3b. Function Calls and Packages

Martin Alfaro
PhD in Economics

INTRODUCTION

Broadly speaking, functions can be broken down into three categories:

- ***i)*** built-in functions,
- ***ii)*** third-party functions, and
- ***iii)*** user-defined functions.

This section focuses on i) and ii), relegating iii) to the next section. We consider in particular how to call functions, which in turn leads us to discuss packages.

> **Notation for Functions**
>
> Julia's developers suggest a **snake-case notation for function names**. This consists of lowercase letters, numbers, and possibly underscores to separate words (e.g., `snake_case123`). Note this is only a convention, not a language's requirement.

PACKAGES

When you start a new session in Julia, only a handful of very basic functions are available (e.g., those for sums, products, and subtractions). This is a deliberate choice made by Julia developers, who rely on **packages** to incorporate functions into the workspace. In fact, both built-in and third-party functions are contained in packages—the only difference is that the former are loaded by default.

The approach is not unique to Julia. However, Julia embraces this philosophy more profoundly than other programming languages. Thus, it doesn't even include standard functions such as averages or standard deviations, which are instead relegated to a package called `Statistics`. [1]

This design philosophy is rooted in a programming principle known as **modularity**. The principle promotes the development of small reusable modules, rather than large intertwined code. Its main advantage is to let packages evolve independently, without bugs and deprecations spreading across the entire Julia ecosystem. The practical implication of this feature is that users need to load several packages in each session, even to perform simple tasks.

LOADING PACKAGES AND CALLING FUNCTIONS

The concept of packages is tightly related to modules. Formally, **modules** are independent blocks of code, each acting as a separate workspace, that export a defined set of functions. In fact, when you start Julia, you're implicitly writing your script in a module called `Main`.

Packages are a special type of modules, which additionally include information about their **dependencies**. Dependencies are defined as the necessary packages that must be loaded to run the package itself.

Getting access to a package's functions requires loading the package via either the keyword `import` or `using`. The primary difference between the two is how functions are eventually called in your code. If the package is loaded via `import`, the function's name must include a prefix with the package's name. On the contrary, no prefix is needed when the package is loaded with `using`.

Below, we demonstrate each approach by calling the function `mean` from the package `Statistics`. This package isn't loaded by default, but it comes pre-installed with Julia.

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

import Statistics   #getting access to its functions will require the prefix `Statistics.`
Statistics.mean(x)
```

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

using Statistics      #no need to add the prefix `Statistics.` to call its functions (although it's possible to do so)
mean(x)
```

BUILT-IN FUNCTIONS

Formally, Julia's built-in functions are contained in two packages known as `Core` and `Base`. Both are automatically loaded in every Julia session, with their functions accessible as if we had executed `using Core` and `using Base`. This determines that their functions don't require adding a prefix to be called. [2]

Among mathematical functions, the syntax of their most common ones is as follows.

Function in Julia	Meaning
`log(x)`	$\ln(x)$
`exp(x)`	e^x
`sqrt(x)`	$\sqrt{x}$

Function in Julia	Meaning
`abs(x)`	$\|x\|$
`sin(x)`	$\sin(x)$
`cos(x)`	$\cos(x)$
`tan(x)`	$\tan(x)$

> **Operators as Functions**
>
> Most of the symbols employed as operators are also available as functions. This is illustrated below for several arithmetic operators:

```
+(2,3)      # same as 2 + 3
-(2,3)      # same as 2 - 3
*(2,3)      # same as 2 * 3
/(2,3)      # same as 2 / 3
^(2,3)      # same as 2 ^ 3
```

WHY USING "IMPORT" IF IT'S MORE VERBOSE?

When a function's name is shared across multiple packages, at least one of the packages must be loaded via `import` to prevent naming conflicts. For instance, given the package `Statistics` and another one called `MyPackage` containing a function called `mean`, Julia will throw an error if you don't load one of them with `import`. [3]

Using `import` not only avoids naming conflicts, but may also reduce ambiguity in the meaning of a function. For instance, consider a function called `rank`. This name could reference a wide range of concepts, depending on the context (e.g., the rank of a matrix, the order in a list). However, explicitly identifying the package when the function is called could shed some light on its intended meaning.

> **Remark**
>
> `import` may also be useful if you have custom functions that are widely applied across your projects. For example, consider a function called `table_in_pdf`, which exports Julia tables to a PDF with some predefined format. While the name of the function makes it clear what it's doing, a user could wonder if this function comes from a standard package. You could hint that this isn't the case, by placing the function in a package called `userDefined`. In this way, you can load the package using `import UserDefined`, and then calling the function via `UserDefined.table_in_pdf`.

APPROACHES TO LOADING PACKAGES AND CALLING FUNCTIONS

The concepts discussed so far will probably be all you need to use packages in Julia. However, there are a few additional features worth mentioning.

First, users can load only a subset of functions from a package. This possibility is particularly relevant for heavy packages, which may take a significant time to fully load. For instance, if we only need the function `mean` from `Statistics`, the following two approaches achieve the same result.

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

import Statistics: mean
mean(x)                    # no prefix needed
```

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

using Statistics: mean
mean(x)
```

Note that this approach deems it unnecessary to add the package's name as a prefix, even when the package is loaded via `import`.

Another handy feature is the possibility of assigning custom names to either packages or functions. This becomes particularly useful when names are lengthy.

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

import Statistics as st
st.mean(x)
```

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

import Statistics: mean as average
average(x)                    # no prefix needed

using Statistics: mean as average
average(x)
```

Again, notice that the function's name doesn't require any prefix when it's called, even with `import`.

MACROS

Macros are ubiquitous in Julia. They enable the automation of tasks that otherwise would be tedious and time-consuming to perform. On this website, we'll only cover how to apply macros, without exploring how to define them. The reason is that creating macros requires knowledge of Julia's metaprogramming capabilities, which is beyond the scope of this website.

While the utility of macros may not be immediately obvious at this point, this will become clearer once we start applying them in subsequent sections.

APPLYING MACROS

Macros and functions share similarities, with both performing operations on inputs and producing outputs. Their key distinction lies in their handling of inputs and outputs: macros manipulate code syntax (statements or expressions), whereas functions process data values (variables or evaluated expressions).

Formally, macros are denoted by prefixing the symbol `@` to their name. They take an entire code expression as their argument and transform it. For example, a macro might take `x = some_function(y)` as input, potentially modifying each individual component (`x`, `=`, or `some_function(y)`), inserting new code, or reorganizing the code structure. The final output is a modified version of the original expression, which is then integrated into the program during execution.

A key purpose of macros is to automate code transformations. For example, consider the `@.` macro in Julia, which appends a dot `.` to every operator and function call in a statement. For now, ignore the impact of adding dots to your code, which will be explained in an upcoming section. Instead, focus on how macros operate at the syntactic level to rewrite entire code blocks.

```
# both are equivalent
   z .= foo.(x .+ y)
@. z  = foo(x  + y)          # @. adds . to =, foo, and +
```

> **Warning!** - Caution with Macro Usage
>
> Applying macros requires extreme caution: they could act as black boxes and hence lead to unexpected behaviors. In fact, macros tend to be a common source of bugs. Make sure you understand which part of the expression a macro modifies and how.

FOOTNOTES

[1.] The extent to which Julia advocates for this principle is evident in `Statistics` itself, where functions for computing distributions are included in another package called `Distributions`.

[2.] Some built-in functions may require a prefix. For instance, this is what occurs with the function called `isgreater`, which must be called via `Base.isgreater`. Furthermore, some submodules are also loaded by default in each session. For instance, the function `Base.Iterators.accumulate` is part of the submodule `Iterators` from `Base`, and can be directly called using `Iterators.accumulate`.

[3.] Defining a function that shares the name of another package's function isn't necessarily an oversight by developers. For instance, we could implement our own `mean` function in a package called `MyPackage`, which aims at computing averages more efficiently in certain applications.