3c. Defining Your Own Functions

Martin Alfaro

PhD in Economics

INTRODUCTION

Recall that functions can be classified into *i)* built-in functions, *ii)* third-party functions, and *iii)* user-defined functions. The previous section has covered the first two, and **we now focus on iii)**.

USER-DEFINED FUNCTIONS

The first step to define your own functions is giving names. Function names follow similar rules to variable names. In particular, they accept Unicode characters, enabling the user to define functions such as `∑(x)`. Once you create them, functions can be called without invoking any prefix. This means that a function `foo` can be called by simply executing `foo(x)`.[1]

There are two approaches to defining functions. We'll refer to each as the **standard form** and the **compact form**. The standard form is the most general and allows you to write both short and long functions. On the other hand, the compact form is employed for single-line functions and is reminiscent of mathematical definitions. To illustrate each form, consider a function `foo` that sums two variables `x` and `y`.

STANDARD FORM

```
function foo(x,y)
    x + y
end
```

COMPACT FORM

```
foo(x,y) = x + y
```

The output of the compact form is given by the only operation contained. For its part, the standard form defaults to returning the last line as its output, allowing you to specify the output via the keyword `return`. To return multiple outputs, you can use a collection, with tuples being the most common choice. [2]

These approaches to specifying an output are illustrated below.

EXPLICIT OUTPUT

```
function foo(x,y)
    term1 = x + y
    term2 = x * y


    return term2
end
```

julia> `foo(10,2)`
20

IMPLICIT OUTPUT

```
function foo(x,y)
    term1 = x + y
    term2 = x * y          # output returned
end
```

julia> `foo(10,2)`
20

MULTIPLE OUTPUTS

```
function foo(x,y)
    term1 = x + y
    term2 = x * y

    return term1, term2     # a tuple, using the
end
```

```
julia> foo(10,2)
2-element Vector{Int64}:
 12
 20
```

AN EXPRESSION AS OUTPUT

```
function foo(x,y)
    term1 = x + y
    term2 = x * y

    return term1 + term2
end
```

```
julia> foo(10,2)
32
```

Functions without Arguments

It's possible to define functions that don't require arguments, as we show below.

FUNCTIONS WITHOUT ARGUMENTS

```
function foo()
    a = 1
    b = 1
    return a + b
end
```

An example of functions without arguments is `Pkg.update()`, which was introduced when we studied packages.

The Order In Which Functions Are Defined is Irrelevant

A function can be defined anywhere in the code. In fact, you can define a function that calls another function, even if the latter hasn't been defined yet. To illustrate this, consider the following two code snippets, which are functionally equivalent.

CODE SNIPPET 1

```
foo1(x) = 2 + foo2(x)

foo2(x) = 1 + x
```

```
julia> foo1(2)
5
```

CODE SNIPPET 2

```
foo2(x) = 1 + x


foo1(x) = 2 + foo2(x)
```

julia> `foo1(2)`
5

FUNCTIONS AS OPERATORS

In the previous section, we noted that most built-in operators are also available as functions. For example, the expression $\boxed{2 + 3}$ can be written equivalently as $\boxed{+(2, \ 3)}$. This works because defining a function whose name is a valid operator symbol automatically creates the corresponding operator.

As an illustration, consider defining the symbol $\boxed{\oplus}$ as a function. This is such that, for scalar inputs $\boxed{x}$ and $\boxed{y}$, $\boxed{\oplus(x, \ y)}$ returns the sum of their logarithms.

FUNCTIONS AS OPERATORS

```
x = 1
y = 1

⊕(x,y) = log(x) + log(y)
```

julia> `⊕(x,y)`
0.0
julia> `x ⊕ y`
0.0

Not all symbols can be used with this purpose. The list of symbols allowed isn't officialy documented, but you can find it here.

POSITIONAL AND KEYWORD ARGUMENTS

Up to this point, we've been defining and calling functions using the notation `foo(x,y)`. A key characteristic of this syntax is that arguments are passed in a specific order, so that `foo(2,4)` assigns the first argument to `x` and the second to `y`. This approach is known as **positional arguments**.

However, a major drawback of positional arguments is their susceptibility to silent errors: if we accidentally swap the positions of the arguments, the function may still provide an output. As the number of arguments grows, the likelihood of

introducing such bugs increases, making it more challenging to identify and resolve errors.

To circumvent this issue, we can rely on **keyword arguments**. This approach requires function calls to explicitly specify their arguments, making their order irrelevant. For example, `foo(x=2,y=4)` and `foo(y=4,x=2)` would then be valid and equivalent.

The following examples illustrate how to define and call functions using both positional and keyword arguments. Additionally, we'll establish that the approaches can be combined. Note that positional arguments necessarily require a semicolon during function definitions, but accept either a semicolon or a comma during function calls.

POSITIONAL ARGUMENTS

```
foo(x, y) = x + y
```

```
julia> foo(1,2)
2
```

KEYWORD ARGUMENTS

```
foo(; x, y) = x + y
```

```
julia> foo(x=1,y=1)
2

julia> foo(; x=1, y=1) # alternative notation (only for
calling 'foo')
2
```

POSITIONAL AND KEYWORD ARGUMENTS

```
foo(x; y) = x + y
```

julia> `foo(1 ; y=1)`
2

julia> `foo(1 , y=1)` # alternative notation
2

KEYWORD ARGUMENTS WITH DEFAULT VALUES

Keyword arguments accept default values, allowing users to omit certain arguments when the function is called. The following examples illustrate how this feature works in practice, where the omitted arguments take on their default values.

POSITIONAL AND KEYWORD ARGUMENTS

```
foo(x; y=1) = x + y
```

julia> `foo(1)` # equivalent to foo(1,y=1)
2

OMITTING POSITIONAL ARGUMENTS

```
foo(; x=1, y=1) = x + y
```

```
julia> foo()  # equivalent to foo(x=1,y=1)
2

julia> foo(x=2)  # equivalent to foo(x=2,y=1)
3
```

PASSING ARGUMENTS AS INPUTS TO OTHER ARGUMENTS

When a function is called, its arguments are evaluated sequentially from left to right. This property enables users to define subsequent arguments in terms of previous ones. For example, given `foo(;x,y)`, the default value of `y` could be set based on the value of `x`.

PRIOR ARGUMENTS AS DEFAULT VALUES

```
foo(; x, y = x+1) = x + y
```

```
julia> foo(x=2)  #function run with implicit value 'y=3'
5
```

SPLATTING

Given a function `foo(x,y)`, you can set the values of `x` and `y` through a tuple or vector `z`. The implementation relies on the splat operator `...`, which unpacks the individual elements of a collection and passes them as separate arguments.

TUPLE SPLATTING

```
foo(x,y) = x + y

z = (2,3)
```
---
julia> `foo(z...)`
5

VECTOR SPLATTING

```
foo(x,y) = x + y

z = [2,3]
```
---
julia> `foo(z...)`
5

ANONYMOUS FUNCTIONS

Anonymous functions offer a third way to define functions. Unlike the previous methods, they're commonly introduced with a different purpose: to serve as inputs to other functions.

3

As the name suggests, anonymous functions aren't referenced by a name. Their syntax resembles the arrow notation from mathematics (e.g. $x \mapsto \sqrt{x}$). Specifically, single-argument functions are expressed as `x -> <body of the function>`. Likewise, functions with two or more arguments are expressed by `(x,y) -> <body of the function>`.

To demonstrate the role of anonymous functions, let's consider the built-in function `map(<function>, <collection>)`. This applies `<function>` element-wise to each element of `<collection>`. For example, `map(add_two, x)` applies the function `add_two(a) = a + 2` to each element of `x = [1,2,3]`, thus returning `[3,4,5]`. Applying `map` in this way requires defining `add_two` beforehand, which unnecessarily pollutes the namespace if `add_two` won't be reused. Anonymous functions provide an elegant solution, by directly embedding the operation within `map`. In this way, an anonymous function effectively eliminates the need of creating a temporary function like `add_two`.

VIA COMPACT FUNCTION

```
x          = [1, 2, 3]
add_two(a) = a + 2


result     = map(add_two, x)
```

```
julia> result
3-element Vector{Int64}:
 3
 4
 5
```

VIA ANONYMOUS FUNCTION

```
x          = [1, 2, 3]



result     = map(a -> a + 2, x)
```

```
julia> result
3-element Vector{Int64}:
 3
 4
 5
```

The function `map` can also demonstrate the syntax of anonymous functions with multiple arguments. In those cases, the syntax becomes `map(<function>, <array1>, <array2>)`. For instance, `map(+, [1,2], [2,4])` provides the sum of each pair of numbers, yielding `[3,6]`.

VIA COMPACT FUNCTION

```
x         = [1,2,3]
y         = [4,5,6]


add(a,b) = a + b
result    = map(add_two, x, y)
```

julia> `result`
3-element Vector{Int64}:
 3
 4
 5

VIA ANONYMOUS FUNCTION

```
x         = [1,2,3]
y         = [4,5,6]



result    = map((a,b) -> a + b, x, y)
```

julia> `result`
3-element Vector{Int64}:
 5
 7
 9

THE "DO-BLOCK" SYNTAX

Anonymous functions can help keep our code tidy, but they may not be practical for functions that span multiple lines.

This inconvenience can be addressed by what's known as **do-blocks**. They allow us to insert the anonymous function separately, and then pass it as the first argument to a function call. Given a function `foo(<inner function>, <vector>)`, its generic implementation is as follows.

PRIOR ARGUMENTS TO DEFINE DEFAULT VALUES

```
foo(<vector>) do <arguments of inner function>
    # body of inner function
    end
```

To illustrate the notation with a concrete scenario, let's revisit the `map` function example and rewrite it using a do-block.

WITH COMPACT FUNCTION

```
x          = [1, 2, 3]
add_two(a) = a + 2


result     = map(add_two, x)
```

```
julia> result
3-element Vector{Int64}:
 3
 4
 5
```

WITH ANONYMOUS FUNCTION

```
x            = [1, 2, 3]


result       = map(a -> a + 2, x)
```

julia> `result`
3-element Vector{Int64}:
 3
 4
 5

WITH DO-BLOCK SYNTAX

```
x            = [1, 2, 3]

result       = map(x) do a
                  a + 2
                  end
```

julia> `result`
3-element Vector{Int64}:
 3
 4
 5

Do-blocks also accept anonymous functions with multiple arguments, as shown below.

WITH COMPACT FUNCTION

```
x = [1,2,3]
y = [4,5,6]


add(a,b) = a + b
result   = map(add_two, x, y)
```

julia> result
3-element Vector{Int64}:
 5
 7
 9

WITH ANONYMOUS FUNCTION

```
x        = [1,2,3]
y        = [4,5,6]



result   = map((a,b) -> a + b, x, y)
```

julia> result
3-element Vector{Int64}:
 5
 7
 9

WITH DO-BLOCK SYNTAX

```
x        = [1,2,3]
y        = [4,5,6]

result   = map(x,y) do a,b        # not (a,b)
                a + b
                end
```

```
julia> result
3-element Vector{Int64}:
 5
 7
 9
```

FUNCTION DOCUMENTATION

To conclude this section, we cover how to document functions. This can be done by adding a string expression immediately before the function definition. Once this is done, the documentation can be accessed in the same manner as with built-in functions: you can type the function's name in the REPL after pressing ?, or directly hover over the function's name in VS Code. [4]

STANDARD FORM

```
"This function is written in a standard way. It t
function add_two(a)
    a + 2
end
```

COMPACT FORM

```
"This function is written in a compact form. It t
add_three(a) = a + 3
```

For further details, see the official documentation.

<u>FOOTNOTES</u>

[1.] The method to call a function actually depends on the **module** in which it's defined, and whether this module has been "imported" or "used". We won't cover modules on this website. However, they're essential when working for large projects, as each module operates as an independent workspace with its own variables. When initiating a new session in Julia, you're actually working within a module called `Main`.

[2.] The reason for this is that tuples are more performant than vectors when the number of elements is small.

[3.] Anonymous functions are also known as *lambda functions* in other languages.

[4.] Here, we explained how to access a function's documentation, under the subtitle "To See The Documentation of a Function".