syntax for an inline function is the function name, followed by an equal sign and the actual function at the end. No end keyword is needed.

```
julia> F(x) = x^2
F (generic function with 1 method)

julia> F(5)
25
```

The function in the example above works as a shorthand way to find the square of a number. Inline functions are pretty simple and are just another way to write a function.

Anonymous functions

If you’re looking for an even simpler way to write a function, anonymous functions might be what you need. They are a function without a name. The primary use for anonymous functions is to be used as arguments for other functions. Some functions, like map (which applies a function to multiple values), take other functions as arguments, and this is where anonymous functions shine.

```
julia> function (x)
           x^2 + 2x - 1
       end
#4 (generic function with 1 method)
```

Anonymous functions are still generic functions, but they're given a name generated by Julia. You can't actually call them, however. The syntax for an inline anonymous function replaces the equal sign with an arrow.

```
julia> x -> x^2 + 2x - 1
#7 (generic function with 1 method)
```

Anonymous functions work with multiple arguments, just like any other function. Type declarations for anonymous functions work as well, which is the next thing we will learn.

```
julia> (x,y,z) -> 2x + y - z
#10 (generic function with 1 method)
```

Methods/Type declarations

However, if I want to restrict the input type, make sure only *some* types work, I can do that. To specify a type, one can use what is called a type declaration. The operator is written with two colons (**::**) and is read as "is an instance of". Instead of working like the function **isa**, which checks if an argument is of a certain type, it calls an error when the argument isn't of the specific type. It kind of acts as a border checkpoint, ensuring that only the acceptable types pass, and stopping any other type.

```
julia> 1::Int64
1

julia> 1::Float64
ERROR: TypeError: in typeassert, expected Float64, got a value of type Int64
```

So when I write "1 is an instance of Float64", the computer freaks out, because 1 is *not* an instance of Float64.

Type declaration can be used in a function to specify input types, but they can also be used to create a function with multiple methods. A method is one possible behavior for a function, and you select which method to use with the arguments. Functions can do different things depending on the type of input. The addition function (+) has many different methods, and the specific method is determined when you decide to add a float to an integer, or two booleans.

To see the number of methods of one function, you just enter the function name, and then either execute the line, or print the output. For example, here's the addition function:

```
julia> +
+ (generic function with 154 methods)
```

154 seems like a lot, but if you need to cover every possible combination of inputs, it might be necessary. If you want to see the specific methods, you can use the methods function.

```
julia> methods(+)
# 154 methods for generic function "+" from Base:
   [1] +(B::BitMatrix, J::LinearAlgebra.UniformScaling)
     @ C:\Users\1254443\AppData\Local\Programs\Julia-1.12.1\shar
:154
   [2] +(x::Bool, z::Complex{Bool})
     @ complex.jl:308
   [3] +(x::Bool, y::Bool)
     @ bool.jl:168
   [4] +(x::Bool)
     @ bool.jl:165
   [5] +(x::Bool, z::Complex)
     @ complex.jl:315
   [6] +(x::Bool, y::T) where T<:AbstractFloat
     @ bool.jl:175
   [7] +(x::Rational{BigInt}, y::Rational{BigInt})
     @ gmp.jl:1062
   [8] +(::Missing, ::Missing)
     @ missing.jl:122
   [9] +(::Missing)
     @ missing.jl:101
  [10] +(::Missing, ::Number)
     @ missing.jl:123
  [11] +(c::BigInt, x::BigFloat)
     @ mpfr.jl:613
  [12] +(x::BigInt, y::BigInt)
     @ gmp.jl:502
  [13] +(a::BigInt, b::BigInt, c::BigInt)
     @ gmp.jl:542
  [14] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt)
     @ gmp.jl:543
  [15] +(a::BigInt, b::BigInt, c::BigInt, d::BigInt, e::BigInt)
     @ gmp.jl:544
  [16] +(x::BigInt, y::BigInt, rest::BigInt...)
     @ gmp.jl:679
  [17] +(x::BigInt, c::Union{UInt16, UInt32, UInt8})
     @ gmp.jl:550
  [18] +(x::BigInt, c::Union{Int16, Int32, Int8})
     @ gmp.jl:556
  [19] +(level::Base.CoreLogging.LogLevel, inc::Integer)
     @ logging\logging.jl:132
  [20] +(z::Complex{Bool}, x::Bool)
     @ complex.jl:309
  [21] +(z::Complex{Bool}, x::Real)
```

And it goes on.

I won't show every single method, but you can easily find them yourself. You won't usually need 154; just a few will work just fine.

To practice writing functions with multiple methods, let's write a program that has a fun fact about the input's type. I think I will write a method for a string, a float, a boolean, and an integer.

```
julia> function funfact(x::String)
           println("You input a string! \n Did you know that strings can be concatenated with a *?")
       end
funfact (generic function with 1 method)

julia> function funfact(x::Float64)
           println("You input a float! \n Did you know that dividing two integers results in a string?")
       end
funfact (generic function with 2 methods)

julia> function funfact(x::Bool)
           println("You input a boolean! \n Did you know that booleans can be added?")
       end
funfact (generic function with 3 methods)

julia> function funfact(x::Int64)
           println("You input a integer! \n Did you know that Julia has a negative infinity?")
       end
funfact (generic function with 4 methods)
```

In the image above, I wrote all the specific types that I wanted to include. Now if I called the function with one of those four types, I would get a fun fact. What would happen if my argument was of another type?

```
julia> typeof((1, 2, 3))
Tuple{Int64, Int64, Int64}

julia> funfact((1, 2, 3))
ERROR: MethodError: no method matching funfact(::Tuple{Int64, Int64, Int64})
```

I would get an error, which I don't want. I want this function to be usable with every type. This is why I can write one last method as a "catch-all", to ensure that every type is accounted for. The type **Any** is what I need. I will go over all the types in depth next lesson, but for now you only need to know that every type is included in **Any**.

```
julia> function funfact(x::Any)
           println("This type isn't yet accounted for. Choose another one.")
       end
funfact (generic function with 5 methods)

julia> funfact((1, 2, 3))
This type isn't yet accounted for. Choose another one.
```

Now, with 5 methods, every possible type will get some sort of result. The cool thing about coding something is that there are probably another 3 ways to achieve the exact same

thing. I can easily think of another two ways to write a similar program that does the same thing.

Functions with Keyword Arguments

Some functions need a large number of arguments or have a large number of behaviors. Remembering how to call such functions can be difficult. Keyword arguments can make these complex interfaces easier to use and allow arguments to be identified by name instead of only by position.

For example, consider a function plot that plots a line. This function might have many options, for controlling line style, width, color, and so on. If it accepts keyword arguments, a possible call might look like:

```
julia> plot(x, y, width=2) = width
plot (generic function with 2 methods)
```

where we have chosen to specify only line width. This serves two purposes. The call is easier to read, since we can label an argument with its meaning. It also becomes possible to pass any subset of a large number of arguments, in any order, because the argument is identified by its keyword and not location.

Varargs Functions and Splatting

Sometimes it is necessary to write functions with a previously undetermined number of arguments. Such functions are traditionally known as "varargs" functions, which is short for "variable number of arguments". Say, for example, that your function creates an array and then applies a function to all the values. There is no way to predict how many values will be in the array, and it's impractical to write a method for each number of arguments. The solution would be to use a varargs function.

A varargs function is defined by following the last positional argument with an ellipsis.

```
julia> function var(x, y...)
           array = collect(y)
           a = 1
           for i in array
               array[a] = x(array[a])
               a += 1
           end
           return array
       end
var (generic function with 1 method)
```

The example above does exactly what I described above. The function creates an array of all the y values, using the collect function. Then, the x function is applied to each element of the array. The output looks like this:

```
julia> var(sqrt, 1.0, 4.0, 9.0)
3-element Vector{Float64}:
 1.0
 2.0
 3.0

julia> var(string, true, 5.0, 69, (1, 2, 3))
4-element Vector{Any}:
 "true"
 "5.0"
 "69"
 "(1, 2, 3)"
```

An ellipsis can be also used in the function argument. This is called splatting. You may "splat" the values in a collection (like array) as individual arguments when calling a function. Note that the splatted argument must be the very last one. Using the example above, say I wanted to use my function, but with my arguments in an array.

```
julia> arguments = [sqrt, 1.0, 4.0, 9.0]
4-element Vector{Any}:
  sqrt (generic function with 19 methods)
 1.0
 4.0
 9.0
```

Instead of figuring out how to take the values out of the array, I can just splat them, using an ellipsis.

```
julia> var(arguments...)
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

The output of the function looks to be working, so everything executed well.

Lazy Assignment

This part isn't directly tied to functions, but it's in the function section of the Julia Documentation, so here it will go. The material was also a little more complicated for the very beginning, so I couldn't cover it then.

In the very first lesson, I taught you how to assign values to variables. I also taught you how to assign multiple values to multiple variables at once, called multiple assignment.

```
julia> a, b, c, d, e, f = 1, 2, 3, 4, 5, 6
(1, 2, 3, 4, 5, 6)
```

These forms of assignment aren't very complicated, so I won't spend too much time on them, but they will be numbered to better keep track of them.

(1) The first is called destructuring assignment and assigns the values of a range to some variables. If the range doesn't match up with the number of variable names, the last values won't be assigned. You can optionally encircle the variables with parentheses.

```
julia> a, b, c = 1:3
1:3

julia> b
2

julia> a, b, c = 1:4
1:4

julia> c
3
```

(2) An easy way to swap variables is to use multiple assignment.

```
julia> x = 5; y = 6
6

julia> x, y = y, x
(6, 5)

julia> y
5

julia> x
6
```

(3) If only a part of the elements of a range are required, a common convention is to assign ignored elements to a variable consisting of only an underscore. An underscore would otherwise not work as a name, so don't use them for variable names.

```
julia> _, _, _, a = 1:10
1:10

julia> a
4
```

Nothing is actually assigned to the underscore when using this method.

```
julia> _
ERROR: syntax: all-underscore identifiers are write-only and their values cannot be used in expressions
```

(4) The very last assignment method is known as "slurping" and functions in a similar way to splatting. A variable name that is followed by an ellipsis will be assigned a collection of the remaining values.

```
julia> a, b... = "hello!"
"hello!"

julia> a
'h': ASCII/Unicode U+0068 (category Ll: Letter, lowercase)

julia> b
"ello!"
```

Slurping can also occur in any other position, like in the middle or very beginning. This can be useful for extracting the very last value.

```
julia> a, b..., c = [1, 2, 3, 4, 5]
5-element Vector{Int64}:
 1
 2
 3
 4
 5

julia> a
1

julia> b
3-element Vector{Int64}:
 2
 3
 4

julia> c
5
```

Recursive functions

These functions will be the last thing you learn before I move on. A recursive function is one that calls itself from within its own code while executing, which might not make sense, but let me explain.

Recursion is a method of problem solving where the solution comes from solutions to smaller versions of the problem. A recursive algorithm divides the problem into sub-problems, and the output, or answer, of the smaller problems, becomes the input to the bigger problem.

A recursive function would call itself with smaller and smaller value or problems until it reaches the “bottom” or the end.

Let's use an example to help. We can all agree that multiplication is the same as adding a number to itself a certain number of times.

```
julia> 5 * 3 == 5 + 5 + 5
true
```

So, if I wanted to create a function that finds the product of two values, I could write a program that uses recursion and addition.

```
julia> function mult(x, y)
           if y == 1
               #this is the 'bottom' bc x*1 == x
               println("mult($x, $y)\t= $x\n")
               return x
           else
               #this is not the 'bottom' so the function calls itself
               println("mult($x, $y)\t= $x + mult($x, $(y - 1))")
               return x + mult(x, y - 1)
           end
       end
mult (generic function with 1 method)
```

And so I did. I colour-coded everything and will explain each part. The yellow are comments I added for clarity and the println functions will be explained later.

First, you would call this function with two values: **x** and **y**. The program will probably skip the if statement (blue), because **y** is unlikely to be equal to one. The program would then execute the return statement (orange). The computer will return the sum of **x** and **mult(x, y-1),** which is the function that is currently executing. The computer will call the function again, to find the value of **mult(x, y-1)**. Then the same process will happen over again (with the computer trying to return the sum of **x** and the function) until y becomes equal to 1. When y becomes equal to 1, the computer reached the 'bottom' of the loop and now has the answer to the previous function (x, because y ==1 and x * 1 is just x). Then the computer adds the numbers all the way back up, until it has the answer to **mult(x, y-1)** and adds it to **x**.

If that doesn't quite make sense, let's examine the output, with some actual values.

```
julia> mult(4, 6)
mult(4, 6)      = 4 + mult(4, 5)
mult(4, 5)      = 4 + mult(4, 4)
mult(4, 4)      = 4 + mult(4, 3)
mult(4, 3)      = 4 + mult(4, 2)
mult(4, 2)      = 4 + mult(4, 1)
mult(4, 1)      = 4

24
```

Here is the function with arguments of four and six. The computer calls the function, and knows that **mult(4, 6)** is equal to **4 + mult(4, 5).** (Mult, of course, is short for multiplication)

```
mult(4, 6)      = 4 + mult(4, 5)
```

So then the computer calls **mult(4, 5)**, to figure out what that is equal to. It learns that **mult(4, 5)** is equal to **4 + mult(4, 4).**

```
mult(4, 5)      = 4 + mult(4, 4)
```

The function then does this until it reaches **mult(4, 1)**. Because a number times one is always equal to one, the answer to the function is 4. It's reached the 'bottom' of the recursion. With the answer to **mult(4, 1)**, the computer goes back up, finding the answers to everything it called.

The problem with recursion is that it is usually incredibly inefficient and performs far more calculations than necessary. Some practical practice is needed with recursion, to figure out the limits of the system.

Using Functions

In this section we will learn the different ways to use functions, and some ways to combine them.

Map

I have briefly touched on this function in the past but will properly explain it now. Map is a function that transforms a collection by applying a specified function to each element. It works in a similar way to the "dot" operator that I covered in a previous lesson. The syntax for the arguments is to first write the function, then follow it by the collection.

```
julia> array = [1, 4, 9]
3-element Vector{Int64}:
 1
 4
 9

julia> map(sqrt, array)
3-element Vector{Float64}:
 1.0
 2.0
 3.0

julia> sqrt.(array)
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

This function can be used with functions that require multiple arguments, like addition. Simply separate the arguments with parentheses.

```
julia> map(+, [1, 2, 3], [10, 20, 30, 40])
3-element Vector{Int64}:
 11
 22
 33
```

You may notice how the last number was not added to anything. This is because the other array had run out of elements, and the function finished executing. This rule applies to both collections, so it could happen the other way around. Map also has a bang version that permanently mutates the collections.

```
julia> map(+, [1, 2, 3, 4], [10, 20, 30])
3-element Vector{Int64}:
 11
 22
 33
```

```
julia> A = [1, 2, 3];

julia> map!(x -> x^2, A)
3-element Vector{Int64}:
 1
 4
 9

julia> A
3-element Vector{Int64}:
 1
 4
 9
```

In Julia, there is a keyword that lets you write the anonymous function of map in another line. The **do** keyword can be used to skip the long anonymous function in the argument and instead put it on another line. It can be used with all functions that take another function as the first argument (like filter and open), but you aren't familiar with those yet.

To use the keyword, write the function (something like **map**) and skip the first argument (which is another function like **sqrt**). After the closing parentheses, write: `do x`. The **x** is the local variable assigned in the function. It's the same in an anonymous function: `julia> x -> x^2`. After the keyword, you write the body of the function and finish with the keyword end.

```
julia> map([1, 2, 3]) do x
           sqrt(x^2)
       end
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

The keyword isn't incredibly intuitive to use and might not fit grammatically when reading the code aloud. Just remember that it is used to write the function outside of the argument. This is useful when writing a long or complicated anonymous function. The example above can be rewritten as:

```
julia> map(x -> sqrt(x^2), [1, 2, 3])
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

Function Composition and Chaining

This part of Julia has concrete ties in mathematics, specifically while using functions. You'll learn this in Precal 11 or in a later class, depending on your teacher. The function composition operator (∘) is used to combine functions or use multiple at once. This operator applies the last function first, and the output becomes the input of the next function. The syntax is to write your functions in a pair of parentheses, separated by the composition operator, and write the input in another set of parentheses.

```
julia> (sqrt ∘ +)(3, 6)
3.0
```

In this example, the input is first added together, then the square root of the sum is found. Three plus six is nine, and three is the square root of nine, so the two functions return a three.

The example above functions in the same way as:

```
julia> sqrt(+(3, 6))
3.0
```

This operation can be written both ways.

It is also possible to use more than two functions with function composition.

```
julia> (floor ∘ sqrt ∘ +)(3, 5)
2.0
```

Here, the computer finds the sum of three and five. After it finds the square root of the sum, it rounds the result down to the nearest whole number.

Another way to apply multiple functions at once is function chaining. It is sometimes called "piping" or "pipelining". This process uses a combination of a vertical bar and a greater than symbol (|>). It almost looks like we're sending the input to the next function. It works in a similar way as function composition, but the initial input goes at the very beginning.

```
julia>  1:10 |> sum |> sqrt
7.416198487095663
```

In the example above, the program finds the sum of the numbers in the range 1:10. Then it returns the square root of the addition.

Chaining also uses the dot operator. This means that one can apply a function to an entire collection with a period. Or, because of the order, pass an entire collection as an argument. This allows you to best manipulate what happens to your input.

```
julia> 1:3 .|> (x -> x^2) |> sum |> sqrt
3.7416573867739413

julia> ["a", "list", "of", "strings"] .|> [uppercase, reverse, titlecase, length]
4-element Vector{Any}:
  "A"
  "tsil"
  "Of"
 7
```

As you can see in the image above, the dot operator allowed me to apply different functions to a specific element of an array.

I will use the examples from the page above to compare piping, function composition, and just applying multiple functions.

```
julia> sqrt(+(3, 6))
3.0

julia> (sqrt ∘ +)(3, 6)
3.0

julia> (3, 6) |> sum |> sqrt
3.0
```

While the first version might be easier to write, I believe that chaining is more legible and intuitive. These functions all execute in the same manner, however, and it is up to you to decide which one you prefer.

Operators With Special Names

In the Julia Documentation, which you can use for further reading, is a list of “operators” that have a not-obvious function name. I wouldn’t call them operators, but more so processes. Indexing, for example, is one of these processes, and the function name is **getindex**. The first argument is the collection, and the second is the index value.

```
julia> array = [4, 3, 2, 1];

julia> getindex(array, 1)
4

julia> getindex(array, 1) == array[1]
true
```

The process for assigning a value to an index also has a function name. The arguments are the collection name, then the element value, then the index number of said element.

```
julia> array = [4, 3, 2, 1];

julia> setindex!(array, 5, 1)
4-element Vector{Int64}:
 5
 3
 2
 1
```

```
julia> setindex!(array, 5, 1) == array[1] = 5
```

Exercises

04E01 Create a function of your own.

04E02 Create a function that determines if a number is positive, negative, or zero.

04E03 Create a function that mutates an array (a function with a “bang”).

04E04 Create a function that removes the whitespace of a sentence, so that the output would look like this:

```
julia> thefunction("I took the trash out today.")
Itookthetrashouttoday.
```

04E05 Create an inline function.

04E06 Create an inline function that finds the square root of the sum of two arguments.

Ex:

```
julia> thefunction(4, 5)
3.0
```

04E07 Create an anonymous function.

04E08 Create an anonymous function that makes the input positive.

Ex:

```
julia> thefunction(3)
3

julia> thefunction(-5)
5
```

04E09 Write a function that returns an error for an input that isn't a float.

Ex:

```
julia> thefunction(3.0)

julia> thefunction(3)
ERROR: MethodError: no method matching thefunction(::Int64)
```

04E10 Write a function with three different methods.

04E11 Rewrite my example function using only one method and an if block.

This is the example function:

```
julia> function funfact(x::String)
           println("You input a string! \n Did you know that strings can be concatenated with a *?")
       end
funfact (generic function with 1 method)

julia> function funfact(x::Float64)
           println("You input a float! \n Did you know that dividing two integers results in a string?")
       end
funfact (generic function with 2 methods)

julia> function funfact(x::Bool)
           println("You input a boolean! \n Did you know that booleans can be added?")
       end
funfact (generic function with 3 methods)

julia> function funfact(x::Int64)
           println("You input a integer! \n Did you know that Julia has a negative infinity?")
       end
funfact (generic function with 4 methods)
```

04E12 Write a function with a keyword argument.

04E13 Write a varargs function that adds 5 to each argument and puts them into an array.

Ex:

```
julia> thefunction(1, 2, 3)
3-element Vector{Int64}:
 6
 7
 8
```

04E14 Write a function that swaps the values for the argument variables.

Ex:

```
julia> x = 6; y = 5;

julia> myfunction(x, y)
6

julia> x
5

julia> y
6
```

04E15 Write a function that finds the result of x to the power of y using recursion, given that the arguments are x and y. This function should be similar to the example I give in the section of recursion, and can be modeled after the example.

(Hint: Exponents is the same as multiplying a number by itself a certain number of times, much like how multiplication is adding a number to itself a certain number of times.)

Ex:

```
julia> myfunction(4, 3)
64

julia> 4^3
64
```

Here's the example I give for recursion:

```
julia> function mult(x, y)
           if y == 1
               #this is the 'bottom' bc x*1 == x
               println("mult($x, $y)\t= $x\n")
               return x
           else
               #this is not the 'bottom' so the function calls itself
               println("mult($x, $y)\t= $x + mult($x, $(y - 1))")
               return x + mult(x, y - 1)
           end
       end
mult (generic function with 1 method)
```

04E16 Rewrite my example function without using recursion.

Here is the example function:

```
julia> function mult(x, y)
           if y == 1
               #this is the 'bottom' bc x*1 == x
               println("mult($x, $y)\t= $x\n")
               return x
           else
               #this is not the 'bottom' so the function calls itself
               println("mult($x, $y)\t= $x + mult($x, $(y - 1))")
               return x + mult(x, y - 1)
           end
       end
mult (generic function with 1 method)
```

04E17 Write a function that uses the map function to find the square root of an array

Ex:

```
julia> myfunction([1, 4, 9, 16])
4-element Vector{Int64}:
 1
 2
 3
 4
```

04E18 Write a function that uses either function chaining or piping to find the product then the square root of a function.

Ex:

```
julia> myfunction([2, 3, 6])
6.0

julia> sqrt(prod([2, 3, 6]))
6.0
```