3c. Defining Your Own Functions

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 in creating your own functions is assigning them names. Function names adhere to similar rules as variable names. In particular, they accept Unicode characters, allowing us to specify functions like `∑(x)`. Once defined, functions can be directly called, without any prefix. Thus, a function named `foo` can be simply called by `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
```

In the compact form, the value of the single expression is automatically returned as the output. By contrast, the standard form returns the result of the last line by default. If greater control is needed, the output can be explicitly specified using the keyword `return`.

It's also possible to return multiple values by defining a collection as the output. Among the available options, tuples are generally preferred when the number of outputs is small. As we'll show in Part II of the book, they provide a lightweight structure that avoids the overhead associated with vectors, making them more efficient in both performance and memory usage. This efficiency explains why tuples are commonly adopted in practice for returning compact sets of results.

Below, we illustrate all this.

EXPLICIT OUTPUT

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

    return term2
end
```

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

IMPLICIT OUTPUT

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

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

MULTIPLE OUTPUTS

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

    return term1, term2    # a tuple (notation that omits the parentheses)
end
```

```
julia> foo(10,2)
(3, 2)
```

AN EXPRESSION AS OUTPUT

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

    return term1 + term2
end
```

```
julia> foo(10,2)
5
```

Functions without Inputs

It's possible to define functions that don't require any input arguments, as shown below.

FUNCTION WITHOUT ARGUMENTS

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

The Order In Which Functions Are Defined is Irrelevant

A function can be defined at any point in the code. In fact, it's possible to define a function that invokes another function, even if the latter hasn't yet been introduced. To illustrate, 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

Most built-in operators in Julia are also available as functions. For example, the expression `2 + 3` that uses `+` as an operator can be equivalently written as `+(2, 3)`, where `+` is employed as a function.

The same principle extends to user-defined functions when their names consist of certain symbols. In such cases, Julia automatically generates a corresponding operator. As an illustration, let's define a function whose name is `⊕`, where `⊕(x, y)` returns the sum of logarithms for scalar inputs `x` and `y`.

FUNCTIONS AS OPERATORS

```
x = 1
y = 1

⊕(x,y) = log(x) + log(y)
```

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

Note that not all symbols can be adopted with this purpose. While the list of allowed symbols isn't officially documented, it can be found in the Julia source code.

POSITIONAL AND KEYWORD ARGUMENTS

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

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 execute and return a wrong result. As the number of arguments grows, the likelihood of introducing such bugs grows, while simultaneously noticing them becomes more difficult.

To circumvent this issue, we can rely on **keyword arguments**. With this approach, each argument must be explicitly named in the function call, rendering their order irrelevant. For example, both `foo(x=2,y=4)` and `foo(y=4,x=2)` would then be valid and equivalent.

The following examples illustrate the difference between positional and keyword arguments when defining and calling functions. In particular, note that positional arguments necessarily require a semicolon during function definitions, but accept either a semicolon or a comma during function calls. Additionally, we show that both approaches can be combined within the same function.

POSITIONAL ARGUMENTS

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

```
julia> foo(1,2)
3
```

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')
\output{./code/region06e}
```

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

An additional advantage of keyword arguments is that they can be given default values. This implies that certain arguments may be omitted when the function is called, in which case the omitted arguments automatically assume their predefined defaults. The following examples illustrate this behavior in practice.

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
```

Default values can also be defined in terms of earlier arguments. For example, in a function `foo(; x, y)`, the default value of `y` can be set based on the value of `x`. This works because, when a function is called, its arguments are evaluated sequentially from left to right.

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)`, it's possible to supply the values of `x` and `y` through a single collection `z`. This is achieved using the splat operator `...`, which unpacks the elements of a collection and passes them as individual arguments to the function.

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 standard or compact forms, they're commonly introduced for a different purpose: to serve as inputs to other functions. [2] Functions that call another function as an argument are referred to as higher-order functions, and will be studied in Part II of the book.

As the name suggests, anonymous functions aren't referenced by a name. Instead, their definition relies entirely on syntax, which resembles the arrow notation from mathematics (e.g. $x \mapsto \sqrt{x}$). In particular, single-argument anonymous functions are expressed as `x -> <body of the function>`, whereas those with two or more arguments are expressed as `(x,y) -> <body of the function>`.

To illustrate, let's consider the built-in function `map(<function>, <collection>)`. This applies `<function>` element-wise to each element of `<collection>`. For example, given the function `add_two(a) = a + 2`, the call `map(add_two, x)` applies `add_two` to each element of `x = [1,2,3]`, thus yielding `[3,4,5]`.

The downside of using `map` in this way is that `add_two` must be defined beforehand, unnecessarily cluttering the namespace if `add_two` won't be reused. Anonymous functions provide an elegant alternative by embedding the operation directly within `map`.

VIA COMPACT FUNCTION

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

output     = map(add_two, x)
```

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

VIA ANONYMOUS FUNCTION

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


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

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

The function `map` can also be employed with multiple arguments, in which case the syntax becomes `map(<function>, <array1>, <array2>)`. For instance, `map(+, [1,2], [2,4])` provides the sum of each pair of numbers, producing `[3,6]`. Using this form, let's apply `a + b` element-wise to each pair of `x` and `y`. Below, we show how this operation can be implemented with an anonymous function.

VIA COMPACT FUNCTION

```
x               = [1,2,3]
y               = [4,5,6]
add_values(a,b) = a + b

output          = map(add_values, x, y)
```

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

VIA ANONYMOUS FUNCTION

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


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

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

THE "DO-BLOCK" SYNTAX

When working with higher-order functions, anonymous functions prevent unnecessary pollution of the namespace. However, when the function body extends beyond a single line, their use can become cumbersome and harder to read.

To address this limitation, we can employ a **do-block**. This groups a sequence of expressions into a block that can be passed as an argument to a function. While this construct isn't limited to higher-order functions, it's particularly valuable when a function expects another function as its first argument. In such situations, the do-block provides a way to define a multi-line anonymous function, without sacrificing readability.

For example, suppose a function of the form `foo(<inner function>, <vector>)`. Instead of embedding a lengthy anonymous function, we can call `foo` through a do-block using the following syntax:

DO-BLOCK SYNTAX

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

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

VIA COMPACT FUNCTION

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

output     = map(add_two, x)
```

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

VIA ANONYMOUS FUNCTION

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


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

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

VIA DO-BLOCK SYNTAX

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

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

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

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

VIA COMPACT FUNCTION

```
x               = [1,2,3]
y               = [4,5,6]
add_values(a,b) = a + b

output          = map(add_values, x, y)
```

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

VIA ANONYMOUS FUNCTION

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


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

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

VIA DO-BLOCK SYNTAX

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

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

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

FUNCTION DOCUMENTATION

To conclude this section, let's discuss how to document functions. A function's documentation, often called a **docstring**, provides a concise description of what the function does, how it should be used, and any important details about its inputs and outputs.

A docstring is added by placing a string expression immediately before the function definition. Once defined, it can be accessed in the same way as the documentation for built-in functions: type the function's name in the REPL after pressing `?`, or directly hover over the function name if you're using VS Code. [3]

STANDARD FORM

```
"This function is written in a standard way. It takes a number and adds two to it."
function add_two(a)
    a + 2
end
```

COMPACT FORM

```
"This function is written in a compact form. It takes a number and adds three to it."
add_three(a) = a + 3
```

For further details about docstrings, see the corresponding section in the official documentation.

FOOTNOTES

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 on large projects, as each module operates as an independent workspace with its own variables. In fact, every new session in Julia defines a module called `Main` in which you're writing code.
2. Anonymous functions are also known as *lambda functions* in other programming languages.
3. See here for an example in VS Code.