3b. Function Calls and Packages

Martin Alfaro
PhD in Economics

INTRODUCTION

Functions can be broadly categorized into three main categories:

- ***i)*** built-in functions, which are provided as part of the core Julia language.
- ***ii)*** third-party functions, typically obtained from external packages.
- ***iii)*** user-defined functions, created by the programmer to perform specific tasks.

This section focuses on i) and ii), with user-defined functions being addressed in the following section.

> **Notation for Functions**
>
> Julia's developers suggest the **snake-case convention for function names**. This style consists of lowercase letters, numbers, and possibly underscores to separate words (e.g., `snake_case123`). Keep in mind that this is just a recommendation, not a language's requirement.

PACKAGES

When a new Julia session is initiated, 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, with the only difference that the former are loaded by default.

The approach isn't unique to Julia. However, Julia embraces this philosophy more deeply 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. The main advantage of modularity is to let packages evolve independently, without bugs and deprecations spreading across the entire Julia ecosystem. In practice, it also implies that users must load several packages during each session, even to perform simple tasks.

LOADING PACKAGES AND CALLING FUNCTIONS

The concept of packages in Julia is closely tied to that of modules. Formally, **modules** are self-contained blocks of code, each providing its own workspace and exporting a defined set of functions. In fact, when a Julia session starts, all code is implicitly executed within a default module called `Main`.

For their part, **packages** are a special type of module that additionally include information about their **dependencies**. These are defined as the necessary packages that must be loaded to run the package itself. This dependency information ensures that Julia can automatically manage and load the required components when the package is used.

To get access to the functions provided by a package, we must load it through either the `import` or `using` keyword. While both mechanisms bring the package into scope, they differ in how functions are subsequently invoked. With `import`, functions must be explicitly qualified by prefixing them with the package name. In contrast, `using` makes the exported functions directly accessible, allowing them to be called without any prefix.

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

Julia's built-in functions are formally organized into two packages called `Core` and `Base`. These packages are automatically loaded at the start of each session. The access to their functions replicates the behavior of `using Core` and `using Base`. As a result, most of their associated functions are directly accessible without requiring a package prefix. [2]

Below, we show the syntax for common built-in mathematical functions.

Function in Julia	Meaning
`log(x)`	$\ln(x)$
`exp(x)`	e^x
`sqrt(x)`	$\sqrt{x}$
`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 in Julia 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 functions share the same name across multiple packages, at least one of the packages must be loaded via `import` to prevent naming conflicts. For instance, if both the package `Statistics` and another one called `MyPackage` contain a function called `mean`, Julia will throw an error unless one of them is loaded with `import`. [3]

Beyond resolving naming conflicts, `import` can also enhance code clarity by reducing 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). Explicitly specifying the package when the function is called could clarify its intended purpose.

> **Remark**
>
> `import` may also be beneficial when defining custom functions that will be reused across several projects. For example, imagine defining a function like `table_in_pdf` to export Julia tables to PDF format. While the function name clearly conveys its purpose, someone reading your code might reasonably wonder whether this function comes from a standard package. To avoid this ambiguity, you could place the function in a package called `userDefined` and load it with `import UserDefined`. This approach makes it immediately clear that `UserDefined.table_in_pdf` is a custom implementation, rather than part of a standard package.

APPROACHES TO LOADING PACKAGES AND CALLING FUNCTIONS

The concepts discussed so far are sufficient for using packages in Julia effectively. Still, there are a few additional features worth highlighting, since they can further enhance how you interact with packages.

First, users have the possibility of loading only a subset of functions from a package. This is particularly valuable when working with heavy packages which may suffer from significant loading times. 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, in both cases, the function `mean` can be called without prefixing it with the package name. This holds even when the package is loaded via `import`.

Another valuable feature is the ability to assign custom names to packages or functions. This becomes particularly useful when names are lengthy, since aliases can shorten them for convenience while maintaining readability in the code.

```
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)
```

In this case as well, the function name can be called without any prefix, even if it's been brought into scope via `import`.

MACROS

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

Although the benefits of macros may not be immediately obvious at this point, their utility will become evident once we apply them in subsequent sections.

APPLYING MACROS

Macros and functions share important similarities: both operate on inputs and produce outputs. Their key distinction lies in the objects taken as inputs and returned as outputs. Functions evaluate data values such as variables or expressions and return a result. Macros, by contrast, manipulate the syntax of code itself, rewriting statements or expressions before they're executed.

Formally, macros are denoted by prefixing the symbol `@` to their name. They take an entire code expression as their argument and transform it at the syntactic level. For instance, a macro might take the expression `x = some_function(y)` as input and then alter its structure: it could modify each individual component (`x`, `=`, or `some_function(y)`), insert new code, or reorganize the statement entirely. The final output is a modified version of the original expression, which is then integrated into the program during execution.

The primary role of macros is to automate code transformations that would be repetitive or error-prone if written manually. A clear example is Julia's `@.` macro, which appends a dot `.` to every operator and function call within a statement. While the semantic consequences of dot notation will be discussed in upcoming sections, the key point here is that macros enable systematic rewriting of entire code blocks.

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

> **Warning!** - Caution with Macro Usage
>
> While powerful, macros should be applied with considerable care. Because they operate by transforming code before runtime, they can behave as "black boxes", making debugging challenging and potentially introducing subtle errors. Indeed, macros are frequently a source of unexpected behavior in programs. Make sure you understand which part of the expression a macro modifies and how that transformation is carried out.

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, the function `isgreater` must be called via `Base.isgreater`. Furthermore, some submodules are preloaded by default. For instance, the function `Base.Iterators.accumulate` belongs to the `Iterators` submodule of `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. For instance, developers could implement their own version of `mean` within a package like `MyPackage`, where averages are computed with a different algorithm.