Julia As Your First Programming Language: A Book for Scientists

Martin Alfaro
PhD in Economics

WELCOME TO THE SITE!

The website is still work in progress in terms of writing, content, and subjects covered.

The chapters included so far can be found **here**. I'll continue adding new content as I go.

If you find any mistakes, typos, or have any suggestions, please open an issue on the **book's GitHub page**. Your feedback is greatly appreciated!

A QUICK OVERVIEW OF THE BOOK

AUDIENCE: The book is intended **for an audience with little or no background in programming**. This doesn't mean that we solely cover basic topics. Rather, it defines the book's approach of starting from elementary concepts, **gradually introducing more advanced concepts as we progress**.

APPROACH: Throughout the book, **I've made a conscious effort to distinguish between what's essential and what's ancillary**, with the latter clearly labelled as optional. My goal is that you don't become bogged down in particular details, while still having the possibility of exploring topics further if you wish.

TOPICS: The book focuses on the foundational concepts of the language, without pursuing an exhaustive examination of all its features. My philosophy is that **you can easily incorporate additional features if you grasp the logic of the language**.

Version
August 20, 2025

Parts and Chapters

PART I: BASICS OF JULIA

INTRODUCTION

1. Installation and Preliminaries
   - *a*. Installation and Resources
   - *b*. Running Julia
   - *c*. VS Code (OPTIONAL)
   - *d*. A Minimal Set of Good Practices
2. Variables
   - *a*. Overview and Goals
   - *b*. Variables, Types, and Operators
   - *c*. Numbers
   - *d*. Strings
   - *e*. Arrays (Vectors and Matrices)
   - *f*. Tuples

CORE CONCEPTS

3. Functions
   - *a*. Overview and Goals
   - *b*. Function Calls and Packages
   - *c*. Defining Your Own Functions
   - *d*. Variable Scope & Relevance of Functions
   - *e*. Map and Broadcasting
4. Control Flow
   - *a*. Overview and Goals
   - *b*. Conditions
   - *c*. Conditional Statements
   - *d*. For-Loops

USING JULIA

5. Vectors: Indexing and Mutations
   - *a*. Overview and Goals
   - *b*. Mutable and Immutable Objects
   - *c*. Assignments vs Mutations
   - *d*. Initializing Vectors

PART II: HIGH PERFORMANCE

10. Vectorization (SIMD)
    a. Overview and Goals
    b. Macros as a Means for Optimizations
    c. Introduction to SIMD
    d. SIMD: Independence of Iterations
    e. SIMD: Unit Strides
    f. SIMD: Branchless Code
    g. Packages For SIMD

11. Multithreading
    a. Overview and Goals
    b. Introduction to Multithreading
    c. Task-Based Parallelism: @spawn
    d. Thread-Safe Operations
    e. Parallel For-Loops: @threads
    f. Applying Parallelization
    g. Packages for Multithreading

1a. Installation and Resources

Martin Alfaro
PhD in Economics

INTRODUCTION

We start by covering the essential steps to install Julia and VS Code. The latter is a code editor to write and execute code in multiple languages. We'll conclude by providing some valuable online resources for Julia's users.

INSTALLING JULIA

> **Remark**
> All the links mentioned on the website are included in Links, located in the left navigation bar.

To download Julia and access its official documentation, visit **Julia's official website**. Note that the installation process depends on your computer's operating system.

INSTALLING VS CODE

Once Julia is installed, you'll need an editor to write scripts and visualize outputs. There are numerous alternatives in this respect. **Our website supposes that you use Visual Studio Code (aka VS Code)**, which is free, officially supported by Julia, and runs on any operating system (i.e., Windows, macOS, and Linux). One of the key benefits of VS Code is the possibility of installing plugins to extend the editor's capabilities. In fact, you'll need to add the *Julia Language Support* plugin for running Julia.

> **Privacy-Oriented Version of VS Code**
> VS Code is open-source software created and maintained by Microsoft. If you want a more private alternative that disables telemetry and tracking, **VSCodium** is a rebuild of VS Code.

Links to other popular editors can be found on **Useful Links**, including Vim, Emacs, NotePad, and Sublime. These editors are officially supported by Julia (except Sublime). I strongly recommend getting proficient in either VS Code or one of these alternatives. This will allow you **to master a single tool for coding in multiple programming languages**.

> **Warning!**

> Avoid getting used to specialized editors built for one particular language, such as RStudio for R (or its newer version **Posit**). The editors I mentioned were designed for coding, regardless of the programming language you choose. Mastering a general code editor will enhance your coding skills—you'll be able to apply the same tools and keyboard shortcuts to every language you work with.

JULIA'S RESOURCES FOR HELP

There are two official resources for learning Julia.

1. **Julia's official documentation**. Written by Julia's developers.
2. **Julia Discourse**. Official forum for asking questions.

INSTALLING R AND PYTHON (*OPTIONAL*)

Julia offers a seamless integration with other programming languages like R and Python, allowing you to export data from Julia, perform specific operations, and then import the results back into Julia. This interoperability is particularly useful when a desired function is only available in one of these languages.

For those familiar with R and Python, **this note** outlines some noteworthy differences with respect to Julia. Additionally, this **cheat sheet** provides a quick reference on syntax differences among Matlab, Python, and Julia.

1b. Running Julia

Martin Alfaro
PhD in Economics

INTRODUCTION

In the following, we cover the basic steps for getting started with Julia. As we haven't introduced any tools available in Julia (e.g., functions), we'll keep the discussion to a bare minimum. Specifically, we'll limit ourselves to setting up Julia in VS Code and presenting methods to add comments and file paths.

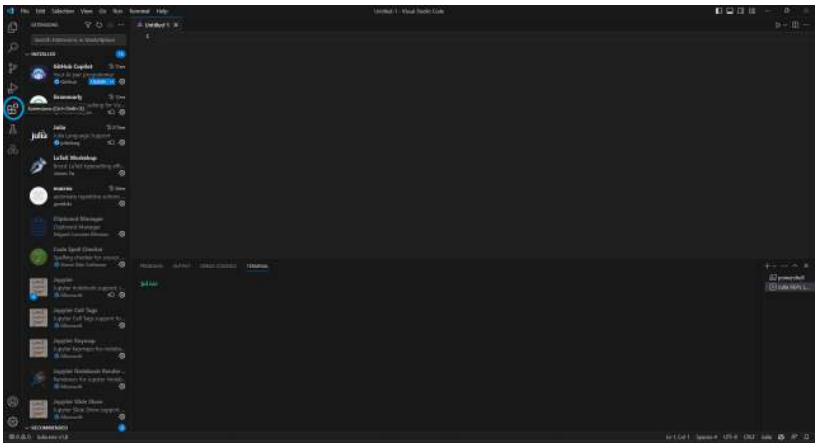
USING JULIA IN VS CODE

The REPL (Read-Eval-Print Loop) is an interactive programming environment that lets users input commands and immediately obtain outputs through a command-line interface. When you run `julia.exe`, the REPL is automatically activated and displays the `julia>` prompt, where you can enter commands.

▼ Screenshot

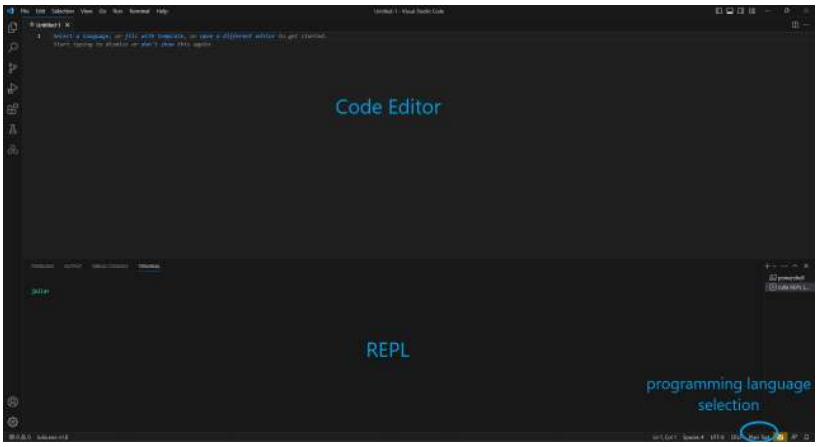
Throughout this website, we'll assume that you're working with a code editor, rather than interacting directly with the REPL. In particular, VS Code will be our code editor of choice. To get started with Julia in VS Code, you'll need to install its Julia extension. This can be found by navigating to the Extensions tab, as indicated below by the blue circle.

▼ Screenshot



The layout of VS Code displays the REPL at the bottom of the screen, where the code must be written in the area above it. To execute the code, you'll also need to specify the programming language you're using. This can be achieved by clicking on the language option located at the bottom corner of the screen, or by using the keyboard shortcut Ctrl+k + M and typing "julia". All this is demonstrated in the screenshot below.

▼ Screenshot



ADDING COMMENTS IN A SCRIPT

In Julia, like in any other programming language, you can include comments in your code. Comments are text annotations ignored during execution, serving as a means to document your code.

To add a *single-line comment*, simply precede the text with the # symbol. This symbol can be placed anywhere on a line, with any text that follows disregarded by Julia. Alternatively, you can add *multi-line comments* by delimiting the text with #= at the beginning and =# at the end.

```
# This is an example of a comment

x = 2   # `x=2` is run, but anything after `#` won't

#=  This is an example of a longer comment.
    It can be split into several lines, and
    can have any length. =#
```

PATHS OF FILES AND FOLDERS

File management systems vary across operating systems, determining that the syntax for file paths also differs. To accommodate this, Julia provides two approaches. The first one provides an operating system-specific syntax. Below, we illustrate its application for a file `C:\user\file.jl` on Windows and `/user/file.jl` on Linux/macOS. There's also a platform-agnostic alternative to make your code more portable, provided by the `joinpath` function. This is the preferred option, as it can be used with any operating system.

```
# On Windows (note the double \\)
"C:\\user\\file.jl"

# On Unix-based systems (e.g., macOS or Linux)
"/user/file.jl"

# on any operating system
joinpath("/", "user", "file.jl")
```

Two special paths have convenient shortcuts that are worth mentioning:

- `@__DIR__` identifies the directory where your script is saved.
  For instance, if your script is in `C:\user\julia`, then `joinpath(@__DIR__, "graphs")` refers to `C:/user/julia/graphs`.

- `homedir()` indicates the user's home directory.
  This refers to `C:\Users\username` on Windows (where "username" is your actual user), and is the equivalent of `~` on Linux. For instance, you could access your Google Drive's folder located on either `C:\Users\username\GoogleDrive` or `\home\username\GoogleDrive` by the command `joinpath(homedir(), "GoogleDrive")`.

EXECUTING CODE FROM A FILE

You can also work **non-interactively** with Julia by executing code from a script stored in a file. The following example illustrates its implementation, running a file located at `C:\user\julia\graphs.jl` on Windows and at `/users/julia/graphs.jl` on macOS/Linux systems.

```
include(joinpath("/", "user", "julia", "graphs.jl"))
```

1c. VS Code (OPTIONAL)

Martin Alfaro
PhD in Economics

FEATURES AND KEYBOARD SHORTCUTS

We present a few keyboard shortcuts and handy features for VS Code. They also apply to its privacy-focused alternative VS Codium. Remarkably, these features are largely language-agnostic, holding regardless of the programming language you're working with.

For visual illustration, the features discussed are accompanied by GIFs. To view these GIFs, simply click "Example", or alternatively press Alt+↑ or Alt+↓ to open and close all of them simultaneously.

TO RUN A SCRIPT

Select the script to be executed and press Ctrl+Enter

▶ Example

TO FORMAT EXPRESSIONS AND MAKE THEM MORE LEGIBLE

Select the script to be formatted and press Ctrl+k + Ctrl+f. Sometimes, activating this tool requires running it twice.

▶ Example

TO ALIGN EQUAL SIGNS

This feature requires the VS Code Extension "Better Align". It aligns consecutive lines by using the equal sign and other symbols as a reference. It's implemented by pressing Alt + a.

▶ Example

See also the extension "Cursor Align", which aligns code by clicking the position on each line.

TO EXTEND THE CURSOR VERTICALLY

Hold down Alt+Ctrl + press ↑ or ↓

▶ Example

TO SEE THE DOCUMENTATION OF A FUNCTION

It requires hovering over the function.

▶ Example

Alternatively, you can go to the REPL, press ?, and then type the function's name you want to search for.

▶ Example

TO AUTOCOMPLETE A WORD

Start typing a word + press Tab when you see the option list.

▶ Example

TO INTRODUCE UNICODE CHARACTERS (TAB COMPLETION)

Type a unicode character/command, press Ctrl + Space to open an option list, and then choose the option and press Tab.

▶ Example

In Julia, Greek letters and math have the same syntax as Latex. To add them, you need to start with \ (e.g., \eq for ≠) and use Tab completion.

TO SELECT THE SAME WORD MULTIPLE TIMES

Select the word and then press Ctrl+d for selecting each additional time it appears. This is useful when you want to change part of the expression.

▶ Example

TO HIDE PART OF THE SCRIPT

Given a code block, add `#region` at the beginning and `#endregion` at the end.

▶ Example

When you have several lines indented, VS Code allows you to hide the block automatically. The following example shows this for a function.

▶ Example

TO TURN MULTIPLE LINES INTO A COMMENT

Select all the lines you want to interpret as a comment rather than code. Then, press Ctrl + /.

▶ Example

1d. A Minimal Set of Good Practices

Martin Alfaro
PhD in Economics

REMARKS

We conclude this chapter by reviewing various principles to write code. They represent a minimum set of good practices that apply regardless of the programming language used. By adhering to these guidelines, you'll be able to write clear and maintainable code. I suggest incorporating these suggestions into your workflow from the very beginning, as it'll render the learning process smoother.

Several of the suggestions we present might seem inconsequential to you at this point, or give the impression that their importance is exaggerated. For small projects, there's some truth to this—they won't have a substantial impact. However, as projects grow in size and complexity, following these principles becomes crucial. [1] It's not uncommon to revisit your own code after a few months (or even days!) and struggle to understand it. When this occurs, extending the code becomes a daunting task, often resulting in non-reusable code.

As usual, the devil is in the details. Thus, the challenge lies in interpreting and implementing these suggestions effectively. Many of them rely on the reader's judgment, as they require a subjective assessment of when and how to apply them. For example, one suggestion we'll present is to use clear and descriptive names. However, determining what constitutes "clear" or "unclear" is ultimately a matter of personal interpretation. Hopefully, the implementation of the suggestions will become apparent as we move forward and apply these concepts.

WRITE EASY-TO-READ CODE

Code is read more often than it's written. I can't stress enough the importance of this statement. It has a stark implication: write code that is easy to read, even if this requires additional effort or some extra verbosity.

If you end up coding extensively in your future career, you'll likely learn this lesson the hard way. I certainly did. One of the first times I had to reuse an old script, I was completely clueless about my own code. As a consequence, I had to rewrite the entire script from scratch, as making sense of the old code would've taken longer.

If you end up coding extensively in your future career, you'll likely learn this lesson the hard way, just as I did. One of my earliest experiences with reusing old code was a humbling one - I was completely baffled by my own script, and rewriting it from scratch proved to be faster than trying to decipher the original code.

> **Remark**

> If you're concerned that more readable code requires excessive typing, remember that you can use Tab Completion to autocomplete names. Additionally, AI tools like GitHub Copilot will suggest code while you type, thereby also mitigating the inconvenience.

To illustrate this point, suppose you're reading a script that cleans some data. Imagine in particular that you come across a line that has two possible expressions: `na.rm=TRUE` and `dropmissing=true`. Even if you're unfamiliar with the language's syntax or the concept of missing data, you could likely infer the meaning of `dropmissing=true`: discard entries with no values provided. On the contrary, `na.rm=TRUE` offers no clue. Although this example may appear somewhat abstract, it actually highlights how to discard missing observations in R and Julia: `na.rm=TRUE` corresponds to R and `dropmissing=true` to Julia. [2]

The example also reveals why typing `na.rm=TRUE` might be tempting: it's short and requires less typing. However, it's essential to weigh the long-term benefits of readable code. Although typing more might seem inconvenient in the short term, it represents a minimal effort compared to the future costs of ambiguous code. Moreover, you may feel confident that you'll remember what you intended to write, but it's common to be puzzled by code you wrote just days before.

The benefits of clear code become apparent when you read a script written in an unfamiliar programming language: if the code is well-written and clearly structured, you might grasp the logic and tasks being performed. [3]

Several tips arise as a consequence of this. We list them below.

USE NAMES WITH A CLEAR MEANING

Clear names don't only refer to variables and functions, but files as well. In particular, you should avoid abbreviating. Code editors can be very helpful in this regard, by offering word auto-completion. This feature requires typing the first letters of each word and then pressing Tab. [4]

Avoiding abbreviations has the additional benefit of making it easier to substitute expressions. For instance, suppose you name a variable `re`, and later decide to replace it with a different name. Then, the substitution process becomes more challenging, as the search will also capture functions like `replace` and `repeat`.

Finally, using descriptive names reduces the need for comments. If the code is self-explanatory, comments become only necessary for exceptionally complex code or clarifications that go beyond what's written.

INDENT AND ALIGN YOUR CODE

The implementation details of this suggestion have already been covered in the previous section. For further details, please refer to that section.

When writing code sequentially, VS Code automatically provides indentation. You can also format a selected portion of code by pressing Ctrl + k, followed by f. Alternatively, to format the entire script, use the shortcut Alt + Shift + L. [5]

To illustrate how this feature improves readability, consider the following (somewhat exaggerated) example.

```
if x>0 display("x is a positive number") else display("x is a non-positive number") end

function example(a,b)
x=a/10#rescaling x
output=2*b+x
return output
end
```

```
if x > 0
    display("x is a positive number")
else
    display("x is a non-positive number")
end

function example(a, b)
    x      = a / 10               # rescaling x
    output = 2 * b +  x

    return output
end
```

To further improve readability, I suggest also aligning code blocks. Several plugins in VS Code can assist with this task, such as "Better Align" and "Cursor Align". Their use is demonstrated below.

```
this_is_a_variable = 1
x = 3
another_var = 2

computations_here = x + another_var
more_calcs = this_is_a_variable * another_var
```

```
this_is_a_variable = 1
x                  = 3
another_var        = 2

computations_here  = x + another_var
more_calcs         = this_is_a_variable * another_var
```

FOOTNOTES

[1.] For real-world examples, read "Brief Story" on this link or the perspective of a former worker from Oracle.

[2.] Python also tends to employ abbreviations that can hinder readability. For instance, to count the number of characters of a variable `x`, Python calls `len(str(x))` while Julia calls `length(string(x))`.

[3.] One way to learn how to write clear code is through AI chatbots, which are pretty good at providing highly readable examples.

[4.] You could eventually use the option of "find and replace", whereby you substitute abbreviations for their full name. However, this is error-prone, and you may end up replacing unrelated expressions by substituting all words at once.

[5.] Unlike Python, Julia only uses indentation for readability purposes. It doesn't affect how code is executed.

2a. Overview and Goals

Martin Alfaro
PhD in Economics

> **Remark**
>
> Throughout the book, I made some deliberate choices regarding whether and when to introduce certain subjects. Considering this, I'll include a section called "Overview and Goals" prior to each chapter, which elucidates my rationale for these choices. The goal is to contextualize the book's approach, offering readers some guidance on the best way to engage with the material.

The current chapter introduces the concept of variables and types, covering single-element objects (numbers and characters) and collections (primarily vectors and tuples). At this early stage, **we only scratch the surface of the topics**. In particular, the chapter doesn't cover any object in depth, and even excludes important ones like dictionaries. The reason is pedagogical: I didn't want to overwhelm readers with details about objects or types, considering that core programmatic concepts like functions and for-loops haven't yet been introduced.

In light of this, Chapter 2 should be understood as a minimal background on objects, sufficient for progressing into the basics of working programmatically.

The main skills you should gain from Chapter 2 are:

- familiarizing yourself with Julia's syntax, and
- distinguishing between scalars (single-element objects) and collections.

2b. Variables, Types, and Operators

Martin Alfaro
PhD in Economics

INTRODUCTION

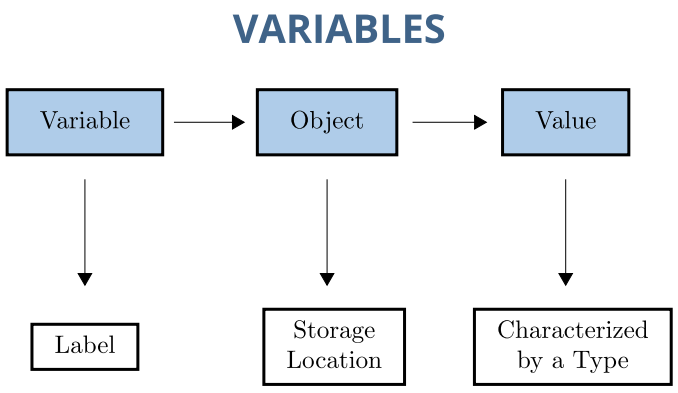
This section introduces the concepts of **variables** and **types**. We'll also present the notion of **operators**, focusing on their syntax. To ensure a smooth learning experience, I've minimized the reliance on objects that we haven't covered yet. The only one introduced is vectors, whose elements are enclosed in brackets (e.g., `[1,2,3]`).

VARIABLES

When a program is executed, the computer stores data in RAM (Random Access Memory). Each piece of data in RAM is referred to as an **object** and is assigned a unique memory address. These addresses are typically represented in hexadecimal format (e.g., `0x00007e0966dc0dd0`).

Furthermore, every object is associated with a value and a type. **Values** represent the actual data contained within the object. In turn, **types** define the nature of the data stored, providing the computer with critical information for handling the object internally.

Since directly referencing memory addresses would be impractical, we instead define **variables**. They act as human-readable **labels** for objects, simplifying our interaction with the data. Linking objects with a variable relies on the so-called **assignment operator** `=`, which creates a binding between the variable name and the object's memory location. [1] This allows developers to interact with data through symbolic identifiers, rather than raw memory locations.

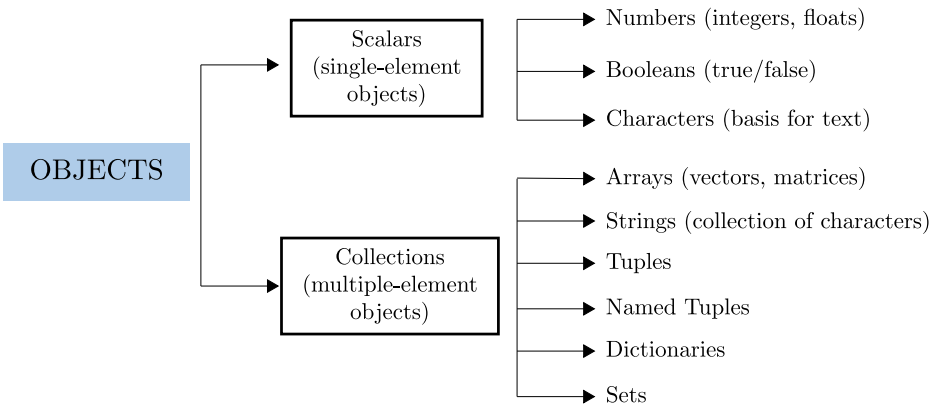


To illustrate this process, let's consider executing `x = "Hello"`. When this is run, several actions take place. First, the computer reserves a memory location (e.g., at address `0x1234`) to store the object. This object is then assigned the specific value `"Hello"`, which in Julia is an instance of the type `String`.

At the same time, we're assigning the label `x` to this object, meaning that `x` points to the memory address `0x1234`. This means that, every time we use `x` in our code, we're actually accessing the object stored at memory address `0x1234`. It's important to note that `x` isn't the object or the value itself, but rather a reference to the memory allocation `0x1234`. This explains why we can define multiple labels to reference exactly the same object (i.e., the same memory address).

CLASSIFICATION OF OBJECTS

Objects are typically characterized according to the number of elements they contain, with **scalars** referring to single-element objects, and **collections** referring to objects containing multiple elements. Below, we outline some objects encompassed in each category.



NAMES FOR VARIABLES

Variable names in Julia can be defined using Unicode characters, thus offering a wide range of possibilities. This feature enables you to use Greek letters, Chinese characters, symbols, and even emoticons. [2] Underscores `_` are also permitted, which can be helpful for separating words within variable names (e.g., `intermediate_result`). [3] Importantly, names are case-sensitive, so that `bar` and `Bar` are treated as two distinct variables.

```
a          = 2
A          = 2              # variable `A` is different from `a`

new_value  = 2              # underscores allowed

β          = 2              # Greek letters allowed

中國        = 2              # Chinese characters allowed

x̄          = 2              # decorations allowed
x₁         = 2
ẋ          = 2

🐒         = 2              # emoticons allowed
```

> **Warning!**

> **Julia doesn't let you delete variables**. Once a variable is created, it remains in memory until the program terminates. If a variable is taking up too much memory, you can free up space by reassigning it to a smaller object.

> **Notation for Variable Names**
>
> Julia's developers adopt the convention of using **snake-case** notation for variable names. This format consists of lowercase letters and numbers, with words separated by underscores. (e.g., `snake_case_var1`). Note that this is only a convention, not a language's requirement.

UPDATING VARIABLES

It's possible to assign a new value to a variable using the variable itself. This approach is referred to as **updating a variable**.

```
x  = 2

x  = x + 3    # 'x' now equals 5
```

Julia offers a concise syntax for updating values, based on the so-called **update operators**. They're implemented by prefixing the assignment operator `=` with the operator to be applied, as demonstrated below.

```
x  = 2

x  = x + 3
x += 3        # equivalent

x  = x * 3
x *= 3        # equivalent

x  = x - 3
x -= 3        # equivalent
```

TYPES

Before diving into the intricacies of Julia, it's essential to familiarize yourself with the basics of its type system. This initial overview will only provide the minimum necessary for the upcoming chapters. A comprehensive treatment of types, including their role in performance optimization, will be deferred to Part II of this website. For now, the focus is on core definition and notation.

> **Notation for Types**
>
> Julia's developers adopt the convention of using **CamelCase** notation for denoting types, where every first letter is capitalized (e.g., `MyType`). Note that this is only a convention, not a language's requirement.

As previously mentioned, types define the nature of values, specifying all the information the computer needs for their storage and manipulation. To better illustrate types, let's split the discussion in terms of scalars and collections.

Common numeric types for scalars include `Int64` for integers, `Float64` for decimal numbers, and `Bool` for binary values (`true` and `false` values). [4] Likewise, the type `Char` represents individual characters, serving as the building block for the `String` type. `String` is the standard type in Julia for representing text, and its values consist of sequences of characters.

Collections, on the other hand, often require **type parameters** for a full characterization of their types. These parameters can be incorporated into any type, and have the goal of providing additional information about its contents.

Type parameters are denoted using `{}` after the type's name. For instance, the type `Vector{Int64}` indicates that the collection represents a vector exclusively containing elements of type `Int64` (e.g., `[2, 4, 6]`). Here, `Int64` serves as a type parameter. Note that type parameters are optional and therefore can be omitted when not needed. Indeed, this is the case with the types for scalars mentioned above.

> **Type Annotations**
>
> You can explicitly declare the type of a variable by using **type annotations**, via the `::` operator. For example, `x::String` ensures that `x` can only store string values throughout the program, resulting in an error if you attempt to reassign `x` with a value of a different type.

CONCRETE TYPES AND ABSTRACT TYPES

In Julia, **types are organized hierarchically**, creating relations of supertypes and subtypes. This hierarchy gives rise to the notions of abstract and concrete types.

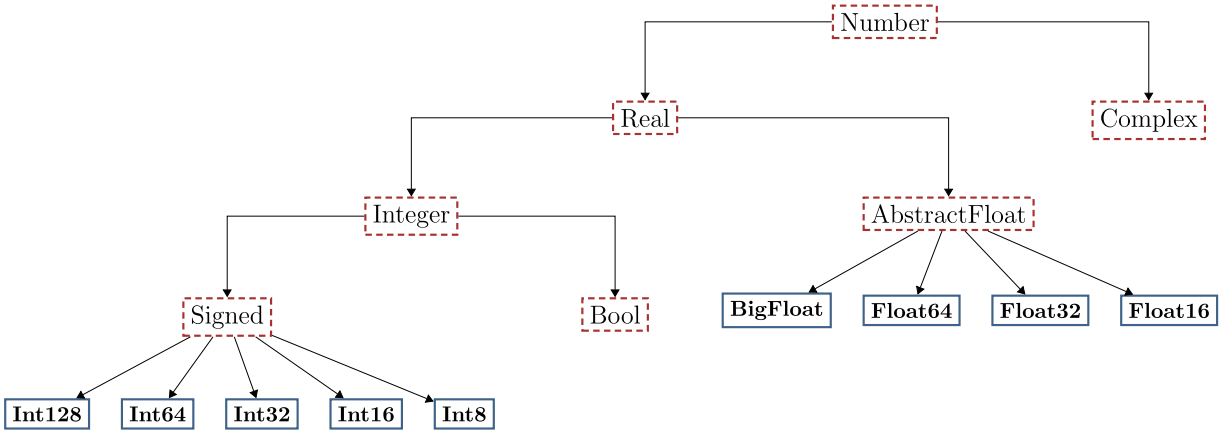
An **abstract type** is a set of types that serve as a parent to other types. The `Any` type in Julia is a prime example of abstract type. It acts as the root of the hierarchy, thus comprising all possible subtypes—by definition, every type in Julia is a subtype of `Any`.

In contrast, a **concrete type** is an irreducible unit, representing a terminal node in the hierarchy and therefore lacking subtypes. Concrete types include in particular primitive types, which represent the most fundamental types that computers use to perform calculations. Examples of primitive types are `Int64` and `Float64`, which directly map to low-level hardware representations.

Abstract types provide great flexibility for writing code. For example, the abstract type `Number` defined in Julia encompasses all possible numeric types (e.g., `Float64`, `Int64`, `Float32`). By declaring a variable as `Number`, programmers avoid unnecessarily constraining their programs to specific numeric representations or precision.

To demonstrate this hierarchy, we consider the concrete types comprised by `Number`. The names included in the table match the exact names in Julia. Note, nonetheless, that the full subtype hierarchy of `Number` is broader than the simplified representation presented. [5]

EXAMPLE OF THE ABSTRACT TYPE "NUMBER"



Note: Dashed red borders indicate abstract types, while solid blue borders indicate concrete types.

OPERATORS

In programming, **operators** are symbols that represent operations performed on objects. They can be thought of as syntactic sugar for functions, as we'll see in the next chapters. In fact, almost all operators in Julia can be employed as functions.

For instance, the symbol `+` in `x + y` is an operator that performs the addition of `x` and `y`. Likewise, the symbols `x` and `y` are referred to as the **operands**, representing the operator's inputs to perform its calculation. *Operators follow specific syntax rules based on the number of operands they require*. Understanding this syntax will prove useful for several topics covered later on the website. Next, we define and illustrate the syntax through several examples. At this point, just focus on how operators are written, even if their specific functions are not yet clear.

- **Unary operators**: They take *one operand*, with the operator written to the left of it. [6] Formally, their syntax is `<operator>x`, such as `√x` or `-x`.

- **Binary operators**: They take *two operands*, and the operator is written between them. [7] Formally, their syntax is `x <operator> y`, such as `x + y` or `x^y` for x^y.

- **Ternary operators**: They take *three operands*. Formally, their syntax is `x <operator1> y <operator2> z`. Ternary operators are rare, which is why the specific operator `x ? y : z` is directly referred to as *the* ternary operator. We'll see that this operator performs a conditional evaluation, returning `y` if `x` is true and `z` returned otherwise.

FOOTNOTES

[1.] While it's common to say that "a variable has a specific type", this is a simplification. Technically, it's the value of the variable that has a specific type, not the variable itself.

[2.] You can insert Unicode characters by copying and pasting them from a list like this one. Alternatively, you can use tab completion with the commands listed in the Julia documentation.

[3.] Not all symbols are allowed. For instance, names with common mathematical symbols like `x^` or `%x` aren't permitted. Additionally, numbers are allowed, but they can't be included as the first character (e.g., `2x` is invalid).

[4.] The suffix `64` in these types represents the precision of the number. This represents the maximum number of significant digits or decimals that a type can accurately represent.

[5.] The subtype `Signed` from `Integers` represents positive or negative integers. Although not included in the graph, there's also a type called `Unsigned`, which only accepts positive integers.

[6.] Operators to the left of the operand are known as **prefix operators**. Conversely, operators written to the right of the operand are known as **postfix operators**, and Julia has a few of them (e.g., `'` to transpose a vector or matrix `x`, which is written as `x'`). Despite this, we won't use postfix operators on this website.

[7.] Operators with this syntax are called **infix operators**.

2c. Numbers

Martin Alfaro
PhD in Economics

INTRODUCTION

The previous section introduced the concept of variables, distinguishing between those containing a single element (scalars) and collections. This section expands on scalars, exclusively focusing on those holding numeric values.

NUMBERS

Computers store numbers in various formats, treating integers and decimal numbers as separate entities. Even within each category of numbers, multiple representations emerge depending on the intended level of precision. This precision is determined by the number of bits allocated to store values in memory, which in turn defines the maximum range of values that a data type supports. [1] The representation just described extends well beyond Julia, and is intrinsic to how computers operate at a fundamental level.

In modern computers, numbers typically have a default size of 64 bits, and Julia's default types for numbers are:

- `Int64` for integers.
- `Float64` for decimal numbers. [2]

> **Remark**
> Julia provides the type `Int` as a more versatile option than `Int64`, which adapts to your computer's architecture: `Int` defaults to `Int64` on 64-bit systems and `Int32` on 32-bit systems. Since most modern machines operate on a 64-bit architecture, `Int` typically defaults to `Int64`. Note that there's no equivalent type `Float` for floating-point numbers, with Julia always defaulting to `Float64`.

It's worth emphasizing that `Int64` and `Float64` are two different data types. Thus, while `1` is a value with type `Int64`, the same value becomes `1.0` as a `Float64` type.

NUMBERS

```
x = 1       # `Int64`

y = 1.0     # `Float64`
z = 1.      # alternative notation for `1.0`
```

Remark

To enhance code readability, you can break up long numbers by inserting underscores `_`.

NOTATION FOR NUMBERS

```
x = 1000000
y = 1_000_000             # equivalent to `x` and more readable

x = 1000000.24
y = 1_000_000.24              # '_' can be used with decimal
numbers
```

The type `Float64` encompasses not only decimal numbers, but also two special values: `Inf` for infinity and `NaN` for indeterminate expressions such as 0/0 (`NaN` stands for "not a number"). Considering this, all the following variables have type `Float64`.

FLOAT64

```
x = 2.5

y = 10/0

z = 0/0
```

```
julia> x
2.5
julia> y
Inf
julia> z
NaN
```

BOOLEAN VARIABLES

A distinct numeric type is `Bool`, which facilitates the representation of **Boolean variables**. These variables can only take on the values `true` and `false`. Internally, they're implemented as integers, with `true` corresponding to `1` and `false` to `0`. Because of this implementation, Julia accepts `1` and `0` interchangeably with `true` and `false`.

Boolean expressions come into play when evaluating conditions, such as checking whether a number exceeds a certain value or whether a string matches a specific pattern. These conditional evaluations yield Boolean values, and can then be employed to control the flow of the program. Some examples of Boolean values are presented below.

NOTATION FOR BOOLEAN

```
x = 2
y = 1

z = (x > y)       # is `x` greater than `y` ?
z = x > y         # same operation (don't interpreted it as 'z = x')
```

```
julia> z
true
```

ARITHMETIC OPERATORS

Numbers can be manipulated through a variety of **arithmetic operators**. These operators are represented by symbols akin to those in other programming languages.

Julia's Arithmetic Operator	Meaning
`x + y`	addition
`x - y`	subtraction
`x * y`	product
`x / y`	division
`x^y`	power (x^y)

It's worth noting that all the operators presented above are *binary*. Consequently, they follow the syntax `x <symbol> y`, as indicated in our **discussion** about the syntax of operators.

FOOTNOTES

[1.] For instance, 8-bit integers can only represent values from -128 to 127. Likewise, 32-bit floating-point numbers, used for decimal numbers, can represent up to 7 significant digits of precision.

[2.] The term "Float" stands for "floating point" and is how computers represent decimal numbers.

2d. Strings

[Martin Alfaro](#)
PhD in Economics

INTRODUCTION

This section presents types for text representation, distinguishing between characters and strings. The coverage will be concise, as the website won't focus on string analysis. However, a minimal coverage is necessary as string variables are important for tasks like specifying paths, saving files, and other core functionalities.

CHARACTERS

In Julia, the `Char` type is used to represent individual characters. A character `x` is defined by using single quotes, as in `'x'`. Given its support for Unicode characters, `Char` encompasses not only numbers and letters, but also a wide range of symbols. This is shown below.

```
# x equals the character 'a'
x = 'a'

# 'Char' allows for Unicode characters
x = 'β'
y = '🐒'
```

Notice that characters must be enclosed in single quotes `' '`, even for symbols like 🐒. Otherwise, Julia will interpret the expression as a variable.

```
# any character is allowed for defining a variable
🐒 = 2          # 🐒 represents a variable, just like if we had defined x = 2

y  = 🐒         # y equals 2
z  = '🐒'       # z equals the character 🐒
```

STRINGS

We'll rarely use the type `Char` directly. Instead, we'll work with the so-called type `String`. This is an ordered collection of characters, making it possible to represent text.

Strings can be defined through either double quotes `" "` or triple quotes `""" """`. The latter is particularly convenient for handling newlines, such as when the text has to span multiple lines. [1]

```
x = "Hello, beautiful world"

x = """Hello, beautiful world"""
```

STRING INTERPOLATION

String interpolation allows you to embed Julia code within a string, which is then evaluated and replaced in the string with its value.

To interpolate an expression, you must simply prefix the string with the `$` symbol. If the expression contains spaces, you'll need to enclose it in curly braces, like `$()`. Both cases are exemplified below.

```
number_students = 10


output_text     = "There are $(number_students) students in the course"
```

```
julia> x
"There are 10 students in the course"
```

```
number_matches  = 50
goals_per_match = 2

output_text     = "Last year, Messi scored $(number_matches * goals_per_match) goals"
```

```
julia> x
"Last year, Messi scored 100 goals"
```

FOOTNOTES

[1] For more on the differences between double and triple quotes, see here

2e. Arrays (Vectors and Matrices)

Martin Alfaro
PhD in Economics

INTRODUCTION

So far, we've explored variables representing single-element objects. Now, we'll shift our focus to **collections**, defined as **variables comprising multiple elements**. Julia provides several forms of collections, including:

- Arrays (including vectors and matrices)
- Tuples and Named Tuples
- Dictionaries
- Sets

Arrays represent one of the most common data structures for collections. They are formally defined as objects with type `Array{T,d}`, where `d` is the array's dimension and `T` is its elements' type (e.g., `Int64` or `Float64`).

Two special categories of arrays are **vectors** (1-dimensional arrays) and **matrices** (2-dimensional arrays). Vectors are represented by the type `Vector{T}`, which is an alias for `Array{T,1}`. For its part, matrices use the type `Matrix{T}`, which is an alias for `Array{T,2}`. Although we provide a subsection about matrices at the end, this is labeled as optional. The reason is that vectors are sufficient for conveying the topics of this website.

> **Remark**
>
> **Julia uses 1 as an array's first index**. This contrasts with many other languages (e.g., Python), where 0 is used as the first index.

VECTORS

Vectors in Julia are defined as *column-vectors*, and their elements are separated by a comma or a semicolon.

```
x = [1, 2, 3]          #= column-vector (defined using commas or semicolons)
                           Vector{Int64} (alias for Array{Int64, 1}) =#

x = [1; 2; 3]          # equivalent notation to define `x`
```

```
julia> x
3-element Vector{Int64}:
 1
 2
 3
```

> **Remark**
>
> Arrays can hold elements of various types, such as numbers and strings. For example, `[1, 2.5, "Hello"]` is a valid vector in Julia, identifying its elements as having type `Any` (recall that `Any` encompasses all the possible types supported by Julia). While arrays mixing types can be created, they're highly discouraged for several reasons, including performance.

ACCESSING A VECTOR'S ELEMENTS

Given a vector `x`, we can access its *i*-th element with `x[i]` and retrieve all its elements with `x[:]`.

```
x = [4, 5, 6]
```

```
julia> x
3-element Vector{Int64}:
 1
 2
 3
```

```
julia> x[2]
5
julia> x[:]
3-element Vector{Int64}:
 4
 5
 6
```

It's also possible to access a subset of `x`'s elements. There are several approaches to achieve this, and we'll only present two basic ones at this point. The simplest method involves setting the indices **via a vector**, using the syntax `x[<vector>]`.

```
x = [4, 5, 6, 7, 8]
```

```
julia> x
3-element Vector{Int64}:
 1
 2
 3
```

```
julia> x[[1,3]] # elements of 'x' with indices 1 and 3
2-element Vector{Int64}:
 4
 6
julia> x[1,3] # be careful! this is the notation used for matrices, indicating 'x[row 1, column 3]'
ERROR: BoundsError: attempt to access 5-element Vector{Int64} at index [1, 3]
```

The second approach sets the indices **via ranges**. These are denoted as `<first>:<steps>:<last>`, with Julia assuming increments of one if we omit `<steps>`. To respectively express the first and last index in a range, you can use the keywords `begin` and `end`.

```
x = [4, 5, 6, 7, 8]
```

```
julia> x
3-element Vector{Int64}:
 1
 2
 3
```

```
julia> x[1:2] # steps with unit increments (assumed by default)
2-element Vector{Int64}:
 4
 5
julia> x[1:2:5] # steps with increments of 2 (explicitly indicated)
3-element Vector{Int64}:
 4
 6
 8
julia> x[begin:end] # all elements. Equivalent to 'x[:]' or 'x[1:end]'
3-element Vector{Int64}:
 4
 5
 6
 7
 8
```

MATRICES (*OPTIONAL*)

Matrices can be defined as collections of row- or column-vectors. If they're created through multiple row vectors, each row has to be separated by a semicolon `;`. If we instead adopt multiple column vectors, their elements need to be separated by a space.

Note that row vectors are considered as special cases of matrices, with their elements separated by a space—they're matrices with multiple columns having one element.

```
X = [1 2 ; 3 4]       #= matrix as a collection of row-vectors, separated by semicolons
                         Matrix{Int64} (alias for Array{Int64, 2})=#

X = [ [1,3] [2,4] ]    # identical to `C`, but defined through a collection of column-
vectors

Y = [1 2 3]           #= row-vector (defined without commas)
                         Matrix{Int64} (alias for Array{Int64, 2}) =#
```

```
julia> X
2×2 Matrix{Int64}:
 1  2
 3  4
julia> Y
1×3 Matrix{Int64}:
 1  2  3
```

ACCESSING A MATRIX'S ELEMENTS

Given a matrix `X`, we can access its element at row `r` and column `c` by `X[r,c]`. Likewise, the i-th element of a row vector is accessed with `X[i]`. [1] Moreover, we can select all elements across the row `r` by `X[r,:]`, and all elements of column `c` by `X[:,c]`.

```
X = [5 6 ; 7 8] # matrix

Y = [4 5 6]     # row-vector
```

```
julia> X
2×2 Matrix{Int64}:
 5  6
 7  8
julia> X[2,1]
7
julia> X[1,:]
2-element Vector{Int64}:
 5
 6
julia> X[:,2]
2-element Vector{Int64}:
 6
 8
julia> Y[2]
5
```

To access a subset of elements, you must follow the same approaches as with vectors, but applied to either rows or columns.

```
X = [5 6 ; 7 8]
```

```
julia> X
2×2 Matrix{Int64}:
 5  6
 7  8
julia> X[[1,2],1]
2-element Vector{Int64}:
 5
 7
julia> X[1:2,1]
2-element Vector{Int64}:
 5
 7
julia> X[begin:end,1]
2-element Vector{Int64}:
 5
 7
```

FOOTNOTES

[1] We could also use this approach for any matrix, as Julia also accepts a linear index for matrices. For instance, a 3x3 matrix accepts indices between 1 and 9. However, unless you want to iterate over all elements of a matrix, the notation `X[r,c]` is easier to interpret.

2f. Tuples

Martin Alfaro
PhD in Economics

INTRODUCTION

We continue our exploration of *collections*, defined as objects storing multiple elements. Having previously focused on arrays, we'll now turn our attention to another form of collection known as *tuples*.

The defining characteristic of tuples is their fixed size and immutability. This implies that, once a tuple is created, its elements cannot be added, removed, or modified. While these restrictions might initially seem limiting, they also bring substantial performance gains when working with small collections.

At this point, we'll only touch on the basics. A more in-depth exploration of tuples will follow in Part II, after we've developed the necessary tools to understand the role of tuples in high-performance scenarios.

> **Warning!**
>
> **Tuples should only be used when the collection comprises a small number of elements**. Large tuples will result in slow computations at best, or directly trigger fatal errors. For large vectors, you should keep relying on vectors.

DEFINITION OF TUPLES

The syntax for accessing the *i*-th element of a tuple `x` is `x[i]`, similar to vectors. Likewise, defining tuples requires enclosing their elements in parentheses `()`, which contrasts with the square brackets `[]` used in vectors.

When tuples comprise more than one element, the use of `()` is optional and its omission is in fact a common practice. On the contrary, single-element tuples have stricter syntax rules: you must use parentheses `()` and a trailing comma `,` after the element. For example, a tuple with the single element `10` is represented as `(10,)`. This notation differentiates a tuple from the expression `(10)`, which would be interpreted simply as the number 10.

```
x = (4,5,6)
x =  4,5,6           #alternative notation
```

```
julia> x
(4, 5, 6)
julia> x[1]
4
```

```
x = (10,)    # not x = (10) (it'd be interpreted as x = 10)
```

```
julia> x
(10,)
julia> x[1]
10
```

TUPLES FOR ASSIGNMENTS

Tuples are particularly useful for simultaneously assigning values to multiple variables. This is achieved by placing **a tuple on the left-hand side of** `=` and **a collection on the right-hand side** (either another tuple or a vector). The following examples demonstrate both options.

```
(x,y) = (4,5)
 x,y  =  4,5       #alternative notation
```

```
julia> x
4
julia> y
5
```

```
(x,y) = [4,5]
 x,y  = [4,5]      #alternative notation
```

```
julia> x
4
julia> y
5
```

As we'll see later, this technique is commonly employed when a function returns multiple values, enabling you to unpack the returned values into individual variables.

3a. Overview and Goals

Martin Alfaro
PhD in Economics

The upcoming Chapters 3 and 4 will cover three core tools for programming: functions, conditional statements, and for-loops. Chapter 3 in particular focuses on **functions**, which constitute the backbone of Julia programming. As they're tightly linked to achieving high performance, we'll dedicate considerable time to discussing their usage.

Our coverage of functions will be organized into three categories, based on who defines them:

i) built-in functions,

ii) third-party functions, and

iii) user-defined functions.

The first two types of functions become available in the workspace via packages, which may be loaded implicitly or explicitly. This connection between packages and functions leads us into exploring the concepts together in Section 3b. Instead, user-defined functions are left for Section 3c.

A firm grasp of functions requires understanding variable scope, including the distinction between global and local variables. By establishing this difference, we'll frame functions as self-contained mini-programs designed to perform a specific task. Both subjects are presented together in Section 3d. This perspective on functions will lead to the identification of good practices for using functions, which will have significant implications for the structure of code as we progress. At this point, nonetheless, it suffices if you start becoming familiar with this view.

Finally, we'll introduce the concept of broadcasting in Section 3e. Mastering this technique is crucial, as it lets you seamlessly apply the same function to each element in a collection. Broadcasting is a widely used technique not only in Julia, but also in other programming languages like Python.

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** format for function names. This consists of lowercase letters, numbers, and possibly underscores to separate words (e.g., `snake_case123`). Note that 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 the following.

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

Function in Julia	Meaning
`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!**
>
> Applying macros requires extreme caution, as 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 is modified by a macro 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.

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 notation that omits the parentheses
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
```

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 COMBINED

```
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.

GENERAL NOTATION

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

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

OMITTING INPUTS

```
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
```

USING ARGUMENTS AS INPUTS OF 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 TO DEFINE 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`.

WITHOUT AN ANONYMOUS FUNCTION

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

result     = map(add_two, x)
```

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

THROUGH AN 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]`.

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}:
 3
 4
 5
```

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
```

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
```

A FUNCTION'S DOCUMENTATION (*OPTIONAL*)

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 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, see 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 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".

3d. Variable Scope & Relevance of Functions

Martin Alfaro
PhD in Economics

INTRODUCTION

Variable scope refers to the code block in which a variable is accessible. The concept allows us to distinguish between **global variables**, which are accessible in any part of the code, and **local variables**, which are confined to specific blocks like functions or loops. The existence of scopes determines that the same variable `x` could refer to different objects, depending on where it's called.

When it comes to functions, Julia adheres to specific rules for variable scope. Specifically, given a variable `x` defined outside a function:

- if a new variable `x` is defined inside a function or is passed to a function as an argument, then `x` is considered *local* to that function. This means that any reference to `x` within the function refers to the local variable, without any relation to the variable `x` defined outside the function,
- if a function doesn't define a new `x` nor `x` is a function argument, then `x` refers to the variable defined outside the function (i.e., the global variable).

In this section, we'll show how these rules work in practice.

GLOBAL AND LOCAL VARIABLES

A variable that is local to a function exists solely within that function's scope. This means that these variables cease to exist once the function finishes executing. Consequently, any attempt to reference local variables outside the function will result in an error.

Variables local to a function encompass:

1. the function arguments,
2. the variables defined in the function body.

Any other variable included in a function that's not *i)* or *ii)* necessarily refers to a global variable.

Understanding which variables are local or global is essential for predicting a program's behavior. This is because a local variable may share the same name as a global one, without them being related. The following examples help clarify the differences between global and local variables.

```
x = "hello"

function foo(x)                 # 'x' is local, unrelated to 'x = hello' above
    y = x + 2                   # 'y' is local, 'x' refers to the function argument

    return x,y
end
```

```
julia> foo(1)
1                           # local x
3                           # local y
julia> x
"hello"
julia> y
ERROR: UndefVarError: y not defined
```

```
z = 2

function foo(x)
    y = x + z                       # 'x' refers to the function argument, 'z' refers to the
global

    return x,y,z
end
```

```
julia> foo(1)
1                           # local x
3                           # local y
2                           # global z
julia> x
ERROR: UndefVarError: x not defined
julia> z
2
```

THE ROLE OF FUNCTIONS

In programming, **functions** can be understood **as self-contained mini-programs to represent specific tasks**. Under this interpretation, local variables simply act as labels that help articulate the mechanics of the task. Consequently, their inaccessibility outside the function emerges naturally. [1]

To explain this view of functions, consider a variable `x`, along with another variable `y` computed by transforming `x` through a function `f`. In particular, assume a transformation that doubles `x`, so that `y = 2 * x`. The following are two approaches to calculating `y`.

```
x = 3

double() = 2 * x
y        = double()
```

```
x = 3

double(x) = 2 * x
y         = double(x)
```

```
x = 3

double(🐒) = 2 * 🐒
y          = double(x)
```

The function in Approach 1 relies on the global variable `x`. This practice is highly discouraged for several reasons. Firstly, it prevents the reusability of the function, as it's specifically designed to double the global variable `x`, rather than acting as a mini-program that doubles *any* variable.

Second, the inclusion of the global variable `x` compromises the function's self-containment, as the function's output depends on the value of `x` at the moment of execution. If you work on a long project, this will turn the code prone to bugs.

Lastly, global variables have a detrimental impact on performance, a topic we'll study later on the website. In fact, global variables in Julia are directly a performance killer.

In contrast, Approach 2 refers to `x` as a local variable. This `x` is unrelated to the global variable `x`—it simply serves as a label to identify the variable to be doubled. Indeed, we could've replaced `x` with any other label, as demonstrated in Approach 3 through the monkey emoji, `🐒`.

By avoiding referencing any variable outside its scope, Approach 2 makes the function self-contained. This allows users to easily anticipate the consequence of executing `double` through a simply inspection of the function, eliminating the need to review the entire codebase. Thus, Approach 2 aligns with the interpretation of a function as a self-contained mini-program: the function embodies the task of doubling a variable, turning the function reusable and applicable to any variable. In this context, applying `double` to the global variable `x` becomes just one possible application.

RECOMMENDATIONS FOR THE USE OF FUNCTIONS

Structuring code around functions offers numerous advantages. However, to fully realize these benefits, users must adhere to certain principles when writing code. This section outlines a few of them and should be considered as a mere introduction to the subject. The topic will be investigated further, when we explore high performance.

AVOID GLOBAL VARIABLES IN FUNCTIONS

Global variables are strongly discouraged. This is not only due to the reasons mentioned previously, but also because they can have a devastating impact on performance. The easiest solution to this issue is to pass global variables as function arguments. This practice will actually become second nature once you start viewing functions as self-contained mini-programs. Specifically, by adopting this

perspective, you'll conceive local variables as labels to describe a task, rather than references to global variables. This shift in mindset can help you write more efficient and maintainable code.

AVOID REDEFINING VARIABLES WITHIN FUNCTIONS

The suggestion applies to both local variables and function arguments. Redefining these variables can have several disadvantages, including reduced code readability and potential performance degradation. Therefore, it's recommended that you define new variables instead of redefining existing ones. This approach is demonstrated in the following example.

```
function foo(x)
   x      = 2 + x          # redefines the argument

   y      = 2 * x
   y      = x + y          # redefines a local variable
end
```

```
function foo(x)
   z      = 2 + x          # new variable

   y      = 2 * x
   output = z + y          # new variable
end
```

(OPTIONAL) - Another Issue of Redefining Variables

MODULARITY

We've emphasized the importance of viewing functions as self-contained mini-programs, designed to perform specific tasks. This perspective leads us to highlight the importance of **modularity**: the practice of breaking down a program into multiple small functions, each with its own distinct purpose, inputs, and outputs.

The primary benefit of modularity is the ability to work with independent code blocks. By keeping these blocks separate, we can decompose complex problems into multiple manageable tasks, making it easier to test and debug code. Additionally, modularity makes it possible to eventually improve or substitute parts of the code, without breaking the entire program.

A helpful way to understand this principle is by considering the analogy of building a Lego minifigure. In the first step, multiple blocks are created independently, each representing a specific part of the figure, such as the legs, torso, arms, and head. Then, in the second stage, these individual blocks are brought together and assembled into an integrated minifigure.

This two-step approach offers several advantages. By focusing on each block individually, we can concentrate and refine each part without worrying about the entire structure. Additionally, it provides great flexibility: since each block is created independently, we can modify specific blocks without having to rebuild the entire figure. For instance, if we want to change the figure's head, we can simply swap out the corresponding block, without starting from scratch.

The principle of modularity is closely tied to the suggestion of writing short functions. Some proponents even argue that functions should be limited to fewer than five lines of code Indeed, entire books have been written based on this principle. Although this viewpoint may be considered rather extreme, it clearly emphasizes the advantages of avoiding lengthy functions.

(OPTIONAL) - Example of Modularity

FOOTNOTES

[1] Local variables play a similar role to integration variables in math. Formally, in $\int f(t)\ \mathrm{d}t$ for some function f, t just represents a symbol indicating over which variable we're integrating. The integral could be equivalently expressed using any other integration variable, such as x in $\int f(x)\ \mathrm{d}x$.

3e. Map and Broadcasting

Martin Alfaro
PhD in Economics

INTRODUCTION

This section explores element-wise operations on **iterable collections**. These are defined as collections whose elements can be accessed sequentially, including examples like vectors, tuples, and ranges.

The first approach covered is the `map` function, which applies a given function to each element of a collection. This function is particularly convenient for transforming collections while avoiding for-loops.

After this, we'll shift our focus to a fundamental technique in Julia known as **broadcasting**. This enables the application of functions and operators element-wise, while maintaining concise and expressive code. Broadcasting is quite versatile, supporting operations on collections of equal size or combinations of scalars and same-size collections. Its distinctive syntax, which involves appending a dot `.` to the function/operator, makes it easily identifiable throughout the code.

> **Remark**
>
> The terms **broadcasting** and **vectorization** will be used interchangeably throughout the website, although strictly speaking they're not equivalent. [1] Furthermore, vectorization has multiple meanings, depending on the context in which the definition is applied.

> **Warning!**
>
> Later on the website, we'll explore **for-loops** as an alternative approach to transforming arrays. Several languages strongly recommend vectorizing operations to improve speed, instead highly discouraging for-loops. **Such advice does not apply to Julia**. In fact, when it comes to optimizing code in Julia, for-loops are often the key to achieving faster performance.
>
> Considering this, the main advantage of vectorization in Julia is to streamline code without sacrificing speed.

THE "MAP" FUNCTION

The `map` function is available in most programming languages, allowing you to take a collection and generate a new one with transformed elements. It can be applied in two ways, depending on the number of inputs passed.

In its simplest form, `map` takes a single-argument function `foo` and a collection `x`. Its syntax is `map(foo,x)`, returning a new collection with `foo(x[i])` as *i*-th element. `map` is commonly applied with an anonymous function playing the role of `foo`, as illustrated below.

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


z = map(log,x)
```

```
julia> z
3-element Vector{Float64}:
 0.0
 0.69315
 1.09861
julia> [log(x[1]), log(x[2]), log(x[3])]
3-element Vector{Float64}:
 0.0
 0.69315
 1.09861
```

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


z = map(a -> 2 * a, x)
```

```
julia> z
3-element Vector{Int64}:
 2
 4
 6
julia> [2*x[1], 2*x[2], 2*x[3]]
3-element Vector{Int64}:
 2
 4
 6
```

The second way to apply `map` arises when the function `foo` takes multiple arguments. In case `foo` is a two-argument function, the syntax is `map(foo, x, y)`, returning a new collection whose *i*-th element is `foo(x[i], y[i])`. When the collections `x` and `y` have different sizes, `foo` **is applied element-wise until the shortest collection is exhausted**. This rule applies even when either `x` or `y` is a scalar, in which case `map` would return a single element.

For demonstrating its use, let's consider the addition operation. As you may recall, `+` denotes both an operator (e.g., `2 + 3`) and a function (e.g., `+(2, 3)`). By using `+` in particular as a function, `map` can perform element-wise additions across multiple collections.

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

z = map(+, x, y)        # recall that `+` exists as both operator and function
```

```
julia> z
3-element Vector{Int64}:
 0
 0
 0
julia> [+(x[1],y[1]), +(x[2],y[2]), +(x[3],y[3])]
3-element Vector{Int64}:
 0
 0
 0
```

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

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

```
julia> z
3-element Vector{Int64}:
 0
 0
 0
julia> [x[1]+y[1], x[2]+y[2], x[3]+y[3]]
3-element Vector{Int64}:
 0
 0
 0
```

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

z = map(+, x, y)        # recall that `+` is both an operator and a function
```

```
julia> z
2-element Vector{Int64}:
 0
 0
julia> [+(x[1],y[1]), +(x[2],y[2])]
2-element Vector{Int64}:
 0
 0
```

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

z = map(+, x, y)        # recall that `+` is both an operator and a function
```

```
julia> z
1-element Vector{Int64}:
 0
julia> [+(x[1],y[1])]
1-element Vector{Int64}:
 0
```

BROADCASTING

The function `map` can rapidly become unwieldy when dealing with complex functions or multiple arguments. This is where broadcasting comes into play, offering a more streamlined syntax.

Next, we'll explore the concept of broadcasting in a step-by-step manner. First, we'll show how it applies to collections of equal size, covering both functions and operators. After this, we'll demonstrate that broadcasting accepts combinations of scalars and collections, even though it typically doesn't support operations with collections of different sizes. In such instances, the scalar is treated as a vector that matches the size of the corresponding collections.

Unlike other programming languages, **broadcasting is an intrinsic feature of Julia** and thereby applicable to *any* function or operator, including user-defined ones.

BROADCASTING FUNCTIONS

Broadcasting expands the versatility of functions, allowing them to be applied element-wise to a collection. This feature is implemented by appending a dot ***after*** the name of the function, as in `foo.(x)`.

Remarkably, **any function `foo` has a broadcasting counterpart `foo.`**. This entails that broadcasting is automatically available for user-defined functions. Furthermore, it determines that broadcasting isn't restricted to numeric collections, but to any type of collection.

Similarly to `map`, broadcasting can be applied to both single- and multiple-argument functions. Each case warrants separate consideration.

As for single-argument functions, broadcasting `foo` over a collection `x` returns a new collection with `foo(x[i])` as its *i*-th element. The following examples demonstrate this.

```
# `log(a)` is a function appying to scalars `a`

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

```
julia> log.(x)
3-element Vector{Float64}:
 0.0
 0.69315
 1.09861
julia> [log(x[1]), log(x[2]), log(x[3])] # identical to log.(x)
3-element Vector{Float64}:
 0.0
 0.69315
 1.09861
```

```
square(a) = a^2     #user-defined function for a single element 'a'

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

```
julia> square.(x)
3-element Vector{Int64}:
 1
 4
 9
julia> [square(x[1]), square(x[2]), square(x[3])] # identical to square.(x)
3-element Vector{Int64}:
 1
 4
 9
```

As for multiple-argument functions, suppose a function `foo` and collections `x` and `y`. Then, `foo.(x,y)` returns a new collection with `foo(x[i],y[i])` as its *i*-th element.

Importantly, **collections with different sizes aren't allowed**, establishing a clear contrast between broadcasting and `map`. The sole exception to this rule is when one of the objects is a scalar, as we'll see later.

Below, we provide several examples. The first example in particular makes use of the built-in function `max`, which provides the maximum value among its scalar arguments.

```
# 'max(a,b)' returns 'a' if 'a>b', and 'b' otherwise

x        = [0, 4, 0]
y        = [2, 0, 8]
```

```
julia> max.(x,y)
3-element Vector{Float64}:
 2
 4
 8

julia> [max(x[1],y[1]), max(x[2]),y[2]), max(x[3]),y[3])] # identical to max.(x,y)
3-element Vector{Float64}:
 2
 4
 8
```

```
foo(a,b) = a + b        # user-defined function for single elements 'a' and 'b'

x        = [-2, -4, -10]
y        = [ 2,  4,  10]
```

```
julia> foo.(x)
3-element Vector{Int64}:
 0
 0
 0

julia> [foo(x[1],y[1]), foo(x[2]),y[2]), foo(x[3]),y[3])] # identical to foo.(x,y)
3-element Vector{Float64}:
 0
 0
 0
```

Remark

Broadcasting applies not only to numeric functions, but to any function. For instance, consider the built-in function `string`, which concatenates its arguments to form a sentence (e.g., `string("hello ","world")` returns `"hello world"`).

```
country = ["France", "Canada"]
is_in   = [" is in "  , " is in "]
region  = ["Europe", "North America"]
```

```
julia> string.(country, is_in, region)
2-element Vector{String}:
 "France is in Europe"
 "Canada is in North America"
```

BROADCASTING OPERATORS

It's also possible to **broadcast operators**, making them apply element-wise. Its use requires prepending a dot ***before*** the operator.

For its application, it's helpful to recall the classification of operators by the number of operands, as this determines their syntax. Specifically, the syntax of *unary operators* is `<symbol>x`, so that `.√x` broadcasts `√`. Likewise, the syntax for *binary operators* is `x <symbol> y`, such that `x .+ y` computes the element-wise sum of vectors `x` and `y`, resulting in `[x[1]+y[1], x[2]+y[2], ...]`.

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

```
julia> x .+ y
3-element Vector{Int64}:
 0
 0
 0
```

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

```
julia> .√x
3-element Vector{Float64}:
 1.0
 1.41421
 1.73205
```

BROADCASTING OPERATORS WITH SINGLE-ELEMENT OBJECTS

In all the cases covered so far, broadcasting was applied with inputs of the same size. In general, collections of dissimilar size, such as `x = [1,2]` and `y=[3,4,5]`, aren't allowed.

One exception to this rule occurs when broadcasting applies to vectors of equal size combined with scalars. In these cases, scalars are treated as objects having the same size as the vectors, with all entries equal to the scalar. For example, given `x = [1,2,3]` and `y = 2`, the expression `x .+ y` produces the same result as defining `y = [2,2,2]` and then executing `x .+ y`. This is demonstrated below.

```
x = [0,10,20]
y = 5
```

```
julia> x .+ y
3-element Vector{Int64}:
 5
 15
 25
```

```
x = [0,10,20]
y = [5, 5, 5]
```

```
julia> x .+ y
3-element Vector{Int64}:
 5
 15
 25
```

Remark

We emphasize that broadcasting can be applied to any iterable collection. Thus, the example based on strings presented above can be rewritten as follows.

▼ Code

```
country       =  ["France", "Canada"]
is_in         =  " is in "
region        =  ["Europe", "North America"]
```

```
julia> string.(country, is_in, region)
2-element Vector{String}:
 "France is in Europe"
 "Canada is in North America"
```

ITERABLE OBJECTS

So far, our examples have focused on broadcasting using vectors as collections. Furthermore, we've explored the technique by treating functions and operators separately, which sheds light on the underlying mechanics of broadcasting. Next, we'll take a more comprehensive perspective, applying broadcasting to other types of collections and to expressions combining functions and operators.

We first show that broadcasting can be applied to any iterable object, including tuples and ranges.

```
x = (1, 2, 3)    # or simply x = 1, 2, 3
```

```
julia> log.(x)
(0.0, 0.69315, 1.09861)
julia> x .+ x
(2, 4, 6)
```

```
x = 1:3
```

```
julia> log.(x)
(0.0, 0.69315, 1.09861)
julia> x .+ x
(2, 4, 6)
```

```
x = (1, 2, 3)    # or simply x = 1, 2, 3
y = 1:3
```

```
julia> x .+ y
(2, 4, 6)
```

Furthermore, it's possible to simultaneously broadcast operators and functions. Given the pervasiveness of such operations, Julia provides the macro `@.` for an effortless application. The macro should be added at the beginning of the statement, and has the effect of automatically adding a "dot" to each operator and function found.

To demonstrate its use, consider adding two vectors element-wise, which we then transform by squaring the elements of the resulting vector.

```
x = [1, 0, 2]
y = [1, 2, 0]

square(x) = x^2
```

```
julia> square.(x .+ y)
3-element Vector{Int64}:
 4
 4
 4
```

```
x = [1, 0, 2]
y = [1, 2, 0]

square(x) = x^2
```

```
julia> @. square(x + y)
3-element Vector{Int64}:
 4
 4
 4
```

```
x = [1, 0, 2]
y = [1, 2, 0]

temp = x .+ y
z    = temp .^ 2
```

```
julia> temp
3-element Vector{Int64}:
 2
 2
 2
julia> z
3-element Vector{Int64}:
 4
 4
 4
```

BROADCASTING FUNCTIONS VS BROADCASTING OPERATORS

We've demonstrated that both functions and operators can be broadcasted. This lets us implement operations in two distinct ways: either broadcast a function that operates on a single element or define a function that directly performs the broadcasted operation.

The examples below demonstrate that the same output is obtained using either approach. For the illustration, we suppose that the goal is to square each element of `x`.

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

number_squared(a) = a ^ 2          # function for a single element 'a'
```

```
julia> number_squared.(x)
3-element Vector{Int64}:
 1
 4
 9
```

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

vector_squared(x) = x .^ 2          # function for a vector 'x'
```

```
julia> vector_squared(x) # '.' not needed (it'd be redundant)
3-element Vector{Int64}:
 1
 4
 9
```

While both approaches yield the same output, **defining a function that operates on a scalar is the more advisable choice**. This is due to a couple of reasons. Firstly, a function like `number_squared(a)` enables users to seamlessly perform computations on both scalars and collections. This is achieved by

simply choosing between executing the function or its broadcasted version. A corollary of this is that scalar functions avoid committing to a specific application. Secondly, the notation `number_square.(x)` explicitly conveys that the operation is element-wise, an aspect that would remain hidden in `vector_squared(x)`.

BROADCASTING OVER ONE ARGUMENT ONLY

When we broadcast a function or operator over some vectors `x` and `y`, both objects are simultaneously iterated. However, there are instances where we only want to iterate over one argument, keeping the other argument fixed. A typical scenario is when we need to check whether elements from `x` match any values in a predefined list `y`.

To illustrate how this can be achieved via broadcasting, we first introduce the function `in(a, list)`. This assesses whether the scalar `a` equals some element in the vector `list`. For instance, executing `in(2, [1,2,3])` returns `true`, because `2` belongs to `[1,2,3]`.

Suppose now that, instead of a scalar `a`, we have a vector `x`. The goal then is to verify whether *each* of the elements in `x` is present in `list = [1,2,3]`. Below, we show that this operation can't be directly implemented by broadcasting `in`.

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

```
julia> in.(x, list)
ERROR: DimensionMismatch: arrays could not be broadcast to a common size; got a dimension
with lengths 2 and 3
```

```
x    = [1, 2, 4]
list = [1, 2, 3]
```

```
julia> in.(x, list)
3-element BitVector:
 1
 1
 0
```

In the first example, `in.(x, list)` errors because `x` and `list` should either have the same size or one of them be a scalar. The second example does produce an output, but not the one we're looking for: it checks whether `1==1`, `2==2`, and `4==3`. Instead, our goal is to determine if `1` is in `[1,2,3]`, if `2` is in `[1,2,3]`, and if `3` is in `[1,2,3]`.

Intuitively, we need a mechanism to inform Julia that `list` should be treated as a single element while iterating over `x`. This can be accomplished in two different ways: either by enclosing `list` in a collection (e.g., a vector or tuple) or by using the `Ref` function.

As for the first approach, let's consider a tuple as the wrapping collection. Then, the implementation would be by writing `(list,)`, which converts the variable into a tuple whose only element is the tuple itself. [2] While explaining the specifics of `Ref` is beyond our current scope, Nonetheless, what matters for practical purposes is that `Ref(list)` makes `list` be treated as a singe element. Below, we demonstrate each approach.

```
x    = [2, 4, 6]
list = [1, 2, 3]        # 'x[1]' equals the element 2 in 'list'
```

```
julia> in.(x, [list])
3-element BitVector:
 1
 0
 0
```

```
x    = [2, 4, 6]
list = [1, 2, 3]        # 'x[1]' equals the element 2 in 'list'
```

```
julia> in.(x, (list,))
3-element BitVector:
 1
 0
 0
```

```
x    = [2, 4, 6]
list = [1, 2, 3]        # 'x[1]' equals the element 2 in 'list'
```

```
julia> in.(x, Ref(list))
3-element BitVector:
 1
 0
 0
```

The output vector we obtain in each case is what's known as a `BitVector`, where `1` corresponds to `true` and `0` to `false`. Therefore, the result is `[true, false, false]`, reflecting that `x[1]` is `2` and `2` belongs to `list`, whereas `x[2]` and `x[3]` don't equal any element in `list`.

> **Warning!**
> It's possible to use any collection to wrap `list`. However, we'll see in Part II of the book that there's some performance penalty involved when vectors are created. Consequently, **you should stick to `(list,)` rather than `[list]` when implementing this approach**.

While the previous example focused on the broadcasting of functions, the same principle applies to operators. This can be illustrated through the `∈` operator, which serves a similar purpose to the `in` function. Just like `in`, the `∈` operator determines whether a particular element exists within a collection. [3]

```
x    = [2, 4, 6]
list = [1, 2, 3]
```

```
julia> x .∈ (list,) # only 'x[1]' equals an element in 'list'
3-element BitVector:
 1
 0
 0
```

```
x    = [2, 4, 6]
list = [1, 2, 3]
```

```
julia> x .∈ Ref(list) # only 'x[1]' equals an element in 'list'
3-element BitVector:
 1
 0
 0
```

CURRYING AND FIXING ARGUMENTS (*OPTIONAL*)

Currying is a technique that transforms the evaluation of a function with multiple arguments into evaluating a sequence of functions, each with a single argument. [4] For instance, the curried version of `f(x,y)` would be written `f(x)(y)` and provide an identical output.

Our interest in currying lies in its ability to simplify broadcasting: it enables the treatment of an argument as a single object, without the need to use `Ref` or encapsulate objects as vectors/tuples. The technique could seem confusing for new users. In particular, it requires a good understanding of functions as first-class objects, entailing that functions can be treated as variables themselves. My primary goal is that you can at least recognize the syntax of currying, and thus be able to read code that applies the technique.

We start by illustrating how currying can be applied in general.

```
addition(x,y) = 2 * x + y
```

```
julia> addition(2,1)
5
```

```
addition(x,y) = 2 * x + y

# the following are equivalent
curried(x)    = (y -> addition(x,y))
curried       = x -> (y -> addition(x,y))
```

```
julia> curried(2)(1)
5
```

```
addition(x,y) = 2 * x + y
curried(x)    = (y -> addition(x,y))

# the following are equivalent
f             = curried(2)          # function of 'y', with 'x' fixed to 2
g(y)          = addition(2,y)
```

```
julia> f(1)
5
julia> g(1)
5
```

The key to understanding the syntax is that `curried(x)` is a function itself, with `y` as its argument. The second tab illustrates this clearly through the equivalence between `f = curried(2)` and `addition(2,y)`. These functions help us understand the logic behind curry, but are only useful for the specific case of `x=2`. Instead, `curried(x)` allows the user to call the function through `curried(x)(y)`, and so be used for any `x`.

As for broadcasting, any function `foo` in Julia can be broadcasted through `f.`. And we've determined that `curried(x)` is a function just like any other. Therefore, `curried(x)` plays the same role as `foo`, and so we can broadcast over `y` for a fixed `x` through `curried(x).(y)`.

```
a             = 2
b             = [1,2,3]

addition(x,y) = 2 * x + y
curried(x)    = (y -> addition(x,y))   # 'curried(x)' is a function, and 'y' its argument
```

```
julia> curried(a).(b)
3-element Vector{Int64}:
 5
 6
 7
```

```
a             = 2
b             = [1,2,3]

addition(x,y) = 2 * x + y
curried(x)    = (y -> addition(x,y))

#the following are equivalent
f    = curried(a)             # 'foo1' is a function, and 'y' its argument
g(y) = addition(2,y)
```

```
julia> f.(b)
3-element Vector{Int64}:
 5
 6
 7
julia> g.(b)
3-element Vector{Int64}:
 5
 6
 7
```

Let's now explore how the currying technique can help treat a vector as a single element in broadcasting. To illustrate this, consider the function `in` used previously. This function has a built-in curried version, which can be applied through `in(list).(x)` for vectors `list` and `x`. To better demonstrate its usage, the following example compares an implementation with `Ref`, the built-in curried `in`, and our own curry implementation.

```
x    = [2, 4, 6]
list = [1, 2, 3]
```

```
julia> in.(x,Ref(list))
3-element BitVector:
 1
 0
 0
```

```
x    = [2, 4, 6]
list = [1, 2, 3]

our_in(list_elements) = (x -> in(x,list_elements))     # 'our_in(list_elements)' is a
function
```

```
julia> our_in(list).(x) # it broadcasts only over 'x'
3-element BitVector:
 1
 0
 0
```

```
x    = [2, 4, 6]
list = [1, 2, 3]
```

```
julia> in(list).(x) # similar to 'our_in'
3-element BitVector:
 1
 0
 0
```

FOOTNOTES

[1.] Vectorization refers to applications restricted to arrays of the same size, with broadcasting being an extension of it that allows for scalars.

[2.] Recall that tuples with a single element must be written with a trailing comma, as in `(list,)`. The expression `(list)` instead would be interpreted as `list`, and hence treated as a vector.

[3.] `∈` can also be applied as a function, with its syntax mirroring that of `in`. Thus, `∈(a, list)` for a scalar `a` yields the same results as `in(a, list)`.

[4.] The name comes from the mathematician Haskell Curry, not the spice!

4a. Overview and Goals

Martin Alfaro
PhD in Economics

After studying functions, Chapter 4 covers another two core tools for programming: **conditions** and **for-loops**.

At this point, we'll simply define the concepts, without emphasizing much on the most effective ways to apply them. Basically, you should focus on the approaches and syntax to express conditions and for-loops.

We also relegate the analysis of techniques that combine functions, conditions, and for-loops. The following chapters will show that their simultaneous use gives rise to important concepts of Julia's language, such as in-place functions.

4b. Conditions

Martin Alfaro
PhD in Economics

INTRODUCTION

This section lays the basics for incorporating conditions into our programs. Formally, conditions are defined as functions and operators that return true or false as their output. A common example of a condition is `x > y`.

To get the most out of this section, you should keep in mind the classification of operators discussed here. This establishes that operators can be categorized according to their number of operands. Specifically, **unary operators** act on a single operand and precede it (i.e. `<operator>x`), whereas **binary operators** take two operands and are placed between them (i.e. `x <operator> y`).

CONDITIONS

Conditions are represented as values with type `Bool`, evaluating to either `true` or `false`. These values are internally represented as integers restricted to `1` and `0`.

The representation of Boolean values in the REPL varies depending on their dimension: scalar `Bool` values are displayed as `true` and `false`, while `Bool` vectors use `1` and `0`. This is illustrated below.

```
x = 2

#`y` provides `true` or `false` as its output
y = (x > 0)
```

```
julia> y
true
```

```
x = 2

#'z' provides 'true' and 'false' as its output, represented by 1s and 0s
z = [x > 0, x < 0]
```

```
julia> z
2-element Vector{Bool}:
 1
 0
```

> **Warning!**
>
> Parentheses are optional when writing single conditions, allowing us to write `y = x > 0` rather than `y = (x > 0)`. Nonetheless, the former

syntax is somewhat ambiguous, with the risk of being potentially misinterpreted as `(y = x) > 0`. To avoid confusion, it's a good practice to always include parentheses. This is especially true when working with multiple conditions, where outcomes can be drastically altered.

The condition in the previous example was defined via the operator `>`. More generally, conditions accept **comparison operators**, which are *binary operators* that compare values of various types (e.g., numbers and strings). The next list defines the most common ones.

Comparison Operator	Meaning
`x == y`	equal
`x ≠ y` or `x != y`	not equal
`x < y`	lower than
`x ≤ y` or `x <= y`	lower or equal than
`x > y`	greater than
`x ≥ y` or `x >= y`	greater or equal than

Remark

The non-standard characters appearing in the table can be written using tab completion:

- `≠` via `\ne`, which stands for "not equal",
- `≥` via `\ge`, which stands for "greater or equal",
- `≤` via `\le`, which stands for "lower or equal".

Remark

Comparison operators are also available as functions. For instance, the following expressions are all valid:

```
==(1,2)     # same as 1 == 2
 ≠(1,2)     # same as 1 ≠ 2
 ≥(1,2)     # same as 1 ≥ 2
>=(1,2)     # same as 1 ≥ 2
 >(1,2)     # same as 1 > 2
```

LOGICAL OPERATORS

Logical operators allow us to combine multiple conditions into a single one. Formally, they take `Bool` expressions as their operands, and return another `Bool` as their output. The following are the main logical operators used in Julia.

Logical Operator	Meaning
`x && y`	`x` and `y`
`x || y`	`x` or `y`
`!x`	negation of `x`

Notice that `&&` and `||` follow the syntax rules of *binary* operators.

```
x = 2
y = 3

# are both variables positive?
z1 = (x > 0) && (y > 0)

# is either `x` or `y` (or both) positive?
z2 = (x > 0) || (y > 0)
```

```
julia> z1
true
julia> z2
true
```

Another operator taking conditions as their operands is the "not" operator, represented by `!`. This is a unary operator that inverts a condition's value, changing `true` to `false` and vice versa. To use it, you simply place `!` at the start of the condition (i.e., before the parentheses).

As an illustration, the variables `y1` and `y2` below become equivalent via `!`.

```
x = 2

# is `x` positive?
y1 = (x > 0)

# is `x` not less than zero nor equal to zero? (equivalent)
y2 = !(x ≤ 0)
```

```
julia> y1 #identical output as 'y2'
true
```

LOGICAL OPERATORS AS SHORT-CIRCUIT OPERATORS

A key feature of `&&` and `||` is that they're **short-circuit operators**. This means that, once an operand is evaluated, the remaining operands are evaluated only if the previous operands didn't establish the truth or falseness of the expression. Specifically:

- `(x > 0) || (y > 0)`
  This expression is true when *at least one condition* is satisfied. Thus, Julia begins by analyzing `x > 0`. If this expression is true, it immediately returns `true`, without evaluating any subsequent expression. Only when `x > 0` is false will Julia evaluate `y > 0`.

- `(x > 0) && (y > 0)`
  This expression is true if *both conditions* are satisfied. Thus, Julia begins by analyzing `x > 0`. If this expression is false, it immediately returns `false`, without evaluating any subsequent expression. Only when `x > 0` is true will Julia evaluate `y > 0`.

Since not all operands are always evaluated, it's possible to get a result even if some operands contain invalid expressions. This is shown in the next example, where we include invalid Julia code as a condition.

```
x = 10
```

```
julia> (x < 0) && (this-is-not-even-legitimate-code)
false
julia> (x > 0) && (this-is-not-even-legitimate-code)
ERROR: UndefVarError: `this` not defined
```

```
x = 10
```

```
julia> (x > 0) || (this-is-not-even-legitimate-code)
true
julia> (x < 0) || (this-is-not-even-legitimate-code)
ERROR: UndefVarError: `this` not defined
```

PARENTHESIS IN MULTIPLE CONDITIONS

The inclusion of parentheses isn't crucial when working with only two conditions. This is because expressions like `(x > 0) && (y > 0)` can be safely written as `x > 0 && y > 0`, without much risk of confusion.

On the contrary, when dealing with three or more conditions, the lack of parentheses can drastically impact the expected behavior of an expression. The following example illustrates this point.

```
x = 5
y = 0
```

```
julia> x < 0 && y > 4 || y < 2
true
```

```
x = 5
y = 0
```

```
julia> (x < 0) && (y > 4 || y < 2)
false
```

```
x = 5
y = 0
```

```
julia> (x < 0 && y > 4) || (y < 2)
true
```

In the example, the expression without parenthesis is equivalent to the last tab's, since `&&` **has higher precedence than** `||` in Julia: when both `&&` and `||` are used, `&&` will be evaluated first.

To avoid confusion when more than two conditions are incorporated, **we'll always add parentheses**. This improves readability and spares us the need to memorize specific rules. The next optional subsection covers Julia's precedence rules in more detail. However, if you'll consistently enclose conditions in parentheses, you can safely skip it.

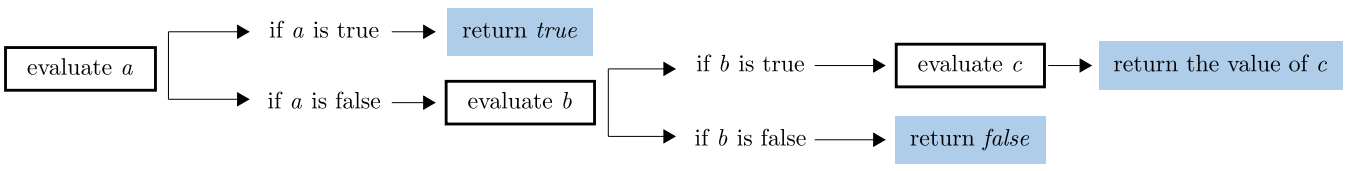
MULTIPLE CONDITIONS WITHOUT PARENTHESES (*OPTIONAL*)

To simplify the explanation, let's focus on cases with three conditions. These conditions will be represented through `Bool` variables `a`, `b`, and `c`, with each variable possibly representing expressions like `x > 0`.

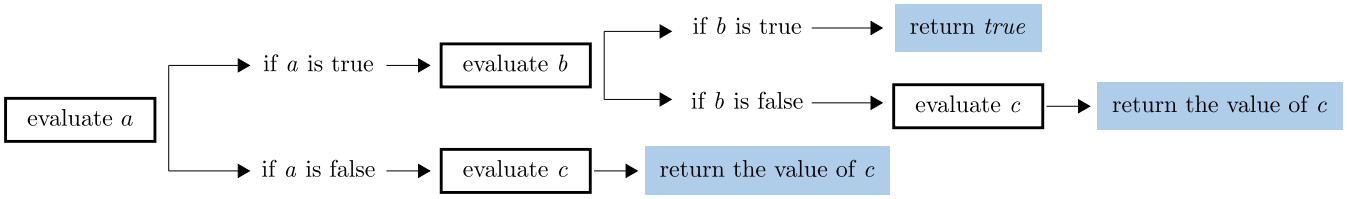
To understand how Julia groups three conditions without parentheses, there are two rules you need to know. First, `&&` has higher precedence than `||`. This means that `a && b || c` is equivalent to `(a && b) || c`, whereas `a || b && c` is equivalent to `a || (b && c)`. Second, `&&` and `||` are short-circuit operators. Thus, `a && b` immediately returns `false` if its first operand `a` is false, without evaluating the second operand `b`. Likewise, `a || b` returns `true` if the first operand `a` is true, without evaluating the second operand `b`.

The following diagrams describe the process for evaluating `a && b || c` and `a || b && c`, based on these two rules.

CASE 1: `a || b && c` is equivalent to `a || (b && c)`



CASE 2: `a && b || c` is equivalent to `(a && b) || c`



To illustrate the rules in practice, let's go through several examples that combine true/false values for `a`, `b`, and `c`. In these examples, we'll use the invalid expression `does-not-matter`. This is to emphasize that some conditions aren't necessarily evaluated thanks to the short-circuit behavior of `&&` and `||`.

```
julia> false || true && true
true
julia> false || true && false
false
julia> true || does-not-matter
true
```

```
julia> true && false || true
true
julia> true && false || false
false
julia> false && does-not-matter || true
true
```

FUNCTIONS TO CHECK CONDITIONS ON VECTORS: "ALL" AND "ANY"

Julia provides two built-in functions called `all` and `any` to evaluate multiple conditions in a collection. The function `all` returns `true` if *every condition* is true, whereas `any` returns `true` if *at least one condition* is true. The functions **require either directly specifying the conditions through a Boolean vector or defining the condition to check through a function**. Next, we cover each case separately.

VECTORS FOR REPRESENTING MULTIPLE CONDITIONS

In the following, we demonstrate the syntax of `all` and `any` when they take a Boolean vector as their argument.

```
a                = 1
b                = -1

# function indicating whether all elements satisfy the condition
are_all_positive = all([a > 0, b > 0])

# function indicating whether at least one element satisfies the condition
is_one_positive  = any([a > 0, b > 0])
```

```
julia> are_all_positive
false
julia> is_one_positive
true
```

The function `all` returns `true` only when all the conditions are satisfied, thus requiring that each vector's entry is positive. This doesn't hold in the example, since `b = -1`. Conversely, `any` returns `true` when at least one of the conditions holds, thus requiring at least one element in the vector to be positive. This is satisfied in the example, since `a = 1`.

As we indicated, `all` and `any` do not support passing multiple conditions as separate arguments. This entails that expressions like `all(a > 0, b > 0)` aren't allowed. Nevertheless, this restriction actually makes the functions more flexible, as they **enable the use of broadcasting operations for checking multiple conditions**. For example, the following code snippet implements the same operations as above, but through a vector `x`.

```
x                = [1, -1]

are_all_positive = all(x .> 0)
is_one_positive  = any(x .> 0)
```

```
julia> are_all_positive
false
julia> is_one_positive
true
```

FUNCTIONS FOR REPRESENTING MULTIPLE CONDITIONS

In addition to expressing conditions through vectors, `all` and `any` allow **passing a function to represent the condition to check**. The syntax for this is `all(<function>, <array>)` and `any(<function>, <array>)`, where `<function>` can be an anonymous function. The following examples demonstrate how to implement `all(x .> 0)` and `any(x .> 0)` using this approach.

```
x                = [1, -1]

are_all_positive = all(i -> i > 0, x)
is_one_positive  = any(i -> i > 0, x)
```

```
julia> are_all_positive
false
julia> is_one_positive
true
```

By passing a function as an argument, `all` and `any` can additionally be employed **to evaluate the same condition across multiple vectors**. This is achieved by broadcasting `all` and `any`.

```
x                = [1, -1]
y                = [1,  1]

are_all_positive = all.(i -> i > 0, [x,y])
is_one_positive  = any.(i -> i > 0, [x,y])
```

```
julia> are_all_positive # all elements in 'y' are positive, but not in 'x'
2-element BitVector:
 0
 1
julia> is_one_positive # at least one element of 'x' or 'y' is positive
2-element BitVector:
 1
 1
```

4c. Conditional Statements

Martin Alfaro
PhD in Economics

INTRODUCTION

Programs routinely perform different operations depending on their execution flow. To handle these possibilities, programs rely on conditional statements, which enable the execution of specific code blocks only when certain conditions are met.

Each code block of a conditional statement is referred to as a **branch**. Based on the number of branches, there are three types of conditional statements:

- **if-then statements**, which consist of a **single branch**. They run a specific operation only if a condition is met, with no operation performed otherwise.
- **if-else statements**, which consist of **two branches**. They run a specific operation if a condition is met, and another if the condition isn't satisfied.
- **if-else-if statements**, which consist of **three or more branches**. They comprise a series of conditions, with each branch executing a different code block.

Next, we cover each in depth. The presentation builds heavily on the logical operators introduced in the previous section. If you haven't read it, I highly recommend doing so before continuing.

IF-THEN STATEMENTS

If-then statements execute an operation only when a condition is met, doing nothing instead when the condition isn't satisfied. These statements can be constructed via:

- the `if` keyword,
- the logical operator `&&`,
- the logical operator `||`.

The approach via `if` keyword is self-explanatory. As for the logical operators, `&&` executes an operation if the condition is true, whereas `||` does it when the condition is *not* satisfied. In fact, `||` is equivalent to `&&` with its condition negated.

Below, we illustrate the syntax for each form. The examples rely on the `println` function, which displays the text passed as argument in the REPL.

```
x = 5

if x > 0
    println("x is positive")
end
```

```
"x is positive"
```

```
x = 5

(x > 0) && (println("x is positive"))
```

```
"x is positive"
```

```
x = 5

(x ≤ 0) || (println("x is positive"))
```

```
"x is positive"
```

Note that if-then statements imply that no action would've been taken if, for instance, we had used `x = -1` as a condition—it's only when `x > 0` that `println` is executed.

The `if` approach offers the most flexibility, making it ideal for complex conditional statements. However, it's somewhat verbose for simple conditional statements. For these cases, `&&` and `||` are preferred, as they help us keep the code streamlined.

A common application of `||` is in conjunction with the function `error` to handle errors. This construct immediately interrupts the script's execution when the condition isn't satisfied, displaying the provided message as the argument of `error`. For instance, consider a function `foo(x)` that requires non-negative values for `x`. To enforce this, you could include `x > 0 || error("x must be positive")` at the beginning of the function. If `foo(x)` is then called with a non-positive `x`, it'll immediately halt its execution and print the error message "x must be positive" in the REPL.

> **Remark**
>
> Note that `&&` and `||` behave like if-then statements when they combine a condition with an operation. This is different from using them exclusively with conditions, where all operands would be `Bool` values.

IF-ELSE STATEMENTS

If-else statements execute an operation when a condition is true and another operation when the condition is false. There are two forms to write these statements.

The first one is the most flexible and uses the `if` keyword in combination with `else`. The second method relies on the so-called **ternary operator**, which requires the keywords `?` and `:` via the syntax `<condition> ? <operation if true> : <operation if false>`. This is referred to as *the* ternary operator because it's the only operator in most programming languages that takes three arguments.

We illustrate the syntax of both approaches below.

```
x = 5

if x > 0
    println("x is positive")
else
    println("x is not positive")
end
```

```
"x is positive"
```

```
x = 5

x > 0 ? println("x is positive") : println("x is not positive")
```

```
"x is positive"
```

The function `ifelse` offers an alternative for constructing if-else expressions. This function takes three arguments: a condition, an expression to be evaluated if the condition is true, and another one if false. [1]

One advantage of using a function for an if-else statement is that it supports broadcasting. This is particularly helpful when creating vectors whose elements vary according to a condition, as demonstrated below.

```
x                    = [4, 2, -6]

are_elements_positive = ifelse.(x .> 0, true, false)
```

```
julia> are_elements_positive
3-element BitVector:
 1
 1
 0
```

```
x                   = [4, 2, -6]

x_absolute_value    = ifelse.(x .≥ 0, x, -x)
```

```
julia> x_absolute_value
3-element Vector{Int64}:
 4
 2
 6
```

> **Remark**
>
> Broadcasting `ifelse` **requires broadcasting both** `ifelse` **and the condition**. The first example, for instance, would throw an error if we execute `ifelse.(x>0, true, false)`. This is because `x > 0` would attempt to check if the *entire vector* is positive, which is an operation undefined in Julia.

IF-ELSE-IF STATEMENTS

So far, we've analyzed conditional statements that handle only two possibilities: one when the condition is met, and another if it isn't. This binary structure can be limiting when multiple alternatives need to be considered. Basically, it forces you to nest several `if` and `else` statements to manage additional conditions.

To simplify this process, we can use the `elseif` keyword to extend the `if` and `else` approach. This is illustrated below.

```
x = -10

if x > 0
    println("x is positive")
elseif x == 0
    println("x is zero")
end
```

```
x = -10

if x > 0
    println("x is positive")
elseif x == 0
    println("x is zero")
else
    println("x is negative")
end
```

```
"x is negative"
```

The first examples showcase the benefits provided by the approach. Specifically, `elseif` eliminates the need to explicitly specify actions for every possible scenario. Instead, it performs an action if `x` is positive, another action if `x` is zero, but it does nothing otherwise. In contrast, using `if` and `else` would require an exhaustive approach, where all possible contingents must be accounted for.

Likewise, the second example demonstrates that combinations of the `if`, `else`, and `elseif` keywords are possible.

FOOTNOTES

[1] The function `ifelse` does *not* behave as a **short-circuit operator**. This means that all the operations are computed, despite that only one of them will ultimately be returned as output.

4d. For-Loops

Martin Alfaro
PhD in Economics

INTRODUCTION

A key feature of programming is its ability to automate repetitive tasks, making **for-loops** play a crucial role in coding. They let you execute the same block of code repeatedly, treating each element in a list as a different input.

While for-loops are fundamental in every programming language, their importance is especially pronounced in Julia: unlike other languages (e.g., Matlab, Python, and R), which often discourage for-loops in performance-critical code, Julia relies on them to achieve high performance.

The role of for-loops in optimizing performance will be explored in Part II. Here, we'll primarily introduce the tool itself, focusing on its syntax, constructions, and common iteration techniques.

SYNTAX

For-loops delimit their scope via the keywords `for` and `end`. To illustrate their syntax, consider the function `println(a)`, which evaluates `a` and displays its output in the REPL. In case `a` is a string, `println(a)` simply displays the word stored in `a`. The following script repeatedly applies `println` to display each word contained in a collection.

FOR-LOOPS SYNTAX

```
for x in ["hello","beautiful","world"]
    println(x)
end
```

```
"hello"
"beautiful"
"world"
```

> **Remark**
>
> The keyword `in` can be replaced by `∈` or `=`. [1] Consequently, the following constructions are all equivalent.
>
> **IN**
>
> ```
> for x in ["hello","beautiful","world"]
>     println(x)
> end
> ```

∈

```
for x ∈ ["hello","beautiful","world"]
    println(x)
end
```

=

```
for x = ["hello","beautiful","world"]
    println(x)
end
```

Furthermore, we can employ any character or term to describe the iteration variable. For instance, we iterate below using `word`.

ALTERNATIVE NAME FOR ITERATION VARIABLE

```
for word in ["hello","beautiful","world"]
    println(word)
end
```

```
"hello"
"beautiful"
"world"
```

Based on this example, we can identify three components that characterize a for-loop:

- **A code block to be executed**: represented in the example by `println(x)`.
- **A list of elements**: represented in the example by `["hello","beautiful","world"]`. This specifies the elements over which we'll apply the code block. The list can contain elements with any data type (e.g., strings, numbers, and even functions). The only requirement is that the list must be an **iterable object**, defined as a collection whose elements can be accessed individually. An example of iterable object is vectors, as in the example. However, we'll also introduce others most commonly used, such as ranges.
- **An iteration variable**: represented in the example by `x`. This serves as a label that takes on the value of each element in the list, one at a time, during each iteration. The iteration variable is a local variable, with no significance outside the for-loop. Its sole purpose is to provide a convenient way to access and manipulate the elements of the list within the loop.

In the following sections, we'll explore different objects that can serve as lists. Furthermore, we'll show that these lists can comprise elements not immediately obvious. A typical example is functions, making it possible to apply different functions to the same object.

> **Always Wrap For-Loops in Functions**
>
> At this stage of the website, we're still introducing fundamental concepts. Thus, we're presenting subjects in their simplest form for

> learning purposes. In particular, this explains why for-loops will be written in the global scope.
>
> However, **you should always wrap for-loops in functions**. Executing for-loops outside a function severely degrades performance, and is additionally subject to different rules regarding variable scoping. [2]

ITERATING OVER INDICES

So far, we've considered a simple list like `["hello", "beautiful", "world"]` to demonstrate how for-loops work. In real applications, however, manually specifying each element in a list is impractical. Fortunately, when a list follows a predictable pattern (e.g., a sequence of numbers), we can simply describe the pattern that generates those elements.

Building on this insight, we'll next explore how to define ranges. They let users define a sequence of numbers, which is particularly useful to access elements of a collection through their indices.

RANGES

Ranges in Julia are defined via the syntax `<begin>:<steps>:<end>`, where `<begin>` represents the starting index and `<end>` the ending index. Likewise, `<steps>` sets the increment between values, defaulting to one when the term is omitted. We can also reverse the order of the sequence, by providing a negative value for `<steps>`. All this is demonstrated below.

RANGE WITH STEPS GIVEN

```
for i in 1:2:5
    println(i)
end
```

```
1
3
5
```

RANGE WITH REVERSE ORDER

```
for i in 3:-1:1
    println(i)
end
```

```
3
2
1
```

> **Remark**
>
> The application of ranges isn't limited to for-loops. They can also define vectors when used in combination with the `collect` function.

CREATING A VECTOR FROM A RANGE

```
x = collect(4:6)
```

```
julia> x
3-element Vector{Int64}:
 4
 5
 6
```

ITERATING OVER INDICES OF AN ARRAY

Ranges can be employed to access elements of a collection. When combined with a for-loop, it makes it possible to apply the same code block to each element of a vector.

Specifically, the expression `1:length(x)`, where `length(x)` returns the number of elements in `x`, allows iteration over all indices of a vector `x`. The same functionality can be achieved with the function `eachindex(x)`. In fact, this is the recommended approach for iterating over all elements, as it returns an iterator optimized for each iterable object.

1:LENGTH(X)

```
x = [4, 6, 8]

for i in 1:length(x)
    println(x[i])
end
```

```
4
6
8
```

EACHINDEX

```
x = [4, 6, 8]

for i in eachindex(x)
    println(x[i])
end
```

```
4
6
8
```

Remark

There are other approaches to iterating over all indices of a vector `x`. For instance, you can use `LinearIndices(x)`, or `firstindex(x):lastindex(x)` to specify a range from the first to the last index of `x`.

This multiplicity of methods exists to handle non-standard indices, such as those provided by the `OffsetArrays.jl` package. This package sets the first index of arrays to 0, a common convention in many programming languages. Nevertheless, unless you're developing a package for other users, you don't need to worry about which approach to implement. Indeed, they can all be used interchangeably, as shown below.

EACHINDEX

```
x = [4, 6, 8]

for i in eachindex(x)
    println(x[i])
end
```

```
4
6
8
```

1:LENGTH(X)

```
x = [4, 6, 8]

for i in 1:length(x)
    println(x[i])
end
```

```
4
6
8
```

LINEARINDICES

```
x = [4, 6, 8]

for i in LinearIndices(x)
    println(x[i])
end
```

```
4
6
8
```

FIRSTINDEX(X):LASTINDEX(X)

```
x = [4, 6, 8]

for i in firstindex(x):lastindex(x)
    println(x[i])
end
```

```
4
6
8
```

> Among the available alternatives, `eachindex` is preferable because it automatically selects the most efficient method for each type of collection. Additionally, the syntax is consistent across all indexing conventions.

RULES FOR VARIABLE SCOPE IN FOR-LOOPS

Similar to functions, for-loops create a new variable scope. In fact, the scoping rules for both are similar, with one key difference: **for-loops can modify global variables, whereas functions cannot**.

> **Warning!**
>
> The general scoping rules presented here apply universally, except in rare edge cases that result from poor coding practices. Since this scenario is uncommon, we only outline it next.
>
> Basically, the issue occurs when *i)* the for-loop is *not* wrapped in a function, *ii)* a local variable shares the same name as a global variable, and *iii)* the script is run non-interactively (i.e., using the function `include` and a script file). [3]
>
> Unless the three conditions hold simultaneously, you don't have to worry about this scenario. And even if this occurs, Julia will display a warning in the REPL indicating that there's a problem with your code.

To formalize the variable scope of for-loops, we'll refer to a variable `x`. The rules governing its scope are:

- the variable of iteration `x` is always local, regardless of whether there's a variable `x` defined outside the for-loop.
- if there's no variable named `x` outside the for-loop, `x` is a new local variable. Moreover, this variable won't be accessible outside the for-loop.
- if there's a variable named `x` outside the for-loop, `x` refers to this variable.

The following code snippets illustrate the first two rules, which exclusively refer to local variables. The second example is particularly noteworthy, as it highlights **a common mistake made by beginners**: running a for-loop that defines a local variable, and then trying to access it outside the for-loop.

ITERATION VARIABLE IS LOCAL

```
x = 2

for x in ["hello"]          # this 'x' is local, not related to 'x = 2'
    println(x)
end
```

```
"hello"
```

DEFINING LOCAL VARIABLE

```
#no `x` outside the for-loop

for word in ["hello"]
    x = word                # `x` is local to the for-loop, not available outside it
end
```

```
julia> x
ERROR: UndefVarError: x not defined
```

Likewise, the following example demonstrates the consequences of the last rule we mentioned. This refers to the consequences of variable scope for global variables.

REFERRING TO THE GLOBAL X

```
x = [2, 4, 6]

for i in eachindex(x)
    x[i] * 10        # it refers to the `x` outside of the for-loop
end
```

```
julia> x
3-element Vector{Int64}:
 20
 40
 60
```

REASSIGNING THE GLOBAL X

```
x = [2, 4, 6]

for word in ["hello"]
    x = word                        # it reassigns the `x` defined outside the for-loop
end
```

```
julia> x
"hello"
```

ARRAY COMPREHENSIONS

To seamlessly create arrays via for-loops, you can use **array comprehensions**. Their syntax is `[<expression> for... if...]`, where `<expression>` denotes either an operation or a function.

For illustration purposes, consider a vector `x`. Suppose that the goal is to create a vector `y` with elements equal to the square of the corresponding element in `x`. The following code snippets show two approaches to creating `y` via array comprehensions.

COMPREHENSION USING AN OPERATION

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


y      = [a^2 for a in x]              # or y = [x[i]^2 for i in eachindex(x)]
```

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

COMPREHENSION USING A FUNCTION

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

foo(a) = a^2
y      = [foo(a) for a in x]           # or y = [foo(x[i]) for i in eachindex(x)]
```

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

Array comprehensions also allow for creating vectors based on conditions. In such instances, the condition must be placed at the end of the expression.

COMPREHENSION WITH CONDITION

```
x = [i for i in 1:4 if i ≤ 3]
```

```
julia> x
3-element Vector{Int64}:
 1
 2
 3
```

> **Remark**
>
> Array comprehensions can also create matrices. Its syntax demands a comma to separate the description of each dimension.

COMPREHENSION FOR MATRICES

```
y = [i * j for i in 1:2, j in 1:2]
```

```
julia> y
2×2 Matrix{Int64}:
 1  2
 2  4
```

ITERATING OVER MULTIPLE OBJECTS

Thus far, we've considered for-loops that iterate over single values. We now extend their application to **simultaneous iterations over multiple values**. Specifically, we'll examine two scenarios: simultaneous iterations over two lists and over both the indices and values of a vector.

ITERATING OVER TWO LISTS

Depending on how elements should be combined, we can define two approaches to simultaneously iterating over two lists `x` and `y`. First, the function `Iterators.product(x,y)` makes it possible to iterate over all the possible combinations of elements. This function is part of the package `Iterators`, imported by default in each Julia session.

Alternatively, you can iterate over all the ordered pairs of `x` and `y`. This is implemented through the function `zip(x,y)`, which provides the pair of *i*-th elements from `x` and `y` in the *i*-th iteration.

MULTIPLE ITERATORS (ALL COMBINATIONS)

```
list1 = [1, 2]
list2 = [3, 4]

for (a,b) in Iterators.product(list1,list2) #it takes all possible combinations
    println([a,b])
end
```

```
[1,3]
[2,3]
[1,4]
[2,4]
```

MULTIPLE ITERATORS (PAIRS)

```
list1 = [1, 2]
list2 = [3, 4]

for (a,b) in zip(list1,list2)              #it takes pairs of elements with the same index
    println([a,b])
end
```

```
[1,3]
[2,4]
```

Using `zip`, we can also iterate over multiple values via array comprehensions.

MULTIPLE ITERATORS (ALL COMBINATIONS)

```
x = [i * j for i in 1:2 for j in 1:2]
```

```
julia> x
4-element Vector{Int64}:
 1
 2
 2
 4
```

MULTIPLE ITERATORS (PAIRS)

```
x = [i * j for (i,j) in zip(1:2, 1:2)]
```

```
julia> x
2-element Vector{Int64}:
 1
 4
```

SIMULTANEOUSLY ITERATING OVER INDICES AND VALUES

To iterate over each pair of index-value of a vector, we can employ the `enumerate` function.

FOR-LOOPS

```
x = ["hello", "world"]

for (index,value) in enumerate(x)
    println("$index $value")
end
```

```
"1 hello"
"2 world"
```

ARRAY COMPREHENSION

```
x = [10, 20]


y = [index * value for (index,value) in enumerate(x)]
```

```
julia> y
2-element Vector{Int64}:
 10
 40
```

ITERATING OVER FUNCTIONS

Functions in Julia are **first-class objects**, also referred to as **first-class citizens**. This means that functions can be manipulated just like any other data type, such as strings and numbers. In particular, this property makes it possible to store functions in a vector and apply them sequentially to an object. The following example illustrates this by computing descriptive statistics of a vector `x`.

ITERATION OVER FUNCTIONS

```
x                       = [10, 50, 100]
list_functions          = [maximum, minimum]

descriptive(vector,list) = [foo(vector) for foo in list]
```

```
julia> descriptive(x, list_functions)
4-element Vector{Real}:
 100
  10
```

FOOTNOTES

[1] Recall that `∈` can be written through tab completion using the command `\in`.

[2] In fact, older versions of Julia were restricting the use of for-loops in the global scope.

[3] There are two methods to execute a script. The first method is what we've been using so far, where you work interactively with Julia. This includes running commands in the REPL's prompt `julia>` and the execution of a script through a code editor. The second method consists of executing files that store scripts through the function `include`.

5a. Overview and Goals

Martin Alfaro
PhD in Economics

Thus far, we've laid the groundwork by introducing the fundamentals of Julia. We've covered in particular variables (single-element and collections) and core programming tools (functions, conditions, and for-loops). At this initial stage, our emphasis was primarily on helping you familiarize with the core approaches and their syntax. However, we didn't delve into any of these concepts, nor did we explore how the tools can be applied and combined.

Equipped now with a foundational knowledge of the concepts, we're ready to explore each in greater depth. **Chapter 5 in particular focuses on mutable collections**, using vectors as their primary example. As we begin to integrate these tools, it may take some time to get fully comfortable with their usage. In fact, you may occasionally need to revisit the sections on functions, conditions, and for-loops.

Despite that our focus is on vectors, many of the lessons we'll learn are applicable across all mutable collections. For instance, this is the case for concepts such as indexing and in-place operations. Other techniques presented extend even further, making their application universal across programming languages. Examples of this include the notion of mutability, along with the distinction between assignments and mutations.

5b. Mutable and Immutable Objects

Martin Alfaro
PhD in Economics

INTRODUCTION

Objects in programming can be broadly classified into two categories: mutable and immutable. **Mutable objects** allow their elements to be modified, appended, or removed at will. They're designed for flexibility, with **vectors** constituting a prime example.

In contrast, **immutable objects** are inherently unchangeable: they prevent additions, removals, or modifications of their elements. A common example of immutable object is **tuples**. Immutability effectively locks variables into a read-only state, safeguarding against unintended changes. Moreover, it can result in potential performance gains, as we'll show in Part II of this website.

This section will be relatively brief, focusing solely on the distinctions between mutable and immutable objects. Subsequent sections will expand on their uses and properties.

> **Remark**
>
> A popular package called `StaticArrays` provides an implementation of **immutable vectors**. We'll explore this package in the context of high performance, as it greatly speeds up computations that involve small vectors.

EXAMPLES OF MUTABILITY AND IMMUTABILITY

To illustrate the consequences of immutability, the following examples attempt to modify existing elements of a collection. The examples rely on vectors as an example of a mutable object and tuples for immutable ones. Additionally, we present the case of strings as another example of immutable object. Recall that strings are essentially sequences of characters, usually employed to represent text.

```
x = [3,4,5]
```

```
julia> x[1] = 0

julia> x
3-element Vector{Int64}:
 0
 4
 5
```

```
x = (3,4,5)
```

```
julia> x[1] = 0
ERROR: MethodError: no method matching setindex!(::Tuple{Int64, Int64, Int64}, ::Int64,
::Int64)
```

```
x = "hello"
```

```
julia> x[1]
'h': ASCII/Unicode U+0068 (category Ll: Letter, lowercase)
julia> x[1] = "a"
ERROR: MethodError: no method matching setindex!(::String, ::Int64, ::Int64)
```

The key characteristic of mutable objects is their ability to modify existing elements. Moreover, mutability also commonly allows for the dynamic addition and removal of elements. In a subsequent section, we'll present various methods for implementing this functionality. For now, we simply demonstrate the concept by using the functions `push!` and `pop!`, which respectively add and remove an element at the end of a collection.

```
x = [3,4]

push!(x, 5)       # add element 5 at the end
```

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

```
x = [3,4,5]

pop!(x)           # delete last element
```

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

```
x = (3,4,5)

pop!(x)           # error, just like push!(x, <some element>)
```

```
ERROR: MethodError: no method matching pop!(::Tuple{Int64, Int64, Int64})
```

5c. Assignments vs Mutations

Martin Alfaro
PhD in Economics

INTRODUCTION

The upcoming sections will be entirely devoted to **vector mutation**. However, to properly cover this subject, we first need to introduce some preliminary concepts, including:

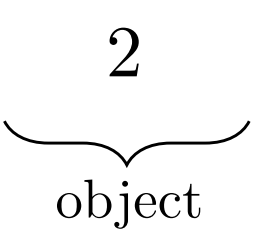
- the distinction between assignments and mutations
- methods for initializing arrays to eventually mutate them
- techniques for vector indexing to select elements

The current section in particular focuses on **distinguishing between assignments and mutations of variables**. The difference between both operations can easily go unnoticed by new users, as both operations use the operator `=`, despite being fundamentally different. Clearly delineating these operations is important not only for Julia, but also other programming languages.

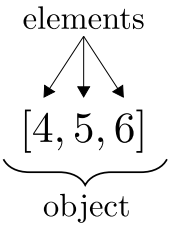
SOME BACKGROUND

Recall that **variables** serve as labels for **objects**, with objects in turn holding **values**. Moreover, objects can be classified according to the number of **elements** contained, ranging from single-element objects (e.g. integers and floating-point numbers) to collections (e.g. vectors).

NUMBER



VECTOR



The distinction between objects and their elements is crucial for the remainder of the section. This is because *assignments apply to objects, whereas mutations apply to elements*. More specifically, assignments rebind variables to new objects, while mutations simply modify existing elements of an existing object.

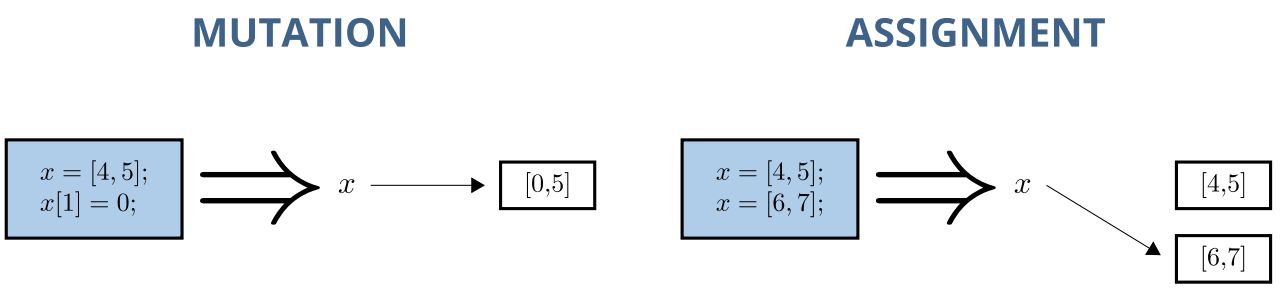
ASSIGNMENTS VS MUTATIONS

Assignments bind variables to objects, a process implemented via the `=` operator. For instance, `x = 3` and `x = [1,2,3]` are examples of assignments, where `3` and `[1,2,3]` represent the objects being assigned to `x`.

Mutations, by contrast, **modify the elements of an object, without creating a new one**. These operations also rely on the `=` operator, with `x[1] = 0` being an example of mutation.

Despite sharing the same operator `=`, assignments and mutations are conceptually distinct. This difference can be better appreciated by **visualizing objects as specific memory addresses**. This implies that assignments like `x = [4,5]` involve two steps: *i*) finding a memory location to store the object with value `[4,5]`, and *ii*) labeling the memory address as `x` for easy access. In contrast, a mutation modifies the contents of the object, but without changing its memory address.

To illustrate this, consider the example `x = [4,5]`, where the object `[4,5]` is stored in a particular memory location. If you then run `x = [6,7]`, `x` becomes associated with a new object containing `[6,7]`, thus constituting an *assignment*. However, if you execute `x[1] = 0` afterwards, the operation modifies the original object `[6,7]` to `[6,0]`. This operation constitutes a *mutation* because `x` continues to reference the same memory address, even though its content has changed.



Remark

You can mutate all the elements of `x`, without this necessarily entailing a new assignment. For example, this occurs when we modify the values of `x` by mutating `x[:]`.

```
x    = [4,5]

x[:] = [0,0]
```

```
julia> x
2-element Vector{Int64}:
 0
 0
```

> The distinction is particularly important since mutations tend to be faster than creating new objects. This will become relevant in Part II, where we explore strategies for speeding up operations.

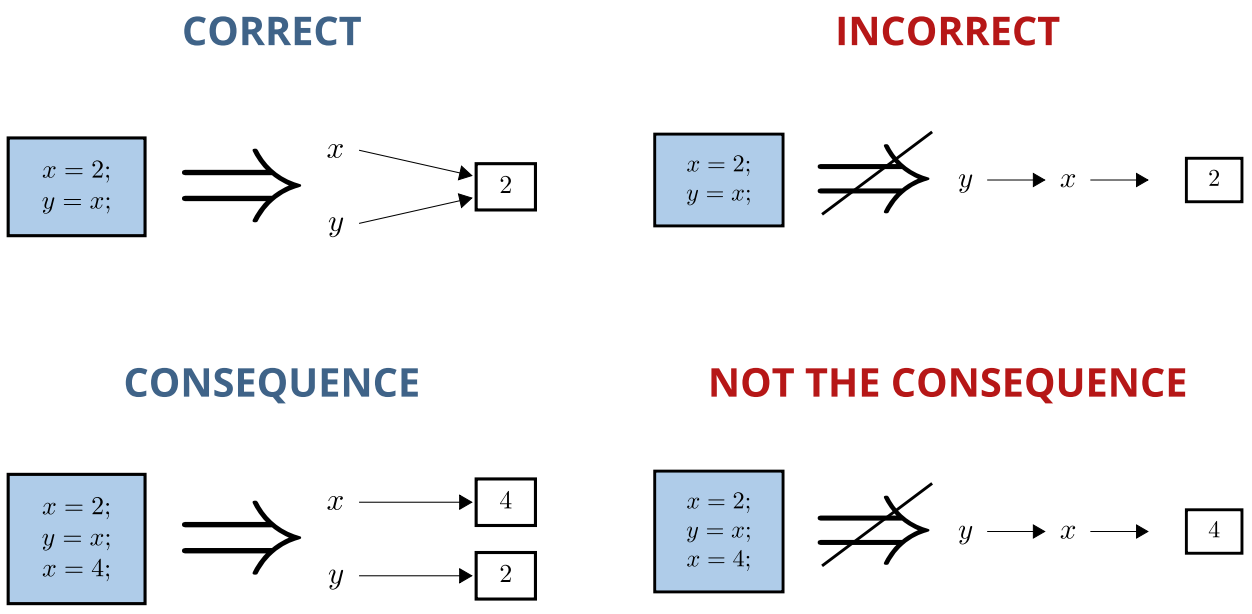
ALIAS VS COPY

So far, we've emphasized the critical distinction between assignments and mutations. Since both operations rely on the `=` operator, we must then inquire when `=` will entail one or the other operation. Next, we explore in particular cases like `y = x`, characterized by entire objects on each side of `=`. Other cases are left for the upcoming sections, after we introduce the concept of slices (i.e, subsets of elements from a vector).

In Julia, executing `y = x` makes `y` another name for the object referenced by `x`. This means that `x` and `y` become different labels for the same underlying object. Formally, it's said that `y` constitutes an **alias** of `x`.

Note that `y = x` shouldn't be understood as binding `y` to `x` itself. Rather, it means that `y` becomes another label for the object that `x` references. This subtle distinction carries a significant practical implication: reassigning `x` to a new object won't affect `y`'s reference.

To clarify this further, let's consider an example where we first execute `x = 2` and then `y = x`. At this point, both `x` and `y` reference the same object, which holds the value `2`. If we eventually execute `x = 4`, the variable `x` will start pointing to a new object that holds the value `4`. However, this won't affect the original object that `x` was referencing. As a result, `y` will still point to the original object with value `2`. This behavior is illustrated below.



```
x = 2   #'x' points to an object with value 2
y = x   #'y' points to the same object as 'x' (do not interpret it as 'y' pointing to 'x')

x = 4   #'x' now points to another object (but 'y' still points to the object holding 2)
```

```
julia> x
4
julia> y
2
```

Remark

Two variables could comprise identical elements and yet refer to different objects.

This can be demonstrated using the operators `==` and `===`, which assess two different types of equality. Specifically, `x == y` checks whether `x` and `y` have equal values, regardless of whether they refer to the same object. In contrast, `x === y` checks whether both `x` and `y` point to the same object, thus verifying if they share the same memory address. By applying these operators, the following example illustrates that objects with identical elements aren't necessarily referencing the same object.

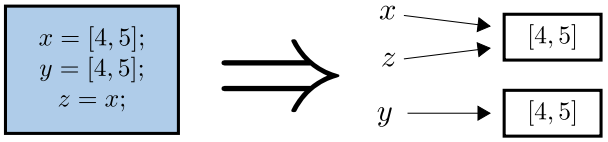
```
x = [4,5]

y = x
```

```
julia> x == y
true                    #`x` and `y` have identical elements
julia> x === y
true                        #`x` and `y` DO point to the same
object
```

```
x = [4,5]

y = [4,5]
```

```
julia> x == y
true                    #`x` and `y` have identical elements
julia> x === y
false                    #`x` and `y` DO NOT point to the same
object
```

GRAPHICAL REPRESENTATION



We've indicated that the operation `y = x` creates an alias of `x`, making `y` and `x` two different labels for the same object. This implies that **modifying the elements of either `x` or `y` will necessarily change the elements held by both**. The following diagram and code snippet illustrate this.

GRAPHICAL REPRESENTATION

```
x    = [4,5]
y    = x

x[1] = 0
```

```
julia> x
2-element Vector{Int64}:
 0
 5
julia> y
2-element Vector{Int64}:
 0
 5
```

If you instead want to treat `x` and `y` as separate objects, you must apply the function `copy`. This creates a *new object* with identical elements as the original. In this way, any modification to the new object won't affect the original one, allowing you to work with `x` and `y` independently.

```
x    = [4,5]
y    = copy(x)

x[1] = 0
```

```
julia> x
2-element Vector{Int64}:
 0
 5
julia> y
2-element Vector{Int64}:
 4
 5
```

5d. Initializing Vectors

Martin Alfaro
PhD in Economics

INTRODUCTION

We continue introducing preliminary concepts for mutations. The previous section distinguished between the use of `=` for assignments and mutations. Now, we'll deal with approaches to creating vectors.

Our presentation starts by outlining the process of initializing vectors, where memory is reserved without assigning initial values. We'll then discuss how to create vectors filled with predefined values such as zeros or ones. Finally, we show how to concatenate multiple vectors into new ones.

INITIALIZING VECTORS

Creating an array involves two steps: reserving memory for holding its content and assigning initial values to its elements. However, if you don't intend to populate the array with values right away, it's more efficient to only initialize the array. This means reserving memory space, but without setting any initial values.

Technically, initializing an array entails creating an array filled with `undef` values. These values represent arbitrary content in memory at the moment of allocation. Importantly, while `undef` displays concrete numbers when you output the array's content, they're meaningless and vary every time you initialize a new array.

There are two methods for creating vectors with `undef` values. The first one requires you to explicitly specify the type and length of the array, which is accomplished via `Vector{<elements' type>}(undef, <length>)`. The second approach is based on the function `similar(x)`, which creates a vector with the same type and dimensions as an existing vector `x`.

```
x_length = 3

x = Vector{Int64}(undef, x_length)  # `x` can hold `Int64` values, and is initialized with
3 undefined elements
```

```
julia> x
3-element Vector{Int64}:
 140724480121488
   2497084710592
   2497285012816
```

```
y = [3,4,5]

x = similar(y)                          # `x` has the same type as `y`, which is Vector{Int64}
(undef, 3)
```

```
julia> x
3-element Vector{Int64}:
 2497063587648
 2497063587664
 2497355825296
```

The example demonstrates that `undef` values don't follow any particular pattern. Moreover, these values vary in each execution, as they reflect any content held in RAM at the moment of allocation. In fact, a more descriptive way to call `undef` values would be **uninitialized values**.

CREATING VECTORS WITH GIVEN VALUES

In the following, we present several approaches to creating arrays filled with predefined values.

VECTORS WITH RANGE

If our goal is to generate a sequence of values, we can employ the function `collect(<range>)`. Recall that the syntax for defining ranges is `<start>: <steps>: <stop>`, where `<steps>` establishes the gap between elements.

```
some_range = 2:5

x          = collect(some_range)
```

```
julia> x
4-element Vector{Int64}:
 2
 3
 4
 5
```

Notice that when a range is created, `<steps>` implicitly dictates the number of elements to be generated. Alternatively, you could specify the number of elements to be stored, letting `<steps>` be implicitly determined. This is achieved by the function `range`, whose syntax is `range(<start>, <end>, <number of elements>)`. [1]

The following code snippet demonstrates the use of `range`, by generating five evenly spaced elements between 0 and 1.

```
x = range(0, 1, 5)
```

```
julia> x
0.0:0.25:1.0
```

```
x = range(start=0, stop=1, length=5)
```

```
julia> x
0.0:0.25:1.0
```

```
x = range(start=0, length=5, stop=1)     # any order for keyword arguments
```

```
julia> x
0.0:0.25:1.0
```

VECTORS WITH SPECIFIC VALUES

We can also create vectors of some given length filled with the same repeated value. In particular, the functions `zeros` and `ones` respectively create vectors with zeros and ones. By default, these functions define `Float64` elements, although you can specify a different type in the first argument of the function.

```
length_vector = 3

x             = zeros(length_vector)
```

```
julia> x
3-element Vector{Float64}:
 0.0
 0.0
 0.0
```

```
length_vector = 3

x             = zeros(Int, length_vector)
```

```
julia> x
3-element Vector{Int64}:
 0
 0
 0
```

```
length_vector = 3

x             = ones(length_vector)
```

```
julia> x
3-element Vector{Float64}:
 1.0
 1.0
 1.0
```

```
length_vector = 3

x             = ones(Int, length_vector)
```

```
julia> x
3-element Vector{Int64}:
 1
 1
 1
```

For creating Boolean vectors, Julia provides two convenient functions called `trues` and `falses`.

```
length_vector = 3

x             = trues(length_vector)
```

```
julia> x
3-element BitVector:
 1
 1
 1
```

```
length_vector = 3

x             = falses(length_vector)
```

```
julia> x
3-element BitVector:
 0
 0
 0
```

VECTORS FILLED WITH A REPEATED OBJECT

To define vectors comprising elements different from zeros or ones, Julia provides the `fill` function. Unlike the previous functions, this accepts any arbitrary scalar to be repeated.

```
length_vector    = 3
filling_object   = 1

x                = fill(filling_object, length_vector)
```

```
julia> x
3-element Vector{Int64}:
 1
 1
 1
```

```
length_vector    = 3
filling_object   = [1,2]

x                = fill(filling_object, length_vector)
```

```
julia> x
3-element Vector{Vector{Int64}}:
 [1, 2]
 [1, 2]
 [1, 2]
```

```
length_vector    = 3
filling_object   = [1]

x                = fill(filling_object, length_vector)
```

```
julia> x
3-element Vector{Vector{Int64}}:
 [1]
 [1]
 [1]
```

CONCATENATING VECTORS

Finally, we can create a vector `z` that merges all the elements of two vectors `x` and `y`. One simple approach for doing this is through `z = [x ; y]`. While this method is suitable for concatenating a few vectors, it becomes impractical with a large number of vectors, and directly infeasible when the number of vectors to concatenate is unknown.

For these scenarios, we can instead employ the function `vcat`, which merges all its arguments into one vector. By use of the splat operator `...`., the function can also be applied with a single argument that comprises a list of vectors. [2]

```
x = [3,4,5]
y = [6,7,8]


z = vcat(x,y)
```

```
julia> x
6-element Vector{Int64}:
 3
 4
 ⋮
 7
 8
```

```
x = [3,4,5]
y = [6,7,8]

A = [x, y]
z = vcat(A...)
```

```
julia> x
6-element Vector{Int64}:
 3
 4
 ⋮
 7
 8
```

Closely related to vector concatenation is the `repeat` function, which defines a vector containing the same object multiple times. Importantly, unlike `fill`, `repeat` **requires an array as its input**, throwing an error if a scalar is passed in particular.

```
nr_repetitions   = 3
vector_to_repeat = [1,2]

x                = repeat(vector_to_repeat, nr_repetitions)
```

```
julia> x
6-element Vector{Int64}:
 1
 2
 ⋮
 1
 2
```

```
nr_repetitions   = 3
vector_to_repeat = [1]

x                = repeat(vector_to_repeat, nr_repetitions)
```

```
julia> x
3-element Vector{Int64}:
 1
 1
 1
```

```
nr_repetitions   = 3
vector_to_repeat = 1

x                = repeat(vector_to_repeat, nr_repetitions)
```

```
ERROR: MethodError: no method matching repeat(::Int64, ::Int64)
```

ADDING, REMOVING, AND REPLACING ELEMENTS (*OPTIONAL*)

> **Warning!**
>
> This subsection requires knowledge of a few **concepts that we haven't discussed yet**. As such, it's marked as optional.
>
> One such concept is **in-place functions**, identified by the symbol `!` appended to the function's name. The symbol is simply a notation added by developers, hinting that the function modifies the value of at least one of its arguments. In-place functions will be explored thoroughly later).
>
> Another concept introduced is **pairs**, which will also be examined comprehensively in a future section. For the purposes of this subsection, it's sufficient to know that pairs are denoted by `a => b`, where `a` in our application refers to some value and `b` represents its corresponding replacement value.

Next, we show how to add, remove, and replace elements of a vector. To add a single element in particular, the methods are as follows.

```
x                = [3,4,5]
element_to_insert = 0


push!(x, element_to_insert)                # add 0 at the end - faster
```

```
julia> x
4-element Vector{Int64}:
 3
 4
 5
 0
```

```
x                = [3,4,5]
element_to_insert = 0


pushfirst!(x, element_to_insert)           # add 0 at the beginning - slower
```

```
julia> x
4-element Vector{Int64}:
 0
 3
 4
 5
```

```
x                = [3,4,5]
element_to_insert = 0
at_index          = 2

insert!(x, at_index, element_to_insert)     # add 0 at index 2
```

```
julia> x
4-element Vector{Int64}:
 3
 0
 4
 5
```

```
x                = [3,4,5]
vector_to_insert  = [6,7]


append!(x, vector_to_insert)               # add 6 and 7 at the end
```

```
julia> x
5-element Vector{Int64}:
 3
 4
 5
 6
 7
```

The function `push!` is particularly helpful to collect results in a vector. This is because, as it doesn't require any prior knowledge about the number of elements to be stored, we can dynamically grow the vector by adding more results. Notice that adding elements at the end via `push!` is faster than doing so at the beginning via `pushfirst!`.

Analogous functions exist to remove elements, as shown below.

```
x                = [5,6,7]


pop!(x)                           # delete last element
```

```
julia> x
2-element Vector{Int64}:
 5
 6
```

```
x                = [5,6,7]


popfirst!(x)                      # delete first element
```

```
julia> x
2-element Vector{Int64}:
 6
 7
```

```
x                 = [5,6,7]
index_of_removal  = 2

deleteat!(x, index_of_removal)      # delete element at index 2
```

```
julia> x
2-element Vector{Int64}:
 5
 7
```

```
x                 = [5,6,7]
indices_of_removal = [1,3]

deleteat!(x, indices_of_removal)    # delete elements at indices 1 and 3
```

```
julia> x
1-element Vector{Int64}:
 6
```

Emulating the behavior of `deleteat!`, we can also indicate which elements should be retained.

```
x              = [5,6,7]
index_to_keep  = 2

keepat!(x, index_to_keep)
```

```
julia> x
1-element Vector{Int64}:
 6
```

```
x              = [5,6,7]
indices_to_keep = [2,3]

keepat!(x, index_to_keep)
```

```
julia> x
1-element Vector{Int64}:
 6
```

Finally, we can replace specific values with new ones. This can be done by creating a new copy via `replace` or by updating the original vector with `replace!`.

Both functions make use of pairs `a => b`, where `a` is some value and `b` its corresponding replacement value. Note that these functions perform substitutions based on values, rather than indices.

```
x = [3,3,5]

replace!(x, 3 => 0)              # in-place (it updates x)
```

```
julia> x
3-element Vector{Int64}:
 0
 0
 5
```

```
x = [3,3,5]

replace!(x, 3 => 0, 5 => 1)      # in-place (it updates x)
```

```
julia> x
3-element Vector{Int64}:
 0
 0
 1
```

```
x = [3,3,5]

y = replace(x, 3 => 0)           # new copy
```

```
julia> y
3-element Vector{Int64}:
 0
 0
 5
```

```
x = [3,3,5]

y = replace(x, 3 => 0, 5 => 1)   # new copy
```

```
julia> y
3-element Vector{Int64}:
 0
 0
 1
```

FOOTNOTES

1. Note that `range` represents a convenient syntax for `<start> : 1 / <number of elements> : <end>`.

2. Recall that the operator `...` splits a collection into multiple arguments. This enables the use of a vector or tuple to denote multiple function arguments. For further details, see here under the subsection "Splatting".

5e. Slices: Copies vs Views

Martin Alfaro
PhD in Economics

INTRODUCTION

This section concludes the coverage of preliminary concepts for studying mutations by focusing on the behavior of a vector's **slice**. This is defined as a subset of a vector's elements, formally represented as `x[<indices>]`.

Importantly, **slices act differently depending on how they're included in a statement**, functioning as either:

- **copies** of the original vector, thus creating a new object at a new memory address.
- **views** of the original vector, where the original object and the slice share the same memory address.

In the following, we explain the distinction between copies and views in detail. Understanding it is crucial for mutating slices, as mutations can only occur when the slice references the original object. In contrast, if a slice acts as a copy, the parent object and the slice are unrelated, with changes to the slice having no impact on the original object.

SLICES AND THE ASSIGNMENT OPERATOR

Vector mutation involves modifying slices through the operator `=`. For this to be possible, a prerequisite is that the slice references the original object. Nonetheless, the behavior of slices in assignments varies depending on their position within the expression.

Specifically, **slices on the left-hand (LHS) side of `=` act as views**. In this case, slices reference the original elements, thus allowing for the mutation of its parent object. In contrast, **slices on the right-hand side (RHS) of `=` create a copy**. Since copies point to a new object, any modification to the slice won't affect the original object.

The following code snippet demonstrates both behaviors.

```
x    = [4,5]


x[1] = 0        # 'x[1]' is a view and mutates 'x'
```

```
julia> x
2-element Vector{Int64}:
 0
 5
```

```
x    = [4,5]
y    = x[1]     # 'y' is unrelated to 'x' because 'x[1]' is a copy

x[1] = 0        # it mutates 'x' but does NOT modify 'y'
```

```
julia> y
4
```

Aliasing vs Copy

Objects on the RHS of `=` are only treated as copies when it comes to **slices**, such as in statements `y = x[<indices>]`. Instead, if we insert the whole object `x` on the RHS of `=`, as in `y = x`, the operation creates an alias. In this case, `y` and `x` will reference the same object, and so any modification made to `y` will also be reflected in `x`.

```
x    = [4,5]
y    = x        # the whole object (a view)

x[1] = 0        # it DOES modify 'y'
```

```
julia> y
2-element Vector{Int64}:
 4
 5
```

```
x    = [4,5]
y    = x[:]     # a slice of the whole object (a copy)

x[1] = 0        # it does NOT modify 'y'
```

```
julia> y
2-element Vector{Int64}:
 0
 5
```

THE FUNCTION 'VIEW'

Identifying when slices act as copies or views is relevant for high performance, as views eliminate the overhead associated with memory allocations. Although this topic will be explored in Part II of the website, such considerations underscore the importance of distinguishing between copies and slices, beyond their use in assignments.

As a rule of thumb, **slices typically default to creating copies**. This is the case when, for instance, a slice is passed as a function argument or when used within an expression not involving an assignment. These scenarios are illustrated below.

```
x = [3,4,5]

#the following slices are all copies
log.(x[1:2])

x[1:2] .+ 2

[sum(x[:]) * a for a in 1:3]

(sum(x[1:2]) > 0) && true
```

In all these cases, transforming slices into views requires an explicit indication. To achieve this, you need to employ the function `view`. Its syntax is `view(x, <indices>)`, where `<indices>` represent the subset of indices defining the slice. To demonstrate its usage, we revisit and compare the previous code snippet.

```
x = [3,4,5]

#we make explicit that we want views
log.(view(x,1:2))

view(x,1:2) .+ 2

[sum(view(x,:)) * a for a in 1:3]

(sum(view(x,:)) > 0) && true
```

```
x = [3,4,5]

#the following slices are all copies
log.(x[1:2])

x[1:2] .+ 2

[sum(x[:]) * a for a in 1:3]

(sum(x[1:2]) > 0) && true
```

These examples reveal the potential verbosity involved when `view` isn't used sparingly. To address this issue, Julia provides the macros `@view` and `@views`.

The `@view` macro is equivalent to `view`, allowing you to write `@view x[1:2]` instead of `view(x, 1:2)`. However, its advantages are somewhat limited: it saves only a few characters, and additionally necessarily requires parentheses when multiple slices are used (e.g., `@view(x[1:2]) .+ @view(x[2:3])`). In contrast, the `@views` macro significantly streamlines notation, by automatically converting *every* slice within an expression into a view.

```
x = [4,5,6]

# the following are all equivalent
       y = view(x, 1:2)  .+ view(x, 2:3)
       y = @view(x[1:2]) .+ @view(x[2:3])
@views y = x[1:2] .+ x[2:3]
```

One of the most notable applications of `@views` is in functions. By placing `@views` at the beginning of a function, you automatically convert every slice within the function body and its arguments into a view.

```
@views function foo(x)
  y = x[1:2] .+ x[2:3]
  z = sum(x[:]) .+ sum(y)

  return z
end
```

```
function foo(x)
  y = @view(x[1:2]) .+ @view(x[2:3])
  z = sum(@view x[:]) .+ sum(y)

  return z
end
```

5f. Array Indexing

Martin Alfaro
PhD in Economics

INTRODUCTION

In order to mutate vectors, you first need to identify the elements you wish to modify. This process is known as **vector indexing**. We've already covered several basic methods for indexing, including vectors and ranges (e.g., `x[[1,2,3]]` or `x[1:3]`). While these approaches are effective for simple selections, they fall short for more complex scenarios, precluding for example selections based on conditions.

This section expands our toolkit by introducing some additional forms of indexing. The techniques presented primarily build on broadcasting Boolean operations.

LOGICAL INDEXING

Logical indexing (also known as *Boolean indexing* or *masking*) allows you to select elements based on conditions. Considering a vector `x`, this is achieved using a Boolean vector `y` of the same length as `x`, which acts as a filter: `x[y]` retains elements where `y` is `true` and excludes those where `y` is `false`.

LOGICAL INDEXING

```
x = [1,2,3]
y = [true, false, true]
```

```
julia> x[y]
2-element Vector{Int64}:
 1
 3
```

OPERATORS AND FUNCTIONS FOR LOGICAL INDEXING

Logical indexing becomes a powerful tool when we leverage broadcasting operations, allowing you to easily specify conditions via Boolean vectors. For instance, to select all the elements of `x` lower than 10, you can broadcast a comparison operator or a custom function.

INDEXING VIA BROADCASTING OPERATOR

```
x           = [1, 2, 3, 100, 200]

y           = x[x .< 10]
```

```
julia> y
3-element Vector{Int64}:
 1
 2
 3
```

INDEXING VIA BROADCASTING FUNCTION

```
x           = [1, 2, 3, 100, 200]

condition(a) = (a < 10)             #function to eventually broadcast
y           = x[condition.(x)]
```

```
julia> y
3-element Vector{Int64}:
 1
 2
 3
```

When dealing with multiple conditions, the conditions must be combined using the logical operators `&&` and `||`. [1] The following example illustrates the syntax for doing this. Note that *all* operators must be broadcasted, since logical operators only work with scalar values.

INDEXING VIA BROADCASTING OPERATOR

```
x           = [3, 6, 8, 100]

# numbers greater than 5, lower than 10, but not including 8
y           = x[(x .> 5) .&& (x .< 10) .&& (x .≠ 8)]
```

```
julia> y
1-element Vector{Int64}:
 6
```

INDEXING VIA @.

```
x           = [3, 6, 8, 100]

# numbers greater than 5, lower than 10, but not including 8
y           = x[@. (x > 5) && (x < 10) && (x ≠ 8)]
```

```
julia> y
1-element Vector{Int64}:
 6
```

INDEXING VIA BROADCASTING FUNCTION

```
x           = [3, 6, 7, 8, 100]

# numbers greater than 5, lower than 10, but not including 8
condition(a) = (a > 5) && (a < 10) && (a ≠ 8)          #function to eventually broadcast
y           = x[condition.(x)]
```

```
julia> y
1-element Vector{Int64}:
 6
```

The example reveals that directly broadcasting *operators* may result in verbose code, due to the repeated use of dots in the expression. In contrast, approaches based on functions or the macro `@.` keep the syntax simple, reducing boilerplate code.

LOGICAL INDEXING VIA `in` AND `∈`

> **Remark**
>
> The symbols `∈` and `∉` used in this section can be inserted via tab completion:
>
> - `∈` by `\in`
> - `∉` by `\notin`

Another approach to selecting elements through logical indexing involves `in` and `∈`. Each of these symbols is available as a function and an operator, and they check whether a *scalar* `a` belongs to a given collection `list`. For simplicity, next we'll refer to `in` as a function and `∈` as an operator.

The function `in(a, list)` evaluates whether the scalar `a` matches any element in the vector `list`, yielding the same result as `a ∈ list`. For example, both `in(2, [1, 2, 3])` and `2 ∈ [1, 2, 3]` return `true`, as `2` is an element of `[1,2,3]`.

By replacing the scalar `a` with a collection `x`, `in` and `∈` can define Boolean vectors via broadcasting. Recall, though, that broadcasting defaults to iterating over pairs of elements. This means that executing `in.(x, list)` or `x .∈ list` will result in a simultaneous iteration over each pair of `x` and `list`. However, this isn't the desired operation. Rather, our goal is to check whether each element of `x` belongs to `list`, which requires treating `list` as a single object. This can be accomplished in several ways, as it was shown here, including wrapping `list` in `Ref`.

As an illustration, below we create a vector `y` that contains the minimum and maximum of the vector `x`.

FUNCTION 'IN' AND '∈'

```
x           = [-100, 2, 4, 100]
list = [minimum(x), maximum(x)]

# logical indexing (both versions are equivalent)
bool_indices = in.(x, Ref(list))    #`Ref(list)` can be replaced by `(list,)`
bool_indices = (∈).(x,Ref(list))

y           = x[bool_indices]
```

```
julia> bool_indices
4-element BitVector:
 1
 0
 0
 1
julia> y
2-element Vector{Int64}:
 -100
  100
```

OPERATOR 'IN' AND '∈'

```
x           = [-100, 2, 4, 100]
list = [minimum(x), maximum(x)]

# logical indexing
bool_indices = x .∈ Ref(list)       #only option, not possible to broadcast `in`


y           = x[bool_indices]
```

```
julia> bool_indices
4-element BitVector:
 1
 0
 0
 1
julia> y
2-element Vector{Int64}:
 -100
  100
```

> **Remark**
>
> The `in` function has an alternative curried version, allowing the user to directly broadcast `in` while treating `list` as a single element. The syntax for doing this is `in(list).(x)`, as shown in the example below.

CURRIED 'IN'

```
x           = [2, 4, 100]
list = [minimum(x), maximum(x)]

#logical indexing
bool_indices = x[in(list).(x)]   #no need to use `Ref(list)`
y            = x[bool_indices]
```

```
julia> bool_indices
4-element BitVector:
 1
 0
 0
 1
julia> y
2-element Vector{Int64}:
 -100
  100
```

Remark

The functions and operators `in` and `∈` allow for negated versions `!in` and `∉` (equivalent to `!∈`), which select elements *not* belonging to a set.

Below, we apply them to retain the elements of `x` that are not its minimum or its maximum.

FUNCTION '!IN' AND '∉'

```
x           = [-100, 2, 4, 100]
list = [minimum(x), maximum(x)]

#identical vectors for logical indexing
bool_indices = (!in).(x, Ref(list))
bool_indices  =  (∉).(x,  Ref(list))                #or  `(!∈).(x,
Ref(list))`
```

```
julia> bool_indices
4-element BitVector:
 0
 1
 1
 0
julia> x[bool_indices]
2-element Vector{Int64}:
 2
 4
```

OPERATORS '!IN' AND '∉'

```
x           = [-100, 2, 4, 100]
list = [minimum(x), maximum(x)]

#vector for logical indexing
bool_indices = x .∉ Ref(list)
```

```
julia> bool_indices
4-element BitVector:
 0
 1
 1
 0
julia> x[bool_indices]
2-element Vector{Int64}:
 2
 4
```

THE FUNCTIONS 'FINDALL' AND 'FILTER'

We close this section by presenting two additional methods for element selection. They're provided by the functions `filter` and `findall`.

The function `filter` returns the *elements* of a vector `x` satisfying a given condition. Despite what the name may suggest, `filter` retains elements rather than discard them. The condition is specified by a function that returns a Boolean scalar.

'FILTER'

```
x = [5, 6, 7, 8, 9]

y = filter(a -> a < 7, x)
```

```
julia> y
2-element Vector{Int64}:
 5
 6
```

The function `findall` does the same as `filter`, but returns the *indices* of `x`. With `findall`, the condition can be stated in two ways: either via a Boolean scalar function or a Boolean vector.

'FINDALL' - VIA FUNCTION

```
x = [5, 6, 7, 8, 9]

y = findall(a -> a < 7, x)
z = x[findall(a -> a < 7, x)]
```

```
julia> y
2-element Vector{Int64}:
 1
 2
julia> z
2-element Vector{Int64}:
 5
 6
```

'FINDALL' - VIA BOOLEAN VECTOR

```
x = [5, 6, 7, 8, 9]

y = findall(x .< 7)
z = x[findall(x .< 7)]
```

```
julia> y
2-element Vector{Int64}:
 1
 2
julia> z
2-element Vector{Int64}:
 5
 6
```

FOOTNOTES

[1] The logical operators `&&` and `||` were introduced in the section about conditional statements.

5g. In-Place Operations

Martin Alfaro
PhD in Economics

INTRODUCTION

This section focuses on **in-place operations**, a term that encompasses any method that mutates collections. These operations are characterized by the reuse of existing objects, rather than generating new ones, giving rise to the expression " *modifying values in place*."

Understanding whether an operation mutates an object or generates a new one is crucial, as the outcomes may differ depending on the application. Furthermore, even if results were identical, in-place modifications commonly entail performance benefits relative to the creation of new objects. This aspect will be explored in Part II of the website, when we discuss high performance.

> **Remark**
>
> Before proceeding, I recommend reviewing the definitions of slices introduced in the previous section. Recall that, given a vector `x`, a **slice** refers to a subset of `x`'s elements selected via `x[<indices>]`.
>
> Moreover, a slice can act as a **copy**, in which case we're creating a new object with its own memory address. Alternatively, the slice can behave as a **view**, thereby referencing the original memory address of `x`. The distinction is crucial, as it'll determine whether modifying the slice will affect the original data or not.

MUTATIONS VIA COLLECTIONS

A simple way to mutate a vector is to replace an entire slice with another collection. This is achieved using statements of the form `x[<indices>] = <expression>`, where `<expression>` must match the length of `x[<indices>]`. The approach effectively mutates `x` because a slice on the left-hand side of `=` behaves as a view, thus referencing the original object. Below, we provide a few examples of this approach.

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

x[2:end]  = [20, 30]
```

```
julia> x
3-element Vector{Int64}:
  1
 20
 30
```

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

x[x .≥ 2] = [2, 3] .* 10
```

```
julia> x
3-element Vector{Int64}:
  1
 20
 30
```

A common application of this method involves defining `<expression>` through elements from either the original vector or the slice being modified. This allows for self-referencing updates of the variable.

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

x[2:end]  = [x[i] * 10 for i in 2:length(x)]
```

```
julia> x
3-element Vector{Int64}:
  1
 20
 30
```

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

x[x .≥ 2] = x[x .≥ 2] .* 10
```

```
julia> x
3-element Vector{Int64}:
  1
 20
 30
```

Importantly for the mutations via for-loops, a scalar can be used on the right-hand side of `=` for single-element slices.

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

x[3]      = 30
```

```
julia> x
3-element Vector{Int64}:
  1
  2
 30
```

> **Warning!** - Vectors can only be mutated by objects of the same type
>
> When a vector is defined, the type of elements that the vector can hold is implicitly defined. Consequently, attempting to replace elements with a different type will result in an error. For instance, the following examples only admit mutations with values of type `Int64`.

```
x         = [1, 2, 3]    # Vector{Int64}

x[2:3]    = [3.5, 4]     # 3.5 is Float64
```

```
ERROR: InexactError: Int64(3.5)
```

```
x         = [1, 2, 3]    # Vector{Int64}

x[2:3]      = [3.0, 4]        # 3.0 is Float64 but accepts
conversion
```

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

MUTATIONS VIA FOR-LOOPS

Replacing a single-element slice with a scalar value enables mutations through for-loops. This is implemented by substituting the value of a single element during each iteration.

To illustrate the procedure, let's consider a typical application of this approach: populating vectors with values. The process involves initializing a vector, and then iterating over its elements to assign desired values.

```
x    = Vector{Int64}(undef, 3)  # `x` is initialized with 3 undefined elements

x[1] = 0
x[2] = 0
x[3] = 0
```

```
julia> x
3-element Vector{Int64}:
 0
 0
 0
```

```
x    = Vector{Int64}(undef, 3)  # `x` is initialized with 3 undefined elements

for i in eachindex(x)
    x[i] = 0
end
```

```
julia> x
3-element Vector{Int64}:
 0
 0
 0
```

The approach relies on `x[i]` on the left-hand side of `=` acting as a view. Alternatively, we could leverage the function `view` to create a variable that contains all the elements to be modified. This allows us to work with for-loops that mutate entire objects, rather than a subset of the original object.

The following example demonstrates this by mutating a vector of zeros. Note that the function `zeros` defaults to zeros with type `Float64`, explaining why `1` is automatically converted to `1.0`.

```
x     = zeros(3)


for i in 2:3
    x[i] = 1
end
```

```
julia> x
3-element Vector{Float64}:
 0.0
 1.0
 1.0
```

```
x     = zeros(3)
slice = view(x, 2:3)

for i in eachindex(slice)
    slice[i] = 1
end
```

```
julia> x
3-element Vector{Float64}:
 0.0
 1.0
 1.0
```

> **Warning!** - For-Loops Should be Wrapped in Functions
>
> Recall that for-loops should always be wrapped in functions, as failing to do so severely affects performance. In the next section, where we'll cover mutating functions, mutations via for-loops will be revisited.

MUTATIONS VIA .=

In terms of syntax, broadcasting serves as a streamlined alternative to for-loops. This principle even extends to mutations, which can be implemented by broadcasting the assignment operator `=`. The syntax for this is `x[<indices>] .= <expression>`, where `<expression>` can be either a *vector* or a *scalar*.

In the specific case of `x[<indices>]` on the left-hand side and a vector for `<expression>`, the `.=` operator produces the same outcome as using `=` with a corresponding collection. In fact, using `=` rather than `.=` in these cases tends to be more performant.

```
x       = [3, 4, 5]

x[1:2]  = x[1:2] .* 10
```

```
julia> x
3-element Vector{Int64}:
 30
 40
  5
```

```
x       = [3, 4, 5]

x[1:2] .= x[1:2] .* 10    # identical output (less performant)
```

```
julia> x
3-element Vector{Int64}:
 30
 40
  5
```

Considering this, the primary use cases of `.=` for mutating `x` involve expressions like:

- `x[<indices>] .= <scalar>`, and
- `y .= <expression>` where `y` is either a view of `x` or `x` itself.

Next, we analyze each case separately.

SCALARS ON THE RIGHT-HAND SIDE OF =

Applying `=` to replace multiple elements with the *same* scalar value requires a collection matching the number of elements being substituted. However, with the `.=` operator, you can streamline the process by simply writing `x[<indices>] .= <scalar>`.

For instance, the following code snippet uses this approach to replace every negative value in `x` with zero.

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

x[x .< 0] .= 0
```

```
julia> x
3-element Vector{Int64}:
 0
 0
 1
```

VARIABLES ON THE LEFT-HAND SIDE OF =

Since both mutations and assignments rely on `=`, it's essential to **distinguish between in-place operations and reassignments**. In particular, the latter doesn't modify the original object, but actually creates a new one.

We've already shown that placing slices on the left-hand side of `=` results in mutations. Now, let's consider cases where an entire object like `x` appears on the left-hand side. In these cases, we need to be careful, as only `.=` will result in a mutation, whereas `=` will perform a reassignment.

For example, suppose our goal is to modify *all* the elements of a vector `x`. All the following approaches determine that `x` holds the same values, but only the last two achieve this by mutating `x`.

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

x    = x .* 10
```

```
julia> x
3-element Vector{Int64}:
 10
 20
 30
```

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

x   .= x .* 10
```

```
julia> x
3-element Vector{Int64}:
 10
 20
 30
```

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

x[:] = x .* 10
```

```
julia> x
3-element Vector{Int64}:
 10
 20
 30
```

Notice that the difference between approaches could go unnoticed, as they all yield the same outcome. However, we'll see in Part II of the website that these approaches can entail big differences in performance. In particular, reusing the original memory address of `x` tends to be more performant than creating a new memory address for `x`.

This difference will also manifest when using the macro `@.` for a seamless broadcasting. Depending on where `@.` is placed relative to `=`, we could end up with an assignment or a mutation.

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

x   .= x .* 10
```

```
julia> x
3-element Vector{Int64}:
 10
 20
 30
```

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

@. x = x  * 10
```

```
julia> x
3-element Vector{Int64}:
 10
 20
 30
```

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

x    = @. x * 10
```

```
julia> x
3-element Vector{Int64}:
 10
 20
 30
```

VIEW ALIASES ON THE LEFT-HAND SIDE OF =

The previous case exemplified a scenario where the ultimate outcome is the same, regardless of whether we employ `=` or `.=`. Instead, the results will differ when view aliases are on the left-hand side. View aliases allow us to work with `slice = view(x, <indices>)`, rather than `x[<indices>]`. Defining view aliases is convenient when we need to perform various operations over the same slice. This avoids repeatedly referencing the slice via `x[<indices>]`, which would be inefficient, error-prone, and tedious.

In these cases, it's only when we use `.=` that we'll perform a mutation.

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

slice  = view(x, x .≥ 2)
slice .= slice .* 10            # same as 'x[x .≥ 2] = x[x .≥ 2] .* 10'
```

```
julia> x
3-element Vector{Int64}:
  1
 20
 30
```

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

slice  = view(x, x .≥ 2)
slice  = slice .* 10            # this does NOT modify `x`
```

```
julia> x
3-element Vector{Int64}:
 1
 2
 3
```

There are a few additional incorrect uses that can emerge with view aliases. To demonstrate them, let's consider replacing all negative values in `x` with zero. Below, only the first approach achieves the desired outcome.

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

slice  = view(x, x .< 0)
slice .= 0
```

```
julia> x
3-element Vector{Int64}:
 0
 0
 1
```

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

slice  = x[x .< 0]          # 'slice' is a copy
slice .= 0                  # this does NOT modify `x`
```

```
julia> x
3-element Vector{Int64}:
 -2
 -1
  1
```

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

slice  = view(x, x .< 0)
slice  = 0                  # this creates a new object, it does NOT modify `x`
```

```
julia> x
3-element Vector{Int64}:
 -2
 -1
  1
```

5h. In-Place Functions

Martin Alfaro
PhD in Economics

INTRODUCTION

This section continues exploring **approaches to mutating vectors**. The emphasis is in particular on **in-place functions**, defined as functions that mutate at least one of their arguments.

Many built-in functions in Julia have a corresponding in-place counterpart. These versions can be easily identified by the symbol `!` appended to their names. In-place functions enable users to store the output in one of the function's arguments, thereby avoiding the creation of a new object. They can also be used to update the values of a variable directly. For example, given a vector `x`, `sort(x)` returns a new vector with ordered elements, but without altering the original `x`. In contrast, the in-place version `sort!(x)` directly stores the result within `x` itself.

The benefits of in-place functions will become evident in Part II, when discussing high-performance computing. Essentially, by reusing existing objects, in-place functions eliminate the overhead associated with creating new objects.

IN-PLACE FUNCTIONS

In-place functions, also known as **mutating functions**, are characterized by their ability to modify at least one of their arguments. For example, given a vector `x`, the following function `foo(x)` constitutes an example of in-place function, as it modifies the content of `x`.

```
y = [0,0]

function foo(x)
    x[1] = 1
end
```

```
julia> y
2-element Vector{Int64}:
 0
 0
julia> foo(y) #it mutates 'y'
julia> y
2-element Vector{Int64}:
 1
 0
```

> **Remark**

> While functions are capable of mutating values, they **can't reassign variables** outside their scope. Any attempt to redefine a variable within a function will be interpreted as the creation of a new local variable. [1]
>
> The following code illustrates this behavior by redefining a function argument and a global variable. The output reflects that `foo` in each example treats the redefined `x` as a new local variable, thus only existing within `foo`'s scope.

```
x = 2

function foo(x)
    x = 3
end
```

```
julia> x
2
julia> foo(x)
julia> x #functions can't redefine variables globally, only mutate them
2
```

```
x = [1,2]

function foo()
    x = [0,0]
end
```

```
julia> x
2-element Vector{Int64}:
 1
 2
julia> foo()
julia> x #functions can't redefine variables globally, only mutate them
2-element Vector{Int64}:
 1
 2
```

BUILT-IN IN-PLACE FUNCTIONS

In Julia, many built-in functions that take vectors as arguments are available in two forms: a "standard" version and an in-place version. To distinguish between them, Julia's developers follow a convention that **any function ending with `!` corresponds to an in-place function**.

> **Remark**

> **Appending `!` to a function has no impact on the function's behavior at all**. It's simply a convention adopted by Julia's developers to emphasize the mutable nature of the operation. Its purpose is to alert users about the potential side effects of the function, thus preventing unintended modifications of objects.

To illustrate these forms, let's start considering single-argument functions. In particular, we'll focus on `sort`. This arranges the elements of a vector in ascending order, with the option `rev=true` implementing a descending order. In its standard form, `sort(x)` creates a new vector containing the sorted elements, leaving the original vector `x` unchanged. In contrast, the in-place version `sort!(x)` updates the original vector `x` directly, overwriting its values with the sorted result.

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

y = sort(x)
```

```
julia> x
3-element Vector{Int64}:
 2
 1
 3
julia> y
3-element Vector{Int64}:
 1
 2
 3
```

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

sort!(x)
```

```
julia> x
3-element Vector{Int64}:
 1
 2
 3
```

Regarding multiple-argument functions, it's common to include an argument whose sole purpose is to store outputs. For instance, given a function `foo` and a vector `x`, the built-in function `map(foo, x)` has an in-place version `map!(foo, <output vector>, x)`.

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


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

```
julia> x
3-element Vector{Int64}:
 1
 2
 3
julia> output
3-element Vector{Int64}:
 1
 4
 9
```

```
x      = [1, 2, 3]
output = similar(x)             # we initialize `output`

map!(a -> a^2, output, x)       # we update `output`
```

```
julia> x
3-element Vector{Int64}:
 1
 2
 3
julia> output
3-element Vector{Int64}:
 1
 4
 9
```

MUTATIONS VIA FOR-LOOPS

Recall that for-loops in Julia should always be wrapped in functions. This not only prevents issues with variable scope, but is also key for performance, as we'll discuss in Part II.

In this context, the ability of functions to mutate their arguments is crucial. It determines that we can initialize vectors with `undef` values, pass them to a function, and fill them through a function via for-loops. The examples below illustrate this application.

```
x = [3,4,5]

function foo!(x)
    for i in 1:2
        x[i] = 0
    end
end
```

```
julia> foo!(x)

julia> x
3-element Vector{Int64}:
 0
 0
 5
```

```
x = Vector{Float64}(undef, 3)          # initialize a vector with 3 elements

function foo!(x)
    for i in eachindex(x)
        x[i] = 0
    end
end
```

```
julia> foo!(x)

julia> x
3-element Vector{Int64}:
 0
 0
 0
```

FOOTNOTES

[1] Strictly speaking, it's possible to reassign a variable by using the `global` keyword. However, its use is typically discouraged, explaining why we won't cover it.

6a. Overview and Goals

Martin Alfaro
PhD in Economics

The previous chapter equipped us with techniques for indexing and modifying vectors, expanding our toolkit for working with data collections. This section builds on this knowledge to achieve several goals.

Firstly, we'll **introduce additional types for collections**, including dictionaries and named tuples. Building on our grasp of tuples and vectors, we're now well-positioned to appreciate the unique features of these alternatives and understand when they're more suitable.

Secondly, we'll **expand on tools for streamlining code**, which will become indispensable in your daily use of Julia. These tools will make your coding experience smoother, by reducing boilerplate code and improving syntax readability. One notable example is the use of pipes.

Thirdly, we'll introduce several standard functions for manipulating vectors, enabling you to perform operations such as removing duplicates and sorting elements.

To conclude the chapter, we'll **put into practice all the tools we've covered**. This will be done through a hypothetical scenario involving a YouTuber's earnings. This hands-on approach will demonstrate how to apply the tools learned, helping you bridge the gap between theory and practice. Furthermore, it'll lay the foundation for more advanced data analysis tools: by mastering the application of fundamentals such as vector indexing, you'll be well-equipped to seamlessly transition to typical data-analysis tools (e.g., the `DataFrames` package).

6b. Named Tuples and Dictionaries

Martin Alfaro
PhD in Economics

INTRODUCTION

Our previous discussions on collections have centered around vectors and tuples. The current section expands on the subject, offering a more comprehensive analysis of tuples and introducing two new types of collections: **named tuples** and **dictionaries**.

In particular, we'll cover how to characterize collections through keys and values, methods for the manipulation of collections, and approaches to transforming one collection into another.

KEYS AND VALUES

Most collections in Julia are characterized by **keys**. They serve as unique identifiers of their elements, and have a corresponding **value** associated with each. [1] For instance, the vector `x = [2, 4, 6]` has the indices `1, 2, 3` as its keys, and `2, 4, 6` as their respective values.

Keys are more general than indices—they encompass all the possible identifiers of a collection's elements (e.g., strings, numbers, or other objects). Instead, indices exclusively employ integers as identifiers.

To identify the keys and values of a collection, Julia offers the functions `keys` and `values`. The following code snippets demonstrate their usage based on vectors and tuples, whose keys are represented by indices. Note that neither `keys` nor `values` return a vector, requiring the `collect` function for this purpose.

```
x = [4, 5, 6]
```

```
julia> collect(keys(x))
3-element Vector{Int64}:
 1
 2
 3
julia> collect(values(x))
3-element Vector{Int64}:
 4
 5
 6
```

```
some_pair = Pair("a", 1)            # equivalent
```

```
julia> collect(keys(x))
"a" => 1
julia> collect(values(x))
1
```

THE TYPE `PAIR`

Collections of key-value pairs in Julia are represented by the type `Pair{<key type>, <value type>}`. Although we won't directly work with objects of this type, they form the basis for constructing other collections such as dictionaries and named tuples.

A key-value pair can be created by using the operator `=>` as in `<key> => <value>`. For instance, `"a" => 1` represents a pair, where `a` is its key and `1` its corresponding value. In addition, we can create pairs through the function `Pair(<key>, <value>)`, making `Pair("a",1)` equivalent to the previous example. Finally, given a pair `x`, its key can be accessed by either `x[1]` or `x.first`, while its value can be retrieved using `x[2]` or `x.second`. All this is demonstrated below.

```
some_pair = ("a" => 1)      # or simply 'some_pair = "a" => 1'

some_pair = Pair("a", 1)    # equivalent
```

```
julia> some_pair
"a" => 1
julia> some_pair[1]
"a"
julia> some_pair.first
"a"
```

```
some_pair = ("a" => 1)      # or simply 'some_pair = "a" => 1'

some_pair = Pair("a", 1)    # equivalent
```

```
julia> some_pair
"a" => 1
julia> some_pair[2]
1
julia> some_pair.second
1
```

THE TYPE `SYMBOL`

The type used to represent keys can vary depending on the collection. An important type used as a key is `Symbol`, which provides an efficient way to represent string-based keys. A symbol labeled `x` is denoted `:x`, and can be created from strings using the function `Symbol(<some string>)`. [2]

```
x = (a=4, b=5, c=6)
```

```
julia> some_symbol
```

```
julia> vector_symbols
3-element Vector{Symbol}:
 :a
 :b
 :c
```

NAMED TUPLES

> **Warning!**
>
> **Tuples and named tuples should only be used for small collections**. Using them for large collections can lead to slow operations or directly result in a fatal error (the so-called stack overflow). Arrays remain the preferred choice for large collections.
>
> Defining what constitutes *small* is challenging, and unfortunately there's no definitive answer. We can only indicate that collections with fewer than 10 elements are undoubtedly small, while those exceeding 100 elements violate the definition.

Named tuples share several similarities with regular tuples, including their **immutability**. However, they also exhibit some notable differences. One of them is that **the keys of named tuples are objects of type** `Symbol`, in contrast to the numerical indices used for regular tuples.

Named tuples also differ syntactically, requiring being enclosed in parentheses `()`—omitting them is not possible, unlike with regular tuples. Furthermore, when creating a named tuple with a single element, the notation requires either a trailing comma `,` after the element (similar to regular tuples) or a leading semicolon `;` before the element. [3]

To construct a named tuple, each element must be specified in the format `<key> = <value>`, such as `a = 10`. Alternatively, you can use a pair `<key with Symbol type> => <value>`, as in `:a => 10`. Once a named tuple `nt` is created, you can access its element `a` by using either `nt[:a]` or `nt.a`.

The following code snippets illustrate all this.

```
# all 'nt' are equivalent
nt = (  a=10, b=20)
nt = (; a=10, b=20)

nt = (  :a => 10, :b => 10)
nt = (; :a => 10, :b => 10)
```

```
julia> nt
(a = 10, b = 20)
julia> nt.a
10
julia> nt[:a] #alternative way to access 'a'
10
```

```
# all 'nt' are equivalent
nt = (  a=10,)
nt = (; a=10 )

nt = (  :a => 10,)
nt = (; :a => 10 )

#not 'nt =  (a = 10)'  -> this is interpreted as 'nt = a = 10'
#not 'nt = (:a => 10)' -> this is interpreted as a pair
```

```
julia> nt
(a = 10, )
julia> nt.a
10
julia> nt[:a] #alternative way to access 'a'
10
```

> **Remark**
>
> To see the list of keys and values, we can employ the functions `keys` and `values`.

```
nt = (a=10, b=20)
```

```
julia> collect(keys(nt))
2-element Vector{Symbol}:
 :a
 :b
julia> values(nt)
(10, 20)
```

DISTINCTION BETWEEN THE CREATION OF TUPLES AND NAMED TUPLES

It's possible to create named tuples from individual variables. For instance, given variables `x = 10` and `y = 20`, you can define `nt = (; x, y)`. This creates a named tuple with keys `x` and `y`, and corresponding values `10` and `20`.

The semicolon `;` is crucial in this construction, as it distinguishes named tuples from regular tuples. Omitting it, as in `nt = (x, y)`, would result in a regular tuple instead.

```
x = 10
y = 20

nt  = (; x, y)
tup = (x, y)
```

```
julia> nt
(x = 10, y = 20)
julia> tup
(10, 20)
```

```
x = 10


nt  = (; x)
tup = (x, )
```

```
julia> nt
(x = 10,)
julia> tup
(10,)
```

DICTIONARIES

Dictionaries are collections of key-value pairs, exhibiting three distinctive features:

- **The keys of dictionaries can be any object**: strings, numbers, and other objects are possible.
- **Dictionaries are mutable**: you can modify, add, and remove elements.
- **Dictionaries are unordered**: keys don't have any order attached.

Dictionaries are created using the function `Dict`, with each argument consisting of a key-value pair denoted by `<key> => <value>`.

```
some_dict = Dict(3 => 10, 4 => 20)
```

```
julia> dict
Dict{Int64, Int64} with 2 entries:
  4 => 20
  3 => 10
julia> dict[1]
10
```

```
dict = Dict("a" => 10, "b" => 20)
```

```
julia> dict
Dict{String, Int64} with 2 entries:
  "b" => 20
  "a" => 10
julia> dict["a"]
10
```

```
some_dict = Dict(:a => 10, :b => 20)
```

```
julia> dict
Dict{Symbol, Int64} with 2 entries:
  :a => 10
  :b => 20
julia> dict[:a]
10
```

```
some_dict = Dict((1,1) => 10, (1,2) => 20)
```

```
julia> dict
Dict{Tuple{Int64, Int64}, Int64} with 2 entries:
  (1, 2) => 20
  (1, 1) => 10
julia> dict[(1,1)]
10
```

Note that regular dictionaries are inherently unordered, meaning access to their elements doesn't follow any pattern. The following example illustrates this, where a vector collects the keys of a dictionary. [4]

```
some_dict = Dict(3 => 10, 4 => 20)

keys_from_dict = collect(keys(some_dict))
```

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

```
some_dict = Dict("a" => 10, "b" => 20)

keys_from_dict = collect(keys(some_dict))
```

```
julia> keys_from_dict
2-element Vector{String}:
 "b"
 "a"
```

```
some_dict = Dict(:a => 10, :b => 20)

keys_from_dict = collect(keys(some_dict))
```

```
julia> keys_from_dict
2-element Vector{Symbol}:
 :a
 :b
```

```
some_dict = Dict((1,1) => 10, (1,2) => 20)

keys_from_dict = collect(keys(some_dict))
```

```
julia> keys_from_dict
2-element Vector{Tuple{Int64, Int64}}:
 (1, 2)
 (1, 1)
```

CREATING TUPLES, NAMED TUPLES, AND DICTIONARIES

Tuples, named tuples, and dictionaries can be constructed from other collections, provided that the source collection possesses a key-value structure. The following examples demonstrate how various collections can be used to create **dictionaries** in particular.

```
vector = [10, 20] # or tupl = (10,20)



dict = Dict(pairs(vector))
```

```
julia> dict
Dict{Int64, Int64} with 2 entries:
  2 => 20
  1 => 10
```

```
keys_for_dict   = [:a, :b]
values_for_dict = [10, 20]


dict = Dict(zip(keys_for_dict, values_for_dict))
```

```
julia> dict
Dict{Symbol, Int64} with 2 entries:
  :a => 10
  :b => 20
```

```
keys_for_dict   = (:a, :b)
values_for_dict = (10, 20)


dict = Dict(zip(keys_for_dict, values_for_dict))
```

```
julia> dict
Dict{Symbol, Int64} with 2 entries:
  :a => 10
  :b => 20
```

```
nt_for_dict = (a = 10, b = 20)



dict = Dict(pairs(nt_for_dict))
```

```
julia> dict
Dict{Symbol, Int64} with 2 entries:
  :a => 10
  :b => 20
```

```
keys_for_dict      = (:a, :b)
values_for_dict    = (10, 20)
vector_keys_values     =     [(keys_for_dict[i],     values_for_dict[i])     for     i     in
eachindex(keys_for_dict)]

dict = Dict(vector_keys_values)
```

```
julia> dict
Dict{Symbol, Int64} with 2 entries:
  :a => 10
  :b => 20
```

Likewise, we can define a **tuple** from other collections, as shown below.

```
a = 10
b = 20

tup = (a, b)
```

```
julia> tup
(10, 20)
```

```
values_for_tup = [10, 20]


tup = (values_for_tup... ,)
```

```
julia> tup
(10, 20)
```

```
values_for_tup = [10, 20]


tup = Tuple(values_for_tup)
```

```
julia> tup
(10, 20)
```

Finally, **named tuples** can be constructed from other collections.

```
a = 10
b = 20


nt = (; a, b)
```

```
julia> nt
(a = 10, b = 20)
```

```
keys_for_nt   = [:a, :b]
values_for_nt = [10, 20]


nt = (; zip(keys_for_nt, values_for_nt)...)
```

```
julia> nt
(a = 10, b = 20)
```

```
keys_for_nt   = [:a, :b]
values_for_nt = [10, 20]


nt = NamedTuple(zip(keys_for_nt, values_for_nt))
```

```
julia> nt
(a = 10, b = 20)
```

```
keys_for_nt   = (:a, :b)
values_for_nt = (10, 20)


nt = NamedTuple(zip(keys_for_nt, values_for_nt))
```

```
julia> nt
(a = 10, b = 20)
```

```
keys_for_nt        = [:a, :b]
values_for_nt      = [10, 20]
vector_keys_values = [(keys_for_nt[i], values_for_nt[i]) for i in eachindex(keys_for_nt)]

nt = NamedTuple(vector_keys_values)
```

```
julia> nt
(a = 10, b = 20)
```

```
dict = Dict(:a => 10, :b => 20)



nt = NamedTuple(vector_keys_values)
```

```
julia> nt
(a = 10, b = 20)
```

DESTRUCTURING TUPLES AND NAMED TUPLES

Previously, we've demonstrated how to create a tuple and a named tuple from variables. Next, we show that the reverse operation is also possible, where **values are extracted from a tuple or named tuple and assigned to individual variables**. This process is known as **destructuring**, and allows users to "unpack" the values of a collection into separate variables.

Destructuring involves the assignment operator `=`, with a tuple or named tuple on the left-hand side. The key difference between them lies in their compatibility with other collections: named tuples on the left-hand side require a matching named tuple, whereas tuples can be paired with a variety of collection types on the right-hand side. We illustrate each case below.

DESTRUCTURING COLLECTIONS THROUGH TUPLES

Given a collection `list` with two elements, destructuring enables the user to create variables `x` and `y` with the values of `list`. This is implemented by the syntax `<tuple> = <collection>`, such as `x,y = list`. The following examples illustrate this operation, according to different objects taken as `list`.

```
list = [3,4]

x,y  = list
```

```
julia> x
3
julia> y
4
```

```
list = 3:4

x,y  = list
```

```
julia> x
3
julia> y
4
```

```
list = (3,4)

x,y  = list
```

```
julia> x
3
julia> y
4
```

```
list = (a = 3, b = 4)

x,y  = list
```

```
julia> x
3
julia> y
4
```

In addition to destructuring all elements in `list`, you can choose to destructure only a subset of elements. The assignment is then performed in sequential order, following the collection's inherent order, without the possibility of skipping any specific value. To explicitly disregard a value, it's common to use the special variable name `_` as a placeholder. Note that this is merely a convention, without any impact on execution.

For illustration purposes, we'll use a vector as an example of `list`, but the same principle applies to any object.

```
list = [3,4,5]

(x,)    = list
```

```
julia> x
3
```

```
list = [3,4,5]

x,y     = list
```

```
julia> x
3
julia> y
4
```

```
list = [3,4,5]

_,_,z   = list                  # _ skips the assignment of that value
```

```
julia> z
5
```

```
list = [3,4,5]

x,_,z   = list                  # _ skips the assignment of that value
```

```
julia> x
3
julia> z
5
```

DESTRUCTURING WITH NAMED TUPLES ON BOTH SIDES

An alternative to standard tuples for destructuring is given by employing named tuples on the left-hand side. This approach lets you extract values by directly referencing field names, rather than relying on their positional order. Its key advantage is that variables can be assigned values in any order, provided their names correspond to the field names in the named tuple.

```
nt             = (; key1 = 10, key2 = 20, key3 = 30)

(; key2, key1) = nt            # keys in any order
```

```
julia> key1
10
julia> key3
30
```

```
nt             = (; key1 = 10, key2 = 20, key3 = 30)

(; key3)       = nt            # only one key
```

```
julia> key3
30
```

Remark

When destructuring with a tuple on the left-hand side and a named tuple on the right-hand side, keep in mind that tuple assignments are strictly positional. This means that variable names don't influence the assignment, which exclusively happens based on the position of values. As a result, the assignment process is unaffected by whether the variable names match the keys of the named tuple.

```
nt = (; key1 = 10, key2 = 20, key3 = 30)

 key2, key1    = nt             # variables defined according to
POSITION
(key2, key1)   = nt            # alternative notation
```

```
julia> key2
10
julia> key1
20
```

```
nt = (; key1 = 10, key2 = 20, key3 = 30)

(; key2, key1) = nt             # variables defined according to
KEY
 ; key2, key1  = nt            # alternative notation
```

```
julia> key1
10
julia> key2
20
```

The same caveat applies to single-variable assignments.

```
nt       = (; key1 = 10, key2 = 20)

(key2,)   = nt                # variable defined according to
POSITION
```

```
julia> key2
10
```

```
nt       = (; key1 = 10, key2 = 20)

(; key2) = nt             # variable defined according to KEY
```

```
julia> key2
20
```

APPLICATIONS OF DESTRUCTURING

Destructuring named tuples is especially useful in models that involve a repeated use of numerous parameters. By storing all these parameters in a named tuple, you can pass a *single* argument to functions. Then, by destructuring the named tuple, you can extract the needed parameters at the beginning of the function body.

```
β = 3
δ = 4
ϵ = 5

# function 'foo' only uses 'β' and 'δ'
function foo(x, δ, β)
    x * δ + exp(β) / β
end
```

```
julia> foo(2, δ, β)
14.695
```

```
parameters_list = (; β = 3, δ = 4, ϵ = 5)



# function 'foo' only uses 'β' and 'δ'
function foo(x, parameters_list)
    x * parameters_list.δ + exp(parameters_list.β) / parameters_list.β
end
```

```
julia> foo(2, parameters_list.β, parameters_list.δ)
14.695
```

```
parameters_list = (; β = 3, δ = 4, ϵ = 5)



# function 'foo' only uses 'β' and 'δ'
function foo(x, parameters_list)
    (; β, δ) = parameters_list

    x * δ + exp(β) / β
end
```

```
julia> foo(2, parameters_list)
14.695
```

Another useful application of destructuring occurs when we need to retrieve multiple outputs of a function. This enables you to store each result in a separate variable. Below, we illustrate this application with a tuple and variables `x`, `y`, and `z`.

```
function foo()
    out1 = 2
    out2 = 3
    out3 = 4

    out1, out2, out3
end

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

```
function foo()
    out1 = 2
    out2 = 3
    out3 = 4

    [out1, out2, out3]
end

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

Another typical application of destructuring is when we need only a subset of a function's outputs. While both tuples and named tuples can be applied for this purpose, the former offers more flexibility since they can be combined with various types of collections. Instead, named tuples are limited to another named tuple as the function's output, further requiring prior knowledge of the output's field names.

The following example demonstrates this functionality by extracting the first and third output of the `foo` function.

```
function foo()
    out1 = 2
    out2 = 3
    out3 = 4

    out1, out2, out3
end

x, _, z = foo()
```

```
function foo()
    out1 = 2
    out2 = 3
    out3 = 4

    [out1, out2, out3]
end

x, _, z = foo()
```

```
function foo()
    out1 = 2
    out2 = 3
    out3 = 4

    (; out1, out2, out3)
end

(; out1, out3) = foo()
```

FOOTNOTES

1. Not all collections map keys to values. For example, the type `Set`, which represents a group of unique unordered elements, doesn't have keys.

2. `Symbol` also enables the creation of variables programmatically. For example, it can be employed for defining new columns in the package `DataFrames`, which provides a table representation of data.

3. The semicolon notation `;` may seem odd, but it actually comes from the syntax for keyword arguments in functions.

4. The package `OrderedCollections` addresses this, by offering a special dictionary called `OrderedDict`. It behaves similarly to regular dictionaries, including their syntax, but endows the dictionary with an order.

6c. Chaining Operations

Martin Alfaro
PhD in Economics

INTRODUCTION

This section introduces two approaches to computing outputs that involve multiple intermediate steps. First, we introduce the so-called **let blocks**, which create a new scope that returns the last line as its output. Let blocks offer a concise way to wrap a sequence of operations, making them similar to functions but with less syntactic clutter. They also help maintain a tidy namespace, as all intermediate variables will be local and therefore inaccessible outside the block.

The second approach is based on **pipes**, which chain a series of operations and return the final result as its output. As the built-in pipe can become unwieldy beyond single-argument functions, we also present an alternative based on the `Pipe` package.

LET BLOCKS

Let blocks are particularly helpful when we need to perform a series of operations, but only care about the ultimate result. To illustrate their utility, suppose we want to compute the rounded logarithm of `a`'s absolute value, formally expressed as $\text{round}\left(\ln\left(|a|\right)\right)$.

In Julia, this operation can be implemented using the expression `round(log(abs(a)))`, where `round(a)` returns the nearest integer to `a`. However, the readability of this expression is less than ideal due to the multiple parentheses, with the issue potentially exacerbated if variables and functions had long names.

To improve clarity, we could break the whole operation into multiple smaller steps: *i)* compute the absolute value of `a`, *ii)* compute the logarithm of the result, and *iii)* round the resulting output. An easy way to implement this is to create three intermediate variables to store the output in each step. Nonetheless, this approach would clutter our namespace and potentially obscure the nested nature of the operations.

A more elegant solution is to introduce a **let-block**, which resembles functions in several respects. This construct creates a new scope delimited by the `let` and `end` keywords, enabling multiple calculations to be performed within it. The result of the last calculation is then returned as the output of the let-block. Like functions, let-blocks also allow arguments to be passed by specifying them after the `let` keyword.

To highlight the benefits of let-blocks, the following examples compare various approaches to computing `round(log(abs(a)))`.

```
a      = -2

output = round(log(abs(a)))
```

```
julia> output
1.0
julia> temp1
julia> temp2
4
```

```
a      = -2

temp1  = abs(a)
temp2  = log(temp1)
output = round(temp2)
```

```
julia> output
1.0
```

```
a      = -2

output = let b = a          # 'b' is a local variable having the value of 'a'
   temp1 = abs(b)
   temp2 = log(temp1)
   round(temp2)
end
```

```
julia> output
1.0
julia> temp1 #local to let-block
ERROR: UndefVarError: `temp1` not defined
julia> temp2 #local to let-block
ERROR: UndefVarError: `temp2` not defined
```

```
a      = -2

output = let a = a          # the 'a' on the left of `=` defines a local variable
   temp1 = abs(a)
   temp2 = log(temp1)
   round(temp2)
end
```

```
julia> output
1.0
julia> temp1 #local to let-block
ERROR: UndefVarError: `temp1` not defined
julia> temp2 #local to let-block
ERROR: UndefVarError: `temp2` not defined
```

> **Let Blocks Can Mutate Variables**

Let blocks behave like functions regarding assignments and mutation. This means that you can mutate their arguments, but can't reassign variables.

```
x = [2,2,2]

output = let x = x
   x[1] = 0
end
```

```
julia> x
3-element Vector{Int64}:
 0
 2
 2
```

```
x = [2,2,2]

output = let x = x
   x = 0
end
```

```
julia> x
3-element Vector{Int64}:
 2
 2
 2
```

Since mutations are possible within let-blocks, exercise caution to prevent unintended side effects in the global scope.

PIPES

Pipes provide an alternative to let-blocks for operations with multiple intermediate steps. Unlike let-blocks, they're specifically designed to chain operations together, with each step taking the output of the previous step as its input. These steps are separated through the `|>` keyword.

Pipes are particularly well-suited for sequential applications of single-argument functions. To illustrate this, let's revisit the example presented above for let blocks.

```
a      = -2

output = round(log(abs(a)))
```

```
julia> output
1.0
```

```
a      = -2

output = a |> abs |> log |> round
```

```
julia> output
1.0
```

Let Blocks and Pipes For Long Names

Both approaches facilitate the creation of temporary aliases for variables with lengthy names. In this way, users can assign meaningful names to variables, while preserving code readability.

```
variable_with_a_long_name = 2

output          =          variable_with_a_long_name          -
log(variable_with_a_long_name) / abs(variable_with_a_long_name)
```

```
julia> output
1.6534264097200273
```

```
variable_with_a_long_name = 2

temp   = variable_with_a_long_name
output = temp - log(temp) / abs(temp)
```

```
julia> output
1.6534264097200273
```

```
variable_with_a_long_name = 2

output = variable_with_a_long_name  |>
         a -> a - log(a) / abs(a)
```

```
julia> output
1.6534264097200273
```

```
variable_with_a_long_name = 2

output = let x = variable_with_a_long_name
    x - log(x) / abs(x)
end
```

```
julia> output
1.6534264097200273
```

BROADCASTING PIPES

Just like any other operator, pipes can be broadcasted by prefixing them with a dot `.`. Thus, `.|>` indicates that the subsequent operation must be applied element-wise to the preceding output. For example, the expression `x .|> abs` is equivalent to `abs.(x)`.

To demonstrate its use, suppose we want to transform `x` by taking the logarithm of its absolute values, and then sum the results.

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

output = sum(log.(abs.(x)))
```

```
julia> output
1.791759469228055
```

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

temp1  = abs.(x)
temp2  = log.(temp1)
output = sum(temp2)
```

```
julia> output
1.791759469228055
```

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

output = x .|> abs .|> log |> sum
```

```
julia> output
1.791759469228055
```

PIPES WITH MORE COMPLEX OPERATIONS

Our examples of pipes so far have followed a simple pattern, with each step consisting of a single-argument function. However, pipes applied in the current form preclude their application to multiple-argument functions or even operations. For example, it prevents the incorporation of expressions like `foo(x,y)` or `2 * x`.

To incorporate such cases, we can **combine pipes with anonymous functions**. In this way, the user can specify how the output of the previous step is incorporated into the subsequent operation. As shown below, the technique greatly expands the utility of pipes.

```
a      = -2

output = round(2 * abs(a))
```

```
a      = -2

temp1  = abs(a)
temp2  = 2 * temp1
output = round(temp2)
```

```
a      = -2

output = a |> abs |> (x -> 2 * x) |> round

#equivalent and more readable
output = a              |>
         abs            |>
         x -> 2 * x     |>
         round
```

PACKAGE PIPE

Combining pipes and anonymous functions can result in cumbersome code, defeating the very own purpose of using pipes to write clean and readable code.

The `Pipe` package provides a convenient solution, eliminating the need for anonymous functions. By prefixing the operation chain with the `@pipe` macro, you can reference the output of the previous step by the symbol `_`. Furthermore, for simple operations that don't require anonymous functions, and therefore don't need `_`, `@pipe` has the same syntax as built-in pipes.

To demonstrate its use, we revisit the last example.

```
#
a      = -2

output = a |> abs |> (x -> 2 * x) |> round

#equivalent and more readable
output =       a            |>
               abs          |>
               x -> 2 * x   |>
               round
```

```
using Pipe
a = -2

output = @pipe a |> abs |> 2 * _ |> round

#equivalent and more readable
output = @pipe a            |>
               abs          |>
               2 * _        |>
               round
```

FUNCTION COMPOSITION (*OPTIONAL*)

An alternative approach to nest functions is through the composition operator `∘`. This symbol can be inserted by tab completion through `\circ`, and its functionality is the same as in Mathematics. Specifically, for some functions `f` and `g`, `(f ∘ g)(x)` is equivalent to `f(g(x))`.

The operator `∘` can be considered as an alternative to piping, as it provides the same output as `x |> f |> g`. Moreover, `∘` is also available as a function, where `∘(f,g)(x)` is equivalent to `(f ∘ g)(x)`. The following examples demonstrate its use.

```
a        = -1



# all `output` are equivalent
output   = log(abs(a))
output   = a |> abs |> log
output   = (log ∘ abs)(a)
output   = ∘(log, abs)(a)
```

```
julia> output
0.0
```

```
a        = 2
outer(a) = a + 2
inner(a) = a / 2

# all `output` are equivalent
output   = (a / 2) + 2
output   = outer(inner(a))
output   = a |> inner |> outer
output   = (outer ∘ inner)(a)
output   = ∘(outer, inner)(a)
```

```
julia> output
3.0
```

The resulting function of the function composition can be broadcasted. The notation for implementing this is easier to understand by thinking of compositions as a new function $h := f \circ g$. This entails that $h(x) := (f \circ g)(x)$ and therefore $h(x) := (f \circ g)(x) = f[g(x)]$. Considering this, broadcasting `h` would require `h.(x)`, which is equivalent to `(f ∘ g).(x)` or `∘(f,g).(x)`.

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



# all `output` are equivalent
output   = log.(abs.(x))
output   = x .|> abs .|> log
output   = (log ∘ abs).(x)
output   = ∘(log, abs).(x)
```

```
julia> output
3-element Vector{Float64}:
 0.0
 0.6931471805599453
 1.0986122886681098
```

```
x        = [1, 2, 3]
outer(a) = a + 2
inner(a) = a / 2

# all `output` are equivalent
output   = (x ./ 2) .+ 2
output   = outer.(inner.(x))
output   = x .|> inner .|> outer
output   = (outer ∘ inner).(x)
output   = ∘(outer, inner).(x)
```

```
julia> output
3-element Vector{Float64}:
 2.5
 3.0
 3.5
```

Lastly, we can broadcast the composition operator `∘` itself, allowing us to apply multiple functions to the same object. For instance, the following example ensures that each function takes the absolute value of its argument.

```julia
a            = -1

inners       = abs
outers       = [log, sqrt]
compositions = outers .∘ inners

# all `output` are equivalent
output       = [log(abs(a)), sqrt(abs(a))]
output       = [foo(a) for foo in compositions]
```

```
julia> compositions
2-element Vector{ComposedFunction{O, typeof(abs)} where O}:
 log ∘ abs
 sqrt ∘ abs
julia> output
2-element Vector{Float64}:
 0.0
 1.0
```

6d. Useful Functions for Vectors

Martin Alfaro
PhD in Economics

INTRODUCTION

This section provides an overview of essential functions for manipulating vectors, including sorting, identifying unique elements, counting occurrences, and ranking data. Our ultimate goal is to apply these functions in a practical context, which we'll do in the next section.

SORTING VECTORS

The `sort` function allows the user to arrange elements in a specific order. By default, it sorts elements in ascending order, but this can be easily reversed to a descending order by setting the keyword argument `rev = true`. The function comes in two variants: `sort`, which returns a new sorted copy, and the in-place version `sort!`, which directly updates the vector.

SORT (ASCENDING)

```
x = [4, 5, 3, 2]

y = sort(x)
```

```
julia> y
4-element Vector{Int64}:
 2
 3
 4
 5
```

SORT (DESCENDING)

```
x = [4, 5, 3, 2]

y = sort(x, rev=true)
```

```
julia> y
4-element Vector{Int64}:
 5
 4
 3
 2
```

SORT!

```
x = [4, 5, 3, 2]

sort!(x)
```

```
julia> x
4-element Vector{Int64}:
 2
 3
 4
 5
```

Both `sort(x)` and `sort!(x)` allow the sorting order to be dictated by transformations of `x`. Specifically, given a function `foo` and leveraging the keyword argument `by`, the sorted order can be determined by the values of `foo(x)`. We demonstrate this below through the function `sort`.

SORT - ABSOLUTE

```
x      = [4, -5, 3]


y      = sort(x, by = abs)      # 'abs' computes the absolute value
```

```
julia> abs.(x)
3-element Vector{Int64}:
 4
 5
 3

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

SORT - QUADRATIC

```
x      = [4, -5, 3]

foo(a) = a^2
y      = sort(x, by = foo)      # same as sort(x, by = x -> x^2)
```

```
julia> foo.(x)
3-element Vector{Int64}:
 16
 25
  9

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

SORT - NEGATIVE

```
x      = [4, -5, 3]

foo(a) = -a
y      = sort(x, by = foo)      # same as sort(x, by = x -> -x)
```

```
julia> foo.(x)
3-element Vector{Int64}:
 -4
  5
 -3
julia> y
3-element Vector{Int64}:
  4
  3
 -5
```

RETRIEVING INDICES OF SORTED ELEMENTS

While `sort` returns the ordered *values* of the vectors, you may also be interested in the *indices* of the sorted elements. This functionality is provided by the function `sortperm`, which returns the indices of `x` that would result in `sort(x)`. In other words, `x[sortperm(x)] == sort(x)` is true. [1]

EXAMPLE 1

```
x          = [1, 2, 3, 4]

sort_index = sortperm(x)
```

```
julia> sort_index
4-element Vector{Int64}:
 1
 2
 3
 4
```

EXAMPLE 2

```
x          = [3, 4, 5, 6]

sort_index = sortperm(x)
```

```
julia> sort_index
4-element Vector{Int64}:
 1
 2
 3
 4
```

EXAMPLE 3

```
x          = [1, 3, 4, 2]

sort_index = sortperm(x)
```

```
julia> sort_index
4-element Vector{Int64}:
 1
 4
 2
 3
```

Analyzing the examples, we can see that the elements in the first two examples are already in ascending order. As a result, `sortperm` returns the trivial permutation `[1, 2, 3, 4]`. In contrast, the last example features an unordered vector `x = [1, 3, 4, 2]`. Thus, the resulting vector `[1, 4, 2, 3]` indicates that the smallest element is at index 1, the second smallest is at index 4, the third smallest is at index 2, and the largest at index 3.

Similar to `sort`, `sortperm` also allows retrieving the indices in descending order. This requires setting the `rev` keyword argument to `true`.

EXAMPLE 1

```
x          = [9, 3, 2, 1]

sort_index = sortperm(x, rev=true)
```

```
julia> sort_index
4-element Vector{Int64}:
 1
 2
 3
 4
```

EXAMPLE 2

```
x          = [9, 5, 3, 1]

sort_index = sortperm(x, rev=true)
```

```
julia> sort_index
4-element Vector{Int64}:
 1
 2
 3
 4
```

EXAMPLE 3

```
x          = [9, 3, 5, 1]

sort_index = sortperm(x, rev=true)
```

```
julia> sort_index
4-element Vector{Int64}:
 1
 3
 2
 4
```

Finally, `sortperm` also supports the keyword argument `by`. This allows users to define a custom transformation, which serves as the sorting criterion for the indices provided.

SORT - ABSOLUTE

```
x      = [4, -5, 3]


value  = sort(x, by = abs)      # 'abs' computes the absolute value
index  = sortperm(x, by = abs)
```

```
julia> abs.(x)
3-element Vector{Int64}:
 4
 5
 3

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

julia> index
3-element Vector{Int64}:
 3
 1
 2
```

SORT - QUADRATIC

```
x      = [4, -5, 3]

foo(a) = a^2
value  = sort(x, by = foo)      # same as sort(x, by = x -> x^2)
index  = sortperm(x, by = foo)
```

```
julia> foo.(x)
3-element Vector{Int64}:
 16
 25
  9

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

julia> index
3-element Vector{Int64}:
 3
 1
 2
```

SORT - NEGATIVE

```
x      = [4, -5, 3]

foo(a) = -a
value  = sort(x, by = foo)      # same as sort(x, by = x -> -x)
index  = sortperm(x, by = foo)
```

```
julia> foo.(x)
3-element Vector{Int64}:
 -4
  5
 -3

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

julia> index
3-element Vector{Int64}:
 1
 3
 2
```

AN EXAMPLE

One common application of `sortperm` is to reorder a variable based on the values of another variable. For example, suppose we want to assess the daily failures of a machine. Focusing on the first three days of the month, the following code snippet ranks these days by their corresponding failure counts.

DAYS SORTED BY LOWEST NUMBER OF FAILURES

```
days     = ["one", "two", "three"]
failures = [8, 2, 4]

index            = sortperm(failures)
days_by_failures = days[index]                    # days sorted by lowest failures
```

```
julia> index
3-element Vector{Int64}:
 2
 3
 1
julia> days_by_earnings
3-element Vector{String}:
 "two"
 "three"
 "one"
```

REMOVING DUPLICATES

The `unique` function eliminates duplicates from a vector, returning a vector containing each element only once. The function comes in two variants, with `unique` providing a new copy, and the in-place version `unique!` directly updating the original vector.

UNIQUE

```
x = [2, 2, 3, 4]

y = unique(x)       # returns a new vector
```

```
julia> x
4-element Vector{Int64}:
 2
 2
 3
 4
julia> y
3-element Vector{Int64}:
 2
 3
 4
```

UNIQUE!

```
x = [2, 2, 3, 4]

unique!(x)          # mutates 'x'
```

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

The `StatsBase` package also offers a related function called `countmap`. This enumerates the number of times each element shows up in a vector. Formally, it returns a dictionary, where the unique elements serve as keys, and their corresponding values represent the number of occurrences of that element.

As the keys in the dictionary are unsorted by design, you must apply the `sort` function to the result if you prefer sorted keys. Note that the application of `sort` will automatically transform an ordinary dictionary into an object with type `OrderedDict`.

UNSORTED COUNT

```
using StatsBase
x           = [6, 6, 0, 5]

y           = countmap(x)              # Dict with `element => occurrences`

elements    = collect(keys(y))
occurrences = collect(values(y))
```

```
julia> y
Dict{Int64, Int64} with 3 entries:
  0 => 1
  5 => 1
  6 => 2
julia> elements
3-element Vector{Int64}:
 0
 5
 6
julia> occurrences
3-element Vector{Int64}:
 1
 1
 2
```

SORTED COUNT

```
using StatsBase
x           = [6, 6, 0, 5]

y           = sort(countmap(x))        # OrderedDict with `element => occurrences`

elements    = collect(keys(y))
occurrences = collect(values(y))
```

```
julia> y
OrderedCollections.OrderedDict{Int64, Int64} with 3 entries:
  0 => 1
  5 => 1
  6 => 2
julia> elements
3-element Vector{Int64}:
 0
 5
 6
julia> occurrences
3-element Vector{Int64}:
 1
 1
 2
```

ROUNDING NUMBERS

Julia provides standard functions to approximate numerical values to a specific precision:

- `round` approximates the number to its nearest integer.
- `floor` approximates the number down to its nearest integer.
- `ceil` approximates the number up to its nearest integer.

Below, we show that these functions are quite flexible. In particular, they allow the user to specify the output's type (e.g., `Int64` or `Float64`), the number of decimals to be included through the keyword argument `digits`, and the significant digits.

ROUND

```
x = 456.175

round(x)                          # 456.0

round(x, digits=1)                # 456.2
round(x, digits=2)                # 456.18

round(Int, x)                     # 456

round(x, sigdigits=1)             # 500.0
round(x, sigdigits=2)             # 460.0
```

FLOOR

```
x = 456.175

floor(x)                          # 456.0

floor(x, digits=1)                # 456.1
floor(x, digits=2)                # 456.17

floor(Int, x)                     # 456

floor(x, sigdigits=1)             # 400.0
floor(x, sigdigits=2)             # 450.0
```

CEIL

```
x = 456.175

ceil(x)                           # 457.0

ceil(x, digits=1)                 # 456.2
ceil(x, digits=2)                 # 456.18

ceil(Int, x)                      # 457

ceil(x, sigdigits=1)              # 500.0
ceil(x, sigdigits=2)              # 460.0
```

RANKINGS

Instead of sorting a vector, you may be interested in determining the rank position of each element. The `StatsBase` package offers two functions for this purpose, `competerank` and `ordinalrank`. The main difference between them lies in how they handle identical elements: `competerank` assigns the same rank to identical elements, while `ordinalrank` assigns different ranks to these elements. Both functions return a rank such that 1 corresponds to the lowest value. If you prefer to invert the ranking, so that the highest value corresponds to a rank of 1, you can add the keyword argument `rev = true`.

RANK (SAME RANK FOR TIES)

```
using StatsBase
x = [6, 6, 0, 5]

y = competerank(x)
```

```
julia> y
4-element Vector{Int64}:
 3
 3
 1
 2
```

RANK (SAME RANK FOR TIES)

```
using StatsBase
x = [6, 6, 0, 5]

y = competerank(x, rev=true)
```

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

RANK (UNIQUE POSITIONS)

```
using StatsBase
x = [6, 6, 0, 5]

y = ordinalrank(x)
```

```
julia> y
4-element Vector{Int64}:
 3
 4
 1
 2
```

RANK (UNIQUE POSITIONS)

```
using StatsBase
x = [6, 6, 0, 5]

y = ordinalrank(x, rev=true)
```

```
julia> y
4-element Vector{Int64}:
 1
 2
 4
 3
```

> **Do not confuse `ordinalrank` and `sortperm`**
>
> The function `ordinalrank` indicates the position of each value in the *sorted* vector, while `sortperm` indicates the position of each value in the *unsorted* vector.

'ORDINALRANK'

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

y = ordinalrank(x)
```

```
julia> y
3-element Vector{Int64}:
 3
 1
 2
```

'SORTPERM'

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

y = sortperm(x)
```

```
julia> y
3-element Vector{Int64}:
 2
 3
 1
```

EXTREMA (MAXIMUM AND MINIMUM)

We conclude by identifying extrema in a vector, along with their corresponding indices. The following examples illustrate the functionality for the maximum, with analogous functions available for the minimum.

VALUE OF MAXIMUM

```
x = [6, 6, 0, 5]

y = maximum(x)
```

```
julia> y
6
```

INDEX OF MAXIMUM

```
x = [6, 6, 0, 5]

y = argmax(x)
```

```
julia> y
1
```

VALUE AND INDEX

```
x = [6, 6, 0, 5]

y = findmax(x)
```

```
julia> y
(6, 1)
```

Julia additionally provides the function `max` and `min`, which respectively return the maximum and minimum of its arguments. These functions are particularly useful when applied in binary operations.

'MAX' FUNCTION

```
x = 3
y = 4

z = max(x,y)
```

```
julia> z
4
```

FOOTNOTES

[1] The name `sortperm` originates from "sorting permutation". Although the name might seem somewhat opaque, it arises because the operation returns the permutation of indices that would sort the original vector.

6e. Illustration - Johnny, the YouTuber

Martin Alfaro
PhD in Economics

INTRODUCTION

Through this section's illustration, we'll show the convenience of the following features:

1. Boolean indexing for working with subsets of the data
2. organizing code around functions
3. pipes to enhance code readability
4. use of views to modify subsets of the data

DESCRIBING THE SCENARIO

We'll explore the stats of Johnny's YouTube channel during a month. He has a median of 50,000 visits per video, with a few viral videos exceeding 100,000 visits. The information at our disposal is:

- `nr_videos`: 30 (one per day).
- `visits`: viewers per video (in thousands).
- `payrates`: Dollars paid per video for 1,000 visits, ranging from $2 to $6. The fluctuation is consistent with YouTube's payment model, which depends on a video's feature (e.g., content, duration, retention).

The scenario is modeled by some mock data. The details of how data are generated are unimportant, but they were added below for the sake of completeness. What matters is that the mock data creates the following two variables.

```
using StatsBase, Distributions
using Random; Random.seed!(1234)

function audience(nr_videos; median_target)
    shape = log(4,5)
    scale = median_target / 2^(1/shape)

    visits = rand(Pareto(shape,scale),  nr_videos)

    return visits
end

nr_videos = 30

visits   = audience(nr_videos, median_target = 50)      # in thousands of visits
payrates = rand(2:6, nr_videos)                         # per thousands of visits
```

```
julia> visits # in thousands
30-element Vector{Float64}:
 44.4608
 57.2323
  ⋮
 86.4182
 36.5051
julia> payrates # per thousand visits
30-element Vector{Int64}:
 2
 6
 ⋮
 3
 4
```

These two variables enable us to calculate the total payment per video.

```
earnings = visits .* payrates
```

```
julia> earnings
30-element Vector{Float64}:
  88.9215
 343.394
   ⋮
 259.254
 146.02
```

SOME GENERAL INFORMATION

We begin by presenting some information about the per-view payments made by YouTube. We first confirm that Johnny's payments ranged from \$2 to \$6. Moreover, using the `countmaps` function from the `StatsBase` package, we conclude that Johnny has eight videos reaching the maximum payment of \$6.

```
range_payrates  = unique(payrates) |> sort
```

```
julia> range_payrates
5-element Vector{Int64}:
 2
 3
 4
 5
 6
```

```
using StatsBase
occurrences_payrates = countmap(payrates) |> sort
```

```
julia> occurrences_payrates
OrderedDict{Int64, Int64} with 5 entries:
  2 => 4
  3 => 5
  4 => 8
  5 => 5
  6 => 8
```

We can also provide some insights into Johnny's most profitable videos. By applying the `sort` function and isolating the top 3 videos, we can obtain information on the highest earnings videos. Moreover, we can apply the `sortperm` function to identify the indices of these videos, allowing us to extract the payment per view and total visits associated with each one.

```
top_earnings    = sort(earnings, rev=true)[1:3]
```

```
julia> top_earnings
3-element Vector{Float64}:
 2708.57
 1083.07
  723.493
```

```
indices         = sortperm(earnings, rev=true)[1:3]

sorted_payrates = payrates[indices]
```

```
julia> sorted_payrates
3-element Vector{Int64}:
 6
 5
 5
```

```
indices         = sortperm(earnings, rev=true)[1:3]

sorted_visits   = visits[indices]
```

```
julia> sorted_visits
3-element Vector{Float64}:
 451.428
 216.615
 144.699
```

BOOLEAN VARIABLES

In the following, we demonstrate how to use Boolean indexing to extract and characterize subsets of data. Our focus will be on characterizing Johnny's viral videos, defined as those that have surpassed a threshold of 100k visits. In particular, we'll determine the number of visits and revenue generated by these videos.

To identify the viral videos, we'll create a `Bool` vector, where `true` identifies a viral video. This vector allows us to selectively extract data points from other variables by using them as indices. For instance, we use it below to compute the total visits and earnings derived from these viral videos.

```
# characterization of viral videos
viral_threshold = 100
is_viral        = (visits .≥ viral_threshold)

# stats
viral_nrvideos  = sum(is_viral)
viral_visits    = sum(visits[is_viral])
viral_revenue   = sum(earnings[is_viral])
```

```
julia> viral_nrvideos
6
julia> viral_visits
1243.63
julia> viral_revenue
6133.19
```

Boolean indexing also makes it possible to subset data satisfying multiple conditions. For instance, we can use this technique to calculate the proportion of viral videos for which YouTube paid more than $3 per thousand visits.

```
# characterization
viral_threshold    = 100
payrates_above_avg = 3

is_viral           = (visits .≥ viral_threshold)
is_viral_lucrative = (visits .≥ viral_threshold) .&& (payrates .> payrates_above_avg)

# stat
proportion_viral_lucrative = sum(is_viral_lucrative) / sum(is_viral) * 100
```

```
julia> proportion_viral_lucrative
83.3333
```

Rounding Outputs

You can express results with rounded numbers via the function `round`. By default, this returns the nearest integer expressed as a `Float64` number.

The function also offers additional specifications. For instance, you can control the number of decimal places in the approximation using the `digits` keyword argument. Furthermore, it's possible to represent the number as an `Int64` using the argument `Int`.

```
rounded_proportion = round(proportion_viral_lucrative)
```

```
julia> rounded_proportion
83.0
```

```
rounded_proportion      =      round(proportion_viral_lucrative,
digits=1)
```

```
julia> rounded_proportion
83.3
```

```
rounded_proportion = round(Int, proportion_viral_lucrative)
```

```
julia> rounded_proportion
83
```

FUNCTIONS TO REPRESENT TASKS

The approach employed so far allows for a rapid exploration of Johnny's viral videos. However, it falls short in providing a systematic analysis that could be extended to other subsets of data. To overcome this limitation, we can automate the process by defining a function.

Recall that a well-designed function should embody a specific task, implying that it must be independent of its specific application. In our case, the goal is to subset data and extract specific statistics, including the number of videos, visits, and revenue generated.

The function below implements this task taking three arguments: the raw data (`visits` and `payrates`) and a condition defining the subset (`condition`). By keeping the condition generic, we can seamlessly apply the same analysis to various subsets of data. The example also showcases the convenience of pipes to compute intermediate temporary steps, using it to retrieve the income earned from a subset of videos.

```
#
function stats_subset(visits, payrates, condition)
    nrvideos = sum(condition)
    audience = sum(visits[condition])

    earnings = visits .* payrates
    revenue  = sum(earnings[condition])

    return (; nrvideos, audience, revenue)
end
```

```
using Pipe
function stats_subset(visits, payrates, condition)
    nrvideos = sum(condition)
    audience = sum(visits[condition])


    revenue  = @pipe (visits .* payrates) |> x -> sum(x[condition])

    return (; nrvideos, audience, revenue)
end
```

```
using Pipe
function stats_subset(visits, payrates, condition)
    nrvideos = sum(condition)
    audience = sum(visits[condition])


    revenue  = @pipe (visits .* payrates) |> sum(_[condition])

    return (; nrvideos, audience, revenue)
end
```

Below, we illustrate how the function enables effortlessly characterizing various subsets of data.

```
viral_threshold  = 100
is_viral         = (visits .≥ viral_threshold)
viral            = stats_subset(visits, payrates, is_viral)
```

```
julia> viral
(nrvideos = 6, audience = 1243.63, revenue = 6133.19)
```

```
viral_threshold  = 100
is_notviral      = .!(is_viral)      # '!' is negating a boolean value and we broadcast it
notviral         = stats_subset(visits, payrates, is_notviral)
```

```
julia> notviral
(nrvideos = 24, audience = 1169.13, revenue = 4971.02)
```

```
days_to_consider = (1, 10, 25)      # days when the videos were posted
is_day           = in.(eachindex(visits), Ref(days_to_consider))
specific_days    = stats_subset(visits, payrates, is_day)
```

```
julia> specific_days
(nrvideos = 3, audience = 118.547, revenue = 414.113)
```

MUTATING VARIABLES

Suppose that, seeking to enhance audience engagement, Johnny has decided to promote his videos through advertising. His projections suggest that ads will boost viewership per video by 20%. However, due to budget constraints, Johnny must choose between promoting either his non-viral videos or his viral ones. To make an informed decision, Johnny decides to leverage the data at his disposal to crunch some rough estimates. In particular, he'll base his decision on the earnings he would've earned during last month if he had run targeted ads.

The computations require creating a modified copy of `visits`, adjusting the audience data after running the ads for the targeted videos. With this updated audience data, we can then apply the previously defined `stats_subset` function to estimate the potential earnings. Comparing the results in each tab, Johnny would conclude that promoting viral videos seems to be a more profitable strategy.

```
# 'temp' modifies 'new_visits'
new_visits      = copy(visits)
temp            = @view new_visits[new_visits .≥ viral_threshold]
temp           .= 1.2 .* temp

allvideos       = trues(length(new_visits))
targetViral     = stats_subset(new_visits, payrates, allvideos)
```

```
julia> targetViral
(nrvideos = 30, audience = 2661.48, revenue = 12330.8)
```

```
# 'temp' modifies 'new_visits'
new_visits      = copy(visits)
temp            = @view new_visits[new_visits .< viral_threshold]
temp           .= 1.2 .* temp

allvideos       = trues(length(new_visits))
targetNonViral  = stats_subset(new_visits, payrates, allvideos)
```

```
julia> targetNonViral
(nrvideos = 30, audience = 2646.58, revenue = 12098.4)
```

Be Careful with Misusing 'view'

Updating `temp` requires an in-place operation to mutate the parent object. In our case, this was achieved via the broadcasted operator `.=`. Below, we state some implementations that fail to produce the desired result.

```
new_visits = copy(visits)


temp  = @view new_visits[new_visits .≥ viral_threshold]
temp .= temp .* 1.2
```

```
new_visits = visits      # it creates an alias, it's a view of
the original object!!!

# 'temp' modifies 'visits' -> you lose the original info
temp  = @view new_visits[new_visits .≥ viral_threshold]
temp .= temp .* 1.2
```

```
new_visits = copy(visits)

# wrong -> not using `temp .= temp .* 1.2`
temp  = @view new_visits[new_visits .≥ viral_threshold]
temp  = temp .* 1.2      # it creates a new variable 'temp', it
does not modify 'new_visits'
```

Use of "Let Blocks" To Avoid Bugs

The code above for "Target Viral" and "Target Non-Viral" refers to each variable by an identical name. This increases the risk of accidentally referring to a variable from a different scenario.

The likelihood of incurring this issue can be alleviated by employing "let blocks". By defining their own scope, they help maintain a clean namespace.

```
targetViral      = let visits = visits, payrates = payrates,
threshold = viral_threshold
    new_visits = copy(visits)
    temp       = @view new_visits[new_visits .≥ threshold]
    temp      .= 1.2 .* temp

    allvideos  = trues(length(new_visits))
    stats_subset(new_visits, payrates, allvideos)
end
```

```
julia> targetViral
(nrvideos = 30, audience = 2661.48, revenue = 12330.8)
```

```
targetNonViral  = let visits = visits, payrates = payrates,
threshold = viral_threshold
    new_visits = copy(visits)
    temp       = @view new_visits[new_visits .< threshold]
    temp      .= 1.2 .* temp

    allvideos  = trues(length(new_visits))
    stats_subset(new_visits, payrates, allvideos)
end
```

```
julia> targetNonViral
(nrvideos = 30, audience = 2646.58, revenue = 12098.4)
```

BROADCASTING OVER A LIST OF FUNCTIONS (*OPTIONAL*)

At the beginning of the analysis, we could've derived descriptive statistics to gain insights about Johnny's videos. This can be accomplished by using the `describe` function from the `StatsBase` package. Despite this, the presentation aims to highlight that functions are first-class objects in Julia. This property entails that a function behaves just like any other variable, allowing the user to define a list of functions and then apply them element-wise to a variable.

```
list_functions = [sum, median, mean, maximum, minimum]

stats_visits   = [fun(visits) for fun in list_functions]
```

```
julia> stats_visits
5-element Vector{Float64}:
 2661.48
   52.7884
   88.716
  541.714
   27.7205
```

By broadcasting the operation, we can also compute stats for multiple variables concurrently. For instance, below we characterize `visits` and `earnings` simultaneously.

```
list_functions = [sum, median, mean, maximum, minimum]

stats_various  = [fun.([visits, payrates]) for fun in list_functions]
```

```
julia> stats_various
5-element Vector{Vector{Float64}}:
 [2661.48, 128.0]
 [52.7884, 4.0]
 [88.716, 4.26667]
 [541.714, 6.0]
 [27.7205, 2.0]
```

One major limitation of the current method is its inability to capture each statistic's name. To overcome this, we can employ a named tuple, which we'll call `stats_visits`. The approach enables us to access stats through their respective names, such as `stats_visits.mean` or `stats_visits[:mean]` for the average value.

The implementation is based on the type `Symbol`. This converts strings into identifiers, which are necessary to programmatically access a named tuple's keys.

```
vector_of_tuples = [(Symbol(fun), fun(visits)) for fun in list_functions]
stats_visits     = NamedTuple(vector_of_tuples)
```

```
julia> stats_visits
(sum = 2661.48, median = 52.7884, mean = 88.716, maximum = 541.714, minimum = 27.7205)
julia> stats_visits.mean
88.716
julia> stats_visits[:median]
52.7884
```

7a. Overview and Goals

Martin Alfaro
PhD in Economics

The first part of the website has laid the groundwork for working with Julia. This demanded introducing fundamental data types, such as scalars, vectors, and tuples. Alongside these, we've covered essential programming constructs, including functions, conditionals, and for-loops. While these concepts may vary in syntax and usage across different programming languages, their underlying principles remain universal.

In the second part of the website, we'll shift our attention to one of Julia's most distinctive strengths: **high-performance computing**. When paired with its intuitive syntax and interactive nature, this feature makes Julia an ideal choice for scientific applications.

The domain of high-performance computing is vast and complex. Moreover, each subject has idiosyncratic features that make certain optimizations more or less relevant. Given this breadth, I've made deliberate choices about what to include and exclude. The challenge lay in striking the right balance between providing sufficient background knowledge for explaining a technique, while avoiding unnecessary specificity.

Considering this inherent trade-off, I've chosen the subjects with the goal of equipping readers with practical knowledge for optimizing code, without overwhelming them with excessive detail. In particular, the primary focus will be on what I consider to be the essentials for performance in Julia: **type stability** and **reductions in memory allocations**. The former in particular constitutes a prerequisite for achieving high performance in Julia, making it necessary for any further optimization.

The discussion of high performance in Julia will lead us to consider its type system. Nonetheless, some valuable concepts related to it have been left out. In particular, the concept of `struct`, which allows users to create their own custom objects, won't be covered. There are two reasons for this omission. First, while important for project development, the subject can be bypassed when analyzing high performance, without compromising its understanding. Second, the section included on types is already long enough—adding more subjects could divert the reader's attention away from the primary focus, which is learning high-performance techniques.

7b. When To Optimize Code?

Martin Alfaro
PhD in Economics

INTRODUCTION

Julia has been praised as solving the "two-language problem". This refers to the difficulty of finding a language that's fast, but still easy to read and write. Although it's true that Julia has some advantages relative to other languages, claims like this can be quite misleading for someone new to programming—it wrongly suggests that Julia is the only language you'll need to learn, regardless of your specific coding domain.

In reality, each programming language is designed with certain purposes in mind. Consequently, it's quite likely that you'll need to learn multiple programming languages, even if your focus is narrow. This is particularly true in data analysis, where a package implementing a specific task may only be available in one language. I, for one, tend to use Julia as my main language for data analysis, but complement it with libraries from R and Python when the task requires it. [1]

Getting the best performance in any language is also not immediate. It requires you to write code appropriately, with implementations that tend to be software-specific and involve several trade-offs. [2] Overall, the claim that "Julia is fast" should be replaced by "Julia *can* be fast." Considering this, the upcoming chapters aim to equip you with the essential tools to unlock Julia's performance capabilities.

WHEN SHOULD WE CARE ABOUT SPEED?

Achieving high performance often comes with trade-offs, and thus should never be the sole consideration when writing code. Optimizing performance frequently means rewriting parts of your script, which can reduce readability and make the code harder to maintain in the long run. Additionally, implementing these improvements requires significant time and effort, including tasks such as testing, identifying bottlenecks, and integrating third-party packages.

Considering this, you should assess your goals before embarking on any optimization efforts. Keep in mind that **most of YOUR time will be spent on writing, reading, and debugging code**—reducing the computer's execution time by a millisecond may not be worth the trade-off if it demands investing hours. Moreover, even if speed is crucial for your project, you should prioritize which parts of the code to optimize. Typically, only a few operations impact runtime critically, with the rest having a negligible effect.

With these caveats in mind, the suggestions we'll present in the upcoming chapters serve a dual purpose. Firstly, they represent essential rules for speed—not adhering to them would severely undermine performance, thereby negating any advantages of using Julia. Secondly, several tips we'll

consider have a minimal impact on code's readability, if any. In summary, the procedures to be presented will help you unlock Julia's speed, without sacrificing code readability or entailing excessive additional work.

FOOTNOTES

[1.] Julia has the capacity of calling programs from other software such as R or Python. Python and R also have this feature.

[2.] This explains the disparate results often seen in online benchmarks, where code can be written inefficiently in one language and highly optimized in another. Moreover, since languages tend to excel at certain tasks, it's possible to cherry-pick examples that make a particular language appear faster.

7c. Benchmarking Execution Time

Martin Alfaro
PhD in Economics

INTRODUCTION

This section introduces standard tools for benchmarking code performance. Our website reports results based on the `BenchmarkTools` package, which is currently the most mature and reliable option in the Julia ecosystem. That said, the newer `Chairmarks` package has demonstrated notable improvements in execution speed compared with `BenchmarkTools`. I recommend adopting `Chairmarks` once it's achieved sufficient stability and adoption within the community.

To set the stage, we'll start by addressing some key points for interpreting benchmark results. We'll also look at Julia's built-in `@time` macro, whose limitations explain why `BenchmarkTools` and `Chairmarks` should be used instead.

TIME METRICS

Julia uses the same time metrics described below, regardless of whether you use `BenchmarkTools` or `Chairmarks`. For quick reference, these metrics can be accessed at any point **in the left bar** under "**Notation & Hotkeys**".

Unit	Acronym	Measure in Seconds
Seconds	`s`	1
Milliseconds	`ms`	10^{-3}
Microseconds	`μs`	10^{-6}
Nanoseconds	`ns`	10^{-9}

Alongside execution times, each package also reports the amount of **memory allocated on the heap**, typically referred to simply as **allocations**. These allocations can play a major role in overall performance, and usually indicate suboptimal coding practices. As we'll explore in later sections, monitoring allocations tends to be crucial for achieving high performance.

"TIME TO FIRST PLOT"

The expression "Time to First Plot" refers to a side effect of how Julia operates, where the first execution in any new session takes longer than subsequent ones. This latency isn't a bug. Rather, it's a direct consequence of the language's design, which relies on a just-in-time (JIT) compiler: Julia compiles the code for executing functions in their first run, translating them into highly optimized machine code on the fly. This compilation process will be thoroughly covered in upcoming sections.

The first time you run any function, Julia generates low-level machine instructions to carry out the function's operations. This process of translating human-readable code into machine-executable instructions is called **compilation**. Unlike other programming languages, Julia relies on a just-in-time (JIT) compiler, where this code is compiled on-the-fly when a function is first run. This compilation process will be thoroughly covered in upcoming sections.

In each new session, this compilation penalty is incurred only once per function and set of argument types. Once a function is compiled, its machine code is cached, making all subsequent calls faster. The consequence is that the resulting overhead isn't a major hindrance for large projects, where startup costs are quickly amortized. However, it does mean that Julia may not be the best option for quick one-off analyses, such as running a simple regression or producing a quick exploratory plot.

The latency caused by this feature varies significantly across functions, making it difficult to generalize its impact. While it may be imperceptible for simple functions like `sum(x)`, it can be noticeable for rendering a high-quality plot. Indeed, drawing a first plot during a session can take several seconds, explaining the origin of the term "Time to First Plot".

> **Warning!**
>
> The Time-to-First-Plot issue has been significantly mitigated since `Julia 1.9`, thanks to improvements in precompilation. Each subsequent release is reducing this overhead even further.

@TIME

Julia comes with a built-in macro called `@time`, allowing you to get a quick sense of an operation's execution time. The results provided by this macro, nonetheless, suffer from several limitations that make it unsuitable for rigorous benchmarking.

First, a measurement based on just a single execution is often unreliable, as runtimes can fluctuate significantly due to background processes on your computer. Additionally, if that run is a function's first call, the measurement will include compilation overhead. The extra time Julia spends generating machine code inflates the reported runtime, making it unrepresentative of subsequent calls.

While running `@time` multiple times can address these issues, its most significant flaw arises when benchmarking functions. This is because `@time` mischaracterizes function arguments as global variables. We'll show in upcoming sections that global variables have a marked detrimental effect on performance. Consequently, the time reported doesn't accurately reflect how the function would perform in practice.

The following example illustrates the use of `@time`, highlighting the difference in execution time between the first and subsequent runs.

```
x = 1:100

@time sum(x)         # first run                      -> it incorporates compilation time
@time sum(x)         # time without compilation time -> relevant for each subsequent run
```

```
 0.002747 seconds (3.56 k allocations: 157.859 KiB, 99.36% compilation time)
 0.000003 seconds (1 allocation: 16 bytes)
```

PACKAGE "BENCHMARKTOOLS"

A more reliable alternative for measuring execution time is provided by `BenchmarkTools`, which addresses the shortcomings of `@time` in several ways.

First, it reduces result variability by running operations multiple times and then computing summary statistics. It also measures the execution time of functions correctly. To account for compilation latency, the package discards the first run, ensuring that overhead isn't included in the reported timing. Additionally, it's possible to handle function arguments correctly: by prefixing an argument with the `$` symbol, you can indicate that the variable shouldn't be treated as a global variable.

The package offers two macros, depending on the level of detail required: `@btime`, which only reports the minimum time, and `@benchmark`, which provides detailed statistics. Below, we demonstrate their use.

```
using BenchmarkTools

x = 1:100
@btime sum($x)        # provides minimum time only
```

```
 2.314 ns (0 allocations: 0 bytes)
```

```
using BenchmarkTools

x = 1:100
@benchmark sum($x)    # provides more statistics than `@btime`
```

In later sections, we'll exclusively benchmark functions. Therefore, you should always prefix each function argument with `$`. **Omitting `$` will lead to inaccurate results**, including incorrect reports of memory allocations.

The following example demonstrates the consequence of excluding `$`, where the runtimes reported are higher than the actual runtime.

```
using BenchmarkTools
x = rand(100)

@btime sum(x)
```

```
  14.465 ns (1 allocation: 16 bytes)
```

```
using BenchmarkTools
x = rand(100)

@btime sum($x)
```

```
  6.546 ns (0 allocations: 0 bytes)
```

PACKAGE "CHAIRMARKS"

A new alternative for benchmarking code is the `Chairmarks` package. Its notation closely resembles that of `BenchmarkTools`, with the macros `@b` and `@be` providing a similar functionality to `@btime` and `@benchmark` respectively. The main benefit of `Chairmarks` is its speed, as it can be orders of magnitude faster than `BenchmarkTools`.

As with `BenchmarkTools`, measuring the execution time of functions requires prepending function arguments with `$`.

```
using Chairmarks
x = rand(100)

display(@b sum($x))        # provides minimum time only
```

```
6.550 ns
```

```
using Chairmarks
x = rand(100)

display(@be sum($x))       # analogous to `@benchmark` in BenchmarkTools
```

```
Benchmark: 3856 samples with 3661 evaluations
 min    6.679 ns
 median 6.815 ns
 mean   6.785 ns
 max    14.539 ns
```

REMARK ON RANDOM NUMBERS FOR BENCHMARKING

When we seek to compare the performance of different methods for a given operation, we must ensure that our measurements aren't skewed by variations in the input data. One way to do this is by making sure that each approach is tested using *the exact same set of numbers*. This guarantees that any differences in execution time can be attributed solely to the efficiency of the method itself, rather than to a change in the inputs.

To achieve this, we can take advantage of random number generators that use a fixed "seed." A **random seed** is simply an initial value that determines the entire sequence of numbers that will be generated. By setting the same seed before each test, we can guarantee that the same deterministic sequence of random numbers is produced when code is run.

Importantly, **any arbitrary number can be used for the seed**. The only requirement is that the same number is utilized, so that you can replicate the exact same set of random numbers.

Random number generation is provided by the package `Random`. Below, we demonstrate its use by setting the seed `1234` before executing each operation, although any other number could have been used.

```
using Random

Random.seed!(1234)      # 1234 is an arbitrary number, use any number you want
x = rand(100)

Random.seed!(1234)
y = rand(100)           # identical to `x`
```

```
using Random

Random.seed!(1234)      # 1234 is an arbitrary number, use any number you want
x = rand(100)


y = rand(100)           # different from `x`
```

To maintain a clear presentation throughout this website, code snippets will omit the lines that set the random seed. While adding these code lines is essential for ensuring reproducibility, their inclusion in every example would create unnecessary clutter. Below, we illustrate the code that will be displayed throughout the website, along with the actual code executed.

```
using Random
Random.seed!(123)

x = rand(100)

y = sum(x)
```

```
# We omit the lines that seet the seed


x = rand(100)

y = sum(x)
```

BENCHMARKS IN PERSPECTIVE

When evaluating approaches for performing a task, execution times are often negligible, typically on the order of nanoseconds. However, this should not lead us to believe that the choice of method has no practical implications.

While operations in isolation may have an insignificant impact on a program's overall runtime, **the relevance of our benchmarks lies in scenarios where these operations are performed repeatedly**. This includes cases where the operation is called in a for-loop or in iterative procedures (e.g., solving systems of equations or the maximization of a function). In these situations, small differences in timing are amplified as they are replicated hundreds, thousands, or even millions of times.

AN EXAMPLE

To illustrate this matter, let's consider a concrete example. Suppose we want to double each element of a vector `x`, and then calculate their sum. In the following, we'll compare two different approaches to accomplish this task.

The first method will be based on `sum(2 .* x)` with `x` entering into the computation as a global variable. As we'll discuss in later sections, this approach is relatively inefficient. A more performance alternative is given by `sum(a -> 2 * a, x)` and passing `x` passed as a function argument. For the purpose of this comparison, you only need to know that both method produces the same result, with the first one being less performance. The runtime of each approach is as follows.

```
x      = rand(100_000)

foo()  = sum(2 .* x)
```

```
  35.519 µs (5 allocations: 781.37 KiB)
```

```
x      = rand(100_000)

foo(x) = sum(a -> 2 * a, x)
```

```
  6.393 µs (0 allocations: 0 bytes)
```

The results reveal that the second approach achieves a significant speedup, taking less than 15% of the slower approach. However, even the "slow" approach is extremely fast, taking less than 0.0001 seconds to execute.

This pattern will be a common theme in our benchmarks, where absolute execution times are often negligible. In such cases, the relevance of our conclusions depends on the context in which the operation is considered. If the operation is only performed once in isolation, readability should be the primary consideration for choosing a method. On the other hand, if the operation is repeated multiple times, small differences in performance might accumulate and become significant, making the faster approach a more suitable choice.

To illustrate this point, let's take the functions from the previous example and call them within a for-loop that runs 100,000 times. Since our sole goal is to repeat the operation, we don't need a meaningful iteration variable. This is a well-established programming convention for so-called **throwaway variables**: placeholders that exist only to satisfy the loop's syntax, without their value being used. It signals to other programmers that the variable can be safely ignored. In our example, `_` will simply reflect that each iteration is performing exactly the same operation.

```
x      = rand(100_000)
foo()  = sum(2 .* x)

function replicate()
   for _ in 1:100_000
      foo()
   end
end
```

```
 5.697 s (500000 allocations: 74.52 GiB)
```

```
x      = rand(100_000)
foo(x) = sum(a -> 2 * a, x)

function replicate(x)
   for _ in 1:100_000
      foo(x)
   end
end
```

```
 677.130 ms (0 allocations: 0 bytes)
```

The example starkly reveals the consequences of calling the function within a for-loop. The execution time of the slow version now jumps to more than 20 seconds, while the fast version finishes in under one second. The outcome highlights the importance of optimizing functions that will eventually be executed repeatedly, as even minor improvements can yield a significant impact on overall performance.

7d. Preliminaries on Types

Martin Alfaro
PhD in Economics

INTRODUCTION

High performance in Julia depends critically on the notion of type stability. The definition of this concept is relatively straightforward: a function is type-stable when the types of its expressions can be inferred from the types of its arguments. When the property holds, Julia can specialize its computation method, resulting in significant performance gains.

Despite its simplicity, type stability is subject to various nuances. In fact, a careful consideration of the property requires a solid foundation in two key areas: **Julia's type system and the inner workings of functions**. The current section equips you with the necessary knowledge to grasp the former, deferring the internals of functions to the next section. **The explanations will focus on the case of scalars and vectors**, leaving more complex objects for subsequent sections.

Before you continue, I recommend reviewing the basics of types introduced here.

> **Warning!**
>
> **The subject is covered only to the extent necessary for understanding type stability**. Julia's type system is indeed quite vast, and a comprehensive exploration would warrant a dedicated chapter.

BASICS OF TYPES

Variables in Julia are mere labels for objects, where objects in turn hold values with certain types. The most common types for scalars are `Float64` and `Int64`, whose vector counterparts are `Vector{Float64}` and `Vector{Int64}`. Recall that `Vector` is an alias for a one-dimensional array, so that a type like `Vector{Float64}` is equivalent to `Array{Float,1}`.

> **`Int` As an Alternative to `Int64`**
>
> You'll notice that packages tend to use `Int` as the default type for integers. The type `Int` is an alias that adapts to your CPU's architecture. Since most modern computers are 64-bit systems, `Int` is equivalent to `Int64`. Nonetheless, `Int` becomes `Int32` on 32-bit systems.

Julia's type system is organized in a hierarchical way. This feature allows for the definition of subsets and supersets of types, which in the context of types are referred to as **subtypes** and **supertypes**. [1] For instance, the type `Any` is a supertype that includes all possible types in Julia, thus occupying the highest position in any type hierarchy. Another example of supertype is `Number`, which encompasses all numeric types (`Float64`, `Float32`, `Int64`, etc.).

Supertypes provide great flexibility for writing code. They enable the grouping of values to define operations in common. For instance, defining `+` for the abstract type `Number` ensures its applicability to all numeric types, regardless of whether they are integers, floats, or their numerical precision.

A special supertype known as `Union` will be instrumental for our examples. This construction is useful for variables that can potentially hold values with different types. They're denoted by `Union{<type1>, <type2>, ...}`, so that a variable with type `Union{Int64, Float64}` could be either an `Int64` or `Float64`. Note that, by definition, union types are always supertypes of their arguments.

> **Union of Types to Account for Missing Values**
>
> Unions of types emerge naturally in data analysis workflows, especially when handling missing observations. In Julia, these values are represented by the type `Missing`. For instance, if we load a column that contains both integers and empty entries, this is usually stored with type `Vector{Union{Missing,Int64}}`.

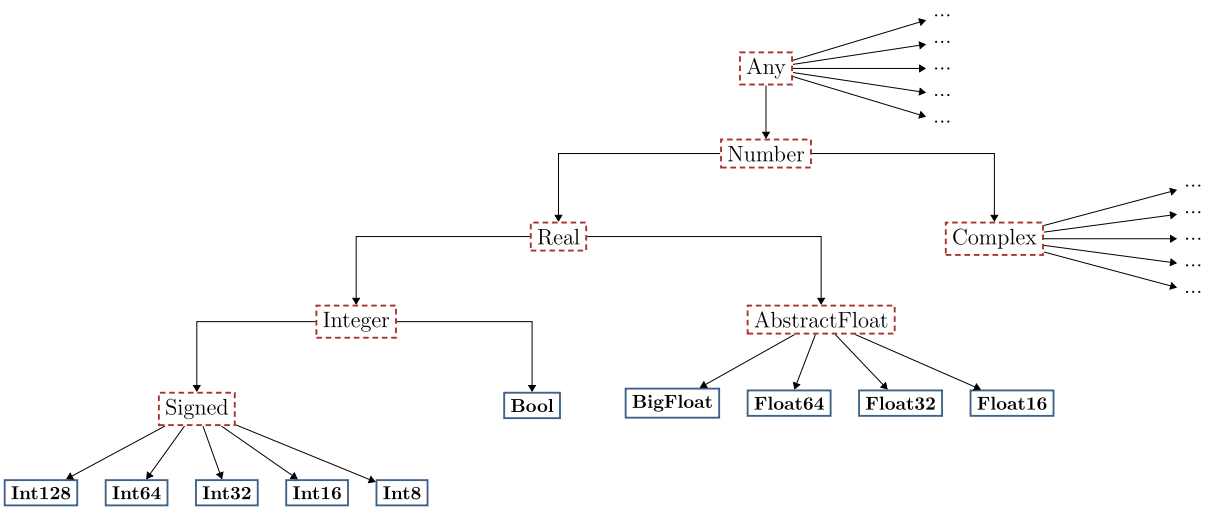
ABSTRACT AND CONCRETE TYPES

The hierarchical nature of types makes it possible to represent subtypes and supertypes as trees. Th structure gives rise to the notions of abstract and concrete types.

An **abstract type** acts as a parent category, necessarily breaking down into subtypes. The type `Any` i Julia is a prime example. In contrast, a **concrete type** represents an irreducible unit that therefor lacks subtypes. Concrete types are considered final, in the sense that they can't be further specialize within the hierarchy.

The diagram below illustrates the difference between abstract and concrete types for scalars. This done by presenting the hierarchy of the type `Number`, where the labels included match th corresponding type name in Julia. [2]

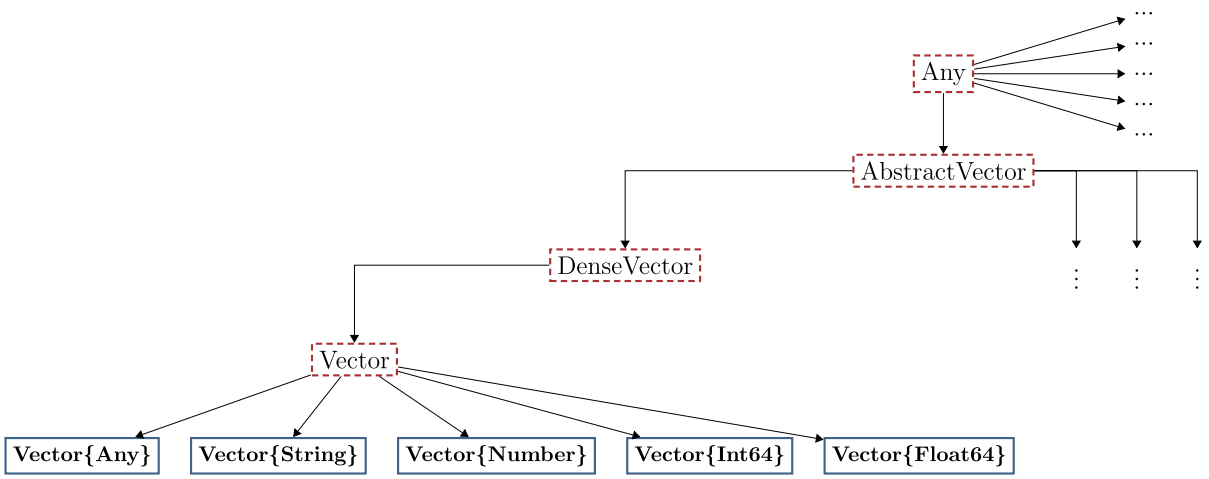
HIERARCHY OF TYPE NUMBER



Note: Dashed red borders indicate abstract types, while solid blue borders indicate concrete types.

The distinction between abstract and concrete types for scalars is relatively straightforward. Instead the same distinction becomes more nuanced when vectors are considered, as shown in the diagram below.

HIERARCHY OF TYPE VECTOR



Note: Dashed red borders indicate abstract types, while solid blue borders indicate concrete types.

The tree reveals that `Vector{T}` **for a given type** `T` **is a concrete type**. By definition, this mean variables can be instances of `Vector{T}` and `Vector{T}` can't have subtypes. The latter in particula implies that a vector like `Vector{Int64}` isn't a subtype of `Vector{Any}`, even though `Int64` is subtype of `Any`. This behavior stands in stark contrast to scalars, where `Any` is an abstract typ However, it aligns perfectly with the concept of vectors as collections of *homogeneous element* meaning they all share the same type.

ONLY CONCRETE TYPES CAN BE INSTANTIATED, ABSTRACT TYPES CAN'T

In Julia, **instantiation** refers to the process of creating an object with a specific type. A key principle o Julia's type system is that only concrete types can be instantiated, implying that values can never b represented by abstract types. This distinction helps clarify the meaning of some widesprea expressions used in Julia. For example, stating that a variable has type `Any` shouldn't be interprete literally. Rather, it means the variable can hold values of any concrete type, since all concrete types i Julia are subtypes of `Any`.

This distinction will become crucial for what follows, particularly for type-annotating variables. implies that declaring a variable with an abstract type restricts the set of possible concrete types it ca hold, even though the variable will ultimately adopt a concrete type.

RELEVANCE FOR TYPE STABILITY

At this point, you may be wondering how all these concepts relate to type stability. The connectio becomes clear when you consider how Julia performs computations.

High performance in Julia relies heavily on specializing the computation method. We'll see that th specialization is unattainable in the global scope, as Julia treats global variables as potentially holdin values of any type. In contrast, when code is wrapped in a function, the execution process begins b determining the concrete types of each function argument. This information is then used to infer th concrete types of all the expressions within the function body.

When this inference succeeds, meaning all expressions have unambiguous concrete types, th function is considered **type stable**. TType stability enables Julia to specialize its computation metho and generate optimized machine code. If, instead, expressions could potentially take on multipl concrete types, performance is substantially degraded, as Julia must consider a separat implementation for each possible type.

For scalars and vectors, type stability essentially requires that expressions ultimately operate o **primitive types**. Examples of numeric primitive types include integers and floating-point number such as `Int64`, `Float64`, and `Bool`. Thus, applying functions like `sum` to a `Vector{Int64}` o `Vector{Float64}` allows for full specialization, whereas applying them to a `Vector{Any}` prevents it.

> **String Objects**
>
> For text representation, the character type `Char` serves as the primitive type. Since a `String` is internally represented as a collection of `Char` elements, operations on `String` objects can also achieve type stability.

THE OPERATOR <: TO IDENTIFY SUPERTYPES

The rest of this section is dedicated to operators and functions for working with types. Specifically, we'll introduce the operator `<:`, which checks whether a given type is a subtype of another, and then explore ways to constrain a variable to certain types.

It's possible that you won't need to apply any of the techniques we present, as Julia automatically attempts to infer types when functions are called. Nonetheless, understanding these operators is essential for grasping upcoming material.

USE OF `<:`

The symbol `:<` tests whether a type `T` is a subtype of another type `S`. It can be used as an operator `T <: S` or as a function `<:(T,S)`. For example, `Int64 <: Number` and `<:(Int64, Number)` verify whether `Int64` is a subtype of `Number`, which would return `true`. Below, we provide further examples.

```
# all the statements below are `true`
Float64 <: Any
Int64   <: Number
Int64   <: Int64
```

```
# all the statements below are `false`
Float64 <: Vector{Any}
Int64   <: Vector{Number}
Int64   <: Vector{Int64}
```

The fact that `Int64 <: Int64` evaluates to `true` illustrates a fundamental principle: **every type is a subtype of itself**. Moreover, in the case of concrete types, this is the only subtype.

THE KEYWORD `WHERE`

By combining `<:` with `Union`, you can also check whether a type belongs to a given set of types. For example, `Int64 <: Union{Int64, Float64}` assesses whether `Int64` equals `Int64` or `Float64`, thus returning `true`.

The approach can be made more widely applicable by using the `where` keyword with a type parameter `T`. [3]. The syntax is `<type depending on T> where T <: <set of types>`. In this way, `T` represents multiple possibilities.

```
# all the statements below are `true`
Float64 <: Any
Int64   <: Union{Int64, Float64}
Int64   <: Union{T, String} where T <: Number       # `String` represents text
```

```
# all the statements below are `true`
Vector{Float64} <: Vector{T} where T <: Any
Vector{Int64}   <: Vector{T} where T <: Union{Int64, Float64}
Vector{Number}  <: Vector{T} where T <: Any

# all the statements below are `false`
Vector{Float64} <: Vector{Any}
Vector{Int64}   <: Vector{Union{Int64, Float64}}
Vector{Number}  <: Vector{Any}
```

```
# all the statements below are `true`
Vector{Float64} <: Vector{<:Any}
Vector{Int64}   <: Vector{<:Union{Int64, Float64}}
Vector{Number}  <: Vector{<:Any}

# all the statements below are `false`
Vector{Float64} <: Vector{Any}
Vector{Int64}   <: Vector{Union{Int64, Float64}}
Vector{Number}  <: Vector{Any}
```

Types that add parameters like `T` are called **parametric types**. In the example above, they allow us to distinguish between a concrete type like `Vector{Any}` and a set of concrete types `Vector{T} where T <: Any`, where the latter encompasses `Vector{Int64}`, `Vector{Float64}`, `Vector{String}`, etc.

> **Warning!** - The Type `Any`
> When we omit `<:` and simply write `where T`, Julia implicitly interprets the statement as `where T <: Any`. This is why we can establish the following equivalences.

```
# all the statements below are `true`
Float64       <: Any
Float64        <: T where T <: Any              # identical to
the line above
Vector{Int64} <: Vector{T} where T <: Any
```

```
# all the statements below are `true`
Float64       <: Any
Float64        <: T where T                     # identical to
the line above
Vector{Int64} <: Vector{T} where T
```

TYPE-ANNOTATING VARIABLES

In the following, we present methods for **type-annotating variables**. The techniques introduced can be used either to assert a variable's type **during an assignment** or to restrict the types of **function arguments**.

Specifically, there are two approaches to type-annotating variables. The first one relies on the binary operator `::`, and its syntax is `x::<type>`. The second approach leverages the Boolean binary operator `<:`, combined with `::` and the keyword `where`. Its syntax is `x::T where T <: <type>` (note that `T` accepts any other letter).

Next, we illustrate both methods, considering type-annotations for assignments and for function arguments separately.

ASSIGNMENTS

Let's start illustrating the approaches for scalar assignments. Each tab below declares an identical type for `x` and for `y`.

```
x::Int64                = 2      # only reassignments to `Int64` are possible

y::Number               = 2      # only reassignments to `Float64`, `Float32`, `Int64`, etc
are possible
```

```
julia> x = 2.5
ERROR: InexactError: Int64(2.5)
julia> y = 2.5
2.5
julia> y = "hello"
ERROR: MethodError: Cannot convert an object of type String to an object of type Number
```

```
x::T where T <: Int64   = 2      # only reassignments to `Int64` are possible

y::T where T <: Number  = 2      # only reassignments to `Float64`, `Float32`, `Int64`, etc
are possible
```

```
julia> x = 2.5
ERROR: InexactError: Int64(2.5)
julia> y = 2.5
2.5
julia> y = "hello"
ERROR: MethodError: Cannot convert an object of type String to an object of type Number
```

> **Warning!** - Modifying Types
>
> Once you assert a type for `x` in an assignment, you can't modify `x`'s type afterwards. The only way to fix this is by starting a new Julia session.

The fact that `x` retains the same type across all tabs follows because `T <: Float64` can only represent `Float64`. This fact arises because `Float64` is a concrete type, which has no subtypes other than itself by definition. Considering this, scalar types are usually asserted using `::` rather than `<:`.

On the contrary, the implications when `::` or `<:` is chosen differs for vectors. Specifically, using `::` in combination with `Vector{Number}` establishes that `Vector{Number}` is the only possible concrete type. Instead, `Vector{T} where T <: Number` indicates that the elements of the vector will adopt a concrete type that's a subtype of `Number`, rather than the object adopting `Vector{Number}`.

```
# `x` will always be `Vector{Any}`
x::Vector{Any}                = [1,2,3]

# `y` will always be `Vector{Number}`
y::Vector{Number}             = [1,2,3]
```

```
julia> typeof(x)
Vector{Any} (alias for Array{Any, 1})
julia> typeof(y)
Vector{Number} (alias for Array{Number, 1})
```

```
# `x` is Vector{Int64} and could eventually become `Vector{Float64}`, `Vector{String}`, etc
x::Vector{T} where T <:Any    = [1,2,3]

# `x` is Vector{Int64} and could eventually become `Vector{Float64}`, `Vector{Int32}`, etc
y::Vector{T} where T <: Number = [1,2,3]
```

```
julia> typeof(x)
Vector{Int64} (alias for Array{Int64, 1})
julia> typeof(y)
Vector{Int64} (alias for Array{Int64, 1})
```

The principles outlined apply even when a variable isn't explicitly type-annotated. The reason is that **an assignment without `::` implicitly assigns the type `Any` to the variable**, where `Any` is the supertype encompassing all possible types. For example, the statements `x = 2` and `x::Any = 2` are equivalent.

The same occurs when omitting `<:` from the expression `where T`, which implicitly takes `T <: Any`. Thus, for instance, `x = 2` is equivalent to `x::T where T = 2` or `x::T where T <: Any = 2`. Considering this, all the variables below have their types restricted in the same way.

```
# all are equivalent
a      = 2
b::Any = 2
```

```
# all are equivalent
a                  = 2
b::T where T       = 2
c::T where T <: Any = 2
```

The default restriction of variables to the type `Any` is the reason why we can reassign variables with any value. For instance, given `a = 1`, executing `a = "hello"` afterwards is valid, since `a` is implicitly type-annotated with `Any`.

> **Warning!** - One-liner Statements Using `where`
>
> Be careful with one-liner statements using `where`, especially when `where T` is shorthand for `where T <: Any`. These concise statements can easily lead to confusion, as demonstrated below.

```
a::T where T = 2                    # this is not `T = 2`, it's
`a = 2`

a::T where {T}          = 2           # slightly less confusing
notation
a::T where {T <: Any} = 2             # slightly less confusing
notation
```

```
foo(x::T) where T = 2               # this is not `T = 2`, it's
`foo(x) = 2`

foo(x::T) where {T}          = 2     # slightly less confusing
notation
foo(x::T) where {T <: Any} = 2      # slightly less confusing
notation
```

FUNCTIONS

Function arguments can also be type-annotated. The examples below illustrate this by restricting the function to accept integer inputs exclusively.

```
function foo1(x::Int64, y::Int64)
    x + y
end
```

```
julia> foo1(1, 2)
3
julia> foo1(1.5, 2)
ERROR: MethodError: no method matching foo1(::Float64, ::Int64)
```

```
function foo2(x::Vector{T}, y::Vector{T}) where T <: Int64
    x .+ y
end
```

```
julia> foo2([1,2], [3,4])
2-element Vector{Int64}:
 4
 6
julia> foo2([1,2], [3.0, 4.0])
ERROR: MethodError: no method matching foo2(::Vector{Int64}, ::Vector{Float64})
```

Note that type-annotating both arguments with the same parameter `T` forces them to have exactly the same type. Also notice that types like `Int64` preclude the use of `Float64`, even for numbers like `3.0`. If you need greater flexibility, you should introduce different type parameters and annotate them with an abstract type like `Number`.

```
function foo2(x::T, y::T) where T <: Number
    x + y
end
```

```
julia> foo2(1.5, 2.0)
3.5
julia> foo2(1.5, 2)
ERROR: MethodError: no method matching foo2(::Float64, ::Int64)
```

```
function foo3(x::T, y::S) where {T <: Number, S <: Number}
    x + y
end
```

```
julia> foo3(1.5, 2.0)
3.5
julia> foo3(1.5, 2)
3.5
```

The greatest flexibility is achieved when we don't type-annotate function arguments at all, as they will implicitly default to `Any`. This can be observed below, where all tabs define identical functions. Ultimately, type-annotating function arguments is only needed to prevent invalid usage (e.g., to ensure that `log` isn't applied to a negative value).

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

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

```
function foo(x::T, y::S) where {T <: Any, S <: Any}
    x + y
end
```

```
function foo(x::T, y::S) where {T, S}
    x + y
end
```

CREATING VARIABLES WITH SOME TYPE

To conclude this section, we present an approach to defining variables with a given type. The approach relies on the so-called **constructors**, which are functions that create new instances of a concrete type. They're useful for transforming a variable `x` into another type.

Constructors are implemented by functions of the form `Type(x)`, where `Type` should be replaced with the literal name of the type (e.g., `Vector{Float64}`). Just like any other function, `Type` supports broadcasting.

```
x = 1

y = Float64(x)
z = Bool(x)
```

```
julia> y
1.0
julia> z
true
```

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

y = Vector{Any}(x)
```

```
julia> y
3-element Vector{Any}:
 1
 2
 3
```

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

y = Float64.(x)
```

```
julia> y
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

Remark

Parametric types can be used as constructors. Moreover, although abstract types can't be instantiated, they may still serve as constructors. In such cases, Julia will attempt to convert the object to a specific concrete type, although not all abstract types can be used for this purpose.

```
x = 1

y = Number(x)
```

```
julia> typeof(y)
Int64
```

```
x = [1, 2]

y = (Vector{T} where T)(x)
```

```
julia> typeof(y)
Vector{Int64}
```

```
x = 1

z = Any(x)
```

```
ERROR: MethodError: no constructors have been defined for Any
```

There's an alternative way to transform `x`'s type into `T`, as long as the conversion is feasible. This is given by the function `convert(T,x)`.

```
x = 1

y = convert(Float64, x)
z = convert(Bool, x)
```

```
julia> y
1.0
julia> z
true
```

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

y = convert(Vector{Any}, x)
```

```
julia> y
3-element Vector{Any}:
 1
 2
 3
```

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

y = convert.(Float64, x)
```

```
julia> y
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

FOOTNOTES

[1.] Types don't necessarily follow a subtype-supertype hierarchy. For example, `Float64` and `Vector{String}` exist independently, without a hierarchical relationship. This fact will become clearer when the concepts of abstract and concrete types are defined.

[2.] The `Signed` subtype of `Integers` allows for the representation of negative and positive integers. Julia also offers the type `Unsigned`, which only accepts positive integers and comprises subtypes such as `UInt64` and `UInt32`.

[3.] `T` can be replaced by any other letter

7e. Functions: Type Inference and Multiple Dispatch

Martin Alfaro
PhD in Economics

INTRODUCTION

In Julia, functions are key for achieving high performance. This is by design, as functions have been engineered from the outset to generate efficient machine code.

However, to fully unlock their potential, we must first understand the underlying process of function calls. Essentially, when a function is called, Julia attempts to identify concrete types for its variables, then selecting the most suitable method for the function. At the heart of the process are three interconnected mechanisms: **dispatch**, **compilation**, and **type inference**. This section will provide a detailed explanation of each concept.

FUNCTIONS AND GLOBAL VARIABLES

To fully grasp how functions enhance performance in Julia, it's essential to examine their relationship with variable scope. **Local variables** encompass all variables defined within a function's scope, including function arguments and variables declared inside the function body. These variables exist only during the function's execution and are inaccessible from outside the function. **Global variables**, on the other hand, refer to any variable defined outside a function's scope and and remain accessible throughout the program's execution.

One of the main takeaway of this section is that **wrapping operations within functions is crucial for achieving high performance in Julia**. Instead, working in the global scope, or more generally relying on global variables, precludes any performance implementation.

The low performance of global variables arises because Julia treats them as potentially embodying any value and therefore any type. This decision was adopted under the logic that, even if a variable holds some value at a specific moment, the user may reassign it at any part of the program.

The detrimental effect of global variables isn't confined to operations in the global scope. It also arises when a function references external variables that haven't been passed as arguments. Considering this, our conclusions apply to all the following cases.

GLOBAL VARIABLE IN GLOBAL SCOPE

```
x     = 2

y     = 3 * x
```

```
julia> y
6
```

GLOBAL VARIABLE IN FUNCTION

```
x     = 2

foo() = 3 * x
```

```
julia> foo()
6
```

Recall that an expression like `x = 2` is shorthand for `x::Any = 2`, reflecting that global variables default to `Any` unless they're explicitly type-annotated. Also remember that only concrete types can be instantiated, meaning that values can only adopt a concrete type. Consequently, `x::Any` shouldn't be understood as `x` having type `Any`, but rather that `x` can take on any concrete type that is a subtype of `Any`. Since `Any` is at the top of Julia's type hierarchy, this simply means that `x`'s types are unrestricted.

Working with a variable like `x` that has type `Any`, prevents specialization of `*`. The reason is that Julia must consider multiple possible methods for its computation, one for each possible concrete type of `x`. In practice, this results in Julia generating code with multiple branches, potentially involving type checks, conversions, and object creations. The consequence is degraded performance.

Even if we had type-annotated `x` with a concrete type like `x::Int64 = 2`, the performance limitations wouldn't completely go away. This is because certain optimizations can only be implemented when both the scope of variables is clearly delimited and their values are known. When both aspects are known, Julia can gain a comprehensive view of all the operations to be performed, creating opportunities for further optimizations.

Functions were designed in Julia to address all these considerations. This is accomplished through a series of steps that functions follow, which we cover next.

FUNCTIONS AND METHODS

A **function** is just a name that groups an arbitrary number of methods. Each **method**, in turn, defines a specific function body for a given combination of number and types of arguments. The list of methods associated with a function `foo` can be retrieved by running the command `methods(foo)`.

To illustrate these concepts, let's define several methods for some function named `foo`. To keep matters simple, let's start considering a scenario where all the methods in `foo` have the same number of arguments. Creating methods requires type-annotating `foo`'s arguments with the

operator `::` during its definition. Then, we can provide a distinct body function for each unique combination of these types.

METHODS

```
foo(a,b)                    = a + b
foo(a::String, b::String) = "This is $a and this is $b"
```

```
julia> methods(foo)
2 methods for generic function "foo" from Main
julia> foo(1,2)
3
julia> foo("some text", "more text")
"This is some text and this is more text"
```

Since `foo(a,b)` is equivalent to `foo(a::Any,b::Any)`, the first method sets the behavior of `foo` for any combination of input types. However, such behavior is overridden by the method `foo(a::String, b::String)`, which provides an alternative function body for `a` and `b` with type `String`. The existence of multiple methods explains the different outputs obtained: the first method of `foo` is called with `foo(1, 2)`, whereas `foo("some text", "more text")` triggers the second method.

The example also reveals that **methods don't need to comprise similar operations**. Although mixing drastically different operations under a single function name isn't recommended, allowing function bodies to differ by method creates opportunities for optimizations. In particular, it allows for computation algorithms tailored to each specific type combination, thus optimizing the overall performance of a function.

Additionally, note that **methods don't need to have the same number of arguments**. For instance, it's possible to define all the following methods for a function `bar`.

METHODS WITH DIFFERENT NUMBERS OF ARGUMENTS

```
bar(x)       = x
bar(x, y)    = x + y
bar(x, y, z) = x + y + z
```

```
julia> methods(bar)
3 methods for generic function "bar" from Main
julia> bar(1)
1
julia> bar(1, 2)
3
julia> bar(1, 2, 3)
6
```

This feature is particularly useful for extending a function's behavior. A prime application is given by the function `sum`. So far, we've only used its simplest form `sum(x)`, which adds all the elements of a collection `x`. However, `sum` also supports additional methods. One of them is `sum(<function>, x)`,

where the elements of `x` are first transformed via `<function>` before being summed.

METHODS FOR 'SUM'

```
x = [2, 3, 4]

y = sum(x)          # 2 + 3 + 4
z = sum(log, x)     # log(2) + log(3) + log(4)
```

FUNCTION CALL

Given a function and its methods, we can now analyze the process triggered when a function is called. In the following, we'll base our explanations on the following function `foo`:

EXAMPLE FUNCTION 'FOO'

```
foo(a, b) = 2 + a * b
```

```
julia> foo(1, 2)
4
julia> foo(3, 2)
8
julia> foo(3.0, 2)
8.0
```

Recall that variables with no type annotation default to `Any`. This implies that the function body `foo(a,b)` holds for any combination of types of `a` and `b`.

MULTIPLE DISPATCH

When `foo(1, 2)` is called, Julia is instructed to evaluate the expression `2 + a * b`. This process relies on a mechanism known as **multiple dispatch**, where Julia decides on the computational approach to be implemented. Importantly, this decision is based on solely on the types of the arguments, not their values.

Multiple dispatch proceeds in several steps. First, the compiler determines the concrete types of the function arguments. In our example, since `a = 1`, and `b = 2`, both are identified as `Int64`.

After this, the information on types is used to select a *method*, which defines the function body and hence the operations to be performed. This process involves searching through all available methods of `foo` until a method that matches the concrete types of `a` and `b` is identified. In our example, `foo` has only one method `foo(a,b) = 2 + a * b`, which applies to all type combinations of `a` and `b`. Consequently, the relevant function body is `2 + a * b`.

The operations to be performed are then forwarded to the **compiler**, which is in charge of the implementation. This involves choosing a **method instance**, which refers to the specific code implementation that will be used to compute the operations defined by the method.

If a method instance already exists for the function signature `foo(a::Int64, b::Int64)`, Julia will directly employ it to compute `foo(1,2)`. Otherwise, a new method instance is generated and stored (cached) in memory.

The following diagram depicts all the process unfolded when `foo(1,2)` is executed.

MULTIPLE DISPATCH

The process determines that the first time you call a function with particular argument types, there's an initial compilation overhead. This phenomenon is referred to in Julia as Time To First Plot. Instead, subsequent calls with the same argument types can reuse this cached code, resulting in faster execution.

To illustrate this mechanism, note that the call `foo(3, 2)` incorporated in the example occurs after `foo(1,2)`. This allows Julia to compute `foo(3, 2)` by directly invoking the method instance `foo(a::Int64, b::Int64)`, without the need of compiling code. Instead, executing a function call like `foo(3.0, 2)` requires the compilation of a new method instance `foo(a::Float64, b::Int64)`, since the types of `3` and `3.0` differ.

TYPE INFERENCE

During a function call, two different stages take place: compilation time and runtime. **Compilation time** consists of the steps just described, during which Julia generates machine code to execute the function's operations. Importantly, this stage involves no computations, and is triggered only when the function is called for the first time with new concrete types.

In contrast, **runtime** is the stage during which code instructions are actually executed. It takes place *after* compilation and every time a function is called.

Most considerations for achieving high performance are related to the compilation process. In particular, Julia employs **Just-In-Time Compilation (JIT)**, a term reflecting that compilation happens on the fly during the function call.

The quality of code generated during JIT critically determines performance. A key mechanism in this process is **type inference**, whereby the compiler attempts to identify concrete types for *all* variables and expressions.

If the compiler succeeds in identifying concrete types, it can specialize instructions for each operation and yield fast code. This is the essence of **type stability**, which we'll cover extensively in the next chapter. For instance, type inference in our example involves determining concrete types for `2`, `a = 1`, and `b = 2`. Since all of them have type `Int64`, the compiler can specialize the computation of `2 + a * b` for variables with type `Int64`.

On the contrary, if the compiler is unable to identify concrete types for some expressions, the compiler must generate generic code that accommodates multiple type possibilities. This forces Julia to defer decisions on methods to runtime, significantly degrading performance.

REMARKS ON TYPE INFERENCE

Below, we provide various remarks about type inference that are worth keeping in mind.

FUNCTIONS DO NOT GUARANTEE CONCRETE TYPES

Notice that merely wrapping operations in a function doesn't guarantee that the compiler will identify concrete types. The following example presents a function call that's unable to do so.

TYPE-UNSTABLE FUNCTION

```
x      = [1, 2, "hello"]    # Vector{Any}

foo(x) = x[1] + x[2]        # type unstable
```

```
julia> foo(x)
3
```

In the example provided, the issue arises because the compiler assigns the type `Any` to `x[1]` and `x[2]`, since they correspond to elements from an object with type `Vector{Any}`. Consequently, the compiler can't specialize the computation of the operation `+`. The example also highlights that compilation is exclusively based on types, not values. Thus, the code is generated ignoring that actually `x[1] = 1` and `x[2] = 2`, and so are `Int64`.

GLOBAL VARIABLES INHERIT THEIR GLOBAL TYPE

Julia's attempt to identify concrete types is restricted to local variables. Instead, any global variable will have its type inherited from the global scope. For instance, consider the following example.

GLOBAL VARIABLE

```
a      = 2
b      = 1

foo(a) = a * b
```

```
julia> foo(a)
2
```

TYPE-ANNOTATED GLOBAL VARIABLE

```
a      = 2
b::Number = 1

foo(a) = a * b
```

```
julia> foo(a)
2
```

In both examples `b` is a global variable. Consequently, `b`'s type in the first tab is inferred to `Any`, while in the second tab to `Number`.

TYPE-ANNOTATING FUNCTION ARGUMENTS DOES NOT IMPROVE PERFORMANCE

Identifying concrete types is key for achieving performance. This might lead you to believe that type-annotating function arguments is essential for performance, or that at least could provide a boost. However, type-annotating arguments is actually redundant due to type inference. In fact, engaging in this practice will unnecessarily constrain the types accepted by the function, reducing the range of potential applications. To better appreciate this loss of flexibility, compare the following scripts.

TYPE-ANNOTATED FUNCTION

```
foo1(a::Float64, b::Float64) = a * b
```

```
julia> foo1(0.5, 2.0)
1.0
julia> foo1(1, 2)
ERROR: MethodError: no method matching foo1(::Int64, ::Int64)
```

UNANNOTATED FUNCTION

```
foo2(a, b) = a * b
```

```
julia> foo2(0.5, 2.0)
1.0
julia> foo2(1, 2)
2
```

The function on the first tab only accepts arguments with type `Float64`. Note that even integer variables are disallowed, as function arguments aren't converted to a common type. On the contrary, the function's second tab entails the same process for `Float64` inputs, but additionally allows for other types. The flexibility stems from the implicit type-annotation `Any` for the function arguments.

> **Packages Commonly Type-Annotate Function Arguments**
>
> When inspect the code of packages, you may notice that function arguments are often type-annotated. The reason for this isn't related to performance, but rather to ensure the function's intended usage, safeguarding against inadvertent type mismatches.
>
> For instance, suppose a function to the revenue of a theater via `nr_tickets * price`. Importantly, the operator `*` in Julia not only implements the product of numbers, but also concatenates words when applied to expressions with type `String`. This opens up the possibility of misusing the function if it's not type-annotated. The first tab demonstrates a potential misuse of this function, with the second tab addressing this possibility by asserting types.

UNANNOTATED FUNCTION

```
revenue1(nr_tickets, price) = nr_tickets * price
```

```
julia> revenue1(3, 2)
6
julia> revenue1("this is ", "allowed")
"this is allowed"
```

TYPE-ANNOTATED FUNCTION

```
revenue2(nr_tickets::Int64, price::Number) = nr_tickets * price
```

```
julia> revenue2(3, 2)
6
julia> revenue2("this is ", "not allowed")
ERROR: MethodError: no method matching revenue2(::String,
::String)
```

8a. Overview and Goals

Martin Alfaro
PhD in Economics

In the upcoming chapters, we'll focus on two essential aspects for performance: type stability and reductions in memory allocation. These core principles represent the most basic procedures to achieve high performance, thus acting as the starting point for further optimizations.

This chapter in particular focuses on type stability, whose importance for Julia can't be overstated —**any attempt to generate fast code without ensuring type stability is destined to fail**.

At its core, type stability is rooted in how computers execute operations at a fundamental level. Specifically, regardless of the programming language used, the approach to computing operations differs depending on the inputs' types. This means, for instance, that the internal process for integer operations differs from computations based on floating-point numbers.

The consequence of this feature for performance is that speed demands the identification of concrete types for each variable. With this information available, the computation method can be specialized. Instead, if concrete types can't be identified, the code generated must accommodate multiple potential approaches, one for each possible combination of input types. This introduces additional runtime checks and type conversions, significantly degrading execution speed.

The discussion of type stability will be intertwined with functions, as *type stability requires wrapping code in function as a prerequisite*. The reason for this is that Julia only attempts to infer the types of variables within a function. Wrapping code in a function is only a necessary condition for type stability, and the chapter will provide additional conditions to guarantee the property.

8b. Defining Type Stability

Martin Alfaro
PhD in Economics

INTRODUCTION

This section formally defines type stability and reviews the tools employed for its verification. In the next section, we'll begin examining how type stability applies in specific scenarios.

AN INTUITION

In a previous section, we described the process that unfolds when a function is called. To briefly review, let's consider a function `foo(x) = x + 2` and executing `foo(a)` for some variable `a`. We assume that `a` has a specific value assigned and therefore a concrete type, although we omit explicitly stating values for `a`. In this way, we highlight that the process depends on types, rather than values.

Calling `foo(a)` prompts Julia to identify the concrete type of `a`, which we'll denote as `T`. If a compiled method instance for `foo` with an argument of type `T` already exists, then `foo(a)` is executed immediately. Otherwise, Julia compiles a method instance for evaluating `a + 2`. This code generation leverages type inference, wherein the compiler attempts to deduce concrete types for all involved terms. The resulting machine code is then stored (cached), making it readily available for subsequent calls of `foo(b)` when `b` has type `T`.

TYPE STABILITY AND PERFORMANCE

The key to generating fast code lies in the information available to the compiler during the compilation stage. This information is primarily gathered through type inference, where Julia identifies the specific type of each variable and expression involved. When the compiler can **accurately predict a single concrete type for the function's output**, the function call is said to be **type stable**.

While this constitutes the formal definition of type stability, a more stringent definition is usually applied in practice: the compiler must be able to **infer unique concrete types for *each* expression within the function**, not only for the final output. This definition aligns with `@code_warntype`, the built-in macro to detect type instabilities.

If the condition is satisfied, the compiler can specialize the computational approach for each operation, resulting in fast execution. Essentially, type stability dictates that there's sufficient information to determine a straight execution path, thus avoiding unnecessary type checks and dispatches at runtime.

In contrast, type-unstable functions generate generic code that accommodates each possible combination of unique concrete types. This results in additional overhead during runtime, where Julia is forced to dynamically gather type information and perform extra calculations based on it. The consequence is a pronounced deterioration in performance.

> **Type Stability Characterizes Function *Calls***
>
> It's common to describe a function as "type stable". Nevertheless, it's not the function itself that's type stable, but rather the function calls for specific concrete types of its arguments. The distinction is crucial in practice, since a function may exhibit type stability for certain input types but not others.

AN EXAMPLE

To see type stability in practice, let's consider the following example.

```
x = [1, 2, 3]                   # `x` has type `Vector{Int64}`

@btime sum($x[1:2])             # type stable
```

```
  22.406 ns (1 allocation: 80 bytes)
```

```
x = [1, 2, "hello"]             # `x` has type `Vector{Any}`

@btime sum($x[1:2])             # type UNSTABLE
```

```
  31.938 ns (1 allocation: 64 bytes)
```

The two operations may seem equivalent, as they both ultimately compute `1 + 2`. However, the methods used in each case differ, with the first approach being faster because the function call is type stable.

Specifically, the output `x[1] + x[2]` in the first tab can be deduced to be `Int64`, thus satisfying the definition of type stability. This occurs because `x[1]` and `x[2]` can be identified as `Int64`, allowing the compiler to generate code specialized for this type. Note that the efficiency of the generated code isn't limited to the given operation: it applies to any call `sum(y)` such that `y` is a `Vector{Float64}`.

In contrast, **the second tab defines a type-unstable function call**. Since `x` has type `Vector{Any}`, it becomes impossible to predict a unique concrete type for `x[1] + x[2]` solely based on `x`'s type. This is because `x[1]` and `x[2]` may embody any concrete type that is a subtype of `Any`. Consequently, the compiler is forced to create code with multiple conditional statements, with each branch handling how to compute `x[1] + x[2]` for a possible type (`Int64`, `Float64`, `Float32`, etc.). This results in slow compiled code, as it'll require extra work during runtime. Furthermore, the degraded performance will be incurred for every call `sum(y)` such that `y` has type `Vector{Any}`.

> **Remark**
>
> Julia's developers are continually refining the compiler, addressing and mitigating the effects of certain type instabilities. As a result, **many operations that were once type unstable are now type stable**. This means that type stability should be considered a dynamic property of the language, subject to change as the compiler evolves.

CHECKING FOR TYPE STABILITY

There are several mechanisms to determine whether a function call is type stable. One of them is based on the `@code_warntype` macro, which reports all the types inferred during a function call. To illustrate its use, consider a function that defines `y` as a transformation of `x`, and then uses `y` to perform some operation.

```
function foo(x)
    y = (x < 0) ?  0  :  x

    return [y * i for i in 1:100]
end
```

```
julia> @code_warntype foo(1.)
```

```
function foo(x)
    y = (x < 0) ?  0  :  x

    return [y * i for i in 1:100]
end
```

```
julia> @code_warntype foo(1)
```

The output of `@code_warntype` can be difficult to interpret. Nonetheless, the addition of colors facilitates its understanding:

- If all lines are **blue**, the function is **type stable**. This means that Julia can identify a unique concrete type for each variable.
- If at least one line is **red**, the function is **type unstable**. It reflects that one variable or more could potentially adopt multiple possible types.
- **Yellow** lines indicate type instabilities that the compiler can handle effectively, in the sense that they have a reduced impact on performance. As a rule of thumb, **you can safely ignore them**.

> **Warning!**

> Throughout the website, we'll refer to **type instabilities** as those indicated by a red warning exclusively. Yellow warnings will be mostly ignored.

In the provided example, the compiler attempts to infer concrete types. This is done by identifying two pieces of information, given `x`'s concrete type:

- **i)** the type of `y`,
- **ii)** the type of `y * i` where `i` has type `Int64`, implicitly defining the type of `[y * i for i in 1:100]`.

The example clearly demonstrates that **the same function can be type stable or unstable depending on the types of its inputs**: `foo` is type stable when `x` has type `Int64`, but type unstable when `x` is `Float64`.

Specifically, in the scenario where `x = 1`, the compiler infers for *i)* that `y` can be equal to either `0` or `x`. Since both `0` and `1` are `Int64`, the compiler identifies a unique type for `y`, given by `Int64`. Regarding *ii)*, `y * i` also yields an `Int64`, as both `i` and `y` have type `Int64`. This determines that `[y * i for i in 1:100]` has type `Vector{Int64}`. Consequently, `foo(1)` is type stable, enabling Julia to invoke a method specialized for integers.

As for `x = 1.0`, the information for *i)* is that `y` could be either `0` or `1.0`. As a result, the compiler can't infer a unique type for `y`, which could be either `Int64` or `Float64`. The `@code_warntype` macro reflects this, identifying `y` as having type `Union{Float64, Int64}`. This ambiguity affects *ii)*, forcing the compiler to consider approaches that handle both `Float64` and `Int64`, and hence preventing specialization. Overall, `foo(1.0)` is type unstable, which has a detrimental impact on performance.

> **Remark**
>
> The conclusions regarding type stability wouldn't have changed if we had considered, for instance, `foo(-2)` or `foo(-2.0)`. This is because the compilation process relies on information about types, not values. More specifically, this means that type stability depends on whether `x` has type `Int64` or `Float64`, regardless of its actual value.

YELLOW WARNINGS MAY TURN RED

Not all instances of type instabilities have the same impact on performance. Their severity is ultimately indicated through a yellow or red warning. Yellow warnings denote a relatively minor impact on performance, typically resulting from isolated computations that Julia can handle effectively. However, repeated execution of these operations may escalate into more serious performance issues, triggering a red warning. The following example demonstrates a scenario like this.

```
function foo(x)
    y = (x < 0) ?  0  :  x

    y * 2
end
```

```
julia> @code_warntype foo(1.)
```

```
function foo(x)
    y = (x < 0) ?  0  :  x

    [y * i for i in 1:100]
end
```

```
julia> @code_warntype foo(1.)
```

```
function foo(x)
    y = (x < 0) ?  0  :  x

    for i in 1:100
      y = y + i
    end

    return y
end
```

```
julia> @code_warntype foo(1.)
```

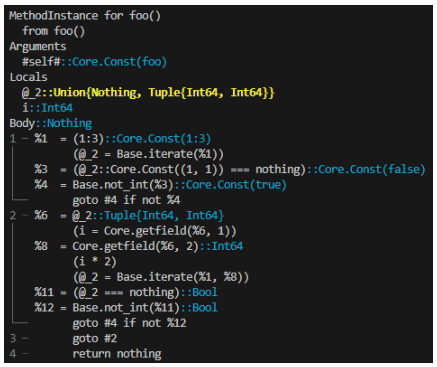
The yellow warning reflects that `y * 2` could return either a `Float64` or `Int64` value. However, this operation is computed only once and based on two types that the compiler can handle efficiently. Instead, the second tab involves multiple computations `y * i` without knowledge of a unique concrete type for `y`, resulting in a red warning.

Despite this, note that not all yellow warnings will necessarily escalate to a red warning when incorporated into a for-loop. The third tab illustrates this point, reinforcing that not all type instabilities are equally detrimental.

> **For-Loops and Yellow Warnings**
>
> A yellow warning will always be displayed when running a for-loop, even if the operation itself is type stable. In such cases, the warning can safely be disregarded, as it simply reflects the inherent behavior of iterators: they return either the next element to iterate over or `nothing` with type `Nothing` when the sequence is exhausted.
>
> ▼ For-Loop

```
function foo()
    for i in 1:100
        i
    end
end
```

```
julia> @code_warntype foo()
```



8c. Type Stability with Scalars and Vectors

Martin Alfaro
PhD in Economics

INTRODUCTION

The previous section has defined type stability, along with approaches to checking whether the property holds. In this section, we start the analysis of type stability for specific objects. We cover in particular the case of scalars and vectors, providing practical guidance for achieving type stability with them.

TYPES OF SCALARS AND VECTORS

Recall that the formal definition of a type-stable function is that the function's output type can be inferred from its argument types. In practice, however, we often rely on a more stringent definition, which requires that the compiler can infer a single concrete type for each expression within the function body. This property guarantees that every operation is specialized, resulting in optimal performance. Nevertheless, simply demanding that the output's type can be inferred already offers benefits, as it ensures that type instability won't be propagated when the function is called in other operations.

The principle applied to scalars is straightforward, demanding operations be performed on variables with the same concrete type (e.g., `Float64`, `Int64`, `Bool`). In contrast, type stability for vectors rather requires that the *elements* have a concrete type. The following table identifies scalars and vectors satisfying this property.

Objects Whose Elements Have Concrete Types

Scalars	Vectors
`Int`	`Vector{Int}`
`Int64`	`Vector{Int64}`
`Float64`	`Vector{Float64}`
`Bool`	`BitVector`

Note: `Int` defaults to `Int64` or `Int32`, depending on your CPU's architecture.

Next, we'll delve into type stability in scalars and vectors, considering each case separately.

TYPE STABILITY WITH SCALARS

To make the definition of type stability for scalars operational, let's revisit some concepts about types. Recall that only concrete types like `Int64` or `Float64` can be instantiated, while abstract types like `Any` or `Number` can't.

Instantiation simply means that all values ultimately adopt a unique concrete type. For instance, a variable `x::Number = 2` shouldn't be interpreted as `x` having the type `Number`. Instead, it means that `x` can only be reassigned to values whose concrete type is a subtype of `Number`. Ultimately, `x` must have a concrete type, which in this case is `Int64`.

In this context, type instability may arise when operations mix `Int64` and `Float64`, although this isn't always the case. To illustrate this, we'll start showing some scenarios where mixing these types doesn't cause issues.

TYPE PROMOTION AND CONVERSION

Julia employs various mechanisms to handle cases combining `Int64` and `Float64`. The first one is part of a concept known as **type promotion**, which converts dissimilar types to a common one whenever possible. The second one emerges when variables are type-annotated, in which case Julia engages in **type conversions**. By transforming values to the respective type declared, this feature could also prevent the mix of types.

Both mechanisms are illustrated below.

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

x          = 2
y          = 0.5

z            = foo(x,y)          # type stable: mixing `Int64` and `Float64` results in
`Float64`
```

```
julia> z
1.0
```

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

x::Float64 = 2               # this is converted to `2.0`
y          = 0.5

z           = foo(x,y)         # type stable: `x` and `y` are `Float64`, so predictable type
of output
```

```
julia> z
1.0
```

In the first tab, the output's type depends on the argument's types. However, in all cases the output's type can be predicted, since mixing `Int64` and `Float64` results in `Float64` due to automatic type promotion. As for the second tab, Julia transforms the value of `x` to make it consistent with the type-annotation declared. Consequently, `x * y` is computed as the product of two values with type `Float64`.

TYPE INSTABILITY WITH SCALARS

While type promotion and conversion can handle certain situations, they certainly don't cover all cases. One such scenario is when a scalar's value depends on a conditional statement and each branch returns a value of a different type. In this situation, since the compiler only considers the types and not values, it can't determine which branch is relevant for the function call. As a result, it'll generate code that accommodates both possibilities, as it happens in the following example.

```
function foo(x,y)
    a = (x > y) ?  x  :  y

    [a * i for i in 1:100_000]
end

foo(1, 2)           # type stable   -> `a * i` is always `Int64`
```

```
julia> @btime foo(1,2)
  23.800 µs (2 allocations: 781.30 KiB)
```

```
function foo(x,y)
    a = (x > y) ?  x  :  y

    [a * i for i in 1:100_000]
end

foo(1, 2.5)         # type UNSTABLE -> `a * i` is either `Int64` or `Float64`
```

```
julia> @btime foo(1,2.5)
  43.200 µs (2 allocations: 781.30 KiB)
```

In the example, type instability will inevitably arise if `x` and `y` have different types. Note that type promotion is of no help here. The reason is that this mechanism only ensures that `a * i` will be converted to `Float64` if `a` is `Float64`, considering that `i` is `Int64`. However, the compiler also needs to consider the possibility that `a` could be `Int64`, in which case `a * i` would be `Int64`.

Given this ambiguity, the method instance created must be capable of handling both scenarios. Then, during runtime, Julia will gather more information to disambiguate the situation, and select the relevant computation implementation.

TYPE STABILITY WITH VECTORS

Vectors in Julia are formally defined as collections of elements sharing a homogeneous type. Since operations based on vectors ultimately handle individual elements, type stability is contingent on whether the type of their elements is concrete.

In this context, it's important to distinguish between the type of the object and of its elements. This is because vectors having elements with a concrete type are themselves concrete, but elements with abstract types will still give rise to vectors with concrete types. This is clearly observed with `Vector{Any}`, a concrete type comprising elements with the abstract type `Any`.

Before the analysis of specific scenarios, we start by considering type conversion applied to vectors. This mechanism prevents the mix of types when vectors are defined.

TYPE PROMOTION AND CONVERSION

By definition, vectors require all their elements to share the same type. This means that if you mix elements with disparate types, such as `String` and `Int64`, Julia will infer the vector's type as `Vector{Any}`. Despite this, there are cases where elements can be converted to a common type, such as when mixing `Float64` and `Int64`.

The following example shows this mechanism in an assignment, where the vector is not type annotated. In this case, all elements are converted to the most general type among the values included.

```
x = [1, 2, 2.5]     # automatic conversion to `Vector{Float64}`
```

```
julia> x
3-element Vector{Float64}:
 1.0
 2.0
 2.5
```

```
y = [1, 2.0, 3.0]    # automatic conversion to `Vector{Float64}`
```

```
julia> y
3-element Vector{Float64}:
 1.0
 2.0
 3.0
```

When assignments are instead declared with type-annotations and values are of different types, Julia will attempt to perform a conversion. If possible, this ensures that the assigned values conform to the declared type.

```
x1                      = [1, 2.0, 3.0]                    # automatic conversion to
`Vector{Float64}`

x2::Vector{Int64}  = y1                            # conversion to `Vector{Int64}`
```

```
julia> z2
3-element Vector{Number}:
 1.0
 2.0
 2.5
```

```
y1                      = [1, 2, 2.5]                      # automatic conversion to
`Vector{Float64}`

y2::Vector{Number} = y1                            # `y2` is still `Vector{Number}`
```

```
julia> z2
3-element Vector{Number}:
 1.0
 2.0
 2.5
```

```
nr_elements  = 3
z            = Vector{Any}(undef, nr_elements)     # `Vector{Any}` always

z           .= 1
```

```
julia> v
3-element Vector{Any}:
 1
 1
 1
```

TYPE INSTABILITY

When evaluating type stability with vectors, two forms of operations must be considered. The first one involves operations that manipulate individual elements, such as `x[i]`. This scenario is analogous to the case of scalars, and therefore type stability follows the same rules.

The second scenario involves functions operating on the entire vector. In this case, type stability requires that vectors have elements with a concrete type. Note that this condition isn't sufficient to guarantee type stability, which ultimately depends on how the function implements the operation executed.

Nevertheless, packages tend to provide optimized versions of functions. Consequently, functions are typically type stable when users provide vectors with elements of a concrete type. For instance, this is illustrated below by the function `sum`, which adds all elements in a vector.

```
x1::Vector{Int}     = [1, 2, 3]

sum(x1)             # type stable
```

```
x2::Vector{Int64}   = [1, 2, 3]

sum(x2)             # type stable
```

```
x3::Vector{Float64} = [1, 2, 3]

sum(x3)             # type stable
```

```
x4::BitVector       = [true, false, true]

sum(x4)             # type stable
```

In contrast, the following vectors have elements with abstract types, which result in type instability.

```
x5::Vector{Number} = [1, 2, 3]

sum(x5)                # type UNSTABLE -> `sum` must consider all possible subtypes of
`Number`
```

```
x6::Vector{Any}    = [1, 2, 3]

sum(x6)             # type UNSTABLE -> `sum` must consider all possible subtypes of `Any`
```

8d. Type Stability with Global Variables

Martin Alfaro
PhD in Economics

INTRODUCTION

Variables can be categorized as local or global according to the code block in which they live: **global variables** can be accessed and modified anywhere in the code, while **local variables** are only accessible within a specific scope. In the context of this section, the scope of interest is a function, so local variables will exclusively refer to function arguments and variables defined within the function.

The distinction between local and global variables is especially relevant for this chapter since **global variables are a common source of type instability**. The issue arises because Julia's type system doesn't assign specific concrete types to global variables. As a result, the compiler is forced to consider multiple possibilities for any computation involving these variables. This limitation prevents specialization, leading to reduced performance.

The current section explores two approaches to working with global variables: type-annotations and constants. Defining global variables as constants is a natural choice when values are truly fixed, such as in the case of `π = 3.14159`. More broadly, constants can be used in any scenario where they remain unmodified throughout the script. Compared to type annotations, constants offer better performance, as the compiler gains knowledge of *both* the type and value, rather than just the type. This feature allows for further optimizations, effectively making **the behavior of constants within a function indistinguishable from that of a literal value**. [1]

> **Warning!** - You Should Still Wrap Code in a Function
>
> Even if you implement the fixes proposed for global variables, optimal performance still calls for wrapping tasks in functions. The reason is that **functions implement additional optimizations** that aren't possible in the global scope.

WHEN ARE WE USING GLOBAL VARIABLES?

Before exploring approaches for handling global variables, let's first identify scenarios in which global variables arise. To this end, we present two cases, each represented in a tab below. The first one considers the simplest scenario possible, where operations are performed directly in the global scope. For its part, the second one illustrates a more nuanced case, where a function accesses and operates on a global variable.

The third tab serves as a counterpoint, implementing the same operations but within a self-contained function. By definition, self-contained functions exclusively operate with locally defined variables. Thus, the comparison of the last two tabs highlights the performance lost by relying on global variables.

```
# all operations are type UNSTABLE (they're defined in the global scope)
x = 2

y = 2 * x
z = log(y)
```

```
x = 2

function foo()
    y = 2 * x
    z = log(y)

    return z
end

@code_warntype foo() # type UNSTABLE
```

```
x = 2

function foo(x)
    y = 2 * x
    z = log(y)

    return z
end

@code_warntype foo(x)  # type stable
```

Self-contained functions offer advantages that extend beyond performance gains: they **enhance readability, predictability, testability, and reusability**. These benefits were briefly considered in a previous section, and come from an interpretation of functions as embodying a specific task.

Among other benefits, self-contained functions are easier to reason about, as understanding their logic doesn't require tracking variables across the entire script. Moreover, a function's output depends solely on its input parameters, without any dependence on the script's state regarding global variables. This makes self-contained functions more predictable, additionally simplifying the code debugging process. Finally, by acting as a standalone program with a clear well-defined purpose, self-contained functions can be reapplied for similar tasks, reducing code duplication and facilitating code maintainability.

ACHIEVING TYPE STABILITY WITH GLOBAL VARIABLES

The previous subsection emphasized the benefits of self-contained functions, providing compelling reasons to avoid global variables. Nonetheless, global variables can still be highly convenient in certain scenarios. For instance, this is the case when we work with true constants. Considering this, next we present two approaches that let us work with global variables, while addressing their performance penalty.

CONSTANT GLOBAL VARIABLES

Declaring global variables as constants requires adding the `const` keyword before the variable's name, such as in `const x = 3`. This approach can be applied to variables of any type, including collections.

```
const a = 5
foo()   = 2 * a

@code_warntype foo()        # type stable
```

```
const b = [1, 2, 3]
foo()   = sum(b)

@code_warntype foo()        # type stable
```

> **Warning!**
>
> **Global variables should only be declared constants if their value will remain unchanged throughout the script**. Although it's possible to redefine constants, this option was only introduced to facilitate testing during interactive use, thereby avoiding the need to restart a Julia session for each new constant value. Importantly, the use of this option assumes that all dependent functions are re-declared when the constant's value is modified: any function that isn't redefined will still rely on the constant's original value. This is why you should re-run the entire script **if you absolutely need to reassign the value of a constant**.
>
> To illustrate the potential consequences of overlooking this practice, let's compare the following code snippets that execute the function `foo`. Both define a constant value of `x=1`, which is subsequently redefined as `x=2`. The first example runs the script without re-executing the definition of `foo`, in which case the value returned by `foo` is still based on `x = 1`. In contrast, the second example emulates the re-execution of the entire script. This is achieved by rerunning `foo`'s definition, thus ensuring that `foo` relies on the updated value of `x`.

```
const x = 1
foo()   = x
foo()             # it gives 1

x     = 2
foo()             # it still gives 1
```

```
const x = 1
foo()   = x
foo()             # it gives 1

x     = 2
foo() = x
foo()             # it gives 2
```

TYPE-ANNOTATING A GLOBAL VARIABLE

The second approach to address type instability involves asserting a *concrete* type for a global variable. This is done by including the operator `::` after the variable's name (e.g., `x::Int64`).

```
x::Int64          = 5
foo()             = 2 * x

@code_warntype foo()     # type stable
```

```
y::Vector{Float64} = [1, 2, 3]
foo()              = sum(y)

@code_warntype foo()     # type stable
```

```
z::Vector{Number}  = [1, 2, 3]
foo()              = sum(z)

@code_warntype foo()     # type UNSTABLE
```

Note that simply declaring a global variable with an abstract type won't resolve the type instability issue.

DIFFERENCES BETWEEN APPROACHES

The two approaches presented for handling global variables have distinct implications for both code behavior and performance. The key to these differences lies in that **type-annotations assert a variable's type, while constants additionally declare its value**. Next, we analyze each consequence.

DIFFERENCES IN CODE

Unlike the case of constants, type-annotations allow you to reassign a global variable without unexpected consequences. This means you don't need to re-run the entire script when redefining the variable.

```
x::Int64 = 5
foo()    = 2 * x
foo()               # output is 10

x        = 2
foo()    = 2 * x
foo()               # output is 4
```

DIFFERENCES IN PERFORMANCE

Type-annotated global variables are more flexible, as we only need to declare their types without committing to a specific value. However, this flexibility comes at the cost of performance, since they prevent certain optimizations that hold with constants. Such optimizations are feasible because constants not only provide information about their types, but also act as a promise that their value will remain fixed throughout the code. Within a function, this feature allows constants to behave like literal values embedded directly in the code. Consequently, the compiler can potentially replace certain expressions with their resulting outcome.

The following code demonstrates a scenario where this occurs. It consists of an operation that can be pre-calculated if the global variable's value is known. Thus, declaring the global variable as a constant enables the compiler to replace this operation by its result, making it equivalent to a hard-coded value. On the contrary, merely type-annotating the global variable only specializes code for the type provided. To starkly reveal the effect, we'll call this operation in a for-loop.

```
const k1  = 2

function foo()
    for _ in 1:100_000
       2^k1
    end
end
```

```
julia> @btime foo()
  0.800 ns (0 allocations: 0 bytes)
```

```
k2::Int64 = 2

function foo()
    for _ in 1:100_000
       2^k2
    end
end
```

```
julia> @btime foo()
  115.600 µs (0 allocations: 0 bytes)
```

Remark

Even without declaring a variable as a constant, the compiler could still recognize the invariance of some operations and perform optimizations accordingly. To illustrate this, suppose we want to reexpress each element of `x` as a proportion relative to the sum of the elements. A naive approach would involve a for-loop with `sum(x)` incorporated into the for-loop body, resulting in the repeated computation of `sum(x)`. If, on the contrary, we calculate shares through `x ./ sum(x)`, the compiler is smart enough to recognize the invariance of `sum(x)` across iterations, therefore proceeding to its pre-computation.

```
x           = rand(100_000)


function foo(x)
    y    = similar(x)

    for i in eachindex(x,y)
        y[i] = x[i] / sum(x)
    end

    return y
end
```

```
julia> @btime foo($x)
  633.245 ms (2 allocations: 781.30 KiB)
```

```
x           = rand(100_000)


foo(x) = x ./ sum(x)
```

```
julia> @btime foo($x)
  49.400 µs (2 allocations: 781.30 KiB)
```

```
x           = rand(100_000)
const sum_x = sum(x)

foo(x) = x ./ sum_x
```

```
julia> @btime foo($x)
  41.500 μs (2 allocations: 781.30 KiB)
```

FOOTNOTES

[1] Literal values refer to values expressed directly in the code (e.g., `1`, `"hello"`, or `true`), in contrast to values coming from a variable input.

8e. Barrier Functions

Martin Alfaro
PhD in Economics

INTRODUCTION

This section presents an approach to mitigating type instability based on the so-called **barrier functions**. These are defined as type-stable functions embedded within a type-unstable function, where variables having uncertain types are passed as arguments. By doing so, the compiler is prompted to infer a concrete type for the variables, effectively creating a "barrier" that prevents the spread of type instability to subsequent operations.

A key benefit of this approach is that **barrier functions are agnostic to the underlying cause of type instability, making them widely applicable**.

> **Warning!** - Barrier Functions Should Be Considered as a Second Option
>
> Typically, barrier functions should be reserved for situations where type instability is either difficult to fix or inherent to the operations performed. This is because the original function will still be type unstable, with different consequences depending on the instability nature. Considering this, it's best to aim for type-stable code from the outset, whenever possible.

APPLYING BARRIER FUNCTIONS

To illustrate the technique, let's revisit a type-unstable function from a previous section. This function defines a variable `y` based on `x`, and subsequently performs an operation involving `y`.

```
function foo(x)
    y = (x < 0) ?  0  :  x

    [y * i for i in 1:100]
end

@code_warntype foo(1)       # type stable
@code_warntype foo(1.)      # type UNSTABLE
```

In the example, `0` is an `Int64`, whereas `x` could be either an `Int64` or `Float64`. When `x` is an `Int64`, `y` will also be an `Int64`, making `foo(1)` type stable. However, when `x` is a `Float64`, the compiler can't determine whether `y` will be an `Int64` or a `Float64`, rendering `foo(1.)` type unstable.

Addressing this type instability through a barrier functions requires embedding a type-stable function into `foo`, passing `y` as an argument. By doing so, the function will attempt to deduce `y`'s type, allowing the compiler to use this information for subsequent operations. The example below in particular defines `operation` as a barrier function. [1]

```
operation(y) = [y * i for i in 1:100]

function foo(x)
    y = (x < 0) ?  0  :  x

    operation(y)
end

@code_warntype operation(1)    # barrier function is type stable
@code_warntype operation(1.)   # barrier function is type stable

@code_warntype foo(1)          # type stable
@code_warntype foo(1.)         # barrier-function solution
```

With the introduction of the barrier function `operation`, the variable `y` in `foo(1.)` can still be either an `Int64` or a `Float64`. Nevertheless, this ambiguity no longer matters, as `operation(y)` will determine the type of `y` before the array comprehension is executed. As a result, the expression `[y * i for i in 1:100]` will be computed using a method specialized for the specific type of `y`, ensuring type stability.

> **Warning!**
>
> Barrier Functions should solve the type instability *before* the type unstable operation is executed. Otherwise, we're back to the original issue, where the compiler has to check `y`'s type at each iteration and select a method accordingly.
>
> For example, `foo` in the example below doesn't apply correctly the barrier-function technique: `y` can be either `Float64` or `Int64`, and `operation(y,i)` only identifies the type inside the for-loop. This determines that the compiler is forced to check `y`'s type at each iteration of the loop, which is the original problem the barrier function was intended to solve.
>
> ```
> operation(y,i) = y * i
>
> function foo(x)
>     y = (x < 0) ?  0  :  x
>
>     [operation(y,i) for i in 1:100]
> end
>
> @code_warntype foo(1)          # type stable
> @code_warntype foo(1.)         # type UNSTABLE
> ```

REMARKS ON @CODE_WARNTYPE

Functions introducing barrier functions hinder the interpretation of `@code_warntype`. This is because barrier functions typically mitigate type instability, rather than completely eliminating it. And even if the barrier function successfully eliminates the type instability, we could still receive a red warning.

To illustrate this, let's start presenting a scenario where the barrier function completely eliminates the type instability. Yet, a red warning shows up.

```
x = ["a", 1]                    # variable with type 'Any'



function foo(x)
    y = x[2]

    [y * i for i in 1:100]
end
```

```
julia> @code_warntype foo(x)
```

```
x = ["a", 1]                    # variable with type 'Any'

operation(y) = [y * i for i in 1:100]

function foo(x)
    y = x[2]

    operation(y)
end
```

```
julia> @code_warntype foo(x)
```

In this example, `y` is defined from an object with type `Vector{Any}`. This leads to a red warning, as `x[2]` has type `Any` and therefore the compiler can't infer a concrete type for `y`. However, no operation is involved at that point, as we're only performing an assignment. Since the only operation performed uses a barrier function, the lack of type information is inconsequential. Therefore, type instability is never impacting performance after introducing a barrier function.

In contrast, the example below demonstrates that a barrier function may only alleviate type instability, rather than eliminate it entirely. In this scenario, the operation `2 * x[2]` is type unstable, forcing the compiler to generate code for each possible concrete type of `x[2]`. Nonetheless, this operation has a negligible performance impact on `foo`, justifying why the barrier function only targets the more demanding operation.

```
x = ["a", 1]                    # variable with type 'Any'



function foo(x)
    y = 2 * x[2]

    [y * i for i in 1:100]
end
```

```
julia> @code_warntype foo(x)
```

```
x = ["a", 1]                    # variable with type 'Any'

operation(y) = [y * i for i in 1:100]

function foo(x)
    y = 2 * x[2]

    operation(y)
end
```

```
julia> @code_warntype foo(x)
```

```
x = ["a", 1]                    # variable with type 'Any'

operation(y) = [y * i for i in 1:100]

function foo(z)
    y = 2 * z

    operation(y)
end
```

```
julia> @code_warntype foo(x)
```

Notice that whether a barrier function is effective in solving performance issues ultimately depends on how the function is applied. In the given example, the barrier-function solution would be sufficient if `foo` is called only once. Instead, if `foo` is eventually called in a tight loop, the type instability of `2 * x[2]` would be incurred multiple times. In such cases, additionally addressing the type instability of `2 * x[2]` could lead to substantial performance benefits.

FOOTNOTES

[1] In this particular example, there's an easier solution for the type instability, where `0` is substituted with `zero(x)`. The function `zero(x)` has been designed to return the null element for the type identified of `x`.

8f. Type Stability with Tuples

Martin Alfaro
PhD in Economics

INTRODUCTION

A function is considered type stable when, given the types of its arguments, the compiler can accurately predict single concrete types for its expressions. This definition, while universal, takes on different forms when applied to specific objects. So far, we've exclusively concentrated on scalars and vectors, whose conditions for type stability are relatively straightforward.

In this section, we begin the analysis of type stability for other data structures. This is done by covering tuples. Guaranteeing type stability with tuples is more nuanced compared to vectors, as their type characterization demands more information. Its exploration will challenge our understanding of type stability, demanding a clear grasp of its definition and subtleties.

> **Warning!** - Tuples Are Only Suitable For Small Collections
>
> Remember that tuples should only be used for collections that comprise a few elements. Using them for large collections will result in significant performance degradation or directly trigger fatal errors.

COMPARING TUPLES AND VECTORS

Tuples and vectors are the most ubiquitous forms of collections in Julia, with tuples playing a vital role for two reasons. Firstly, tuples are more performant for small objects, as they avoid the overhead of memory allocation. This feature will be expanded on when we explore *static vectors*, which are essentially tuples that can be handled as vectors. The second reason is that tuples automatically encompass the case of **named tuples**, which are merely tuples having symbols as keys instead of indices.

In comparison to vectors, tuples possess a more intricate type system. To appreciate this, let's compare the information needed for each type description.

Vectors represent collections of elements sharing a *homogeneous* type, additionally allowing for varying number of elements. Thus, the information needed to describe the types of vectors is relatively minor. For instance, a type like `Vector{Float64}` establishes that *all* elements must have type `Float64`, without any restriction on number of elements to be contained.

For their part, tuples are fixed-size collections that can accommodate *heterogeneous* types. This makes the characterization of a tuple's type more demanding, requiring both the number of elements and the type of *each* element. For instance, the variable `tup = ("hello", 1)` has type `Tuple{String,`

`Int64}`, indicating that the first element has type `String` and the second one `Int64`. Furthermore, it implicitly sets the number of elements to two, as there's no possibility of appending or removing elements.

The fact that the number of elements is part of the type becomes clear when tuples contain `N` elements of the same type `T`. For this case, Julia provides the convenient alias `Ntuple{N, Float64}`, which is just syntactic sugar for `Tuple{T, T,...,T}` where `T` appears `N` times. [1]

In the following, we show that the choice between tuples and vectors may have different implications for type stability.

SLICES OF HETEROGENEOUS TUPLES CAN STILL BE TYPE STABLE

The type `Tuple` provides explicit information about each element's type. In contrast, vectors necessarily hold elements with a uniform type, entailing that mixing concrete types leads Julia to choose the smallest type that can encompass all of them.

In particular, vectors whose elements' types are extremely different require an abstract type to characterize all of them. In the worst case scenario, Julia could define `Vector{Any}` as the type. Defining this type propagates to slices, which will inherit the type of the parente vector. Consequenly, operations on these slices will result in type instability. Such a feature constrasts with **operations on slices of tuples, which identify a specific type for each element**.

TUPLE

```
tup    = (1, 2, "hello")        # type is `Tuple{Int64, Int64, String}`

foo(x) = sum(x[1:2])

@code_warntype foo(tup)         # type stable (output is `Int64`)
```

VECTOR

```
vector = [1, 2, "hello"]        # type is `Vector{Any}`

foo(x) = sum(x[1:2])

@code_warntype foo(vector)      # type UNSTABLE
```

Notice that type promotion could solve this issue. Through this mechanism, Julia attempts to convert each element of a vector into a common *concrete type*, thus avoiding the need of *abstract* types like `Any`. This is what occurs below, where numbers holding different types are converted to the most general concrete type.

TUPLE

```
tup    = (1, 2, 3.5)            # type is `Tuple{Int64, Int64, Float64}`

foo(x) = sum(x)

@code_warntype foo(tup)         # type stable (output returned is `Int64`)
```

VECTOR

```
vector = [1, 2, 3.5]            # type is `Vector{Float64}` (type promotion)

foo(x) = sum(x)

@code_warntype foo(vector)      # type stable (output returned is `Float64`)
```

TUPLES CONTAIN MORE INFORMATION THAN VECTORS

Given the differences in type information, conversions between tuples and vectors can pose several challenges for type stability. To see this, let's start with the simplest case, where a tuple is converted into a vector. The conclusions drawn from this case are straightforward, as they're essentially a corollary from the previous analysis: type stability will hold when the tuple contains type-homogeneous elements or when the types are heterogeneous but can be promoted to a common type.

For the examples, recall that **each type automatically creates a function** that transforms variables into that type. In particular, below we introduce the function `Vector` with the purpose of converting variables.

TYPE-HOMOGENEOUS TUPLES

```
tup = (1, 2, 3)                 # `Tuple{Int64, Int64, Int64}` or just `NTuple{3, Int64}`


function foo(tup)
    x = Vector(tup)             # 'x' has type `Vector(Int64)}`
    sum(x)
end

@code_warntype foo(tup)         # type stable
```

TYPE PROMOTION

```
tup = (1, 2, 3.5)               # `Tuple{Int64, Int64, Float64}`


function foo(tup)
    x = Vector(tup)             # 'x' has type `Vector(Float64)}`
    sum(x)
end

@code_warntype foo(tup)         # type stable
```

TYPE-HETEROGENEOUS TUPLES

```
tup = (1, 2, "hello")          # `Tuple{Int64, Int64, String}`


function foo(tup)
    x = Vector(tup)            # 'x' has type `Vector{Any}`
    sum(x)
end

@code_warntype foo(tup)        # type UNSTABLE
```

For its part, **creating a tuple from a vector will inevitably cause type instability**, regardless of the vector's characteristics. The reason is that vectors don't store information about the number of elements they contain. Consequently, the compiler must treat tuples as having a variable number of arguments, with each possible number corresponding to a different concrete type.

VECTOR WITH NON-PRIMITIVE TYPES

```
x   = [1, 2, "hello"]          # 'Vector{Any}' has no info on each individual type


function foo(x)
    tup = Tuple(x)             # 'tup' has type `Tuple`

    sum(tup[1:2])
end

@code_warntype foo(x)          # type UNSTABLE
```

VECTOR WITH PRIMITIVE TYPES

```
x   = [1, 2, 3]                # 'Vector{Int64}' has no info on the number of elements


function foo(x)
     tup = Tuple(x)               # 'tup' has type `Tuple{Vararg{Int64}}` (`Vararg` means
"variable arguments")

    sum(tup[1:2])
end

@code_warntype foo(x)          # type UNSTABLE
```

ADDRESSING VARIABLE ARGUMENTS: DISPATCH BY VALUE

A key takeaway from the previous subsection is that defining tuples from vectors invariably introduce type instability. A simple remedy for this is to convert tuples outside the function, which we then pass as function arguments. This is demonstrated in the code snippet below.

TUPLE AS FUNCTION ARGUMENT

```
x   = [1, 2, 3]
tup = Tuple(x)

foo(tup) = sum(tup[1:2])

@code_warntype foo(tup)         # type stable
```

The approach presented should be your first option when transforming vectors to tuples. Nonetheless, there may be scenarios where defining the tuple inside the function is unavoidable. In such cases, there are a few alternatives that can be implemented.

Note first that simply passing the vector's number of elements as a functoin argument doesn't solve the issue. The reason is that the compiler generates method instances based on information about types, not values. This means that a function argument like `length(x)` merely informs the compiler that the number of elements can be described as an object with type `Int64`, without providing any additional insight.

Instead, one effective solution is to define the tuple's length using a literal value, as demonstrated below.

NOT A SOLUTION

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

function foo(x)
    tup = NTuple{length(x), eltype(x)}(x)

    sum(tup)
end

@code_warntype foo(x)        # type UNSTABLE
```

INFLEXIBLE SOLUTION

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

function foo(x)
    tup = NTuple{3, eltype(x)}(x)

    sum(tup)
end

@code_warntype foo(tup)       # type stable
```

The downside of this solution is that it defeats the purpose of having generic code, as it restricts the function to tuples of a single predetermined size. To eliminate the type instability without constraining functionality, we need to introduce a more advanced solution. This is based on a technique known as **dispatch by value**. Since this approach is more complex to implement, *I recommend using it only when passing the tuple as a function argument is unfeasible*.

Next, we lay out the principles of dispatch by value, and then apply the technique to the specific case of tuples.

DEFINITING DISPATCH BY VALUE

Dispatch by value enables passing information about values to the compiler. Implementing this feature, nonetheless, requires a workaround, since the compiler only gathers information about types. The hack consists of creating a type that stores values as type parameters. In the case of tuples, this type parameter is simply the vector's number of elements.

The functionality is implemented via the built-in type `Val`, whose use is best explained through an example. Suppose a function `foo` and a value `a` that you wish the compiler to know. The technique requires defining `foo` with a type-annotated argument having no name, `::Val{a}`. After this, you must call `foo` passing an argument `Val(a)`, which instantiates a type with parameter `a`.

To illustrate the use of `Val`, we revisit an example included in previous sections. This considers a variable `y` that could be an `Int64` or `Float64`, contingent upon a condition. The ambiguity of `y`'s type is then transmitted to any subsequent operation, leading to type instability.

Dispatch by value is implemented by defining the condition as a type parameter of `Val`. In this way, the compiler will receive information about whether condition is `true` or `false`, and therefore know `y`'s type. This makes it possible to specialize its operations.

TYPE UNSTABLE

```
function foo(condition)
    y = condition ? 1 : 0.5      # either `Int64` or `Float64`

    [y * i for i in 1:100]
end

@code_warntype foo(true)         # type UNSTABLE
@code_warntype foo(false)        # type UNSTABLE
```

SOLUTION "VAL"

```
function foo(::Val{condition}) where condition
    y = condition ? 1 : 0.5      # either `Int64` or `Float64`

    [y * i for i in 1:100]
end

@code_warntype foo(Val(true))    # type stable
@code_warntype foo(Val(false))   # type stable
```

> **Warning!**
>
> The function argument `Val` must be defined with `{}`, but called with `()`. This is because types define their parameters with `{}`, while instances of types require functions.

DISPATCHING BY VALUE WITH TUPLES

Let's now revisit the conversion of vectors to tuples. As we previously discussed, type instability arises because vectors don't store the size as part of their type information, leaving the compiler without sufficient information to determine the tuple's type.

Dispatch by value provides a solution to this issue: by passing the vector's length as a type parameter, the function call becomes type stable.

TYPE UNSTABLE

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

function foo(x, N)
    tuple_x = NTuple{N, eltype(x)}(x)

    2 .+ tuple_x
end

@code_warntype foo(x, length(x))        # type UNSTABLE
```

SOLUTION "VAL"

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

function foo(x, ::Val{N}) where N
    tuple_x = NTuple{N, eltype(x)}(x)

    2 .+ tuple_x
end

@code_warntype foo(x, Val(length(x)))   # type stable
```

FOOTNOTES

[1] Don't confuse `NTuple` as an abbreviation for the type `NamedTuples`. The "N" in the former case is referring to a number "N" of elements.

8g. Type Stability with Higher-Order Functions

Martin Alfaro
PhD in Economics

INTRODUCTION

Functions in Julia are **first-class objects**, a concept also referred to as **first-class citizens**. This means that functions can be treated like any other variable, thus allowing for vectors of functions, functions returning other functions, and more.

In particular, the property makes it possible to define **higher-order functions**, which are functions that take another function as an argument. We've already encountered several examples of higher-order functions, often in the form of anonymous functions passed as arguments. A classic example of a higher-order function is `map(<function>, <collection>)`, given that `<function>` is one of its arguments. Throughout the explanations, we follow common terminology and refer the function passed as an argument as the *callback function*.

In this section, we'll examine conditions under which higher-order functions are type-stable. As we'll see, these functions present some challenges for specializing their computation method.

FUNCTIONS AS ARGUMENTS: THE ISSUE

For exploring type stability, a distinctive feature of functions is that *each function defines its own unique concrete type*. In turn, this concrete type is a subtype of an abstract type called `Function`, which encompasses all possible functions defined in Julia. This type system creates challenges when specializing the computation method of higher-order functions, as it can potentially lead to a combinatorial explosion of methods, with a unique method generated for each callback function.

To address this issue, Julia takes a conservative approach, **often choosing not to specialize the methods of high-order functions**. The performance in such cases can be severely degraded, as the execution runtime would become similar to performing operations in the global scope.

Taking this into account, it's important to pinpoint the scenarios where specialization is inhibited and monitor its consequences. If it occurs that performance is severely impaired, there are still ways to enforce specialization. In the following section, we explore several techniques for doing so.

> **Warning!**
>
> Exercise caution when inducing specialization. Overly aggressive specialization can degrade performance severely, explaining why Julia's default approach is deliberately conservative. In particular, you should

> avoid specialization when your script repeatedly calls a high-order function with many unique functions. [1]

AN EXAMPLE OF NO SPECIALIZATION

To illustrate when higher-order functions don't specialize, let's consider a specific scenario where we sum the transformed elements of a vector `x`. The only requirement we impose is that the transforming function should be generic, allowing us to possibly apply different functions for the transformation.

Below, we implement this operation through a higher-order function `foo`. This first uses `map` to transform `x` through some function `f`, and then applies the function `sum` to add the transformed elements. For demonstrating how `foo` works, we also call it with the function `abs` as its specific transformed function.

```
x = rand(100)

function foo(f, x)
    y = map(f, x)

    sum(y)
end
```

```
julia> @code_warntype foo(abs,x)
```

Although we can show that `foo(abs,x)` isn't specialized, `@code_warntype` **fails to detect any type-stability issues**. This is a consequence of `@code_warntype` evaluating type stability *under the assumption that specialization is attempted*. In our example, this assumption doesn't hold and therefore `@code_warntype` is of no use.

The specific reason for this behavior is that Julia **avoids specialization when a callback function isn't explicitly called within the function**. In the example, the function `f` only enters `foo` as an argument of `map`, but there's no explicit line calling `f`.

To obtain indirect evidence regarding the lack of specialization, we can compare the runtimes of the original `foo` function with a version that explicitly calls `f`.

```
x = rand(100)

function foo(f, x)
    y = map(f, x)


    sum(y)
end
```

```
julia> foo(abs, x)
48.447
julia> @btime foo(abs, $x)
  195.579 ns (3 allocations: 928 bytes)
```

```
x = rand(100)

function foo(f, x)
    y = map(f, x)
    f(1)                # irrelevant computation to force specialization

    sum(y)
end
```

```
julia> foo(abs, x)
48.447
julia> @btime foo(abs, $x)
  45.745 ns (1 allocation: 896 bytes)
```

The comparison lays bare a significant reduction in time when `f(1)` is added. Furthermore, there's also a notable decrease in memory allocations. As we'll demonstrate when exploring the subject, excessive allocations often serve as a telltale sign of type instability.

FORCING SPECIALIZATION

Explicitly calling the function to circumvent the no-specialization issue isn't optimal, since it introduces an unnecessary computation. Fortunately, alternative solutions exist to address the problem. One approach is to type-annotate `f`, which provides Julia with a hint to specialize. Another solution is to wrap the function in a tuple and then call it. This ensures the identification of the function's type, as tuples identify a concrete type for each element.

Below, we outline both approaches.

```
x = rand(100)


function foo(f::F, x) where F
    y = map(f, x)


    sum(y)
end
```

```
julia> foo(abs, x)
48.447
julia> @btime foo(abs, $x)
  46.686 ns (1 allocation: 896 bytes)
```

```
x = rand(100)
f_tup = (abs,)

function foo(f_tup, x)
    y = map(f_tup[1], x)


    sum(y)
end
```

```
julia> foo(f_tup, x)
48.447
julia> @btime foo($f_tup, $x)
  45.101 ns (1 allocation: 896 bytes)
```

FOOTNOTES

[1] For discussions about the issue of excessive specialization, see here and here.

8h. Gotchas for Type Stability

Martin Alfaro
PhD in Economics

INTRODUCTION

This section considers scenarios where type instabilities aren't immediately obvious. For this reason, we dub them as "gotchas". We also provide recommendations for addressing them. To make the section self-contained, we revisit some examples of type instability that were covered previously.

GOTCHA 1: INTEGERS AND FLOATS

When working with numeric scalars, it's essential to remember that `Int64` and `Float64` are distinct types. Mixing them can inadvertently introduce type instability.

To illustrate this case and how to handle it, consider a function `foo`. This takes a numeric variable `x` as its argument and performs two tasks. Firstly, it defines a variable `y` by transforming `x` in a way that all negative values are replaced with zero. Secondly, it executes an operation based on the resulting `y`.

The following example illustrates two implementations of `foo`. The first one suffers from type instability, while the second provides a revised implementation that addresses this issue.

```
function foo(x)
    y = (x < 0) ?  0  :  x

    return [y * i for i in 1:100]
end

@code_warntype foo(1)      # type stable
@code_warntype foo(1.)     # type UNSTABLE
```

```
function foo(x)
    y = (x < 0) ?  zero(x)  :  x

    return [y * i for i in 1:100]
end

@code_warntype foo(1)      # type stable
@code_warntype foo(1.)     # type stable
```

The first implementation uses the literal `0`, which has type `Int64`. If `x` is also `Int64`, no type instability arises. However, if `x` is `Float64`, the compiler must consider that `y` could be either `Int64` or `Float64`, thus causing type instability. [1]

Julia can handle combinations of `Int64` and `Float64` quite effectively. Therefore, the latter type instability wouldn't be a significant issue if the operation involving `y` calls `y` only once. Indeed, `@code_warntype` would only issue a yellow warning that therefore could be safely ignored. However, `foo` in our example repeatedly performs an operation that involves `y`, incurring the cost of type instability multiple times. As a result, `@code_warntype` issues a red warning, indicating a more serious performance issue.

The second tab proposes a **solution** based on a function that returns the zero element corresponding to the type of `x`. This approach can be extended to other values by using either the function `convert(typeof(x), <value>)` or `oftype(x, <value>)`. Both convert `<value>` to the same type as `x`. For instance, below we reimplement `foo`, but using the value `5` instead of `0`.

```
function foo(x)
    y = (x < 0) ?  5  :  x

    return [y * i for i in 1:100]
end

@code_warntype foo(1)      # type stable
@code_warntype foo(1.)     # type UNSTABLE
```

```
function foo(x)
    y = (x < 0) ?  convert(typeof(x), 5)  :  x

    return [y * i for i in 1:100]
end

@code_warntype foo(1)      # type stable
@code_warntype foo(1.)     # type stable
```

```
function foo(x)
    y = (x < 0) ?  oftype(x, 5)  :  x

    return [y * i for i in 1:100]
end

@code_warntype foo(1)      # type stable
@code_warntype foo(1.)     # type stable
```

GOTCHA 2: COLLECTIONS OF COLLECTIONS

When working in data analysis, collections of collections emerge naturally. An example of this data structure is given by the `DataFrames` package, which defines a table with each column representing a different variable. As we haven't introduced this package, we'll consider a more basic scenario involving a vector of vectors, represented by the type `Vector{Vector}`. This exhibits a similar structure, and also the same potential issues regarding type stability.

Essentially, the issue arises because, by not constraining the types of its inner vectors, a collection of collections like `Vector{Vector}` offers a high degree of flexibility. This flexibility is particularly valuable in data analysis, where datasets often comprise diverse columns that may contain disparate data types (e.g., `String`, `Float64`, `Int64`). However, it also comes at a cost: the type `Vector{Vector}` only guarantees that its elements are vectors, without providing any information about the concrete types they contain. As a result, when a function operates on one of these inner vectors, the type system is unable to infer the concrete type of the data, leading to type instability.

To illustrate this scenario, suppose a vector `data` comprising multiple inner vectors. Moreover, consider a function `foo` that takes `data` as its argument, and operates on one of its inner vectors `vec2`. The first tab below shows that this case leads to type instability. The simplest **solution** is presented in the second tab, and consists of including a barrier function that takes the inner vector `vec2` as its argument. The technique rectifies the type instability, as the barrier function attempts to identify a concrete type for `vec2`. Note that the barrier function is defined in-place). This implies that the value of `vec2`, and hence of `data`, is updated when `foo` is executed.

```
vec1 = ["a", "b", "c"] ; vec2 = [1, 2, 3]
data = [vec1, vec2]

function foo(data)
    for i in eachindex(data[2])
        data[2][i] = 2 * i
    end
end

@code_warntype foo(data)            # type UNSTABLE
```

```
vec1 = ["a", "b", "c"] ; vec2 = [1, 2, 3]
data = [vec1, vec2]

foo(data) = operation!(data[2])

function operation!(x)
    for i in eachindex(x)
        x[i] = 2 * i
    end
end

@code_warntype foo(data)            # barrier-function solution
```

GOTCHA 3: BARRIER FUNCTIONS

Barrier functions are an effective technique to mitigate type instabilities. However, it's essential to remember that **the parent function may remain type unstable**. When this is the case, if we fail to resolve the type instability before executing a repeated operation, the performance cost of the type instability will be incurred multiple times.

To illustrate this point, let's revisit the last example involving a vector of vectors. Below, we present two incorrect approaches to using a barrier function, followed by a demonstration of its proper application.

```
vec1 = ["a", "b", "c"] ; vec2 = [1, 2, 3]
data = [vec1, vec2]

operation(i) = (2 * i)

function foo(data)
    for i in eachindex(data[2])
        data[2][i] = operation(i)
    end
end

@code_warntype foo(data)            # type UNSTABLE
```

```
vec1 = ["a", "b", "c"] ; vec2 = [1, 2, 3]
data = [vec1, vec2]

operation!(x,i) = (x[i] = 2 * i)

function foo(data)
    for i in eachindex(data[2])
        operation!(data[2], i)
    end
end

@code_warntype foo(data)            # type UNSTABLE
```

```
vec1 = ["a", "b", "c"] ; vec2 = [1, 2, 3]
data = [vec1, vec2]

function operation!(x)
    for i in eachindex(x)
        x[i] = 2 * i
    end
end

foo(data) = operation!(data[2])

@code_warntype foo(data)            # barrier-function solution
```

GOTCHA 4: INFERENCE IS BY TYPE, NOT BY VALUE

Julia's compiler generates method instances solely based on types, without considering the actual values. To demonstrate this, let's consider the following concrete example.

```
function foo(condition)
    y = condition ?  2.5  :  1

    return [y * i for i in 1:100]
end

@code_warntype foo(true)          # type UNSTABLE
@code_warntype foo(false)         # type UNSTABLE
```

At first glance, we might erroneously conclude that `foo(true)` is type stable: the value of `condition` is `true`, so that `y = 2.5` and therefore `y` will have type `Float64`. However, values don't participate in multiple dispatch, meaning that Julia's compiler ignores the value of `condition` when inferring `y`'s type. Ultimately, `y` is treated as potentially being either `Int64` or `Float64`, leading to type instability.

The issue in this case can be easily resolved by replacing `1` by `1.`, thus ensuring that `y` is always `Float64`. More generally, we could employ similar techniques to the first "gotcha", where values are converted to a specific concrete type.

An alternative solution relies on dispatching by value, a technique we already explored and implemented for tuples. This technique makes it possible to pass information about values to the compiler. It's based on the type `Val`, along with the keyword `where` introduced here.

Specifically, for any function `foo` and value `a` that you seek the compiler to know, you need to include `::Val{a}` as an argument. In this way, `a` is interpreted as a type parameter, which you can identify by including the keyword `where`. Finally, we need to call `foo` by passing `Val(a)` as its input.

Applied to our example, type instability in `foo` emerges because the value of `condition` isn't known by the compiler. Dispatching by `a` enables us to pass the value of `condition` to the compiler.

```
function foo(condition)
    y = condition ?  2.5  :  1

    return [y * i for i in 1:100]
end

@code_warntype foo(true)          # type UNSTABLE
@code_warntype foo(false)         # type UNSTABLE
```

```
function foo(::Val{condition}) where condition
    y = condition ?  2.5  :  1

    return [y * i for i in 1:100]
end

@code_warntype foo(Val(true))    # type stable
@code_warntype foo(Val(false))   # type stable
```

GOTCHA 5: VARIABLES AS DEFAULT VALUES OF KEYWORD ARGUMENTS

Functions accept positional and keyword arguments. In the particular case that functions are defined with keyword arguments, it's possible to assign default values. However, when these default values are specified through variables rather than literal values, a type instability is introduced. The reason is that the variable is then treated as a global variable.

```
foo(; x) = x

β = 1
@code_warntype foo(x=β)          #type stable
```

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


@code_warntype foo()             #type stable
```

```
foo(; x = β) = x

β = 1
@code_warntype foo()             #type UNSTABLE
```

When setting a variable as a default value is unavoidable, there are still a few strategies you could follow to restore type stability.

One set of solutions leverages the techniques we introduced for global variables. These include type-annotating the global variable (*Solution 1a*) or defining it as a constant (*Solution 1b*).

Another strategy involves defining a function that stores the default value. By doing so, you can take advantage of type inference, where the function attempts to infer a concrete type for the default value (*Solution 2*).

You can also adopt a local approach, by adding type annotations to either the keyword argument (*Solution 3a*) or the default value itself (*Solution 3b*). Finally, type instability does not arise when positional arguments are used as default values of keyword arguments (*Solution 4*).

All these cases are stated below.

```
foo(; x = β) = x

const β = 1
@code_warntype foo()            #type stable
```

```
foo(; x = β) = x

β::Int64 = 1
@code_warntype foo()            #type stable
```

```
foo(; x = β()) = x

β() = 1
@code_warntype foo()            #type stable
```

```
foo(; x::Int64 = β) = x

β = 1
@code_warntype foo()            #type stable
```

```
foo(; x = β::Int64) = x

β = 1
@code_warntype foo()            #type stable
```

```
foo(β; x = β) = x

β = 1
@code_warntype foo(β)           #type stable
```

GOTCHA 6: CLOSURES CAN EASILY INTRODUCE TYPE INSTABILITIES

Closures are a fundamental concept in programming. A typical situation where they arise is when **a function is defined inside another function**, granting the inner function access to the outer function's scope. In Julia, closures explicitly show up when defining functions within a function, but also implicitly when using anonymous functions within a function.

While closures offer a convenient way to write self-contained code, they can easily introduce type instabilities. Furthermore, although there have been some improvements in this area, their surrouding issues have been around for several years. This is why it's crucial to be aware of its subtle consequences and how to address them.

CLOSURES ARE COMMON IN CODING

There are several scenarios where nesting functions emerges naturally. One such scenario is when you aim to keep a task within a single self-contained unit of code. For instance, this approach is particularly useful if a function needs to perform multiple interdependent steps, such as data preparation (e.g., setting parameters or initializing variables) and subsequent computations based on that data. By nesting a function within another, you can keep related code organized and contained within the same logical block, promoting code readability and maintainability.

To illustrate the patterns involved with and without closures, we'll use generic code. This isn't intended to be executed, but rather to demonstrate the underlying structure of the code. We also suppose a task that lends itself to nested functions.

```
function task()
        # <here, you define parameters and initialize variables>

    function output()
        # <here, you do some computations with the variables and parameters>
    end

    return output()
end

task()
```

```
function task()
        # <here, you define parameters and initialize variables>

    return output(<variables>, <parameters>)
end

function output(<variables>, <parameters>)
        # <here, you do some computations with the variables and parameters>
end

task()
```

Although the approach using closures may seem more intuitive, it can easily introduce type instability. This occurs under certain conditions, such as:

- Redefining variables or arguments (e.g., when updating a variable in an output)
- Altering the order in which functions are defined
- Utilizing anonymous functions

Each of these cases is explored below, where we refer to the containing function as the *outer function* and the closure as the *inner function*.

WHEN THE ISSUE ARISES

Let's start examining three examples. They cover all the possible situations where closures could result in type instability, allowing us to identify real-world scenarios where they could emerge.

The first examples reveal that the placement of the inner function could matter for type stability.

```
function foo()
    x            = 1
    bar()        = x

    return bar()
end

@code_warntype foo()      # type stable
```

```
function foo()
    bar(x)       = x
    x            = 1

    return bar(x)
end

@code_warntype foo()      # type stable
```

```
function foo()
    bar()        = x
    x            = 1

    return bar()
end

@code_warntype foo()      # type UNSTABLE
```

```
function foo()
    bar()::Int64 = x::Int64
    x::Int64     = 1

    return bar()
end

@code_warntype foo()      # type UNSTABLE
```

```
function foo()
    x = 1

    return bar(x)
end

bar(x) = x

@code_warntype foo()      # type stable
```

The second example establishes that type instability arises when closures are combined with reassignments of variables or arguments. This issue even persists when you reassign the same object to the variable, including trivial expressions such as `x = x`. The example also reveals that type annotating the redefined variable or the closure doesn't resolve the problem.

```
function foo()
    x            = 1
    x            = 1            # or 'x = x', or 'x = 2'

    return x
end

@code_warntype foo()            # type stable
```

```
function foo()
    x            = 1
    x            = 1            # or 'x = x', or 'x = 2'
    bar(x)       = x

    return bar(x)
end

@code_warntype foo()            # type stable
```

```
function foo()
    x            = 1
    x            = 1            # or 'x = x', or 'x = 2'
    bar()        = x

    return bar()
end

@code_warntype foo()            # type UNSTABLE
```

```
function foo()
    x::Int64     = 1
    x            = 1
    bar()::Int64 = x::Int64

    return bar()
end

@code_warntype foo()            # type UNSTABLE
```

```
function foo()
    x::Int64     = 1
    bar()::Int64 = x::Int64
    x            = 1

    return bar()
end

@code_warntype foo()            # type UNSTABLE
```

```
function foo()
    bar()::Int64 = x::Int64
    x::Int64     = 1
    x            = 1

    return bar()
end

@code_warntype foo()            # type UNSTABLE
```

```
function foo()
    x            = 1
    x            = 1            # or 'x = x', or 'x = 2'

    return bar(x)
end

bar(x) = x

@code_warntype foo()            # type stable
```

Finally, the last example deals with situations involving multiple closures. It highlights that the order in which you define them could matter for type stability. The third tab in particular demonstrates that passing subsequent closures as arguments can sidestep the issue. However, such an approach is at odds with how code is generally written in Julia.

```
function foo(x)
    closure1(x) = x
    closure2(x) = closure1(x)

    return closure2(x)
end

@code_warntype foo(1)            # type stable
```

```
function foo(x)
    closure2(x) = closure1(x)
    closure1(x) = x

    return closure2(x)
end

@code_warntype foo(1)            # type UNSTABLE
```

```
function foo(x)
    closure2(x, closure1) = closure1(x)
    closure1(x)           = x

    return closure2(x, closure1)
end

@code_warntype foo(1)            # type stable
```

```
function foo(x)
    closure2(x) = closure1(x)

    return closure2(x)
end

closure1(x) = x

@code_warntype foo(1)            # type stable
```

In the following, we'll examine specific scenarios where these patterns emerge. The examples reveal that the issue can occur more frequently than we might expect. For each scenario, we'll also provide a solution that enables the use of a closure approach. Nonetheless, if the function captures a performance critical part of your code, it's probably wise to avoid closures.

"BUT NO ONE WRITES CODE LIKE THAT"

i) Transforming Variables through Conditionals

```
x = [1,2]; β = 1

function foo(x, β)
    (β < 0) && (β = -β)          # transform 'β' to use its absolute value

    bar(x) = x * β

    return bar(x)
end

@code_warntype foo(x, β)         # type UNSTABLE
```

```
x = [1,2]; β = 1

function foo(x, β)
    (β < 0) && (β = -β)         # transform 'β' to use its absolute value

    bar(x,β) = x * β

    return bar(x,β)
end

@code_warntype foo(x, β)        # type stable
```

```
x = [1,2]; β = 1

function foo(x, β)
    δ = (β < 0) ? -β : β        # transform 'β' to use its absolute value

    bar(x) = x * δ

    return bar(x)
end

@code_warntype foo(x, β)        # type stable
```

```
x = [1,2]; β = 1

function foo(x, β)
    β = abs(β)                    # 'δ = abs(β)' is preferable (you should avoid redefining
variables)

    bar(x) = x * δ

    return bar(x)
end

@code_warntype foo(x, β)        # type stable
```

Recall that the compiler doesn't dispatch by value, and so whether the condition holds is irrelevant. For instance, the type instability would still hold if we wrote `1 < 0` instead of `β < 0`. Moreover, the value used to redefine `β` is also unimportant, with the same conclusion holding if you write `β = β`.

ii) Anonymous Functions inside a Function

Using an anonymous function inside a function is another common form of closure. Considering this, type instability also arises in the example above if we replace the inner function `bar` for an anonymous function. To demonstrate this, we apply `filter` with an anonymous function that keeps all the values in `x` that are greater than `β`.

```
x = [1,2]; β = 1

function foo(x, β)
    (β < 0) && (β = -β)          # transform 'β' to use its absolute value

    filter(x -> x > β, x)        # keep elements greater than 'β'
end

@code_warntype foo(x, β)         # type UNSTABLE
```

```
x = [1,2]; β = 1

function foo(x, β)
    δ = (β < 0) ? -β : β         # define 'δ' as the absolute value of 'β'

    filter(x -> x > δ, x)        # keep elements greater than 'δ'
end

@code_warntype foo(x, β)         # type stable
```

```
x = [1,2]; β = 1

function foo(x, β)
    β = abs(β)                     # 'δ = abs(β)' is preferable (you should avoid redefining
variables)

    filter(x -> x > β, x)        # keep elements greater than β
end

@code_warntype foo(x, β)         # type stable
```

iii) Variable Updates

```
function foo(x)
    β = 0                       # or 'β::Int64 = 0'
    for i in 1:10
        β = β + i               # equivalent to 'β += i'
    end

    bar() = x + β               # or 'bar(x) = x + β'

    return bar()
end

@code_warntype foo(1)           # type UNSTABLE
```

```
function foo(x)
    β = 0
    for i in 1:10
        β = β + i
    end

    bar(x,β) = x + β

    return bar(x,β)
end

@code_warntype foo(1)          # type stable
```

```
x = [1,2]; β = 1

function foo(x, β)
    (1 < 0) && (β = β)

    bar(x) = x * β

    return bar(x)
end

@code_warntype foo(x, β)        # type UNSTABLE
```

iv) The Order in Which you Define Functions Could Matter Inside a Function

To illustrate this claim, suppose you want to define a variable x that depends on a parameter β. However, β is measured in one unit (e.g., meters), while x requires β to be expressed in a different unit (e.g., centimeters). This implies that, before defining x, we must rescale β to the appropriate unit.

Depending on how we implement the operation, a type instability could emerge.

```
function foo(β)
    x(β)                  =  2 * rescale_parameter(β)
    rescale_parameter(β)  =  β / 10

    return x(β)
end

@code_warntype foo(1)      # type UNSTABLE
```

```
function foo(β)
    rescale_parameter(β)  =  β / 10
    x(β)                  =  2 * rescale_parameter(β)

    return x(β)
end

@code_warntype foo(1)      # type stable
```

FOOTNOTES

[1] A similar problem would occur if we replaced `0` by `0.` and `x` is an integer.

9a. Overview and Goals

Martin Alfaro
PhD in Economics

INTRODUCTION

In the previous chapter, we began our exploration of high performance in Julia by focusing on type stability. We now shift our attention to memory allocations, a critical aspect of performance optimization.

Memory allocations occur whenever a new object is created, involving the reservation of memory space to store its values. The aspect is crucial for performance, since the approach selected to handle the process can significantly slow down computations. In particular, memory allocations on the heap, simply referred to as *memory allocations*, incur a notable cost due to the additional CPU instructions required for memory management.

Despite this, the interplay between memory allocation and performance is complex. In fact, **reducing memory allocation is neither necessary nor sufficient for speeding up computations**—we'll present instances where the approach allocating more memory turns out to be faster. This apparent paradox arises from a trade-off involved when creating a new object: although allocations can lead to a significant overhead, the resulting objects store their data in contiguous blocks of memory, enabling the CPU to access information more efficiently.

From a practical perspective, it's essential to closely monitor memory usage if performance is critical. **Excessive memory allocation often serves as a red flag**: if two approaches exhibit large differences in memory allocation, their execution speeds are likely to differ significantly as well.

9b. Stack vs Heap

Martin Alfaro
PhD in Economics

INTRODUCTION

Memory allocations are a fundamental process in computer programming. This occurs whenever a new object is created, and involves reserving memory space to store the object's values, typically in the computer's Random Access Memory (RAM). In turn, RAM is logically divided into two main areas: the stack and the heap. Importantly, these areas aren't physical locations, but rather conceptual models that govern how memory is managed.

The distinction is crucial for performance since memory allocations on the heap are a costly operation. This approach to handle memory requires searching for free memory, tracking memory information, and freeing unused memory (a process known as garbage collection). Stack allocations, by contrast, are simpler and therefore more efficient.

The performance difference between stack and heap allocations can be significant, quickly becoming a major performance bottleneck if the operation is performed repeatedly. This disparity in performance explains the common convention in programming, including Julia, where **memory allocations will exclusively refer to heap allocations**.

In the current section, we begin the exploration of memory allocations, by briefly comparing how the stack and the heap work.

STACK ALLOCATIONS

In Julia, typical objects stored on the stack include integers, numbers, characters, and small fixed-size collections like tuples.

Objects on the stack are characterized by having a fixed size, precluding the possibility of dynamically growing or shrinking in size. These characteristics make allocating and deallocating memory on the stack extremely efficient.

The primary limitation of the stack is its limited capacity, making it suitable only for objects with a few elements. Indeed, attempting to allocate more memory than the stack can accommodate will result in a "stack overflow" error, causing program termination. And, even if an object fits on the stack, allocating too many elements can significantly degrade performance. [1]

HEAP ALLOCATIONS

Common objects stored on the heap include arrays (such as vectors and matrices) and strings. Unlike the stack, the heap is designed to allocate and free memory in any order, allowing it to accommodate larger and more complex data structures. Thus, the heap can handle objects as large as the available RAM permits, which additionally could grow or shrink dynamically.

While the heap offers greater flexibility than the stack, its more complex memory management comes at the cost of slower performance. [2] Due to their reduced speed, Chapter 9 will outline approaches to minimizing heap allocations. Strategies for achieving this includes utilizing the stack whenever possible, and favoring mutation over the creation of new objects.

FOOTNOTES

1. There's no hard and fast rule about how many elements are "too many". Benchmarking is the only reliable way to determine the performance consequences for each particular case. As a rough guideline, objects with more than 100 elements will certainly suffer from poor performance, while those with fewer than 15 elements are likely to benefit from stack allocation.

2. To handle the memory of heap-allocated objects, Julia uses what's known as a *garbage collector*. This mechanism automatically identifies and frees memory no longer in use, which can be computationally expensive.

9c. Objects Allocating Memory

Martin Alfaro
PhD in Economics

INTRODUCTION

In the previous section, we introduced the fundamentals of memory allocations, highlighting that objects can be stored on either the heap or the stack. Furthermore, we emphasized the application of conventional terminology in programming and Julia, where **allocations exclusively refer to those on the heap**. This definition also gives rise to the expression that "an object allocates" when the object is stored on the heap.

The distinction isn't merely to economize on words. Rather, it reflects that heap allocations are the ones that matter when it comes to performance: they involve a more complex memory management, which can significantly hinder performance.

In fact, the close relationship between performance and heap allocations can be appreciated through the macros `@time` and `@btime`. To provide a comprehensive measure of performance, they not only return the total runtime of an operation, but additionally the heap allocations involved.

In the following, we initiate our analysis of memory allocation by categorizing objects into two groups: those that allocate and those that don't.

NUMBERS, TUPLES, NAMED TUPLES, AND RANGES DON'T ALLOCATE

We start by focusing on objects that don't allocate memory. They include:

- Numbers
- Tuples
- Named Tuples
- Ranges

As these objects don't allocate, neither does creating, accessing, and operating on them. This is demonstrated below.

```
function foo()
    x = 1; y = 2

    x + y
end
```

```
julia> @btime foo()
  0.800 ns (0 allocations: 0 bytes)
```

```
function foo()
    tup = (1,2,3)

    tup[1] + tup[2] * tup[3]
end
```

```
julia> @btime foo()
  0.800 ns (0 allocations: 0 bytes)
```

```
function foo()
    nt = (a=1, b=2, c=3)

    nt.a + nt.b * nt.c
end
```

```
julia> @btime foo()
  0.800 ns (0 allocations: 0 bytes)
```

```
function foo()
    rang = 1:3

    rang[1] + rang[2] * rang[3]
end
```

```
julia> @btime foo()
  0.800 ns (0 allocations: 0 bytes)
```

ARRAYS AND SLICES DO ALLOCATE MEMORY

The most common object that allocates memory is arrays. These allocations occur not only when we create an array and assign it to a variable, but also when computations returning arrays are performed on the fly. The following example illustrates this point.

```
foo()  = [1,2,3]
```

```
julia> @btime foo()
  13.714 ns (1 allocation: 80 bytes)
```

Slicing is another operation that creates an array, and therefore allocates. This is due to the default behavior of slicing, which returns a new copy rather than a view of the original object. The sole exception to this rule is when a single element is accessed, in which case no new allocation occurs.

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

foo(x) = x[1:2]                 # ONE allocation, since ranges don't allocate (but 'x[1:2]'
itself does)
```

```
julia> @btime foo($x)
  16.116 ns (1 allocation: 80 bytes)
```

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

foo(x) = x[[1,2]]                    # TWO allocations (one for '[1,2]' and another for
'x[[1,2]]' itself)
```

```
julia> @btime foo($x)
  31.759 ns (2 allocations: 160 bytes)
```

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

foo(x) = x[1] * x[2] + x[3]
```

```
julia> @btime foo($x)
  1.400 ns (0 allocations: 0 bytes)
```

Other operations that involve array creation include array comprehensions and broadcasting. Remarkably, broadcasting even involves memory allocation when intermediate results are computed internally, but not explicitly returned. This specific case is demonstrated in "Broadcasting 2" below.

```
foo()  = [a for a in 1:3]
```

```
julia> @btime foo()
  13.514 ns (1 allocation: 80 bytes)
```

```
x      = [1,2,3]
foo(x) = x .* x
```

```
julia> @btime foo($x)
  15.916 ns (1 allocation: 80 bytes)
```

```
x      = [1,2,3]
foo(x) = sum(x .* x)                # 1 allocation from temporary vector 'x .* x'
```

```
julia> @btime foo($x)
  21.242 ns (1 allocation: 80 bytes)
```

9d. Slice Views to Reduce Allocations

Martin Alfaro
PhD in Economics

INTRODUCTION

In our previous discussion on Slices and Views, we defined the concept of a **slice** as a subvector of the parent vector `x`. Typical examples of slices are expressions such as `x[1:2]` or `x[x .> 0]`. By default, slices create a copy of the data and therefore incurs memory allocation, with the only exception of slices comprising a single object.

Next, we present an approach to avoiding the overhead of memory allocation. This is based on the concept of **views**, which bypass the need for a copy by directly referencing the parent object. This method is particularly effective when slices are indexed through ranges. However, it's not suitable for slices that employ Boolean indexing, in which case allocations will still occur.

Finally, we demonstrate that **copying data could be faster than using views**, despite the additional memory allocation involved. This seeming paradox arises because creating a vector ensures that elements are stored in a contiguous block in memory, which facilitates more efficient access to them.

VIEWS OF SLICES

We start showing that views don't allocate memory *if the slice is indexed by a range*. This property can lead to performance improvements over regular slices, which create a copy by default.

SLICE AS A COPY

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

foo(x) = sum(x[1:2])           # it allocates ONE vector -> the slice 'x[1:2]'
```

```
julia> @btime foo($x)
  15.015 ns (1 allocation: 80 bytes)
```

SLICE AS A VIEW

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

foo(x) = sum(@view(x[1:2]))    # it doesn't allocate
```

```
julia> @btime foo($x)
  1.200 ns (0 allocations: 0 bytes)
```

However, **views under Boolean indexing won't reduce memory allocations or be more performant**. Therefore, don't rely on views of these objects to speed up computations. This fact is illustrated below.

BOOLEAN INDEX (COPY)

```
x = rand(1_000)

foo(x) = sum(x[x .> 0.5])
```

```
julia> @btime foo($x)
  662.500 ns (4 allocations: 8.34 KiB)
```

BOOLEAN INDEX (VIEW)

```
x = rand(1_000)

foo(x) = @views sum(x[x .> 0.5])
```

```
julia> @btime foo($x)
  759.770 ns (4 allocations: 8.34 KiB)
```

COPYING DATA MAY BE FASTER

Although views can reduce memory allocations, there are scenarios where copying data can be the faster approach. This is due to an inherent trade-off between memory allocation and data access patterns. On the one hand, newly created vectors store data in contiguous blocks of memory, enabling more efficient CPU access. On the other hand, while views avoid allocation, they require accessing data scattered throughout memory.

In certain cases, the overhead of creating a copy may be outweighed by the benefits of contiguous memory access, making copying the more efficient choice. This possibility is illustrated below.

COPY

```
x = rand(100_000)

foo(x) = max.(x[1:2:length(x)], 0.5)
```

```
julia> @btime foo($x)
  30.100 μs (4 allocations: 781.34 KiB)
```

VIEW

```
x = rand(100_000)

foo(x) = max.(@view(x[1:2:length(x)]), 0.5)
```

```
julia> @btime foo($x)
  151.700 μs (2 allocations: 390.67 KiB)
```

9e. Pre-Allocations

Martin Alfaro
PhD in Economics

INTRODUCTION

This section explores scenarios where for-loops entail the creation of new vectors in each iteration, which leads to repeated memory allocation. Specifically, we focus on situations where vectors represent intermediate results that don't need to be stored for future use. In such cases, the issue can be addressed by a technique known as pre-allocation.

Pre-allocation involves initializing a vector before the for-loop executes, and then reusing it to temporarily store results during each iteration. By allocating memory upfront and modifying it in place, the approach effectively bypasses the overhead of creating new vectors repeatedly.

The performance gains from pre-allocation can be substantial. Remarkably, the technique isn't specific to Julia, but rather applicable across programming languages. Ultimately, its effectiveness relies on favoring the mutation of pre-allocated memory, which minimizes the reliance on the heap.

The presentation begins by reviewing methods to initialize vectors, which constitutes a prerequisite for pre-allocation. We then present two scenarios where pre-allocation proves advantageous. In particular, one of them highlights the benefits of pre-allocating in the context of nested for-loops.

> **Remark**
> The review of vector initialization will be relatively brief and focused on performance. For more details, I recommend reviewing the section about vector creation, as well as the sections on in-place assignments) and in-place functions).

INITIALIZING VECTORS

Vector initialization refers to the process of creating a vector to subsequently fill it with values. The process typically involves two steps: reserving space in memory and populating the space with some initial values. An efficient way to initialize a vector is by only performing the first step, keeping whatever content is held in the memory address at the moment of creation. Although these values will display a specific number, they are essentially arbitrary and meaningless, explaining why they're referred to as `undef`.

There are two methods for initializing a vector with `undef` values. The first ones requires specifying the type and length of the array, and its syntax resembles the creation of new vectors. The second one is based on the function `similar(y)`, which creates a vector with the same type and dimension as

another existing vector `y`.

Below, we compare the performance of approaches to initializing a vector. In particular, we show that working with `undef` values is faster than setting specific values. To starkly show these differences, we create a vector with 100 elements and repeat the procedure 100,000 times.

```
x           = collect(1:100)
repetitions = 100_000                       # repetitions in a for-loop

function foo(x, repetitions)
    for _ in 1:repetitions
        Vector{Int64}(undef, length(x))
    end
end
```

```
julia> @btime foo($x, $repetitions)
  1.581 ms (100000 allocations: 85.45 MiB)
```

```
x           = collect(1:100)
repetitions = 100_000                       # repetitions in a for-loop

function foo(x, repetitions)
    for _ in 1:repetitions
        similar(x)
    end
end
```

```
julia> @btime foo($x, $repetitions)
  1.623 ms (100000 allocations: 85.45 MiB)
```

```
x           = collect(1:100)
repetitions = 100_000                       # repetitions in a for-loop

function foo(x, repetitions)
    for _ in 1:repetitions
        zeros(Int64, length(x))
    end
end
```

```
julia> @btime foo($x, $repetitions)
  7.530 ms (100000 allocations: 85.45 MiB)
```

```
x           = collect(1:100)
repetitions = 100_000                       # repetitions in a for-loop

function foo(x, repetitions)
    for _ in 1:repetitions
        ones(Int64, length(x))
    end
end
```

```
julia> @btime foo($x, $repetitions)
  4.674 ms (100000 allocations: 85.45 MiB)
```

```
x           = collect(1:100)
repetitions = 100_000                       # repetitions in a for-loop

function foo(x, repetitions)
    for _ in 1:repetitions
        fill(2, length(x))                  # vector filled with integer 2
    end
end
```

```
julia> @btime foo($x, $repetitions)
  4.877 ms (100000 allocations: 85.45 MiB)
```

> **Remark**
>
> Recall that `_` is a convention adopted for denoting **dummy variables**. They're variables that have a value, but aren't used or referenced anywhere in the code. In the context of a for-loop, the sole purpose of `_` is to satisfy the syntax requirements, which expects a variable to iterate over.
>
> The symbol `_` is arbitrary and any other could be used in its place. Throughout the website, we've consistently used `_` when our intention is to repeatedly compute the same operation.

APPROACHES TO INITIALIZING VECTORS

We can initialize `output` by passing it to the function as a keyword argument. This enables using `similar(x)`, where `x` is a previous function's argument. Considering this, the following two implementations turn out to be equivalent.

```
function foo(x)
   output = similar(x)
   # <some calculations using 'output'>
end
```

```
function foo(x; output = similar(x))

   # <some calculations using 'output'>
end
```

When it comes to initializing multiple variables, we can leverage array comprehension to obtain a concise syntax. The only requisite for this approach is that all variables to be initialized share the same type. Below, we additionally present a more efficient approach based on what's known as generators.

This subject is covered in a subsequent section. At this point, you should only know that the method based on generators doesn't allocate. Furthermore, its syntax is similar to array comprehension, with the only difference that brackets `[]` are replaced with parentheses `()`.

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

function foo(x)
    a,b,c = [similar(x) for _ in 1:3]
    # <some calculations using a,b,c>
end
```

```
julia> @btime foo($x)
  49.848 ns (4 allocations: 320 bytes)
```

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

function foo(x)
    a,b,c = (similar(x) for _ in 1:3)
    # <some calculations using a,b,c>
end
```

```
julia> @btime foo($x)
  35.348 ns (3 allocations: 240 bytes)
```

The demonstration uses `similar(x)` as an example, but the same principle applies to other initialization methods such as `Vector{Float64}(undef, length(x))`.

PRE-ALLOCATING VECTORS IN NESTED FOR-LOOPS

When working with vectors, certain operations inherently require the creation of new vectors, whether as intermediate steps or final results. These operations commonly arise with for-loops and broadcasting. The following examples demonstrate this. Note that both approaches in the example create a new vector, even when the operation ultimately yields a scalar value.

```
x = rand(100)

function foo(x)
    output = similar(x)                 # you need to create this vector to store the results

    for i in eachindex(x)
        output[i] = 2 * x[i]
    end

    return output
end
```

```
julia> @btime foo($x)
  45.416 ns (1 allocation: 896 bytes)
```

```
x = rand(100)

foo(x) = sum(2 .* x)                    # 2 .* x implicitly creates a temporary vector
```

```
julia> @btime foo($x)
  36.779 ns (1 allocation: 896 bytes)
```

When the result needs to be stored, allocating a new vector is unavoidable. This is particularly true when the computed result is the final output. However, the operation could serve as an intermediate step in a larger computation, which may involve another for-loop. We refer to these scenarios as **nested for-loops**.

The following example illustrates how each iteration in the second for-loop generates a new vector for the intermediate result.

```
x = rand(100)

function foo(x; output = similar(x))
    for i in eachindex(x)
        output[i] = 2 * x[i]
    end

    return output
end

calling_foo_in_a_loop(output,x) = [sum(foo(x)) for _ in 1:100]
```

```
julia> @btime calling_foo_in_a_loop($x)
  6.160 μs (101 allocations: 88.38 KiB)
```

```
x = rand(100)

foo(x) = 2 .* x

calling_foo_in_a_loop(x) = [sum(foo(x)) for _ in 1:100]
```

```
julia> @btime calling_foo_in_a_loop($x)
  6.433 μs (101 allocations: 88.38 KiB)
```

Scenarios like this lead to unnecessary memory allocations, making them well-suited for pre-allocation of the intermediate result. By adopting this strategy, we can reuse the same vector across iterations, effectively bypassing the memory allocations stemming from creating a new vector multiple times.

To implement it, we need an in-place function that takes the output of the for-loop as one of the arguments. This function will eventually be called iteratively, updating its output in each iteration. There are two ways to implement this strategy, and we analyze each separately in the following.

VIA A FOR-LOOP

The first approach defines an in-place function that updates the values of `output` through a for-loop.

```
x      = rand(100)
output = similar(x)

function foo!(output,x)
    for i in eachindex(x)
        output[i] = 2 * x[i]
    end

    return output
end
```

```
julia> @btime foo!($output, $x)
  5.100 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)
output = similar(x)

function foo!(output,x)
    for i in eachindex(x)
        output[i] = 2 * x[i]
    end

    return output
end

calling_foo_in_a_loop(output,x) = [sum(foo!(output,x)) for _ in 1:100]
```

```
julia> @btime calling_foo_in_a_loop($output, $x)
  1.340 μs (1 allocation: 896 bytes)
```

VIA BROADCASTING

The second option relies on the operator `.=` to update a vector's values. Relative to the example above, this allows for an update through a simpler syntax, where `foo!` is defined in one line.

```
x      = rand(100)
output = similar(x)

foo!(output,x) = (output .= 2 .* x)
```

```
julia> @btime foo!($output, $x)
  5.800 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)
output = similar(x)

foo!(output,x) = (@. output = 2 * x)
```

```
julia> @btime foo!($output, $x)
  5.400 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)
output = similar(x)

foo!(output,x) = (@. output = 2 * x)

calling_foo_in_a_loop(output,x) = [sum(foo!(output,x)) for _ in 1:100]
```

```
julia> @btime calling_foo_in_a_loop($output,$x)
  1.320 μs (1 allocation: 896 bytes)
```

> **Warning!** - Use of @. to update values
>
> When your goal is to update values of a vector, recall that `@.` has to be *placed at the beginning* of the statement.
>
> ```
> # the following are equivalent and define a new variable
>    output  = @. 2  * x
>    output  =    2 .* x
> ```
>
> ```
> # the following are equivalent and update 'output'
> @. output  = 2  * x
>    output .= 2 .* x
> ```

PRE-ALLOCATIONS FOR INTERMEDIATE STEPS

So far, our discussion has centered around the benefits of pre-allocating vectors in nested for-loops. However, its applicability extends beyond this specific scenario.

Broadly speaking, pre-allocating proves useful when: ***i***) the vector serves an intermediate result that feeds into another operation, and ***ii***) the intermediate result is computed inside a for-loop. If these two conditions are met, reusing the same pre-allocated vector outperforms a strategy based on a new vector for each iteration.

Next, we analyze a case where these conditions hold, even though `foo` isn't called in a for-loop as it'd be the case in a nested for-loop. The usefulness of a pre-allocation emerges because `output` demands a complex computation, making it convenient to split the calculation in several steps.

To illustrate the procedure, consider an operation where `temp` represents an intermediate variable to compute `output`. All the implementations below don't pre-allocate `temp`, and hence create a new vector in each iteration.

```
x = rand(100)

function foo(x; output = similar(x))
    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo($x)
  14.700 μs (201 allocations: 11.81 KiB)
```

```
x = rand(100)

foo(x) = [sum(x .> x[i]) for i in eachindex(x)]
```

```
julia> @btime foo($x)
  14.600 μs (201 allocations: 11.81 KiB)
```

```
x = rand(100)

function foo(x)
    temp   = [x .> x[i] for i in eachindex(x)]
    output = sum.(temp)
end
```

```
julia> @btime foo($x)
  15.100 μs (202 allocations: 12.69 KiB)
```

In the following, we pre-allocate `temp`, although the scenario considered exhibits some differences relative to a nested for-loop. These differences result in some subtle aspects regarding their implementation.

First, as we're assuming that this function won't be called in a for-loop, the pre-allocation can be performed within the function, rather than defining `temp` as an argument of `foo`. Second, all the iterations occur within the same for-loop, making the broadcasting option more convenient. The consequences of these differences for the implementation are discussed below.

VIA A FOR-LOOP OR BROADCASTING

Pre-allocating `temp` and update its values via broadcasting is the simplest way for the scenario considered. In fact, this method doesn't require departing from the original syntax. On the contrary, the use a for-loop is more involved.

```
x = rand(100)

function foo!(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
        for j in eachindex(x)
            temp[j] = x[j] > x[i]
        end
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo!($x)
  1.850 µs (2 allocations: 1.75 KiB)
```

```
x = rand(100)

function foo!(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
        temp     .= x .> x[i]
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo!($x)
  1.660 µs (2 allocations: 1.75 KiB)
```

```
x = rand(100)

function foo!(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
        @. temp      = x > x[i]
           output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo!($x)
  1.660 µs (2 allocations: 1.75 KiB)
```

Note that the two allocations observed are due to the creation of `temp` and `output`, which are incurred only once rather than in each iteration.

VIA IN-PLACE FUNCTION

Given the features of the scenario considered, we can also implement the pre-allocation via an in-place function. We refer to it as `update_temp!`, which is defined outside the for-loop and updated in each iteration. An advantage of this approach is that we separate `update_temp!` from the for-loop,

and hence we can focus on `update_temp!` if the operation is performance critical.

```
x = rand(100)

function update_temp!(x, temp, i)
    for j in eachindex(x)
       temp[j] = x[j] > x[i]
    end
end

function foo!(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
        update_temp!(x, temp, i)
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo!($x)
  1.790 μs (2 allocations: 1.75 KiB)
```

```
x = rand(100)

update_temp!(x, temp, i) = (@. temp = x > x[i])

function foo!(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
           update_temp!(x, temp, i)
           output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo!($x)
  1.680 μs (2 allocations: 1.75 KiB)
```

ADDING A NESTED FOR-LOOP

We know combine both cases, where `foo` requires an intermediate variable `temp` and then is called in another for-loop. In such a scenario, both `output` and `temp` needs to be initialized outside the functions and used as function arguments. The following example illustrates this by considering only `update_temp!` using broadcasting.

```
x      = rand(100)



update_temp!(x, temp, i) = (@. temp = x > x[i])

function foo!(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
           update_temp!(x, temp, i)
           output[i] = sum(temp)
    end
    return output
end

calling_foo_in_a_loop(x) = [foo!(x) for _ in 1:1_000]
```

```
julia> @btime calling_foo_in_a_loop($x)
  1.734 ms (2001 allocations: 1.72 MiB)
```

```
x      = rand(100)
output = similar(x)
temp   = similar(x)

update_temp!(x, temp, i) = (@. temp = x > x[i])

function foo!(x, output, temp)
    for i in eachindex(x)
           update_temp!(x, temp, i)
           output[i] = sum(temp)
    end
    return output
end

calling_foo_in_a_loop(x, output, temp) = [foo!(x, output, temp) for _ in 1:1_000]
```

```
julia> @btime calling_foo_in_a_loop($x, $output, $temp)
  1.666 ms (1 allocation: 7.94 KiB)
```

9f. Reductions

Martin Alfaro
PhD in Economics

INTRODUCTION

Reductions are a powerful technique for **operations that take collections as inputs and return a single element**. They arise naturally when we need to compute summary statistics such as an average, variance, or maximum of a collection.

The reduction process works by iteratively applying an operation to pairs of elements, accumulating the results at each step until the final output is obtained. For example, to compute `sum(x)` for a vector `x`, a reduction would start by adding the first two elements, then add the third element to the total, and continue this cumulative process until all elements have been summed.

The method is particularly convenient when we need to transform a vector's elements prior to computing a summary statistic. By operating on scalars, **reductions sidestep the need to materialize intermediate outputs**, thereby reducing memory allocations. This means that, if for example you have to compute `sum(log.(x))`, a reduction would avoid the creation of the intermediate vector `log.(x)`.

INTUITION

Reductions are typically implemented using a for-loop, with an operator applied to pairs of elements and the resulting output updated in each iteration. A classic example is the summation of all numeric elements in a vector. This involves applying the addition operator `+` to pairs of elements, iteratively updating the accumulated sum. This is demonstrated below.

```
x = rand(100)

foo(x) = sum(x)
```

```
julia> foo(x)
48.447
```

```
x = rand(100)

function foo(x)
    output = 0.

    for i in eachindex(x)
        output = output + x[i]
    end

    return output
end
```

```
julia> foo(x)
48.447
```

```
x = rand(100)

function foo(x)
    output = 0.

    for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
48.447
```

In reductions, it's common to see implementations like the last tab, which are based on update operators. This entails that an expression like `x += a` is equivalent to `x = x + a`.

IMPLEMENTING REDUCTIONS

Reductions are implemented by applying either a binary operator or a two-argument function during each iteration. An example of a binary operator is `+`, as we used above. However, we could have also used `+` as a two-argument function, replacing `output = output + x[i]` with `output = +(output, x[i])`. The possibility of using functions broadens the scope of reductions. For instance, it allows us to compute the maximum value of a vector `x` by the function `max`, where `max(a,b)` returns the maximum of the scalars `a` and `b`.

Formally, a reduction requires the binary operation to satisfy **two mathematical requirements**:

- **Associativity**: the way in which operations are grouped does not change the result. For example, addition is associative because `(a + b) + c = a + (b + c)`.
- **Existence of an identity element**: there exists an element that leaves any other element unchanged when the binary operation is applied. For example, the identity element of addition is 0 because `a + 0 = a`.

The following list indicates the identity elements of each operation.

Operation	Identity Element
Sum	0
Product	1
Maximum	-Inf
Minimum	Inf

The relevance of identity elements lies in that they constitute the initial values of the iterative process. Based on these identity elements, we next implement reductions for several operations. The examples make use of the function `foo1` to show the desired outcome, while `foo2` provides the same output via a reduction.

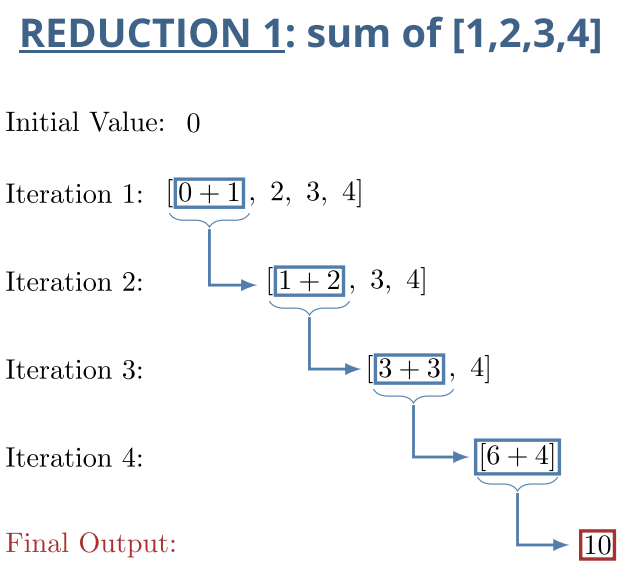
```
x = rand(100)

foo1(x) = sum(x)

function foo2(x)
    output = 0.

    for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
x = rand(100)

foo1(x) = prod(x)

function foo2(x)
    output = 1.

    for i in eachindex(x)
        output *= x[i]
    end

    return output
end
```

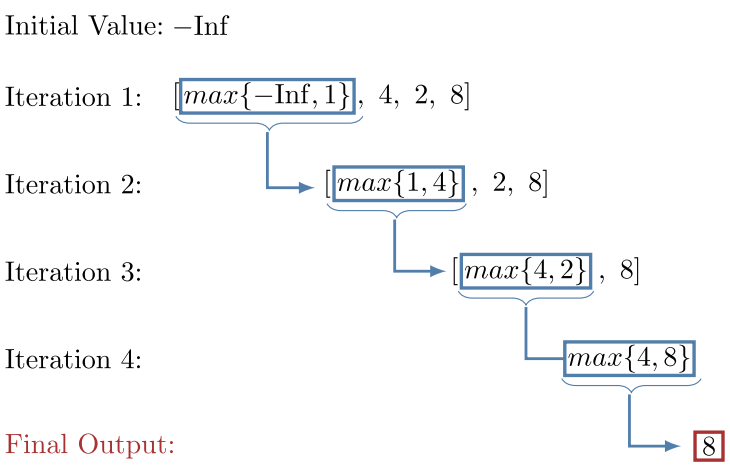
```
x = rand(100)

foo1(x) = maximum(x)

function foo2(x)
    output = -Inf

    for i in eachindex(x)
        output = max(output, x[i])
    end

    return output
end
```

```
x = rand(100)

foo1(x) = minimum(x)

function foo2(x)
    output = Inf

    for i in eachindex(x)
        output = min(output, x[i])
    end

    return output
end
```

We can also visually illustrate how reductions operate in these examples, as we do below.



REDUCTION 2: maximum of [1,4,2,8]



AVOIDING MEMORY ALLOCATIONS VIA REDUCTIONS

One of the primary advantages of reductions is that they avoid the memory allocation of intermediate results. To illustrate this, consider the operation `sum(log.(x))` for a vector `x`. This operation involves transforming `x` into `log.(x)` and then summing the transformed elements. By default, broadcasting creates a new vector to store the result of `log.(x)`, thereby allocating memory for it. However, we're only interested in the final sum and not the intermediate result itself. Therefore, an approach that bypasses the allocation of `log.(x)` is beneficial. Reductions accomplish this by defining a scalar `output`, which is iteratively updated by summing the transformed values of `x`. [1]

```
x = rand(100)

foo1(x) = sum(log.(x))

function foo2(x)
    output = 0.

    for i in eachindex(x)
        output += log(x[i])
    end

    return output
end
```

```
julia> @btime foo1($x)
  315.584 ns (1 allocation: 896 bytes)
julia> @btime foo2($x)
  296.119 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100)

foo1(x) = prod(log.(x))

function foo2(x)
    output = 1.

    for i in eachindex(x)
        output *= log(x[i])
    end

    return output
end
```

```
julia> @btime foo1($x)
  311.840 ns (1 allocation: 896 bytes)
julia> @btime foo2($x)
  296.061 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100)

foo1(x) = maximum(log.(x))

function foo2(x)
    output = -Inf

    for i in eachindex(x)
        output = max(output, log(x[i]))
    end

    return output
end
```

```
julia> @btime foo1($x)
  482.602 ns (1 allocation: 896 bytes)
julia> @btime foo2($x)
  374.961 ns (0 allocations: 0 bytes)
```

```
x = rand(100)

foo1(x) = minimum(log.(x))

function foo2(x)
    output = Inf

    for i in eachindex(x)
        output = min(output, log(x[i]))
    end

    return output
end
```

```
julia> @btime foo1($x)
  487.156 ns (1 allocation: 896 bytes)
julia> @btime foo2($x)
  368.502 ns (0 allocations: 0 bytes)
```

REDUCTIONS VIA BUILT-IN FUNCTIONS

The previous examples implemented reductions through explicit for-loops. Unfortunately, this approach can compromise readability due to the verbosity of for-loops. To address the issue, Julia offers several streamline alternatives to implement reductions.

For common operations, Julia also implements `mapreduce` as additional methods of the functions `sum`, `prod`, `maximum`, and `minimum`. These built-in implementations are more efficient than `mapreduce`, as they've been optimized for each respective case. Their syntax is given by `foo(<transforming function>, x)`, where `foo` is one of the functions mentioned and `x` is the vector to be transformed. For instance, the following examples consider reductions for the transformed vector `2 .* x`.

```
x      = rand(100)

foo(x) = sum(log, x)        #same output as sum(log.(x))
```

```
julia> @btime foo($x)
  294.889 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = prod(log, x)       #same output as prod(log.(x))
```

```
julia> @btime foo($x)
  294.763 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = maximum(log, x)    #same output as maximum(log.(x))
```

```
julia> @btime foo($x)
  579.940 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = minimum(log, x)    #same output as minimum(log.(x))
```

```
julia> @btime foo($x)
  577.516 ns (0 allocations: 0 bytes)
```

While we've used the built-in function `log` for transforming `x`, the approach can be employed through anonymous functions.

```
x      = rand(100)

foo(x) = sum(a -> 2 * a, x)       #same output as sum(2 .* x)
```

```
julia> @btime foo($x)
  6.493 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = prod(a -> 2 * a, x)      #same output as prod(2 .* x)
```

```
julia> @btime foo($x)
  6.741 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = maximum(a -> 2 * a, x)   #same output as maximum(2 .* x)
```

```
julia> @btime foo($x)
  172.547 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = minimum(a -> 2 * a, x)   #same output as minimum(2 .* x)
```

```
julia> @btime foo($x)
  171.490 ns (0 allocations: 0 bytes)
```

Finally, all these functions accept transforming functions that require multiple arguments. To incorporate this possibility, it's necessary to enclose the multiple variables using `zip`, and referring to each variable through indexes. We illustrate this below, where the transforming function is `x .* y`.

```
x = rand(100); y = rand(100)

foo(x,y) = sum(a -> a[1] * a[2], zip(x,y))          #same output as sum(x .* y)
```

```
julia> @btime foo($x)
  29.127 ns (0 allocations: 0 bytes)
```

```
x = rand(100); y = rand(100)

foo(x,y) = prod(a -> a[1] * a[2], zip(x,y))         #same output as prod(x .* y)
```

```
julia> @btime foo($x)
  48.031 ns (0 allocations: 0 bytes)
```

```
x = rand(100); y = rand(100)

foo(x,y) = maximum(a -> a[1] * a[2], zip(x,y))      #same output as maximum(x .* y)
```

```
julia> @btime foo($x)
  172.580 ns (0 allocations: 0 bytes)
```

```
x = rand(100); y = rand(100)

foo(x,y) = minimum(a -> a[1] * a[2], zip(x,y))      #same output as minimum(x .* y)
```

```
julia> @btime foo($x)
  166.969 ns (0 allocations: 0 bytes)
```

THE "REDUCE" AND "MAPREDUCE" FUNCTIONS

Beyond the specific functions we've considered, we can also implement reductions as long as the operation satisfies the requirements for their application. This is implemented through the functions `reduce` and `mapreduce`. Their difference lies in that `reduce` applies the reduction directly, while `mapreduce` transforms the collection's elements prior to doing it.

It's worth remarking that reductions with `sum`, `prod`, `max`, and `min` should still be done via the dedicated functions. The reason is that they've been optimized for their respective tasks. In this context, our primary use case of `reduce` and `mapreduce` is for other types of reductions not covered or when packages provide their own implementations of these functions. [2]

FUNCTION "REDUCE"

The function `reduce` uses the syntax `reduce(<function>,x)`, where `<function>` is a two-argument function. The following example demonstrates its use.

```
x      = rand(100)

foo(x) = reduce(+, x)           #same output as sum(x)
```

```
julia> @btime foo($x)
  6.168 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = reduce(*, x)           #same output as prod(x)
```

```
julia> @btime foo($x)
  6.176 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = reduce(max, x)         #same output as maximum(x)
```

```
julia> @btime foo($x)
  167.905 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = reduce(min, x)         #same output as minimum(x)
```

```
julia> @btime foo($x)
  167.440 ns (0 allocations: 0 bytes)
```

Note that all the examples provided could've been implemented as we did previously, where we directly applied `sum`, `prod`, `maximum` and `minimum`.

FUNCTION "MAPREDUCE"

The function `mapreduce` combines the functions `map` and `reduce`: before applying the reduction, `mapreduce` transforms vectors via the function `map`. Recall that `map(foo,x)` transforms each element of the collection `x` by applying `foo` element-wise. Thus, `mapreduce(<transformation>, <reduction>,x)` first transforms `x`'s elements through `map`, and then applies a reduction to the resulting output.

To illustrate its use, we make use of a `log` transformation.

```
x      = rand(100)

foo(x) = mapreduce(log, +, x)       #same output as sum(log.(x))
```

```
julia> @btime foo($x)
  294.805 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = mapreduce(log, *, x)       #same output as prod(log.(x))
```

```
julia> @btime foo($x)
  294.618 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = mapreduce(log, max, x)     #same output as maximum(log.(x))
```

```
julia> @btime foo($x)
  579.808 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = mapreduce(log, min, x)     #same output as minimum(log.(x))
```

```
julia> @btime foo($x)
  577.505 ns (0 allocations: 0 bytes)
```

Just like with `reduce`, note that the examples could've been implemented directly as we did previously, through the functions `sum`, `prod`, `maximum`, and `minimum`.

`mapreduce` can also be used with anonymous functions and transformations requiring multiple arguments. Below, we illustrate both, whose implementation is the same as with `sum`, `prod`, `maximum`, and `minimum`.

```
x = rand(100); y = rand(100)

foo(x,y) = mapreduce(a -> a[1] * a[2], +, zip(x,y))       #same output as sum(x .* y)
```

```
julia> @btime foo($x)
  29.165 ns (0 allocations: 0 bytes)
```

```
x = rand(100); y = rand(100)

foo(x,y) = mapreduce(a -> a[1] * a[2], *, zip(x,y))       #same output as prod(x .* y)
```

```
julia> @btime foo($x)
  48.221 ns (0 allocations: 0 bytes)
```

```
x = rand(100); y = rand(100)

foo(x,y) = mapreduce(a -> a[1] * a[2], max, zip(x,y))     #same output as maximum(x .* y)
```

```
julia> @btime foo($x)
  175.634 ns (0 allocations: 0 bytes)
```

```
x = rand(100); y = rand(100)

foo(x,y) = mapreduce(a -> a[1] * a[2], min, zip(x,y))     #same output as minimum(x .* y)
```

```
julia> @btime foo($x)
  166.995 ns (0 allocations: 0 bytes)
```

REDUCE OR MAPREDUCE?

`reduce` can be considered as a special case of `mapreduce`, where the latter transforms `x` through the identity function, `identity(x)`. Likewise, `mapreduce(<transformation>,<operator>,x)` produces the same result as `reduce(<operator>, map(<transformation>,x))`. However, `mapreduce` is more efficient, since it avoids the allocation of the transformed vector. This is demonstrated below, where we compute `sum(2 .* x)` through a reduction.

```
x = rand(100)

foo(x) = mapreduce(a -> 2 * a, +, x)
```

```
julia> @btime foo($x)
  6.372 ns (0 allocations: 0 bytes)
```

```
x = rand(100)

foo(x) = reduce(+, map(a -> 2 * a, x))
```

```
julia> @btime foo($x)
  43.476 ns (1 allocation: 896 bytes)
```

FOOTNOTES

[1.] In the section Lazy Operations, we'll explore an alternative with broadcasting that doesn't materialize intermediate results either.

[2.] For instance, the package `Folds` provides a parallelized version of both `map` and `mapreduce`, enabling the utilization of all available CPU cores. Its syntax is identical to Julia's built-in functions.

9g. Static Vectors for Small Collections

Martin Alfaro
PhD in Economics

INTRODUCTION

Due to the memory-allocation overhead involved, the creation of vectors can rapidly become a performance bottleneck. The issue has far-reaching implications, as vector creation not only occurs when we explicitly define a variable holding a new vector. It also happens internally in various scenarios, such as when referencing a slice like `x[1:2]` or computing intermediate results on the fly like `x .* y` in the operation `sum(x .* y)`.

This section introduces a way to address this limitation, while still preserving the use of vectors for collections. The solution leverages the so-called **static vectors**, provided by the `StaticArrays` package. Unlike built-in vectors, which are allocated on the heap, static vectors are stack-allocated.

Internally, static vectors are build on top of tuples. This feature determines that **static vectors are only suitable for collections comprising a few elements**. As a rule of thumb, consider **using static vectors for collections with up to 75 elements**. Exceeding this threshold can lead to increased overhead during creation and access, potentially offsetting any performance benefits or even resulting in a fatal error. [1]

Static vectors offer additional benefits relative to tuples. Firstly, they maintain their performance benefits even at sizes where tuples would typically lose their advantages. Secondly, they're more convenient to work with, as they are manipulated similarly to regular vectors. In fact, they support any array type, including matrices. Finally, the `StaticArrays` package provides mutable variants. This makes static vectors more flexible than tuples, which are only available in an immutable form. It's worth indicating tough that, while the mutable version provides performance benefits relative to regular vectors, the immutable option still offers the best performance.

> **Warning!**
>
> To avoid repetition, **the entire section assumes all collections are small**. Taking this into account, all the benefits highlighted are contingent upon this assumption. We also suppose that the `StaticArrays` package is already available in the workspace, so the command `using StaticArrays` is omitted from each code snippet.

CREATING STATIC VECTORS

The package `StaticArrays` include several variants of static vectors. Our focus in particular we'll be on the type `SVector`, whose objects will be simply referred to as SVectors.

There are two approaches to creating an SVector, each serving a distinct purpose. The first one creates an SVector through literal values, while the other option included converts a standard vector into an SVector. The several implementations of each approach are illustrated below.

```
# all 'sx' define the same static vector '[3,4,5]'


sx = SVector(3,4,5)
sx = SVector{3, Int64}(3,4,5)
sx = SA[3,4,5]
sx = @SVector [i for i in 3:5]
```

```
julia> sx
3-element SVector{3, Int64} with indices SOneTo(3):
 3
 4
 5
```

```
# all 'sx' define a static vector with same elements as 'x'
x = collect(1:10)

sx = SVector(x...)
sx = SVector{length(x), eltype(x)}(x)
sx = SA[x...]
sx = @SVector [a for a in x]
```

```
julia> sx
10-element SVector{10, Int64} with indices SOneTo(10):
  1
  2
  3
  ⋮
  9
 10
```

Of these approaches, we'll primarily rely on the function `SVector`, occasionally employing `SA` for indexing purposes. [2] Note the use of the splatting operator `...` to turn a regular vector into an SVector. This operator is necessary for the function `SVector` to transform a collection into a sequence of arguments. [3]

Regarding slices of SVectors, the approach used for their creation could result in either a regular vector or an SVector. This depends on the indexing employed: a slice remains an SVector when indices are given as SVectors, whereas the slice becomes a regular vector for indices provided by ranges or regular vectors. The sole exception to this rule is when the slice references the whole object (i.e., `sx[:]`), in which case an SVector is returned.

```
x = collect(3:10) ; sx = SVector(x...)

# both define static vectors
slice1 = sx[:]
slice2 = sx[SA[1,2]]        # or slice2 = sx[SVector(1,2)]
```

```
julia> slice1
8-element SVector{8, Int64} with indices SOneTo(8):
  3
  4
  ⋮
  9
 10
julia> slice2
2-element SVector{2, Int64} with indices SOneTo(2):
 3
 4
```

```
x = collect(3:10) ; sx = SVector(x...)

# both define and ordinary vector
slice2 = sx[1:2]
slice2 = sx[[1,2]]
```

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

SVECTORS DON'T ALLOCATE MEMORY AND ARE FASTER

One of the key advantages of SVectors is that they're internally built on top of tuples. Consequently, SVectors don't allocate memory.

```
x = rand(10)

function foo(x)
    a = x[1:2]              # 1 allocation (copy of slice)
    b = [3,4]               # 1 allocation (vector creation)

    sum(a) * sum(b)         # 0 allocation (scalars don't allocate)
end
```

```
julia> @btime foo($x)
  29.819 ns (2 allocations: 160 bytes)
```

```
x = rand(10)

@views function foo(x)
    a = x[1:2]              # 0 allocation (view of slice)
    b = [3,4]               # 1 allocation (vector creation)

    sum(a) * sum(b)         # 0 allocation (scalars don't allocate)
end
```

```
julia> @btime foo($x)
  15.015 ns (1 allocation: 80 bytes)
```

```
x = rand(10);   tup = Tuple(x)

function foo(x)
    a = x[1:2]              # 0 allocation (slice of tuple)
    b = (3,4)               # 0 allocation (tuple creation)

    sum(a) * sum(b)         # 0 allocation (scalars don't allocate)
end
```

```
julia> @btime foo($tup)
  1.400 ns (0 allocations: 0 bytes)
```

```
x = rand(10);   sx = SA[x...]

function foo(x)
    a = x[SA[1,2]]          # 0 allocation (slice of static array)
    b = SA[3,4]             # 0 allocation (static array creation)

    sum(a) * sum(b)         # 0 allocation (scalars don't allocate)
end
```

```
julia> @btime foo($sx)
  1.600 ns (0 allocations: 0 bytes)
```

The decrease in memory allocations from SVectors is especially relevant for operations that result in temporary vectors, such as broadcasting.

```
x = rand(10)

foo(x) = sum(2 .* x)
```

```
julia> @btime foo($x)
  17.936 ns (1 allocation: 144 bytes)
```

```
x = rand(10);   sx = SVector(x...)

foo(x) = sum(2 .* x)
```

```
julia> @btime foo($sx)
  1.800 ns (0 allocations: 0 bytes)
```

Interestingly, the performance benefits of SVectors extend beyond memory allocation. This entails that, even when operations on regular vectors don't allocate memory, SVectors can still provide a speed boost. Below, we demonstrate this through the function `sum(f, <vector>)`, which sums the elements of `<vector>` after they're transformed via `f`. The example shows that the implementation with SVectorsyields faster execution times, even though regular vectors already don't incur memory allocations.

```
x  = rand(10)

foo(x) = sum(a -> 10 + 2a +  3a^2, x)
```

```
julia> @btime foo($x)
  4.400 ns (0 allocations: 0 bytes)
```

```
x  = rand(10);  sx = SVector(x...);

foo(x) = sum(a -> 10 + 2a +  3a^2, x)
```

```
julia> @btime foo($sx)
  2.900 ns (0 allocations: 0 bytes)
```

SVECTOR TYPE AND ITS MUTABLE VARIANT

Similar to tuples, **SVectors are immutable**, meaning that its elements can't be added, removed, or modified. Nevertheless, when mutable collections are needed, the package `StaticArrays` provides a variant given by the type `MVector`. The creation of MVector instances and their slices follow the same syntax as SVectors, with the only difference being the substitution of the function `Svector` with `MVector`. This is illustrated below.

```
x  = [1,2,3]
sx = SVector(x...)

sx[1] = 0
```

```
ERROR: setindex!(::SVector{3, Int64}, value, ::Int) is not defined
```

```
x  = [1,2,3]
mx = MVector(x...)

mx[1] = 0
```

```
julia> mx
3-element MVector{3, Int64} with indices SOneTo(3):
 0
 2
 3
```

The mutability of MVectors makes them ideal for initializing a vector that will eventually be filled via a for-loop. In fact, executing `similar(sx)` when `sx` is an SVector automatically returns an MVector.

▼ 'similar' for SVectors

```
sx = SVector(1,2,3)

mx = similar(sx)        # it defines an MVector with undef elements
```

```
3-element MVector{3, Int64} with indices SOneTo(3):
        2073873450800
 -1152921504606846976
                    0
```

TYPE STABILITY: SIZE IS PART OF THE STATIC VECTORS' TYPE

SVectors are formally defined as objects with type `SVector{N,T}`, where `N` specifies the number of elements and `T` denotes the element's type. For instance, `SVector(4,5,6)` has type `SVector{3,Int64}`, indicating that it comprises 3 elements with type `Int64`. Importantly, this implies that **the number of elements is part of the `SVector` type**. This feature, which is shared with MVectors and inherited from tuples, can readily introduce type instabilities if not handled carefully.

The approaches to ensuring type stability are similar to those employed for tuples. This means that we should either pass SVectors and MVectors as function arguments, or dispatch by the number of elements through the `Val` technique.

```
x = rand(50)

function foo(x)
    output = SVector{length(x), eltype(x)}(undef)
    output = MVector{length(x), eltype(x)}(undef)

    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end

@code_warntype foo(x)                       # type unstable
```

```
x = rand(50);   sx = SVector(x...)

function foo(x)

    output = MVector{length(x), eltype(x)}(undef)

    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end

@code_warntype foo(sx)                      # type stable
```

```
x = rand(50)

function foo(x, ::Val{N}) where N
    sx     = SVector{N, eltype(x)}(x)
    output = MVector{N, eltype(x)}(undef)

    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end

@code_warntype foo(x, Val(length(x)))     # type stable
```

PERFORMANCE COMPARISON

MVectors offer performance benefits over regular vectors. However, you should bear in mind that they're never more performant than SVectors and may additionally result in memory allocations. Considering this, it's recommended to use SVectors if you're certain that the collection won't be mutated.

Below, we illustrate the performance of SVectors and MVectors. The examples demonstrate that they may exhibit similar performance, with the possibility of SVectors being more performant. Furthermore, SVectors and MVectors consistently outperform regular vectors for small collections, regardless of their relative performance. To emphasize this point, we include the benchmark time for the same operation using a regular vector.

```
x  = rand(10)
sx = SVector(x...);  mx = MVector(x...)

foo(x) = sum(a -> 10 + 2a +  3a^2, x)
```

```
julia> @btime foo($x)
  4.400 ns (0 allocations: 0 bytes)
julia> @btime foo($sx)
  2.800 ns (0 allocations: 0 bytes)
julia> @btime foo($mx)
  2.900 ns (0 allocations: 0 bytes)
```

```
x  = rand(10)
sx = SVector(x...);  mx = MVector(x...)

foo(x) = 10 + 2x +  3x^2
```

```
julia> @btime foo.($x)
  19.739 ns (1 allocation: 144 bytes)
julia> @btime foo.($sx)
  1.600 ns (0 allocations: 0 bytes)
julia> @btime foo.($mx)
  6.600 ns (1 allocation: 96 bytes)
```

STATIC VECTORS VS PRE-ALLOCATIONS

Considering the advantages of static vectors over regular vectors, we can now compare their performance to other strategies that reduce memory allocations. In particular, we'll examine how they stack up against pre-allocating memory for intermediate outputs. Our examples demonstrate that static vectors can efficiently store intermediate results, making pre-allocation techniques unnecessary. Moreover, they reveal that storing the final output in an MVector can lead to performance gains over using a regular vector.

For the illustration, we consider a for-loop that requires an intermediate result during each iteration `i`. This involves counting the number of elements in `x` that are greater than `x[i]`, which can be formally implemented as `sum(x .> x[i])`. To make the comparison starker, we isolate the computation of the intermediate step `x .> x[i]`. Note that every implementation below requires pre-allocating the vector `output`, leaving us with only one decision to make: whether to pre-allocate the temporary vector `temp`. This also explains why all implementations involve at least one memory allocation.

```
x = rand(50)

function foo(x; output = similar(x))
    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo($x)
  3.188 µs (101 allocations: 5.17 KiB)
```

```
x = rand(50)

function foo(x; output = similar(x), temp = similar(x))
    for i in eachindex(x)
        @. temp      = x > x[i]
           output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo($x)
  695.745 ns (2 allocations: 992 bytes)
```

```
x = rand(50);   sx = SVector(x...)

function foo(x; output = Vector{Float64}(undef, length(x)))
    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo($sx)
  183.661 ns (1 allocation: 496 bytes)
```

```
x = rand(50);   sx = SVector(x...)

function foo(x; output = similar(x))
    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo($sx)
  148.817 ns (1 allocation: 448 bytes)
```

```
x = rand(50);   sx = SVector(x...)

function foo(x; output = MVector{length(x),eltype(x)}(undef))
    for i in eachindex(x)
        temp      = x .> x[i]
        output[i] = sum(temp)
    end

    return output
end
```

```
julia> @btime foo($sx)
  148.975 ns (1 allocation: 448 bytes)
```

The "No-Preallocation" tab serves as our baseline, providing a reference point for the other methods. As for the "Pre-allocating" tab, it reuses a regular vector to compute `temp`. In contrast, the "SVector" tab converts `x` to an SVector `sx` without pre-allocating `temp`. The benchmarks reveal that latter approach is more performant, thanks to the memory allocation avoidance and additional optimizations provided by SVectors.

As for the last two tabs, they continue defining `x` as an SVector, but additionally treat `output` as an MVector. The last tab in particular does this by using `similar(sx)` to initialize `output`, whereas the other tab explicitly specifies an MVector. Comparing these cases, the benchmarks demonstrate that integrating MVectors into the operation yields further performance gains.

FOOTNOTES

[1.] The recommended number of elements I provide is actually lower than the documentation's suggested (100 elements). The reason for this discrepancy is that as you approach the upper limit, the performance benefits of static vectors compared to regular vectors decrease significantly. As a result, the time spent benchmarking with collections of 100 elements will likely offset any potential advantage.

[2.] The approach based on the macro `@SVector` requires some caveats. For instance, it doesn't support definitions based on local variables, thus precluding the use of `eachindex(x)` in the array comprehension, unless `x` is a global variable.

[3.] For instance, `foo(x...)` is equivalent to `foo(x[1], x[2], x[3])` given a vector or tuple `x` with 3 elements.

9h. Lazy Operations

Martin Alfaro
PhD in Economics

INTRODUCTION

Computational approaches can be broadly classified into "lazy" and "eager" categories. **Eager operations** are characterized by their immediate execution upon definition, providing instant access to the results. So far, most operations on this website have fallen under this category.

In contrast, **lazy operations** define the code to be executed, deferring computation until the results are actually needed. The approach is particularly valuable for operations involving heavy intermediate computations, as lazy evaluation can **sidestep unnecessary memory allocations**: by fusing operations, it becomes possible to perform intermediate calculations on the fly and fed them directly into the final calculation.

This section provides various implementations for lazy computations. The first approach presented is based on the so-called **generators**, which are the lazy analogous of array comprehensions. After this, we'll introduce several functions from the package **Iterators**, which provides lazy implementations of functions like `map` and `filter`.

GENERATORS

Array comprehensions offer a convenient technique for creating vectors, using a similar syntax to for-loops to define their elements. These elements are computed and stored right away, making array comprehensions an eager operation. For their part, **generators** represent **the lazy counterpart of array comprehensions**, deferring the creation of elements until they're actually needed.

In terms of syntax, generators are identical to array comprehensions, with the sole difference they're enclosed in parentheses `( )` instead of square brackets `[ ]`. Just like array comprehensions, generators also retain the ability to add conditions and simultaneous iterate over multiple collections.

```
x = [a for a in 1:10]

y = [a for a in 1:10 if a > 5]
```

```
julia> x
10-element Vector{Int64}:
  1
  2
  ⋮
  9
 10
julia> y
5-element Vector{Int64}:
  6
  7
  8
  9
 10
```

```
x = (a for a in 1:10)

y = (a for a in 1:10 if a > 5)
```

```
julia> x
Base.Generator{UnitRange{Int64}, typeof(identity)}(identity, 1:10)
```

The examples show that array comprehensions compute all their elements at the moment of defining the vector, giving immediate access to them. In contrast, generators formally define an object with type `Base.Generator`, where operations are described, but no output is materialized.

This characteristic of generators makes them particularly useful for computing reductions with transformed values. By producing values on-demand and fusing them with the reduction function, generators avoid the materialization of temporary vectors, thus reducing memory allocations.

To illustrate the performance benefits this entails, let's compute the sum of all elements in a vector `y`. In particular, `y` is obtained by doubling each element of a vector `x`. One way to compute this operation is by first creating the vector `y` and then sum all its elements. Alternatively, we can describe the transformation through a generator, which bypasses the storage of the intermediate output `y` and instead feeds the transformation directly into the `sum` function. This allows the compiler to perform the addition as a cumulative operation on scalars, thereby reducing memory usage.

```
x = rand(100)

function foo(x)
    y = [a * 2 for a in x]                  # 1 allocation (same as y = x .* 2)

    sum(y)
end
```

```
julia> @btime foo($x)
  46.945 ns (1 allocation: 896 bytes)
```

```
x = rand(100)

function foo(x)
    y = (a * 2 for a in x)                  # 0 allocations

    sum(y)
end
```

```
julia> @btime foo($x)
  23.996 ns (0 allocations: 0 bytes)
```

```
x = rand(100)


foo(x) = sum(a * 2 for a in x)              # 0 allocations
```

```
julia> @btime foo($x)
  23.996 ns (0 allocations: 0 bytes)
```

The last tab shows that generators can be incorporated directly as a function argument, resulting in a compact syntax. Remarkably, this syntax is applicable to *any* function that accepts a collection as its input.

ITERATORS

Iterators are formally defined as lazy objects that create sequential values on demand, rather than storing them all in memory upfront. Throughout the website, we've already encountered numerous scenarios involving iterators. A typical example of an iterator is a range, such as `1:length(x)`, which defines a sequence of numbers to be generated on the fly. Their lazy evaluation explains why the function `collect` is needed when we want to materialize the entire sequence into a vector. Without `collect`, iterators merely describe the numbers to be created, without actually creating and storing them in memory.

Beyond simple ranges, we've also covered other types of iterators hat offer more specialized functionality. They included `eachindex` for accessing array indices, `enumerate` for pairing elements with their positions, and `zip` for combining multiple sequences.

The lazy nature of iterators makes them particularly efficient in for-loops: by generating each value as the for-loop progresses, we eliminate unnecessary memory allocations that would arise from materializing the list being iterated over.

```
x = 1:10
```

```
julia> x
1:10
```

```
x = collect(1:10)
```

```
julia> x
10-element Vector{Int64}:
  1
  2
  ⋮
  9
 10
```

The built-in package `Iterators`, which is automatically "imported" in every Julia session, provides multiple functions for generating lazy sequences. Additionally, it offers lazy counterparts of various functions such as `filter` and `map`, which can be accesed as `Iterators.filter` and `Iterators.map`. [1]

The following example demonstrates the use of these functions to avoid memory allocations of intermediate computations.

```
x = collect(1:100)

function foo(x)
    y = filter(a -> a > 50, x)              # 1 allocation

    sum(y)
end
```

```
julia> @btime foo($x)
  53.163 ns (1 allocation: 896 bytes)
```

```
x = collect(1:100)

function foo(x)
    y = Iterators.filter(a -> a > 50, x)    # 0 allocations

    sum(y)
end
```

```
julia> @btime foo($x)
  55.239 ns (0 allocations: 0 bytes)
```

```
x = rand(100)

function foo(x)
    y = map(a -> a * 2, x)                  # 1 allocation

    sum(y)
end
```

```
julia> @btime foo($x)
  47.963 ns (1 allocation: 896 bytes)
```

```
x = rand(100)

function foo(x)
    y = Iterators.map(a -> a * 2, x)        # 0 allocations

    sum(y)
end
```

```
julia> @btime foo($x)
  23.972 ns (0 allocations: 0 bytes)
```

FOOTNOTES

[1] The `IterTools` package further extends the functionality of `Iterators`, offering even more tools for working with lazy sequences.

9i. Lazy Broadcasting and Loop Fusion

Martin Alfaro
PhD in Economics

INTRODUCTION

This section continues the analysis of lazy and eager operations as a means of reducing memory allocations. The focus now shifts to broadcasting operations, which strike a balance between code readability and performance.

Central to the upcoming discussion is the eager default behavior of broadcasting in Julia. This means that broadcasted operations compute their outputs immediately upon execution, thereby inevitably leading to memory allocation when applied to allocating-objects like vectors. This characteristic is particularly relevant in scenarios involving multiple intermediate broadcasted operations, resulting in multiple temporary, and therefore avoidable, memory allocations.

Next, we present various approaches to reduce allocations in such scenarios. We'll start highlighting the notion of **loop fusion**, which allows multiple broadcasting operations to be combined into a more efficient single operation. After this, we'll explore the `LazyArrays` package, which evaluates broadcasting operations in a lazy manner. The technique is particularly useful for reductions requiring intermediate transformations, entirely circumventing memory allocations for these intermediates.

HOW DOES BROADCASTING WORK?

To gain a deeper understanding of the optimizations we'll be discussing, let's first examine the internal mechanics of broadcasting. Under the hood, broadcasting operations are converted into optimized for-loops during compilation, rendering the two approaches computationally equivalent. In this context, broadcasting serves as syntactic sugar, eliminating the need for explicit for-loops. This allows developers to write more concise and expressive code, without compromising performance.

Despite the equivalence between broadcasting and for-loops, you'll often notice performance differences in practice. These discrepancies are largely driven by compiler optimizations, rather than inherent differences between the two approaches. Essentially, the fact that an operation supports a broadcasted form reveals further information about its underlying structure, allowing the compiler to automatically apply certain optimizations. In contrast, the generality of for-loops precludes this possibility, as the same assumptions can't be taken for granted. Nevertheless, with careful manual optimization, for-loops can always match or surpass the performance of broadcasting, thanks to their greater flexibility and potential for fine-tuning.

The following code snippets demonstrate the equivalence between the two approaches. The first tab describes the operation being performed, while the second tab provides a rough translation of broadcasting's internal implementation. The third tab further highlights this equivalence, by providing the exact code used. This requires including the `@inbounds` macro in the for-loop, which is automatically applied with broadcasting. The specific effect of this macro on computations can be disregarded, as it'll be discussed in a later section. Its sole purpose here is to illustrate the equivalence between the two approaches.

```
x      = rand(100)

foo(x) = 2 .* x
```

```
julia> @btime foo($x)
  34.506 ns (1 allocation: 896 bytes)
```

```
x      = rand(100)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = 2 * x[i]
    end

    return output
end
```

```
julia> @btime foo($x)
  48.188 ns (1 allocation: 896 bytes)
```

```
x      = rand(100)

function foo(x)
    output = similar(x)

    @inbounds for i in eachindex(x)
        output[i] = 2 * x[i]
    end

    return output
end
```

```
julia> @btime foo($x)
  32.832 ns (1 allocation: 896 bytes)
```

> **Warning!** - About `@inbounds`
>
> In the example provided, `@inbounds` **was added to illustrate the internal implementation of broadcasting, rather than a recommended practice for general use**. In fact, `@inbounds` used incorrectly **can lead to severe issues**.

> To understand what this macro does, Julia by default enforces bounds checking on array indices to prevent out-of-range access (e.g., it checks that a vector `x` with 3 elements isn't accessed at index `x[4]`). Adding `@inbounds` to a for-loop instructs Julia to bypass this check, thus speeding up computations. However, it simultaneously makes our code unsafe, including the possibility of returning incorrect results and other more pronounced issues.

A key implication of the example is that Julia's broadcasting is eager by default, meaning that the result is immediately computed and stored. In the example, this is reflected in `2 .* x` being computed and stored in `output`, which also explains the observed memory allocation.

Importantly, **memory allocations under broadcasting arise even if the result isn't explicitly stored**. For example, computing `sum(2 .* x)` involves the computation and temporary storage of `2 .* x`, before the sum is performed.

REMARK (OPTIONAL) Differing Optimizations With Broadcasting and For-Loops

BROADCASTING: LOOP FUSION

While eager broadcasting makes results readily available, their outputs may not be important in themselves. Instead, they could represent intermediate steps in a larger computation, with these outputs eventually being passed as inputs to subsequent operations.

In the following, we address scenarios like this, where broadcasting is employed for intermediate results. The first approach we explore leverages a technique called **loop fusion**, which combines multiple broadcasting operations into a single loop. By doing so, the compiler can perform all operations in a single pass over the data. This not only eliminates the creation of multiple intermediate vectors, but also provides the compiler with a holistic view of the operations, thus allowing for further optimizations.

Remarkably, when all broadcasting operations are nested within a single operation, the compiler automatically implements loop fusion. However, for complex expressions, writing a single lengthy expression can be impractical. To overcome this limitation, we'll show a method that enables us to break down operations into partial calculations, while still preserving loop fusion. This approach is based on the lazy design of functions definitions, making it possible to delay operations until their eventual combination.

```
x       = rand(100)

foo(x)  = x .* 2 .+ x .* 3      # or @. x * 2 + x * 3
```

```
julia> @btime foo($x)
  35.361 ns (1 allocation: 896 bytes)
```

```
x        = rand(100)

function foo(x)
    a      = x .* 2
    b      = x .* 3

    output = a .+ b
end
```

```
julia> @btime foo($x)
  124.420 ns (3 allocations: 2.62 KiB)
```

```
x        = rand(100)

term1(a) = a * 2
term2(a) = a * 3

foo(a)   = term1(a) + term2(a)
```

```
julia> @btime foo.($x)
  34.330 ns (1 allocation: 896 bytes)
```

VECTOR OPERATIONS ALLOCATE AND BREAK LOOP FUSION

A common scenario where loop fusion is prevented is when a single expression combines broadcasting and vector operations. This possibility arises because **some vector operations yield similar results to their broadcasting equivalents**, making it possible to combine these operations without dimensional mismatches. For instance, we show below that adding two vectors using `+` produces the same result as summing them element-wise by employing `.+`.

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

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

```
julia> foo(x,β)
3-element Vector{Int64}:
 5
 7
 9
```

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

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

```
julia> foo(x,β)
3-element Vector{Int64}:
 5
 7
 9
```

The same occurs with a vector product when one of the operands is a scalar.

```
x        = [1, 2, 3]
β        = 2

foo(x,β) = x .* β
```

```
julia> foo(x,β)
3-element Vector{Int64}:
 2
 4
 6
```

```
x        = [1, 2, 3]
β        = 2

foo(x,β) = x * β
```

```
julia> foo(x,β)
3-element Vector{Int64}:
 2
 4
 6
```

OMITTING DOTS AVOIDS LOOP FUSION

Mixing vector operations and broadcasting is problematic for performance. The reason is that each vector operation will allocate memory, in a context that operations aren't fused. In particular, the issue arises when the following conditions are met:

- The final output requires combining multiple operations
- The operations yield the same result whether implemented through broadcasting or a vector operation
- We mix broadcasting and vector implementations, by omitting the inclusion of some dots

If all these conditions are satisfied, Julia will partition the operations and compute each separately. The consequence of this is the emergence of multiple temporary vectors, with each separately allocating memory.

The following example illustrates this possibility in the extreme case where *all* broadcasting dots `.` are omitted. It demonstrates that vector operations aren't fused, even when expressed in a single operation. Moreover, it establishes that vector operations are similar to obtaining the final result by separately calculating each operation.

```
x = rand(100)

foo(x)  = x * 2 + x * 3
```

```
julia> @btime foo($x)
  129.269 ns (3 allocations: 2.62 KiB)
```

```
x = rand(100)

function foo(x)
    term1  = x * 2
    term2  = x * 3

    output = term1 + term2
end
```

```
julia> @btime foo($x)
  130.798 ns (3 allocations: 2.62 KiB)
```

While the previous example exclusively consists of vector operations, the same principle applies when mixing broadcasting and non-broadcasting operations. In such cases, loop fusion is partially achieved, with only a subset of operations being internally computed through a single for-loop.

```
x      = rand(100)

foo(x) = x * 2 .+ x .* 3
```

```
julia> @btime foo($x)
  85.034 ns (2 allocations: 1.75 KiB)
```

```
x      = rand(100)

function foo(x)
    term1  = x * 2

    output = term1 .+ x .*3
end
```

```
julia> @btime foo($x)
  85.763 ns (2 allocations: 1.75 KiB)
```

Overall, the key takeaway from these examples is that **guaranteeing loop fusion requires appending a dot to every operator and function to be broadcasted**. Note that this can be error-prone, especially in large expressions where a single missing dot can be easily overlooked.

Fortunately, there are two alternatives that mitigate this risk.

One option is to prefix the expression with `@.`, as shown in the tab `Equivalent 1` below. This ensures that *all* operators and functions are broadcasted. Alternatively, all operations could be combined into a *scalar* function, which you eventually broadcast. This is presented in the tab `Equivalent 2` below.

```
x      = rand(100)

foo(x) = x .* 2 .+ x .* 3
```

```
julia> @btime foo($x)
  36.456 ns (1 allocation: 896 bytes)
```

```
x      = rand(100)

foo(x) = @. x * 2 + x * 3
```

```
julia> @btime foo($x)
  36.573 ns (1 allocation: 896 bytes)
```

```
x      = rand(100)

foo(a) = a * 2 + a * 3
```

```
julia> @btime foo.($x)
  34.536 ns (1 allocation: 896 bytes)
```

When multiple long operations are combined, the need to split operations is inevitable. In this case, we can apply a similar trick as we did before, where we leverage that function definitions are inherently lazy. Specifically, this allows us to achieve loop fusion by defining each operation as a separate scalar function.

▼ Loop Fusion Splitting Operations

```
x        = rand(100)

term1(a) = a * 2
term2(a) = a * 3

foo(a)   = term1(a) + term2(a)
```

```
julia> @btime foo.($x)
  35.346 ns (1 allocation: 896 bytes)
```

LAZY BROADCASTING

To handle intermediate computations, we can also transform broadcasting into a lazy operation. Such functionality is provided by the `LazyArrays` package, whose use requires prepending the `@~` macro to the broadcasting operation. Similar to a function definition, lazy broadcasting defers the actual computation of the operation until it's needed.

```
x         = rand(100)

function foo(x)
    term1  = x .* 2
    term2  = x .* 3

    output = term1 .+ term2
end
```

```
julia> @btime foo($x,$y)
  109.803 ns (3 allocations: 2.62 KiB)
```

```
x         = rand(100)

function foo(x)
    term1  = @~ x .* 2
    term2  = @~ x .* 3

    output = term1 .+ term2
end
```

```
julia> @btime foo($x,$y)
  37.304 ns (1 allocation: 896 bytes)
```

Note that, up to this point, all the cases considered had a vector as its end result. In this context, our goal becomes to reduce memory allocations to the single unavoidable allocation, which is given by storage of the final result.

Instead, when scalar values are the final output, lazy broadcasting offers a significant advantage: it enables us to completely remove memory allocations. This is because lazy broadcasting fuses broadcasting and reduction operations, allowing the former to be computed on-the-fly. This is illustrated in the example provided below.

```
# eager broadcasting (default)
x      = rand(100)

foo(x) = sum(2 .* x)
```

```
julia> @btime foo($x,$y)
  48.012 ns (1 allocation: 896 bytes)
```

```
using LazyArrays
x      = rand(100)

foo(x) = sum(@~ 2 .* x)
```

```
julia> @btime foo($x,$y)
  7.906 ns (0 allocations: 0 bytes)
```

Note that completely eliminating allocations can't be accomplished simply by using functions. This is because functions enable the splitting of broadcasting operations, but do not fuse them with reduction operations.

```
x = rand(100)

term1(a) = a * 2
term2(a) = a * 3
temp(a)  = term1(a) + term2(a)

foo(x)   = sum(temp.(x))
```

```
julia> @btime foo($x,$y)
  48.307 ns (1 allocation: 896 bytes)
```

```
x = rand(100)

term1(a) = a * 2
term2(a) = a * 3
temp(a)  = term1(a) + term2(a)

foo(x)   = sum(@~ temp.(x))
```

```
julia> @btime foo($x,$y)
  10.474 ns (0 allocations: 0 bytes)
```

```
x = rand(100)

function foo(x)
    term1  = @~ x .* 2
    term2  = @~ x .* 3
    temp   = @~ term1 .+ term2

    output = sum(temp)
end
```

```
julia> @btime foo($x,$y)
  13.766 ns (0 allocations: 0 bytes)
```

> **Remark**
>
> An additional advantage of `@~` is that it performs additional optimizations when possible. As a result, you'll typically observe that

`@~` is faster than alternatives like a lazy map, despite that neither allocates memory. This performance benefit can be appreciated in the following comparison.

```
x = rand(100)

term1(a) = a * 2
term2(a) = a * 3
temp(a)  = term1(a) + term2(a)

foo(x)   = sum(@~ temp.(x))
```

```
julia> @btime foo($x,$y)
  10.474 ns (0 allocations: 0 bytes)
```

```
x = rand(100)

term1(a) = a * 2
term2(a) = a * 3
temp(a)  = term1(a) + term2(a)

foo(x)   = sum(Iterators.map(temp, x))
```

```
julia> @btime foo($x,$y)
  28.909 ns (0 allocations: 0 bytes)
```

10a. Overview and Goals

Martin Alfaro
PhD in Economics

INTRODUCTION

The previous chapters started our study of techniques for improving performance. The focus was in particular on type stability and memory allocations, which are not only critical for achieving optimal performance, but also universally applicable. However, specific applications can often benefit from more specialized strategies. This chapter takes a step in this direction by introducing some of these techniques. In particular, we offer two key insights that extend beyond any particular application, and therefore can be applied broadly.

First, any of the new techniques we'll introduce **inherently involve trade-offs**. This occurs because, after applying fundamental optimizations to operate at the performance frontier, any further gains can only be achieved at the expense of precision, safety, or generality. This stands in contrast to fundamental optimizations like type stability and reduced memory allocations, which may hinder readability but don't entail compromises in other respects. The presence of these trade-offs also explains why the techniques aren't part of Julia's default implementation, which consistently prioritizes correctness and safety over speed.

The second important takeaway is related to the concept of **code transformation via macros**. This represents a general strategy that allows developers to implement sophisticated computational algorithms, without requiring users to grasp the underlying complexities for their applications. Macros are particularly well-suited suited for this purpose, as they essentially take expressions in a code and modifies it before compilation. This makes it possible, for example, to identify all operations within a for-loop, subsequently adapting the algorithm for a more efficient computation.

10b. Macros as a Means for Optimizations

Martin Alfaro
PhD in Economics

INTRODUCTION

Customized computational approaches often have an edge over general-purpose built-in solutions, as they can tackle the unique challenges of a given scenario. However, the complexity of these specialized techniques often deters their adoption among practitioners, who may lack the necessary expertise to implement them. Macros offer a practical solution to bridge this gap, making specialized computational approaches more accessible to users. They're particularly well-suited for this purpose due to their ability to take entire code blocks as inputs and transform them into an optimized execution approach. This capability allows practitioners to benefit from specialized algorithms, without the need to implement them themselves.

In the upcoming sections, the role of macros in boosting performance will be central. By leveraging them, we'll be able to effectively separate the benefits provided by an algorithm from its actual implementation details. This decoupling will let us shift our focus from the nitty-gritty details of how to implement algorithms, to the more practical question of when to apply them. The current section in particular will concentrate on the procedure for applying macros, paying special attention to some subtle considerations arising in practice.

ABOUT MACROS FOR OPTIMIZATIONS

Macros bear a resemblance to functions in that they take an input and return an output. Their primary difference lies in that macros take an entire code block as their input, possibly yielding another code block as its output.

This unique feature enables macros to be applied for tasks that functions can't handle. One common application is **code simplification**. By automating repetitive tasks and eliminating redundant code, macros are capable of significantly improving code readability. For instance, suppose a function requires multiple slices of `x` to be converted into views. Without macros, this would involve repeatedly invoking `view(x, <indices>)`, resulting in verbose and error-prone code. Instead, prepending the function definition with `@views` will automatically handle all the slice conversions for us.

Another application of macros is **to modify how operations are computed**, which is the focus of this section. This functionality allows developers to package sophisticated optimization techniques, making advanced solutions accessible to users. In this context, users who might not be familiar with the underlying complexities of the method, only need to focus on selecting the most suitable computational approach, rather than grappling with implementation details.

While macros are powerful tools, they're not without their limitations. Their black-box nature means that incorrect usage can lead to unexpected results or compromise computational safety. That's why it's important to identify the suitable scenarios of each macro. Although this requires some initial investment, it's considerably less demanding than implementing the functionality from scratch.

APPLYING MACROS IN FOR-LOOPS: @INBOUNDS AS AN EXAMPLE

One distinctive feature of Julia is its ability to execute for-loops with exceptional speed. In fact, carefully optimized for-loops tend to reach the highest possible performance within the language. This efficiency stems from the versatility of for-loops, which lets users fine-tune them for their specific needs. As a result, it's no surprise that one prominent application of macros is to implement specific computational approaches for for-loops.

To illustrate this use, let's consider the `@inbounds` macros. Although strictly speaking this doesn't implement a new computational approach, it does modify how for-loops are executed. Additionally, it's simple enough to easily illustrate this role of macros.

To appreciate the impact of `@inbounds`, we first need to understand how for-loops typically behave in Julia. By default, the language implements **bounds checking**: when an element `x[i]` is accessed during the *i*-th iteration, Julia verifies that *i* falls within the valid range of indices for `x`. This built-in mechanism safeguards against errors and security issues caused by out-of-bounds access.

While bounds checking prevents bugs, it comes at a performance cost: these additional checks not only introduce computational overhead, but also limit the compiler's ability to implement certain optimizations. However, there are situations where iterations are guaranteed to stay within array bounds. In those cases, we can safely boost performance by disabling bounds checking through the `@inbounds` macro.

> **Trade-Offs Entailed by @inbounds**
>
> The `@inbounds` macro perfectly illustrates both the power and risks associated with macro usage. When applied judiciously, it can yield substantial performance gains, especially when multiple slices are involved.
>
> However, disabling bounds checking simultaneously renders code unsafe: it increases the risk of crashes and silent errors, additionally creating security vulnerabilities. In this context, `@inbounds` shifts the responsibility of applying the macro onto the user, who must be absolutely certain that the iteration range is within the arrays' bounds.

ILLUSTRATING @INBOUNDS

Broadly speaking, using a macro within a for-loop to modify its computational approach requires its addition at the beginning of the for-loop. For instance, to disable bounds checking for every indexed element within a for-loop, we simply need to prepend the for-loop with `@inbounds`. We can alternatively apply `@inbounds` individually to any specific line within the loop. Nonetheless, this possibility is specific to `@inbounds`, only arising because the macro can actually be employed even outside for-loops.

```
x = rand(1_000)

function foo(x)
    output = 0.

    @inbounds for i in eachindex(x)
                  a      = log(x[i])
                  b      = exp(x[i])
                  output += a / b
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  5.002 µs (0 allocations: 0 bytes)
```

```
x = rand(1_000)

function foo(x)
    output = 0.

    for i in eachindex(x)
        @inbounds a      = log(x[i])
        @inbounds b      = exp(x[i])
                  output += a / b
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  5.093 µs (0 allocations: 0 bytes)
```

The performance advantages of `@inbounds` don't only come from the elimination of bounds checking itself. Bounds checking is a form of conditional, where the iteration is executed contingent on all indices being within range. In the next sections, we'll see that conditional statements commonly limit the compiler's ability to apply further optimizations. Once we remove these checks, you give the compiler more leeway to implement additional enhancements.

To illustrate such possibility, the next example shows that the application of `@inbounds` triggers the so-called SIMD instructions. They're a form of parallelism within a core and will be explored in the upcoming sections.

```
v,w,x,y = (rand(100_000) for _ in 1:4)       # it assigns random vectors to v,w,x,y

function foo(v, w, x, y)
    output = 0.0

    for i in 2:length(v)-1
        output += v[i-1] / v[i+1] / w[i-1] * w[i+1] + x[i-1] * x[i+1] / y[i-1] * y[i+1]
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  231.242 μs (0 allocations: 0 bytes)
```

```
v,w,x,y = (rand(100_000) for _ in 1:4)       # it assigns random vectors to v,w,x,y

function foo(v, w, x, y)
    output = 0.0

    @inbounds for i in 2:length(v)-1
        output += v[i-1] / v[i+1] / w[i-1] * w[i+1] + x[i-1] * x[i+1] / y[i-1] * y[i+1]
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  154.179 μs (0 allocations: 0 bytes)
```

Warning! - Function Calls in For-Loop Bodies Can Disable Macro Effects

The use of functions without direct reference to slices could prevent the application of `@inbounds`. This can be observed below, where we compare approaches with and without `@inbounds` when a function is involved.

```
v,w,x,y = (rand(100_000) for _ in 1:4) # it assigns random
vectors to v, w, x, y
compute(i, v,w,x,y) = v[i-1] / v[i+1] / w[i-1] * w[i+1] +
                      x[i-1] * x[i+1] / y[i-1] * y[i+1]

function foo(v,w,x,y)
    output = 0.0

    @inbounds for i in 2:length(v)-1
        output += compute(i, v,w,x,y)

    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  271.897 μs (0 allocations: 0 bytes)
```

```
v,w,x,y = (rand(100_000) for _ in 1:4) # it assigns random
vectors to v, w, x, y



function foo(v,w,x,y)
    output = 0.0

    @inbounds for i in 2:length(v)-1
        output += v[i-1] / v[i+1] / w[i-1] * w[i+1] +
                  x[i-1] * x[i+1] / y[i-1] * y[i+1]
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  154.194 μs (0 allocations: 0 bytes)
```

MACROS COULD BE DISREGARDED OR APPLIED AUTOMATICALLY BY THE COMPILER

The influence of macros on code execution is complex. In many cases, macros might have no impact at all because compilers ultimately decide the best strategy for the problem at hand. Thus, they could already be implementing the functionality we suggest through the macro, or simply disregard it entirely. The lack of any discernible impact is easily inferred through execution times, which remain unchanged with and without the macro.

This occurs with the `@inbounds` macro, in cases where compiler is already skipping bound checks. This is only implemented automatically by the compiler in very simple cases, such as when we define iterations by `eachindex`. In such scenarios, the compiler can recognize that memory access is safe and automatically disable bounds checking, rendering the `@inbounds` macro redundant.

```
x = rand(1_000)

function foo(x)
    output = 0.

    for i in eachindex(x)
        output += log(x[i])
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  3.151 µs (0 allocations: 0 bytes)
```

```
x = rand(1_000)

function foo(x)
    output = 0.

    @inbounds for i in eachindex(x)
        output += log(x[i])
    end

    return output
end
```

```
julia> @btime foo($v,$w,$x,$y)
  3.098 µs (0 allocations: 0 bytes)
```

Macros could also serve as a mere hint to the compiler, without dictating its use. In such scenarios, the hint provided indicates that certain assumptions are met, allowing the compiler to implement more aggressive optimizations. The compiler will then carefully analyze the operations involved and decide whether the suggested approach is actually beneficial. If it is, it'll apply the optimizations. If not, it'll disregard the hint and opt for a different approach. This determines that macros guide the compiler towards better performance, but without imposing strict directives.

An example along these lines is `@simd`, which suggests the application of SIMD instructions a technique that we'll be explored in the next sections. When `@simd` is introduced, the compiler maintains complete autonomy in deciding whether to implement the suggested optimization. In fact, it'll only adopt SIMD instructions if it concludes that they'll potentially improve performance in the specific application. In the following example, `@simd` is ignored by the compiler, explaining why the execution time remains the same with and without the macro. [1]

```
x = rand(2_000_000)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = if (200_000 > i > 100_000)
                        x[i] * 1.1
                    else
                        x[i] * 1.2
                    end
    end

    return output
end
```

```
julia> @btime foo($x)
  1.056 ms (2 allocations: 15.26 MiB)
```

```
x = rand(2_000_000)

function foo(x)
    output = similar(x)

    @simd for i in eachindex(x)
        output[i] = if (200_000 > i > 100_000)
                        x[i] * 1.1
                    else
                        x[i] * 1.2
                    end
    end

    return output
end
```

```
julia> @btime foo($x)
  1.066 ms (2 allocations: 15.26 MiB)
```

FOOTNOTES

[1] The fact that the code implemented is the same is confirmed by inspecting the internal code executed.

10c. Introduction to SIMD

Martin Alfaro
PhD in Economics

INTRODUCTION

Single Instruction, Multiple Data (SIMD) is an optimization technique widely embraced in modern CPU architectures. At its core, SIMD allows a single CPU instruction to process multiple data points concurrently instead of sequentially processing each one individually. This approach can yield substantial performance gains, particularly for workloads involving simple identical calculations repeated across multiple data elements. [1]

To illustrate the power of SIMD, consider a computation consisting of four separate addition operations. Without SIMD, you'd have to execute four distinct instructions, one for each addition. Instead, SIMD makes it possible to bundle the four additions into a single instruction, with the CPU processing them all at once. In an ideal scenario, the time required to complete four additions using SIMD or one addition without it could be the same.

The efficiency of SIMD comes from its ability to leverage parallelism within a single CPU core. By operating on vectors rather than individual elements, SIMD instructions can execute the same operation on multiple data simultaneously. This is why SIMD is usually referred to as **vectorization**.

Throughout the sections, we'll cover two approaches for implementing SIMD instructions.

- Julia's native capabilities.
- The package `LoopVectorization`.

In this section, we'll concentrate on the built-in tools for applying SIMD. In particular, we'll present conditions that trigger its automatic application. Furthermore, we'll introduce the `@simd` macro, which allows for manual implementation in for-loops. Instead, the study of `LoopVectorization` is relegated to subsequent sections. As we'll see, this package implements more aggressive optimizations, relative to Julia's base.

WHAT IS SIMD?

SIMD is a type of instruction-level parallelism that occurs within a single processor core. It's particularly effective for basic arithmetic operations, such as addition and multiplication, when the same operation must be applied to multiple data. Given the nature of these operations, it's no surprise that one of the primary applications of SIMD is in linear algebra.

At the heart of SIMD lies the process of vectorization, where data is split into sub-vectors that can be processed as single units. To facilitate this, modern processors include specialized SIMD registers designed for this purpose. Desktop and laptop processors these days typically feature 256-bit wide

registers for vectorized operations, which can hold four 64-bit floating-point numbers.

To illustrate the workings of SIMD, consider the task of adding two vectors, each comprising four elements. Specifically, let `x = [1, 2, 3, 4]` and `y = [10, 20, 30, 40]`. In traditional scalar processing, performing the operation `x + y` would require four separate addition operations, one for each pair of numbers. In contrast, all four additions can be performed with a single instruction under SIMD, producing the result `[11, 22, 33, 44]` in one step.

For larger vectors, the process remains fundamentally the same. The only difference is that the processor first partitions the vectors into sub-vectors that fit the register's capacity. After this, the processor computes all the operations within each segment simultaneously, repeating the procedure for each sub-vector.

BROADCASTING VS FOR-LOOPS

The previous analysis shows that SIMD applies to computations involving collections. Based on this, we can identify **two types of operations that can potentially benefit from SIMD instructions: for-loops and broadcasting**. The latter is automatically implemented by the compiler, without requiring any special consideration from the user.

Instead, the upcoming sections will focus on the application of SIMD in for-loops. This will require exploring the conditions under which SIMD instructions can be applied. If these conditions aren't met, SIMD will become infeasible or reduce its effectiveness substantially. In addition to elaborating on these conditions, we'll provide guidance on how to address scenarios that don't conform to them.

To pave the way and shift our attention to for-loops, we conclude this section by illustrating the automatic application of SIMD in broadcasting.

SIMD IN BROADCASTING

The decision whether to apply SIMD instructions is always handled by the compiler, which relies on heuristics to determine when it's beneficial to do so. One situation where Julia strongly favors the application of SIMD is with broadcasting, as can be noticed in the following example.

```
x      = rand(1_000_000)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = 2 / x[i]
    end

    return output
end
```

```
julia> @btime foo($x)
  789.564 μs (2 allocations: 7.629 MiB)
```

```
x      = rand(1_000_000)

foo(x) = 2 ./ x
```

```
julia> @btime foo($x)
  414.250 μs (2 allocations: 7.629 MiB)
```

The example compares the same operation implemented using a for-loop and broadcasting. While broadcasting automatically leverages SIMD, this isn't necessarily the case with for-loops. Indeed, in this particular example, we'd need to explicitly instruct the compiler to enable SIMD, which accounts for the observed time differences.

FOOTNOTES

1. SIMD isn't exclusive to CPUs. In fact, GPUs also take advantage of it. They're a natural fit for SIMD, as their architecture was conceived for parallel processing of simple identical operations.

10d. SIMD: Independence of Iterations

Martin Alfaro
PhD in Economics

INTRODUCTION

Broadcasting heavily favors the application of SIMD instructions. In contrast, whether and when for-loops apply SIMD is more complex. Furthermore, the heuristics of the compiler, while powerful, aren't without flaws. Indeed, it's entirely possible that SIMD is implemented when it actually reduces performance or not applied when it would've been advantageous. To address this, Julia provides the `@simd` macro to manually implement SIMD, giving developers a more granular control over the optimization process.

An effective application of SIMD requires identifying the conditions under which this optimization can be applied. Failing to meet these criteria can render SIMD infeasible or necessitate code adaptations that end up slowing down computation. The ideal conditions for leveraging SIMD instructions are:

- **Independence of Iterations**: Except for reductions, which are specifically handled to ensure their feasibility.
- **Unit Strides**: Elements in collections must be accessed sequentially.
- **No Conditional Statements**: The loop body should consist solely of straight-line code.

In the upcoming sections, we'll elaborate on each of these items, additionally providing guidance on how to address scenarios not conforming to them. This section in particular exclusively focuses on the independence of iterations.

> **Warning!** - Determining Whether SIMD Has Been Implemented
>
> Assessing whether SIMD instructions are implemented requires inspecting the compiled code. Due to the complexity of this approach, we'll instead rely on execution times as a practical indicator.

REMARKS ABOUT @SIMD IN FOR-LOOPS

Recall from the previous section that the impact of macros on computational methods is intricate. The reason is that macros only serve as a hint to the compiler, rather than a strict directive. Consequently, they suggest techniques that the compiler may eventually discard or would have implemented regardless—the compiler has the final say on which optimizations are worth adopting. In this context, the inclusion of `@simd` in a for-loop is far from a guarantee that SIMD will actually be implemented.

Furthermore, it's notoriously difficult to predict whether SIMD instructions are beneficial in particular scenarios. This is due to several factors. Firstly, different CPU architectures provide varying levels of support for SIMD instructions. [1] This diversity in SIMD capabilities means that the benefits of SIMD tend to vary greatly by hardware.

Second, as we already mentioned, it's hard to anticipate when and how SIMD will be applied in our code. The compiler relies on sophisticated heuristics to determine when SIMD may be advantageous, but they aren't infallible. Indeed, it's entirely possible that SIMD is implemented when it actually reduces performance or not applied when it would've been advantageous.

Despite these complexities, structuring operations in certain ways can improve the likelihood of implementing SIMD beneficially. By identifying these conditions, we'll be able to write code that's more amenable to SIMD optimization. It's worth remarking, though, that **the recommendations we'll present should be interpreted as general principles, rather than absolute rules**. Given the complexity of SIMD, benchmarking remains necessary to validate the existence of any performance improvement.

> **Safety of SIMD**
>
> Strictly speaking, SIMD is a form of parallelization. We'll see in subsequent sections that parallelization may render code unsafe and lead to catastrophic errors when used improperly. `@simd` doesn't involve these types of risks, since it's been designed to apply only when it's safe to do so. Specifically, the compiler will disregard SIMD if the conditions for its safe application aren't met.

INDEPENDENCE OF ITERATIONS

To effectively apply SIMD, iterations should be independent. This means that no iteration should depend on the results of previous iterations or affect the results of subsequent ones. When this condition is met, each iteration can be executed in parallel. A typical scenario is when we need to apply some function $f\left(x_i\right)$ to each element x_i of a vector `x`.

In the following, we illustrate this case via a polynomial transformation of `x`. The transformation will be done through for-loops with and without SIMD. We'll also compare these approaches with broadcasting, which applies SIMD automatically.

Importantly, as we'll explain in a subsequent section, applying `@simd` in for-loops requires the `@inbounds` macro. We'll see that, essentially, checking index bounds introduces a condition, giving rise to execution branches that hinder or directly prevent the application of SIMD.

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = x[i] / 2 + x[i]^2 / 3
    end

    return output
end
```

```
julia> @btime foo($x)
  806.606 μs (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    @inbounds @simd for i in eachindex(x)
        output[i] = x[i] / 2 + x[i]^2 / 3
    end

    return output
end
```

```
julia> @btime foo($x)
  464.734 μs (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

foo(x) = @. x / 2 + x^2 / 3
```

```
julia> @btime foo($x)
  447.074 μs (4 allocations: 7.629 MiB)
```

A SPECIAL CASE OF DEPENDENCE: REDUCTIONS

SIMD requires that iterations are independent. One exception to this rule is given by reductions, which have been carefully designed for their proper handling.

Julia leverages SIMD automatically for reductions involving integers. Instead, reductions with floating-point numbers require the explicit addition of the `@simd` macro. The following example demonstrates this fact. For the case of integers, we see that there are no differences in execution times with and without `@simd`.

```
x = rand(1:10, 10_000_000)   # random integers between 1 and 10

function foo(x)
    output = 0

    for a in x
        output += a
    end

    return output
end
```

```
julia> @btime foo($x)
  2.606 ms (0 allocations: 0 bytes)
```

```
x = rand(1:10, 10_000_000)   # random integers between 1 and 10

function foo(x)
    output = 0

    @simd for a in x
        output += a
    end

    return output
end
```

```
julia> @btime foo($x)
  2.636 ms (0 allocations: 0 bytes)
```

This behavior contrasts with a sum reduction consisting of floating-point operations, as shown below.

```
x = rand(10_000_000)

function foo(x)
    output = 0.0

    for a in x
        output += a
    end

    return output
end
```

```
julia> @btime foo($x)
  5.033 ms (0 allocations: 0 bytes)
```

```
x = rand(10_000_000)

function foo(x)
    output = 0.0

    @simd for a in x
        output += a
    end

    return output
end
```

```
julia> @btime foo($x)
  2.753 ms (0 allocations: 0 bytes)
```

Why Floating Points Are Treated Differently

Unlike integers, addition of floating-point numbers doesn't obey associativity: due to the inherent imprecision of floating-point arithmetic, `(x+y)+z` may not be exactly equal to `x+(y+z)`. This is one of several reasons why floating-point numbers are distinct from mathematical real numbers: they are finite-precision approximations that don't always follow the same mathematical properties we expect from real numbers.

The following code shows this feature of floating points.

```
x = 0.1 + (0.2 + 0.3)
```

```
julia> x
0.6
```

```
x = (0.1 + 0.2) + 0.3
```

```
julia> x
0.6000000000000001
```

By instructing the compiler to ignore the non-associativity of floating-point arithmetic, SIMD instructions can optimize performance by reordering terms. However, this approach assumes that the operations do not rely on a specific order of operations. Fortunately, this assumption rarely causes issues in scientific applications, as they typically involve mathematical models that inherently assume real number properties from the outset.

FOOTNOTES

[1.] For instance, x86 architectures (Intel and AMD processors) offer SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions). In turn, each comprises variants supporting different vector widths and operations (e.g., the variant AVX-512 in Intel Xeon processors).

10e. SIMD: Unit Strides

Martin Alfaro
PhD in Economics

INTRODUCTION

SIMD improves computational performance by simultaneously executing the same operation on multiple data elements. Technically, this is achieved through the use of specialized vector registers, which can hold several values (e.g., 4 floating-point numbers or 8 integers). They allow operations such as multiple additions or multiplications to be completed with a single instruction.

For SIMD to fully exploit this vector-based processing, **data should be organized and accessed in memory** in specific ways. Two fundamental concepts for understanding this are contiguous memory layout and unit stride access.

Contiguous memory layout means that data elements reside in adjacent memory addresses, therefore not exhibiting gaps. Newly allocated arrays satisfy this condition, enabling the load of entire segments directly into vector registers. In contrast, array views reference the original data structure, and thus don't guarantee contiguity. This can potentially result in highly irregular memory access patterns.

In addition, **strides** refer to the step size between consecutive memory accesses, with **unit strides** in particular entailing that elements are accessed sequentially. For example, consider a freshly allocated vector `x`. Then, accessing its elements through `eachindex(x)` (or equivalently `1:length(x)`) ensures a unit stride, as each access moves sequentially to the next element in memory. This contrasts with ranges having a non-unit stride such as `1:2:length(x)` or indices with no predictable pattern (e.g., `[1, 5, 3, 4]` as an index vector).

When it comes to performance, **SIMD is most effectively applied when data is stored in a contiguous memory block and accessed in a unit stride pattern**. This has significant practical implications for creating slices, which can take the form of copies or views. Essentially, this choice gives rise to a trade-off, as views reduce memory allocations but at the expense of hindering computational efficiency. This section explores such a decision.

REVIEW OF INDEXING

For the explanations, we'll utilize the various methods for creating slices, with each differing in how indices are defined: vector indexing, Boolean indexing, and ranges. We present the examples used for each, and then proceed to their explanation.

```
x         = [10, 20, 30]

indices   = sortperm(x)
elements  = x[indices]    # equivalent to `sort(x)`
```

```
julia> sorted_indices
3-element Vector{Int64}:
 1
 2
 3
julia> sorted_elements
3-element Vector{Int64}:
 10
 20
 30
```

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

indices_1 = 1:length(x)         # unit strides, equivalent to 1:1:length(x)
indices_2 = 1:2:length(x)       # strides two
```

```
julia> collect(indices_1)
5-element Vector{Int64}:
 1
 2
 3
 4
 5
julia> collect(indices_2)
3-element Vector{Int64}:
 1
 3
 5
```

```
x         = [20, 10, 30]

indices   = x .> 15
elements  = x[indices]
```

```
julia> sorted_indices
3-element BitVector:
 1
 0
 1
julia> sorted_elements
2-element Vector{Int64}:
 20
 30
```

As for **vector indexing**, the demonstrations will be based on the function `sortperm`. Given some vector `x`, this function is such that, while `sort(x)` returns a vector with `x`'s elements sorted in ascending order, `sortperm(x)` returns the corresponding indices of these elements.

Similarly, **ranges** can be understood as a special case of vector indexing. They differ in that ranges lazily reference consecutive elements for some given strides. Recall that strides represent the gap between successive elements, and they're included in between the first and last index, i.e. `<first index>:<stride>:<last index>`. The absence of a stride implicitly assumes a step equal to 1.

While vector indexing and ranges reference the indices of `x`, **Boolean indexing** returns a Boolean vector where `1` indicates the element must be kept. This approach will be used for the creation of slices through broadcasted conditions, as in the example provided.

CONTIGUOUS BLOCKS IN MEMORY

In the section discussing decreases in memory allocations, we highlighted the benefits of using views over copies when handling slices. Specifically, views maintain references to the original data, thereby avoiding the cost of additional memory allocation. However, views can lead to irregular memory access patterns if data are too scattered. This is why this section also remarked that copies could outperform views in some scenarios. We're now in a position to explain in more depth why this occurs.

Creating copies of some data structure involves allocating the information in a new contiguous block of memory. This ensures that all elements are stored sequentially, thus offering two key advantages: a quicker fetching of elements and a more effective use of SIMD instructions.

To illustrate this, consider retrieving books from a library. If every book you need resides on a single shelf, collecting them is straightforward—you move once, grab the entire stack, and proceed. This mirrors contiguous memory access. Conversely, if the books are dispersed across different floors and sections, each retrieval demands additional time and effort, akin to non-contiguous access.

In addition to this, our recollection of books would be even more efficient if we had a cart capable of carrying multiple books at once—SIMD operations act like this cart. Note that even without such a cart, the act of gathering contiguous books remains inherently faster, as the physical effort (or computational cycles) is minimized. In other words, the benefits of SIMD are on top of the faster memory access.

VECTOR AND BOOLEAN INDEXING

The following examples highlight the two advantages from contiguous memory access. To isolate their performance effect, we create the slices outside the function to be benchmarked. In this way, the functions we'll present don't entail memory allocations.

For the case of vector indexing:

```
x       = rand(5_000_000)

indices = sortperm(x)
y       = @view x[indices]

function foo(y)
    output = 0.0

    for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  21.481 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)

indices = sortperm(x)
y       = @view x[indices]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  20.754 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)

indices = sortperm(x)
y       = x[indices]

function foo(y)
    output = 0.0

    for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  2.281 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)

indices = sortperm(x)
y       = x[indices]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  902.847 μs (0 allocations: 0 bytes)
```

while for Boolean indexing:

```
x       = rand(5_000_000)

indices = x .> 0.5
y       = @view x[indices]

function foo(y)
    output = 0.0

    for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  2.206 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)

indices = x .> 0.5
y       = @view x[indices]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  1.878 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)

indices = x .> 0.5
y       = x[indices]

function foo(y)
    output = 0.0

    for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  1.015 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)

indices = x .> 0.5
y       = x[indices]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  246.526 μs (0 allocations: 0 bytes)
```

Finally, comparing unit strides with non-unit strides:

```
x = rand(1_000_000)
y = @view x[1:2:length(x)]

function foo(y)
    output = 0.0

    for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  902.479 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)
y = @view x[1:2:length(x)]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  889.059 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)
y = x[1:2:length(x)]

function foo(y)
    output = 0.0

    for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  196.497 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)
y = x[1:2:length(x)]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  38.274 μs (0 allocations: 0 bytes)
```

SOME REMARKS

Note that views don't necessarily impede a sequential memory access. If the view consists of sequential elements, then we'd obtain the same performance relative to a copy created outside the function.

```
x       = rand(1_000_000)

indices = 1:length(x)
y       = @view x[indices]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  76.950 μs (0 allocations: 0 bytes)
```

```
x       = rand(1_000_000)

indices = 1:length(x)
y       = x[indices]

function foo(y)
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($y)
  76.777 μs (0 allocations: 0 bytes)
```

A corollary of this example is that views should be employed in cases like this, as they additionally avoid memory allocations.

Another remark is that *storing* elements contiguously is only a necessary condition for contiguous memory *access*. To demonstrate this, let's consider a scenario where we access vectors through the ranges.

Specifically, consider a vector `x`. Moreover, there's another vector `y` that comprises the same elements as `x`, but with zeros placed in between. Formally:

```
x_size = 1_000_000

x = rand(x_size)

y = zeros(eltype(x),x_size * 2)
    temp  = view(y, 2:2:length(y))
    temp .= x
```

```
julia> x[1:3]
3-element Vector{Float64}:
 0.906299638797481
 0.44349373245960455
 0.7456733811393941
julia> y[1:6]
6-element Vector{Float64}:
 0.0
 0.906299638797481
 0.0
 0.44349373245960455
 0.0
 0.7456733811393941
```

Given this, we'd get the same value whether we add all elements from `x` or add all elements in `y` skipping zeros. However, the results below reveal that, despite both `x` and `y` being contiguous blocks in memory, unit-stride access only holds for `x`. This explains the differences in the performance observed.

```
function foo(x)
    output = 0.0

    @inbounds @simd for i in 1:length(x)
        output += x[i]
    end

    return output
end
```

```
julia> @btime foo($y)
  78.349 μs (0 allocations: 0 bytes)
```

```
function foo(y)
    output = 0.0

    @inbounds @simd for i in 2:2:length(y)
        output += y[i]
    end

    return output
end
```

```
julia> @btime foo($y)
  188.483 μs (0 allocations: 0 bytes)
```

COPIES VS VIEWS: OVERALL EFFECTS

When slices are created, the choice between copies and views requires weighing in the overhead of additional memory allocations against the performance benefits of sequential memory accesses (including a more performant application of SIMD).

One scenario where views always outperform copies was given above, where the view elements are accessed sequentially. Instead, a common scenario where copies tend to outperform views is when we need to perform multiple operations over the same slice. In this case, the cost of an additional memory allocation is usually outweighed by the performance benefits of contiguous memory access. This is illustrated below.

```
x       = rand(5_000_000)
indices = sortperm(x)

function foo(x, indices)
    y      = @view x[indices]
    output1, output2, output3 = (0.0 for _ in 1:3)

    @simd for a in y
        output1 += a^(3/2)
        output2 += a / 3
        output3 += a * 2.5
    end

    return output1, output2, output3
end
```

```
julia> @btime foo($y)
  248.861 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)
indices = sortperm(x)

function foo(x, indices)
    y      = x[indices]
    output1, output2, output3 = (0.0 for _ in 1:3)

    @simd for a in y
        output1 += a^(3/2)
        output2 += a / 3
        output3 += a * 2.5
    end

    return output1, output2, output3
end
```

```
julia> @btime foo($y)
  125.033 ms (2 allocations: 38.147 MiB)
```

In general, though, benchmarking is the only way to decide whether copies or views are faster. For instance, views are faster in the following example:

```
x       = rand(5_000_000)
indices = sortperm(x)

function foo(x, indices)
    y      = @view x[indices]
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($x, $indices)
  22.741 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)
indices = sortperm(x)

function foo(x, indices)
    y      = x[indices]
    output = 0.0

    @simd for a in y
        output += a
    end

    return output
end
```

```
julia> @btime foo($x, $indices)
  36.151 ms (2 allocations: 38.147 MiB)
```

Instead, the following scenario establishes that an approach with copies outperforms views.

```
x       = rand(5_000_000)
indices = sortperm(x)

function foo(x, indices)
    y      = @view x[indices]
    output = 0.0

    @simd for a in y
        output += a^(3/2)
    end

    return output
end
```

```
julia> @btime foo($x, $indices)
  268.653 ms (0 allocations: 0 bytes)
```

```
x       = rand(5_000_000)
indices = sortperm(x)

function foo(x, indices)
    y      = x[indices]
    output = 0.0

    @simd for a in y
        output += a^(3/2)
    end

    return output
end
```

```
julia> @btime foo($x, $indices)
  135.816 ms (2 allocations: 38.147 MiB)
```

10f. SIMD: Branchless Code

Martin Alfaro
PhD in Economics

BRANCHES

SIMD accelerates computations by executing the same set of instructions in parallel across multiple data elements. Yet, certain programming constructs, particularly conditional statements, can severely degrade SIMD efficiency. The issue arises since conditional statements inherently lead to different instruction paths, thus disrupting the single instruction execution that SIMD relies on. While the compiler attempts to mitigate this issue by transforming code into SIMD-compatible forms, these adaptations often incur a performance penalty.

This section explores strategies for efficiently applying SIMD in the presence of conditional operations. We'll first examine scenarios where the compiler introduces conditional statements as an artifact of its internal computation techniques. By employing alternative coding strategies, we'll show how these conditional statements can be bypassed.

After this, we'll explore conditional statements that are intrinsic to program logic and therefore unavoidable. This includes standard scenarios where conditions are explicitly introduced in the code. In this respect, we'll revisit the usual approaches to expressing conditions, focusing on their internal implementation. We'll outline their relative strengths and limitations, indicating which approaches are more conducive to SIMD optimizations. Finally, we'll show that conditional statements can be equivalently recast as algebraic operations, which effectively removes the branching logic that disrupts SIMD execution.

TYPE STABILITY AND BOUNDS CHECKING AS AVOIDABLE CONDITIONS

Two patterns in Julia introduce hidden branches that hurt SIMD performance: type-unstable functions and bounds checking in array indexing. These conditions arise internally from compiler decisions, rather than explicit code, making them easy to overlook.

When a function is type-unstable, Julia generates multiple execution branches, one for each type. Those extra branches, while invisible to you, still disrupt the uniform instruction flow required by SIMD. The remedies for this case are the same as those for fixing type instabilities. Regardless of any SIMD consideration, recall that you should always strive for type stability. Type instability is a major performance bottleneck, with any attempt to achieve high performance becoming nearly impossible without addressing it.

The other source of hidden conditionals arises in for-loops, which perform bounds checking by default. This operation represents a subtle form of conditional execution, where each iteration is executed only when indices remain within bounds.

In relation to this scenario, the example below demonstrates two key insights. First, merely adding `@inbounds` can be enough to induce the compiler to apply SIMD instructions, rendering `@simd` annotations redundant for performance improvements. This explains why using `@inbounds @simd` in the example has a negligible impact on execution times. [1] Second, the example highlights that **adding `@inbounds` is a necessary precondition for the application of SIMD**. Simply using `@simd` on its own won't trigger the implementation of SIMD instructions, as the compiler may still be hindered by the bounds checks. Overall, if we aim to apply SIMD in a for-loop, we should prepend it with `@inbounds @simd`.

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = 2/x[i]
    end

    return output
end
```

```
julia> @btime foo($x)
  775.469 μs (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    @simd for i in eachindex(x)
        output[i] = 2/x[i]
    end

    return output
end
```

```
julia> @btime foo($x)
  792.687 μs (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    @inbounds for i in eachindex(x)
        output[i] = 2/x[i]
    end

    return output
end
```

```
julia> @btime foo($x)
  474.154 μs (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    @inbounds @simd for i in eachindex(x)
        output[i] = 2/x[i]
    end

    return output
end
```

```
julia> @btime foo($x)
  452.921 μs (2 allocations: 7.629 MiB)
```

Broadcasting and For-Loops

Broadcasting disables bounds checking and strongly favors SIMD by default, often making it appear more performant than a simple for-loop. Despite this, broadcasting essentially serves as a concise notation for implementing a for-loop. As the example below demonstrates, a for-loop that has been optimized with `@inbounds` and `@simd` will typically exhibit a similar level of performance to a broadcasted operation.

```
x      = rand(1_000_000)
foo(x) = 2 ./ x
```

```
julia> @btime foo($x)
  431.487 μs (2 allocations: 7.629 MiB)
```

```
x      = rand(1_000_000)

function foo(x)
    output = similar(x)

    @inbounds @simd for i in eachindex(x)
        output[i] = 2/x[i]
    end

    return output
end
```

```
julia> @btime foo($x)
  435.973 μs (2 allocations: 7.629 MiB)
```

```
x      = rand(1_000_000)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = 2/x[i]
    end

    return output
end
```

```
julia> @btime foo($x)
  809.359 μs (2 allocations: 7.629 MiB)
```

APPROACHES TO CONDITIONAL STATEMENTS

When conditions are part of the program's logical flow and therefore unavoidable, we need to inquire on what approach is better for the introduction.

Specifically, conditional statements can be evaluated either eagerly or lazily. To illustrate, let's consider the computation of `1 + 1` but only if certain condition `C` is met. A lazy approach evaluates whether `C` holds true, before proceeding with the computation of `1 + 1`. Thus, the operation is deferred until it's confirmed that `C` holds. In contrast, an eager approach computes `1 + 1`, regardless of whether `C` is satisfied. If `C` turns out to be false, the computation remains unused.

When conditional statements are applied only once, a lazy approach is almost always more performant as it avoids needless computations. However, inside a for-loop, SIMD can compute multiple operations simultaneously. Consequently, it may be beneficial to evaluate all conditions and branches upfront, selecting the relevant branches afterward. The possibility is especially true when branches involve inexpensive computations.

In Julia, whether a conditional statement is evaluated lazily or eagerly depends on how it's written. Next, we explore this nuance in more detail.

IFELSE VS IF

The `ifelse` function in Julia follows an eager evaluation strategy, where both the condition and possible outcomes are computed before deciding which result to return. In contrast, `if` and `&&` favor lazy computations, only evaluating the necessary components based on the truth value of the condition.

The following example demonstrates this computational difference through a reduction operation that's contingent on a condition. [2]

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    for i in eachindex(x)
        if x[i] > 0.5
            output += x[i]/2
        end
    end

    return output
end
```

```
julia> @btime foo($x)
  415.373 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if x[i] > 0.5
            output += x[i]/2
        end
    end

    return output
end
```

```
julia> @btime foo($x)
  414.155 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    for i in eachindex(x)
        output += ifelse(x[i] > 0.5, x[i]/2, 0)
    end

    return output
end
```

```
julia> @btime foo($x)
  393.046 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += ifelse(x[i] > 0.5, x[i]/2, 0)
    end

    return output
end
```

```
julia> @btime foo($x)
  87.192 μs (0 allocations: 0 bytes)
```

As the example reveals, an eager computation doesn't automatically imply the application of SIMD. This is precisely why `@simd` is included, which provides a hint to the compiler that vectorizing the operation might be beneficial. In fact, we'll show later that adding `@simd` when conditions comprise multiple statements could prompt the compiler to vectorize conditions, while still relying on a lazy evaluation.

It's also worth remarking that applying SIMD instructions doesn't necessarily increase performance. The example below demonstrates this point, where the compiler adopts a SIMD approach through `ifelse`.

```
x      = rand(5_000_000)
output = similar(x)

function foo!(output,x)
    for i in eachindex(x)
        output[i] = ifelse(x[i] > 0.5, x[i]/2, 0)
    end
end
```

```
julia> foo!($output,$x)
  2.806 ms (0 allocations: 0 bytes)
```

```
x      = rand(5_000_000)
output = similar(x)

function foo!(output,x)
    @inbounds for i in eachindex(x)
        output[i] = ifelse(x[i] > 0.5, x[i]/2, 0)
    end
end
```

```
julia> foo!($output,$x)
  2.888 ms (0 allocations: 0 bytes)
```

```
x      = rand(5_000_000)
output = similar(x)

function foo!(output,x)
    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(x[i] > 0.5, x[i]/2, 0)
    end
end
```

```
julia> foo!($output,$x)
  16.026 ms (0 allocations: 0 bytes)
```

TERNARY OPERATORS

Ternary operators are an alternative approach for conditional statements, consisting of the form `<condition> ? <action if true> : <action if false>`. Unlike the previous methods, this form relies on heuristics to determine whether an eager or lazy approach should be used. The decision depends on which strategy would more likely be faster in the application considered.

For the illustrations, we'll consider examples where we directly add `@inbounds` and `@simd` in each approach.

DIFFERENT CHOICES

Starting with the same example as above, we show that the ternary operator could choose an eager approach.

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if x[i] > 0.5
            output += x[i]/2
        end
    end

    return output
end
```

```
julia> foo!($output,$x)
  422.480 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += ifelse(x[i]>0.5, x[i]/2, 0)
    end

    return output
end
```

```
julia> foo!($output,$x)
  85.895 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += x[i]>0.5 ? x[i]/2 : 0
    end

    return output
end
```

```
julia> foo!($output,$x)
  87.881 μs (0 allocations: 0 bytes)
```

Instead, the ternary operator implements a lazy approach in the following example.

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if x[i] > 0.99
            output += log(x[i])
        end
    end

    return output
end
```

```
julia> foo!($output,$x)
  405.304 μs (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += ifelse(x[i] > 0.99, log(x[i]), 0)
    end

    return output
end
```

```
julia> foo!($output,$x)
  3.470 ms (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += x[i]>0.99 ? log(x[i]) : 0
    end

    return output
end
```

```
julia> foo!($output,$x)
  421.493 μs (0 allocations: 0 bytes)
```

TERNARY OPERATOR COULD CHOOSE A LESS PERFORMANT APPROACH

It's worth remarking that the method chosen by the ternary operator isn't foolproof. In the following scenario, it actually chooses the slowest approach.

```
x      = rand(5_000_000)
output = similar(x)

function foo!(output,x)
    @inbounds @simd for i in eachindex(x)
        if x[i] > 0.5
            output[i] = log(x[i])
        end
    end
end
```

```
julia> foo!($output,$x)
  26.620 ms (0 allocations: 0 bytes)
```

```
x      = rand(5_000_000)
output = similar(x)

function foo!(output,x)
    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(x[i] > 0.5, log(x[i]), 0)
    end
end
```

```
julia> foo!($output,$x)
  16.864 ms (0 allocations: 0 bytes)
```

```
x      = rand(5_000_000)
output = similar(x)

function foo!(output,x)
    @inbounds @simd for i in eachindex(x)
        output[i] = x[i]>0.5 ? log(x[i]) : 0
    end
end
```

```
julia> foo!($output,$x)
  26.517 ms (0 allocations: 0 bytes)
```

SCENARIOS UNDER WHICH EACH APPROACH IS BETTER

As a rule of thumb, **an eager approach is potentially more performant when branches comprise simple algebraic computations**. On the contrary, conditional statements with **computational-demanding operations will more likely benefit from a lazy implementation**. In fact, this is a heuristic that ternary operators commonly follow.

To demonstrate this, the following example considers a conditional statement where only one branch has a computation, which in turn is straightforward. An eager approach with SIMD is faster, and coincides with the approach chosen when a ternary operator is chosen.

```
x             = rand(1_000_000)
condition(a)   = a > 0.5
computation(a) = a * 2

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if condition(x[i])
            output += computation(x[i])
        end
    end

    return output
end
```

```
julia> foo!($output,$x)
  416.681 μs (0 allocations: 0 bytes)
```

```
x             = rand(1_000_000)
condition(a)   = a > 0.5
computation(a) = a * 2

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += ifelse(condition(x[i]), computation(x[i]), 0)
    end

    return output
end
```

```
julia> foo!($output,$x)
  86.242 μs (0 allocations: 0 bytes)
```

```
x             = rand(1_000_000)
condition(a)   = a > 0.5
computation(a) = a * 2

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += condition(x[i]) ? computation(x[i]) : 0
    end

    return output
end
```

```
julia> foo!($output,$x)
  86.370 μs (0 allocations: 0 bytes)
```

Instead, the following scenario considers a branch with more computational-intensive calculations. In this case, a lazy approach is faster, which is the approach implemented by the ternary operator.

```
x              = rand(2_000_000)
condition(a)   = a > 0.5
computation(a) = exp(a)/3 - log(a)/2

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if condition(x[i])
            output += computation(x[i])
        end
    end

    return output
end
```

```
julia> foo!($output,$x)
  12.346 ms (0 allocations: 0 bytes)
```

```
x              = rand(2_000_000)
condition(a)   = a > 0.5
computation(a) = exp(a)/3 - log(a)/2

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += ifelse(condition(x[i]), computation(x[i]), 0)
    end

    return output
end
```

```
julia> foo!($output,$x)
  16.552 ms (0 allocations: 0 bytes)
```

```
x             = rand(2_000_000)
condition(a)   = a > 0.5
computation(a) = exp(a)/3 - log(a)/2

function foo(x)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        output += condition(x[i]) ? computation(x[i]) : 0
    end

    return output
end
```

```
julia> foo!($output,$x)
  12.492 ms (0 allocations: 0 bytes)
```

VECTOR OF CONDITIONS

Next, we consider scenarios where you already have a vector holding conditions. This could occur either because the vector is already part of your dataset, or because the conditions will be reused multiple times over your code, in which case storing the conditions is worthy.

Storing conditions in a vector could be done through an object with type `Vector{Bool}` or `BitVector`. The latter is the default type returned by Julia, as when you define objects like `x .> 0`. Although this type offers certain performance advantages, it can also hinder the application of SIMD. In cases like this, transforming `BitVector` to `Vector{Bool}` could speed up computations.

The following example demonstrates this, where the execution time is faster even considering the vector transformation.

```
x         = rand(1_000_000)
bitvector = x .> 0.5

function foo(x,bitvector)
    output     = similar(x)


    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(bitvector[i], x[i]/i, x[i]*i)
    end

    return output
end
```

```
julia> foo($x,$bitvector)
  3.393 ms (2 allocations: 7.629 MiB)
```

```
x         = rand(1_000_000)
bitvector = x .> 0.5

function foo(x,bitvector)
    output     = similar(x)
    boolvector = Vector(bitvector)

    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(boolvector[i], x[i]/i, x[i]*i)
    end

    return output
end
```

```
julia> foo($x,$bitvector)
  862.798 µs (4 allocations: 8.583 MiB)
```

No Vector of Conditions

The conclusions stated here assumes that you already have the vector holding the conditions. If this isn't the case and you want to apply SIMD instructions, you should implement `ifelse` without a vector of conditions. This allows you to avoid memory allocations, while still applying SIMD effectively. The following example illustrates this point. [3]

```
x = rand(1_000_000)

function foo(x)
    output     = similar(x)
    bitvector  = x .> 0.5


    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(bitvector[i], x[i]/i, x[i]*i)
    end

    return output
end
```

```
julia> foo($x)
  3.628 ms (6 allocations: 7.753 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output     = similar(x)
    boolvector = Vector{Bool}(undef,length(x))
        boolvector .= x .> 0.5

    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(boolvector[i], x[i]/i, x[i]*i)
    end

    return output
end
```

```
julia> foo($x)
  774.952 μs (4 allocations: 8.583 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output     = similar(x)



    @inbounds @simd for i in eachindex(x)
        output[i] = ifelse(x[i]>0.5, x[i]/i, x[i]*i)
    end

    return output
end
```

```
julia> foo($x)
  501.114 μs (2 allocations: 7.629 MiB)
```

ALGEBRAIC OPERATIONS AS COMPOUND CONDITIONS

We leverage algebraic equivalences to express conditions in ways that allow us to avoid the creation of branches. Mathematically, given a set $\{b_i\}_{i=1}^n$ where $b_i \in \{0,1\}$:

- all conditions are satisfied when

$$\prod_{i=1}^{n} c_i = 1$$

- at least one condition is satisfied when

$$1 - \prod_{i=1}^{n} (1 - c_i) = 1$$

In terms of Julia, given two Boolean scalars `c1` and `c2`, these equivalences become

- `c1 && c2` is `Bool(c1 * c2)`

- `c1 || c2` is `Bool(1 - !c1 * !c2)`

For instance, with for-loops:

```
x                 = rand(1_000_000)
y                 = rand(1_000_000)


function foo(x,y)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if (x[i] > 0.3) && (y[i] < 0.6) && (x[i] > y[i])
            output += x[i]
        end
    end

    return output
end
```

```
julia> foo($x)
  2.116 ms (0 allocations: 0 bytes)
```

```
x                 = rand(1_000_000)
y                 = rand(1_000_000)


function foo(x,y)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if (x[i] > 0.3)  * (y[i] < 0.6)  * (x[i] > y[i])
            output += x[i]
        end
    end

    return output
end
```

```
julia> foo($x)
  905.078 μs (0 allocations: 0 bytes)
```

```
x                = rand(1_000_000)
y                = rand(1_000_000)


function foo(x,y)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if (x[i] > 0.3) || (y[i] < 0.6) || (x[i] > y[i])
            output += x[i]
        end
    end

    return output
end
```

```
julia> foo($x)
  2.724 ms (0 allocations: 0 bytes)
```

```
x                = rand(1_000_000)
y                = rand(1_000_000)


function foo(x,y)
    output = 0.0

    @inbounds @simd for i in eachindex(x)
        if Bool(1 - !(x[i] > 0.3)  * !(y[i] < 0.6)  * !(x[i] > y[i]))
            output += x[i]
        end
    end

    return output
end
```

```
julia> foo($x)
  889.879 μs (0 allocations: 0 bytes)
```

While with broadcasting:

```
x              = rand(1_000_000)
y              = rand(1_000_000)


foo(x,y)       = @. ifelse((x>0.3) && (y<0.6) && (x>y), x,y)
```

```
julia> foo($x)
  5.356 ms (2 allocations: 7.629 MiB)
```

```
x               = rand(1_000_000)
y               = rand(1_000_000)


foo(x,y)        = @. ifelse((x>0.3) *  (y<0.6) *  (x>y), x,y)
```

```
julia> foo($x)
  541.621 μs (2 allocations: 7.629 MiB)
```

```
x               = rand(1_000_000)
y               = rand(1_000_000)


foo(x,y)        = @. ifelse((x>0.3) || (y<0.6) || (x>y), x,y)
```

```
julia> foo($x)
  3.354 ms (2 allocations: 7.629 MiB)
```

```
x               = rand(1_000_000)
y               = rand(1_000_000)


foo(x,y)        = @. ifelse(Bool(1 - !(x>0.3) * !(y<0.6) * !(x>y)), x,y)
```

```
julia> foo($x)
  536.276 μs (2 allocations: 7.629 MiB)
```

FOOTNOTES

[1.] Recall that the compiler may automatically disable bounds checking in some cases, especially in straightforward cases. For instance, this would be the case in our example if only `x` had been indexed and `eachindex(x)` were employed as the iteration range. This is in contrast to scenarios like the one below, where we're indexing both `x` and `output`.

[2.] Note that `ifelse` requires specifying an operation for when the condition is true and another when it's not. For a sum reduction, this is handled by returning zero when the condition isn't met.

[3.] Note that the approach for `Vector{Bool}` is somewhat different to the examples we considered above. As we don't have a vector of conditions already defined, it's optimal to create `Vector{Bool}` directly, rather than defining it as a transformation of the `BitVector`. In this way, we avoid unnecessary memory allocations too.

10g. Packages For SIMD

Martin Alfaro
PhD in Economics

INTRODUCTION

So far, we've been using the built-in macro `@simd` to apply SIMD instructions. This macro is relatively limited in certain respects. For one, it only hints at the potential advantages of applying SIMD, leaving the final decision implementation to the compiler's discretion. Moreover, it only provides basic features of SIMD, prioritizing code safety over performance.

Next, we introduce the macro `@turbo` from the package `LoopVectorization`, which offers several distinct advantages. First, it enforces SIMD optimizations when invoked, rather than merely suggesting them. Furthermore, it applies more aggressive optimizations compared to `@simd`. Finally, `@turbo` supports both for-loops and broadcasting operations, contrasting with `@simd`'s exclusive applicability to for-loops.

CAVEATS ABOUT IMPROPER USE OF @TURBO

In contrast to `@simd`, applying `@turbo` requires some caution, as it may lead to incorrect results if misapplied. This issue arises because the macro makes additional assumptions about the operations being performed, with the goal of applying optimizations more aggressively. In particular:

- `@turbo` never checks index bounds, potentially leading to out-of-bounds memory access.
- `@turbo` assumes the outcome is independent of the iteration order (except for reduction operations).

An example of the latter is when computing a vector holding cumulative sums of another vector. This can be observed below, where we verify the final result by summing all values in the output vector.

NO MACRO

```
x = rand(1_000_000)

function foo(x)
    output = copy(x)

    for i in 2:length(x)
        output[i] = output[i-1] + x[i]
    end

    return output
end
```

```
julia> sum(foo(x))
2.50038e11
```

@SIMD

```
x = rand(1_000_000)

function foo(x)
    output = copy(x)

    @inbounds @simd for i in 2:length(x)
        output[i] = output[i-1] + x[i]
    end

    return output
end
```

```
julia> sum(foo(x))
2.50038e11
```

@TURBO

```
x = rand(1_000_000)

function foo(x)
    output = copy(x)

    @turbo for i in 2:length(x)
        output[i] = output[i-1] + x[i]
    end

    return output
end
```

```
julia> sum(foo(x))
1.03169e6
```

CASES COVERED

Considering that `@turbo` isn't suitable for all operations, let's present two of its primary applications. The first one arises **when iterations are completely independent**, making their execution order irrelevant.

For instance, the following code snippet applies a polynomial transformation to each element of a vector.

DEFAULT

```
x              = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

function foo(x)
    output     = similar(x)

    for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  3.098 ms (2 allocations: 7.629 MiB)
```

@SIMD

```
x              = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

function foo(x)
    output     = similar(x)

    @inbounds @simd for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  5.070 ms (2 allocations: 7.629 MiB)
```

@TURBO (FOR-LOOP)

```
x              = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

function foo(x)
    output     = similar(x)

    @turbo for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  492.031 µs (2 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

foo(x)         = @turbo calculation.(x)
```

```
julia> @ctime foo($x)
  406.089 μs (2 allocations: 7.629 MiB)
```

The second application is **reductions**. Although reductions inherently involve dependent iterations, they represent a special case that `@turbo` handles properly.

DEFAULT

```
x              = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

function foo(x)
    output     = 0.0

    for i in eachindex(x)
        output += calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  3.083 ms (0 allocations: 0 bytes)
```

@SIMD

```
x              = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

function foo(x)
    output     = 0.0

    @inbounds @simd for i in eachindex(x)
        output += calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  3.299 ms (0 allocations: 0 bytes)
```

@TURBO

```
x              = rand(1_000_000)
calculation(a) = a * 0.1 + a^2 * 0.2 - a^3 * 0.3 - a^4 * 0.4

function foo(x)
    output     = 0.0

    @turbo for i in eachindex(x)
        output += calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  179.722 μs (0 allocations: 0 bytes)
```

SPECIAL FUNCTIONS

The package `LoopVectorization` leverages the library *SLEEF*, which is an acronym for "SIMD Library for Evaluating Elementary Functions". SLEEF is available in Julia through the package `SLEEFPirates` and it's designed to boost the mathematical computations of some functions by utilizing SIMD instructions. In particular, it speeds up the computations of the exponential, logarithmic, power, and trigonometric functions.

Below, we illustrate the use of `@turbo` for each type of function. See **here** for a list of all the functions supported.

LOGARITHM

DEFAULT

```
x              = rand(1_000_000)
calculation(a) = log(a)

function foo(x)
    output     = similar(x)

    for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  3.395 ms (2 allocations: 7.629 MiB)
```

@SIMD

```
x              = rand(1_000_000)
calculation(a) = log(a)

function foo(x)
    output     = similar(x)

    @inbounds @simd for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  3.414 ms (2 allocations: 7.629 MiB)
```

@TURBO (FOR-LOOP)

```
x              = rand(1_000_000)
calculation(a) = log(a)

function foo(x)
    output     = similar(x)

    @turbo for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  1.229 ms (2 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = log(a)

foo(x) = @turbo calculation.(x)
```

```
julia> @ctime foo($x)
  1.237 ms (2 allocations: 7.629 MiB)
```

EXPONENTIAL FUNCTION

DEFAULT

```
x              = rand(1_000_000)
calculation(a) = exp(a)

function foo(x)
    output     = similar(x)

    for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  1.962 ms (2 allocations: 7.629 MiB)
```

@SIMD

```
x              = rand(1_000_000)
calculation(a) = exp(a)

function foo(x)
    output     = similar(x)

    @inbounds @simd for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  1.950 ms (2 allocations: 7.629 MiB)
```

@TURBO (FOR-LOOP)

```
x              = rand(1_000_000)
calculation(a) = exp(a)

function foo(x)
    output     = similar(x)

    @turbo for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  600.413 µs (2 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = exp(a)

foo(x) = @turbo calculation.(x)
```

```
julia> @ctime foo($x)
  596.388 μs (2 allocations: 7.629 MiB)
```

POWER FUNCTIONS

DEFAULT

```
x              = rand(1_000_000)
calculation(a) = a^4

function foo(x)
    output     = similar(x)

    for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  3.100 ms (2 allocations: 7.629 MiB)
```

@SIMD

```
x              = rand(1_000_000)
calculation(a) = a^4

function foo(x)
    output     = similar(x)

    @inbounds @simd for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  3.305 ms (2 allocations: 7.629 MiB)
```

@TURBO (FOR-LOOP)

```
x              = rand(1_000_000)
calculation(a) = a^4

function foo(x)
    output     = similar(x)

    @turbo for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  498.762 μs (2 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = a^4

foo(x) = @turbo calculation.(x)
```

```
julia> @ctime foo($x)
  434.407 μs (2 allocations: 7.629 MiB)
```

The implementation of power functions includes square roots.

DEFAULT

```
x              = rand(1_000_000)
calculation(a) = sqrt(a)

function foo(x)
    output     = similar(x)

    for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  1.232 ms (2 allocations: 7.629 MiB)
```

@SIMD

```
x              = rand(1_000_000)
calculation(a) = sqrt(a)

function foo(x)
    output     = similar(x)

    @inbounds @simd for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  1.231 ms (2 allocations: 7.629 MiB)
```

@TURBO (FOR-LOOP)

```
x              = rand(1_000_000)
calculation(a) = sqrt(a)

function foo(x)
    output     = similar(x)

    @turbo for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  614.203 μs (2 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = sqrt(a)

foo(x) = @turbo calculation.(x)
```

```
julia> @ctime foo($x)
  614.241 μs (2 allocations: 7.629 MiB)
```

TRIGONOMETRIC FUNCTIONS

Among others, `@turbo` can handle the functions `sin`, `cos`, and `tan`. Below, we demonstrate its use with `sin`.

DEFAULT

```
x              = rand(1_000_000)
calculation(a) = sin(a)

function foo(x)
    output     = similar(x)

    for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  3.841 ms (2 allocations: 7.629 MiB)
```

@SIMD

```
x              = rand(1_000_000)
calculation(a) = sin(a)

function foo(x)
    output     = similar(x)

    @inbounds @simd for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  3.767 ms (2 allocations: 7.629 MiB)
```

@TURBO (FOR-LOOP)

```
x              = rand(1_000_000)
calculation(a) = sin(a)

function foo(x)
    output     = similar(x)

    @turbo for i in eachindex(x)
        output[i] = calculation(x[i])
    end

    return output
end
```

```
julia> @ctime foo($x)
  1.386 ms (2 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = sin(a)

foo(x) = @turbo calculation.(x)
```

```
julia> @ctime foo($x)
  1.384 ms (2 allocations: 7.629 MiB)
```

11a. Overview and Goals

Martin Alfaro
PhD in Economics

INTRODUCTION

Programming languages typically execute code sequentially, following a single path of execution that utilizes one core at a time. This linear approach simplifies reasoning about program behavior, as each operation completes before the next begins. However, hardware these days is commonly equipped with multiple processor cores. Consequently, a sequential execution does all the work on one core, while the others sit idle. This leaves substantial computational power untapped.

Multithreading addresses this limitation by running different segments of our program simultaneously across multiple cores. While this capability opens up significant opportunities for performance improvement, it also introduces new challenges that developers need to navigate carefully. In fact, simple operations that work flawlessly in single-threaded programs may yield incorrect results in a multithreaded setting. Furthermore, writing multithreaded code requires a fundamental shift in the user's mindset regarding program execution. All this makes multithreaded code inherently more difficult to write, test, and debug than its single-threaded counterpart.

Despite these challenges, the potential performance benefits of multithreading make it an essential tool in modern programming. This is particularly true for applications that are computationally intensive or demand that the same code be applied to multiple objects.

11b. Introduction to Multithreading

Martin Alfaro
PhD in Economics

INTRODUCTION

A proper implementation of multithreading demands some basic understanding of the inner workings of computers. In particular, it's essential to know how programming languages manage dependencies between operations. This knowledge is especially relevant for multithreading, since the technique creates the possibility of writing unsafe code, where a flawed multithreaded implementation may yield incorrect results.

This section will only present preliminary concepts, setting the stage for subsequent sections. Moreover, the focus will be on explanations, rather than actual implementations of multithreading. In fact, most of the macros and functions introduced here won't be utilized again on this website.

NATURE OF COMPUTATIONS

An operation can be broadly classified by data dependency as dependent or independent. A **dependent operation** is one whose outcome is influenced by the result of another operation. In such cases, the order of execution is critical, because changing the sequence can alter the final outcome. By contrast, an **independent operation** produces the same result, regardless of the order in which it's executed relative to others—its computation does not rely on the outputs of preceding or subsequent operations.

The following code gives rise to a dependent or independent operation, depending on which values are summed by operation B.

```
job_A()  = 1 + 1
job_B(A) = 2 + A

function foo()
    A = job_A()
    B = job_B(A)

    return A,B
end
```

```
job_A() = 1 + 1
job_B() = 2 + 2

function foo()
    A = job_A()
    B = job_B()

    return A,B
end
```

Likewise, regardless of dependency status, operations can be computed either sequentially or concurrently. A **sequential** procedure involves executing operations one after the other, ensuring each operation completes before the next one begins. Conversely, **concurrency** allows multiple operations to be processed simultaneously, opening up opportunities for parallel execution.

Like most programming languages, **Julia defaults to a sequential execution**. This is a deliberate choice that prioritizes result correctness, based on that **concurrent execution with dependent operations can yield incorrect results if mishandled**. Basically, the issue arises because concurrency can deal with dependencies in multiple ways, potentially involving timing inconsistencies for reading and writing data. A sequential approach precludes this possibility, as it guarantees a predictable order of execution and therefore timing.

Despite its advantages regarding safety, a sequential approach can be quite inefficient for independent tasks: by restricting computations to one at a time, computational resources may go underutilized. In contrast, **a simultaneous approach allows for operations to be calculated in parallel, thereby fully utilizing all our available computational resources**. This can lead to significant reductions in computation time.

Because most programming languages default to sequential execution, certain nuances of concurrent programming can be difficult to grasp (e.g., concurrency doesn't necessarily imply simultaneity). Misunderstandings in this regard can lead to flawed program design or incorrect handling of concurrent processes. To address this potential issue, we next revisit this topic in light of the fundamental concepts of tasks and threads.

TASKS AND THREADS

When computing an operation, Julia internally defines a set of instructions to be processed through the concept of **task**. Each of these tasks must be assigned to **a computer thread** for its computation. Since a single task runs on exactly one thread at a time, **the number of threads available on your computer determines the number of tasks that can be computed simultaneously**.

Importantly, each session in Julia begins with a predefined pool of threads. Julia defaults to a single thread, regardless of your computer's hardware. We'll start considering this case, as it provides a convenient starting point for understanding concurrency.

To build intuition, consider two workers A and B, whom we'll think of as employees working for a company. B's job consists of performing the same operation continuously for a certain period of time. In the code, this is represented by summing `1+1` repeatedly for one second. Instead, A's job consists of receiving some delivery, which will arrive after a certain period of time. In the code, this job is represented by performing no computations for two seconds, captured by calling the function `sleep(2)`.

```
function job_A(time_working)
    sleep(time_working)        # do nothing (waiting for some delivery in the example)

    println("A completed his task")
end
```

```
function job_B(time_working)
    start_time = time()

    while time() - start_time < time_working
        1 + 1                  # compute `1+1` repeatedly during `time_working` seconds
    end

    println("B completed his task")
end
```

Due to the lazy nature of function definitions, these code snippets simply describe a set of operations without performing any computation. It's only when we add lines like `job_A(2)` and `job_B(1)` that the operations are sent for computation.

To lay bare the internal steps Julia follows to compute them, let's use a lower-level approach by defining `job_A(2)` and `job_B(1)` as tasks. As shown below, tasks aren't mere abstractions to organize our discussion, but are actual constructs in Julia's codebase.

```
A = @task job_A(2)      # A's task takes 2 seconds
B = @task job_B(1)      # B's task takes 1 second
```

Once tasks are defined, the first step for their computation is to **schedule** them. This means the task is added to the queue of operations the computer's processor will execute. Essentially, scheduling instructs the machine to compute a task as soon as a thread becomes available.

Importantly, multiple tasks can be *processed* concurrently, without implying that they'll be *computed* simultaneously. Indeed, this is the case in a single-thread session. The distinction can be understood through an analogy with juggling: a juggler manages multiple balls at the same time, but only holds one ball at any given moment. Similarly, multiple tasks can be processed simultaneously, even when only one is actively executing on the CPU.

Although true parallelism isn't feasible in single-threaded sessions, concurrency can still offer some benefits. This is due to the possibility of **task switching**, which is enabled by a **task yielding** mechanism. When a task becomes idle, it can voluntarily relinquish control of the thread, allowing

other tasks to utilize the thread's time. By fostering a cooperative approach, concurrency ensures plenty of computer resource utilization at any given time.

In the following, we describe this mechanism in more detail.

SEQUENTIAL AND CONCURRENT COMPUTATIONS

While code is executed sequentially by default, **tasks are designed to compute concurrently**. As a result, adopting a sequential approach requires instructing Julia to execute tasks one at a time. This is achieved by introducing a wait instruction immediately after scheduling a task, ensuring that the task completes its calculation before proceeding.

The code snippet below demonstrates this mechanism by introducing the functions `schedule` and `wait`.

```
A = job_A(2)            # A's task takes 2 seconds
B = job_B(1)            # B's task takes 1 second
```

```
A = @task job_A(2)      # A's task takes 2 seconds
B = @task job_B(1)      # B's task takes 1 second

schedule(A) |> wait
schedule(B) |> wait
```

```
A = @task job_A(2)      # A's task takes 2 seconds
B = @task job_B(1)      # B's task takes 1 second

(schedule(A), schedule(B)) .|> wait
```

Note that `wait` was added even in the concurrent case. Moreover, `wait` call was incorporated after both tasks had been scheduled. Its purpose is to ensure that both tasks can be processed at the same time, while preventing that subsequent operations are processed alongside them.

The example reveals the benefits of task switching under concurrency: although only one task can run at any moment, task A can yield control of the thread to task B when it becomes idle. In the code, the idle state is simulated by the function `sleep`, during which the computer performs no operations. Once task A becomes idle, its state is saved, allowing it to eventually resume execution from where it left off. In the meantime, task B can use that thread's processing time, explaining why B finishes first.

By taking turns efficiently and sharing the single available thread, tasks make the most of the CPU's processing power. This contrasts with a sequential approach, where task A must finish before moving to the next task. The difference is reflected in their execution times, resulting in 2 seconds for the concurrent approach and 3 seconds for the sequential one.

Examples of idle states emerge naturally in real-world scenarios. For instance, it's common when a program is waiting for user input, such as a keystroke or mouse click. It can also arise when browsing the internet, where the CPU may idle while waiting for a server to send data. Task switching is so ubiquitous in certain contexts that we often take it for granted. For instance, I bet you never questioned whether you could use the computer while a document prints in the background!

Note, though, that concurrency with a single thread offers no benefits if both tasks require active computations. This is because the CPU would be fully utilized, leaving no opportunity for task switching. In such cases, the sequential and concurrent approaches are equivalent. In our example, this would occur if task B consisted of computing `1+1` repeatedly, resulting in an execution time of 3 seconds for both approaches.

```
function job(name_worker, time_working)
    start_time = time()

    while time() - start_time < time_working
        1 + 1                  # compute `1+1` repeatedly during `time_working` seconds
    end

    println("$name_worker completed his task")
end
```

```
function schedule_of_tasks()
    A = @task job("A", 2)      # A's task takes 2 seconds
    B = @task job("B", 1)      # B's task takes 1 second

    schedule(A) |> wait
    schedule(B) |> wait
end
```

```
function schedule_of_tasks()
    A = @task job("A", 2)      # A's task takes 2 seconds
    B = @task job("B", 1)      # B's task takes 1 second

    (schedule(A), schedule(B)) .|> wait
end
```

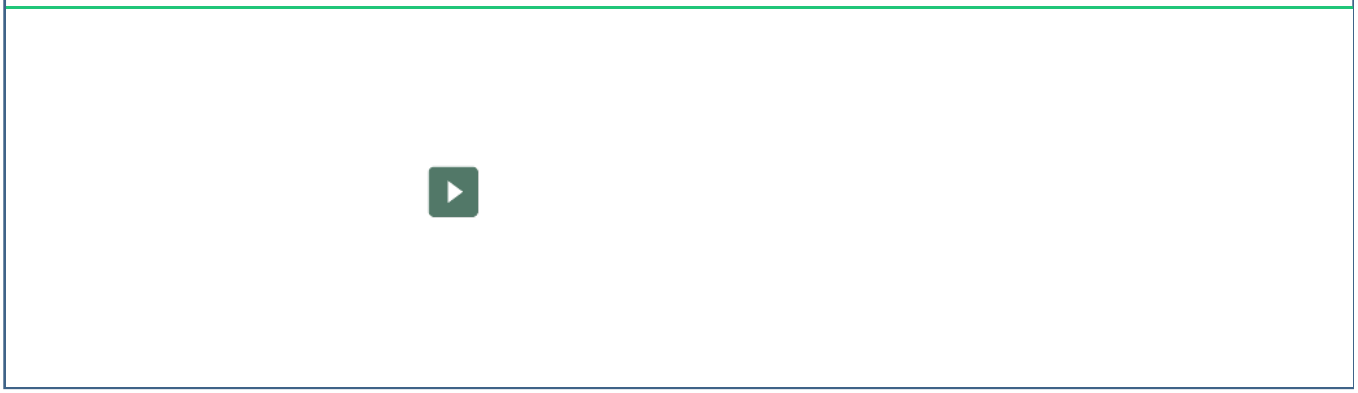
Nonetheless, the key insight from the examples isn't that concurrency is ineffective in a single-thread session. Rather, the main takeaway is the underlying procedure: **when a task is scheduled, the computer attempts to find an available thread for its computation**. For concurrency, this implies

that **starting a session with multiple threads enables parallel code execution**, which is simply called **multithreading**. In the following, we explain this case in more detail.

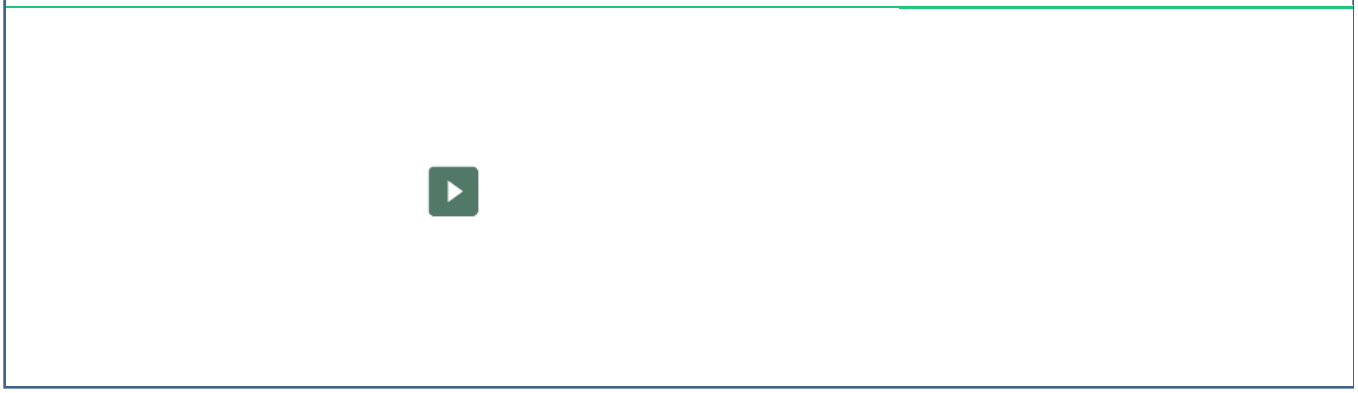
MULTITHREADING

Let's continue considering the last scenario, where both workers A and B perform meaningful computations. The only change we introduce is that Julia's session now starts with more than one thread available. For the concurrent approach, the only code adjustment added is that tasks are no longer "sticky". This is just a technicality indicating that a task can be run on any thread, rather than the thread on which it was first scheduled. Non-sticky tasks allow for a better use of resources, as the task can be computed as soon as a thread becomes available.

```
function schedule_of_tasks()
    A = @task job("A", 2)                    # A's task takes 2 seconds
    B = @task job("B", 1)                    # B's task takes 1 second

    schedule(A) |> wait
    schedule(B) |> wait
end
```



```
function schedule_of_tasks()
    A = @task job("A", 2) ; A.sticky = false      # A's task takes 2 seconds
    B = @task job("B", 1) ; B.sticky = false      # B's task takes 1 second

    (schedule(A), schedule(B)) .|> wait
end
```



Once there's more than one thread available, concurrency implies simultaneity. This means each task runs on a different thread, which is why task B finishes first.

Previewing some of the approaches employed in the next section, let's compare Julia's standard implementation with a multithreaded one. The macro `@spawn`, which will be covered in the next section, offers a simple way to run tasks in a multithreaded environment. It's essentially equivalent to creating and scheduling a non-sticky task. The following code snippets demonstrate both the standard and multithreaded approaches.

```
function schedule_of_tasks()
    A = job("A", 2)             # A's task takes 2 seconds
    B = job("B", 1)             # B's task takes 1 second
end
```

```
function schedule_of_tasks()
    A = @spawn job("A", 2)      # A's task takes 2 seconds
    B = @spawn job("B", 1)      # B's task takes 1 second

    (A,B) .|> wait
end
```

THE IMPORTANCE OF WAITING FOR THE RESULTS

Before concluding this section, it's worth stressing a crucial point: you must always instruct the computer to wait for all operations to complete, before it proceeds with any subsequent computation. This holds true even for concurrent computations. **Failing to wait may produce incorrect results,**

even in a single-threaded environment.

To illustrate this, consider mutating a vector in a single-threaded session, with a one-second delay for each value update. If we don't wait for the mutation to finish, any subsequent operation will be based on the vector's value at the moment it's accessed. This value doesn't necessarily reflect its final state after the mutation, but merely its value at the moment of reference.

For instance, suppose we want to mutate the vector `x = [0,0,0]` into `x = [1,2,3]`. Julia's default sequential execution ensures that the mutation must complete, before continuing with any other operation.

```
# Description of job
function job!(x)
    for i in 1:3
        sleep(1)     # do nothing for 1 second
        x[i] = 1     # mutate x[i]

        println("`x` at this moment is $x")
    end
end

# Execution of job
function foo()
    x = [0, 0, 0]

    job!(x)          # slowly mutate `x`

    return sum(x)
end

output = foo()
println("the value stored in `output` is $(output)")
```

Let's now consider the same implementation but through tasks. In particular, a task performing a mutation is defined in the following way.

```
function job!(x)
    @task begin
        for i in 1:3
            sleep(1)    # do nothing for 1 second
            x[i] = 1    # mutate x[i]

            println("`x` at this moment is $x")
        end
    end
end
```

The following code snippets show the consequences of waiting and not waiting for the mutation to complete.

```
function foo()
    x = [0, 0, 0]

    job!(x) |> schedule             # define job, start execution, don't wait for job to be
done

    return sum(x)
end

output = foo()
println("the value stored in `output` is $(output)")
```

```
function foo()
    x = [0, 0, 0]

     job!(x) |> schedule |> wait       # define job, start execution, only continue when
finished

    return sum(x)
end

output = foo()
println("the value stored in `output` is $(output)")
```

As we can see, without waiting for the mutation to take place, the subsequent operation takes the value of `x` at the moment of execution. Since the mutation hasn't started, `x` is still `x = [0,0,0]`.

11c. Task-Based Parallelism: @spawn

Martin Alfaro
PhD in Economics

INTRODUCTION

The previous section explained the basics of multithreading. In particular, we've shown that operations can be computed either sequentially (Julia's default) or concurrently. The latter approach enables multiple operations to be processed simultaneously, with operations running as soon as a thread becomes available. When Julia's session is initialized with more than one thread, this implies that computations can be executed in parallel.

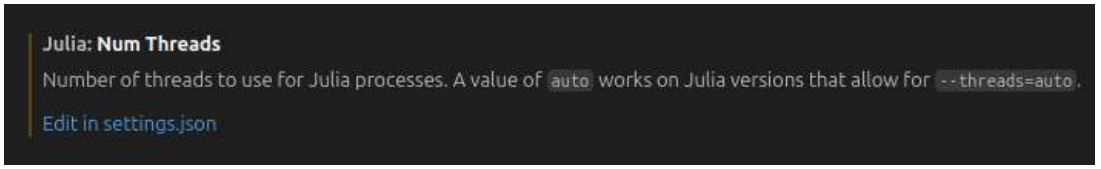
This section will focus on Julia's native multithreading mechanisms, a topic that will span several sections. Our primary goal here is to demonstrate how to write multithreaded code, rather than exploring how and when to apply the technique.

We've deliberately structured our explanation in this way to smooth subsequent discussions. However, a crucial caveat at this point remains necessary: while multithreading can offer significant performance advantages, it's not applicable in all scenarios. In particular, multithreading demands extreme caution in handling dependencies between operations, as mismanagement can lead to silent catastrophic bugs. We'll defer the topic of unsafe-thread operations for now, as identifying them presupposes a basic understanding of parallelism techniques.

ENABLING MULTITHREADING

Julia initializes every session with a given pool of threads available. Each of these threads is responsible for executing a given set of instructions. Consequently, the number of threads delimits the number of instructions that the CPU can handle simultaneously.

By default, Julia only operates with a single thread, requiring setting an alternative number of threads to enable multithreading. You can achieve this in VSCode or VSCodium by going to *File* > *Preferences* > *Settings*. Then, you should search for the keyword *threads*, prompting the following line:



After pressing *Edit in settings.json*, you should add the line `"julia.NumThreads": "auto"`. This will automatically identify the number of threads based on your computer's features (either logical or physical threads available). **Notice that the effects won't take place on the current session.**

To check whether the effects have taken place, use the command `Threads.nthreads()`. This displays the number of threads available in the session. Any number greater than one will indicate that multithreading is activated. Notice also that the changes are permanent, so that every new Julia session will start with the number of threads specified.

Once we have a session with more than one thread, there are several packages for performing multithreaded computations. The focus on this section will be on the built-in package `Threads`, which is automatically imported when you start Julia.

```
# package Threads automatically imported when you start Julia

Threads.nthreads()
```

```
2
```

```
using Base.Threads      # or `using .Threads`

nthreads()
```

```
2
```

> **Warning!** - Loaded Package
>
> All the scripts below assume that you've executed the line `using Base.Threads`. Furthermore, all the examples are based on **a session with two worker threads**.

TASK-BASED PARALLELISM: @SPAWN

The first approach we'll cover is implemented through the macro `@spawn`, which streamlines the application of the previous section's techniques. Specifically, by prepending any operation with `@spawn`, we create a (non-sticky) task that's scheduled right away for its execution. Recall that once a task is scheduled, it'll immediately start its computation if there's a thread available.

Unlike other approaches that we'll present, `@spawn` requires explicitly instructing Julia to wait for the task to complete. The way to do this depends on the nature of the output. For tasks that perform computation and additionally return an output, we have the function `fetch`. This waits for calculations of a task to finish and then returns its output. Since parallel computation requires spawning multiple tasks, the function argument of `fetch` should comprise all the tasks spawned and `fetch` be broadcasted.

In the following, we illustrate `fetch` with two spawned tasks that return vectors as their output.

```
x = rand(10); y = rand(10)

function foo(x)
    a = x .* -2
    b = x .*  2

    a,b
end
```

```
x = rand(10); y = rand(10)

function foo(x)
    task_a = @spawn x .* -2
    task_b = @spawn x .*  2

    a,b = fetch.((task_a, task_b))
end
```

It's important to distinguish between `task_a` and `a`: while `a` refers to the vector created (i.e., the task's output), `task_a` denotes the task creating the vector `a`. The distinction is essential since the function `fetch` only takes a task as its input.

Alternatively, for operations that don't return any output, we can use either the function `wait` or the macro `@sync`. The function `wait` is applied similarly to `fetch`. Instead, the macro `@sync` requires wrapping all operations to be synchronized, which is done by enclosing the operations with the keywords `begin` and `end`.

For the demonstration, let's consider a mutating function. Mutating functions are suitable as an example, since they only modify values of a collection, without returning any output.

```
x = rand(10); y = rand(10)

function foo!(x,y)
    @. x = -x
    @. y = -y
end
```

```
x = rand(10); y = rand(10)

function foo!(x,y)
    task_a = @spawn (@. x = -x)
    task_b = @spawn (@. y = -y)

    wait.((task_a, task_b))
end
```

```
x = rand(10); y = rand(10)

function foo!(x,y)
    @sync begin
        @spawn (@. x = -x)
        @spawn (@. y = -y)
    end
end
```

MULTITHREADING OVERHEAD

To see the advantages of `@spawn` in action, let's compute the sum and maximum of a vector `x`, for which we present a sequential and a simultaneous approach. To clearly shed light on the benefits of parallelization, we also include the time to execute each operation in isolation. The results establish that the time of the sequential procedure is equivalent to the sum of each computation. Instead, thanks to parallelism, the execution time under multithreading is roughly equivalent to the maximum time required for either computation.

```
x = rand(10_000_000)

function non_threaded(x)
    a           = maximum(x)
    b           = sum(x)

    all_outputs = (a,b)
end
```

```
julia> @btime maximum($x)
  7.705 ms (0 allocations: 0 bytes)
julia> @btime sum($x)
  3.131 ms (0 allocations: 0 bytes)
julia> @btime non_threaded($x)
  10.917 ms (0 allocations: 0 bytes)
```

```
x = rand(10_000_000)

function multithreaded(x)
    task_a      = @spawn maximum(x)
    task_b      = @spawn sum(x)

    all_tasks   = (task_a, task_b)
    all_outputs = fetch.(all_tasks)
end
```

```
julia> @btime maximum($x)
  7.705 ms (0 allocations: 0 bytes)
julia> @btime sum($x)
  3.131 ms (0 allocations: 0 bytes)
julia> @btime multithreaded($x)
  7.741 ms (21 allocations: 1.250 KiB)
```

As we can see, the execution time under multithreaded is roughly equivalent to the maximum time for a single operation to complete in isolation. Nonetheless, this equivalence isn't exact. The reason is that **multithreading has a non-negligible overhead**, stemming from the creation and scheduling of tasks. This determines **multithreading isn't beneficial for operations involving small objects**, as the added overhead negates any potential benefits.

To illustrate this, let's compare the execution times of a sequential and multithreaded approach for different sizes of `x`. In the case considered, the single-threaded approach dominates for sizes smaller than 100,000.

```
x_small  = rand(    1_000)
x_medium = rand(  100_000)
x_big    = rand(1_000_000)

function foo(x)
    a           = maximum(x)
    b           = sum(x)

    all_outputs = (a,b)
end
```

```
julia> @btime foo($x_small)
  866.758 ns (0 allocations: 0 bytes)
julia> @btime foo($x_medium)
  59.934 μs (0 allocations: 0 bytes)
julia> @btime foo($x_big)
  620.869 μs (0 allocations: 0 bytes)
```

```
x_small  = rand(    1_000)
x_medium = rand(  100_000)
x_big    = rand(1_000_000)

function foo(x)
    task_a      = @spawn maximum(x)
    task_b      = @spawn sum(x)

    all_tasks   = (task_a, task_b)
    all_outputs = fetch.(all_tasks)
end
```

```
julia> @btime foo($x_small)
  3.245 µs (14.33 allocations: 1.068 KiB)
julia> @btime foo($x_medium)
  55.853 µs (21 allocations: 1.250 KiB)
julia> @btime foo($x_big)
  549.445 µs (21 allocations: 1.250 KiB)
```

11d. Thread-Safe Operations

Martin Alfaro
PhD in Economics

INTRODUCTION

Multithreading allows running multiple threads simultaneously in a single process, enabling operations to be executed in parallel within the same computer. Unlike other forms of parallelization such as multiprocessing, multithreading is distinguished by **the sharing of a common memory space among all tasks**.

This shared memory environment introduces several complexities, determining that **running code in parallel may create side effects if handled improperly**. Basically, the issue arises when multiple threads access and modify shared data, potentially causing unintended consequences in other threads. These potential issues have led to the concept of **thread-safe operations**. They're characterized by the possibility of being executed in parallel without causing any issues (e.g., data corruption, inconsistencies, or crashes).

The section starts by identifying features that make operations unsafe. They'll reveal that common operations such as reductions aren't thread safe, giving rise to incorrect results if multithreading is applied naively. We'll also explore the concept of embarrassingly parallel problems, which are a prime example of thread-safe operations. As the name suggests, these problems can be parallelized directly, without requiring significant program adaptations.

UNSAFE OPERATIONS

We start by presenting some operations that aren't thread-safe. The examples highlight the need for caution when tasks exhibit some degree of dependency, either in terms of operations or shared resources.

WRITING ON A SHARED VARIABLE

The following example highlights the potential pitfalls of writing to a shared variable in a concurrent environment. The scenario considered is such that a scalar variable `output` is initialized to zero. Then, this value is updated within a for-loop that iterates twice, with `output` set to `i` in the *i*-th iteration. The script is as follows.

```
function foo()
    output = 0

    for i in 1:2
        sleep(1/i)
        output = i
    end

    return output
end
```

```
julia> foo()
2
```

```
function foo()
    output = 0

    @threads for i in 1:2
        sleep(1/i)
        output = i
    end

    return output
end
```

```
julia> foo()
1
```

To illustrate the challenges of concurrent execution, we've deliberately introduced a decreasing delay before updating `output`. This delay is implemented using `sleep(1/i)`, causing the first iteration to pause for 1 second and the second iteration to pause for half a second. Although this delay is artificially introduced through `sleep`, it represents the potential delays caused by intermediate computations, which could prevent an immediate update of `output`.

The delay is inconsequential for a sequential procedure, with `output` taking on the values 0, 1, and 2 as the program progresses. However, when executed concurrently, the first iteration completes only after the second iteration has finished. As a result, the sequence of values for `output` is 0, 2, and 1.

While the problem may seem apparent, it can manifest in more complex and subtle ways. In fact, the issue can be exacerbated when each iteration additionally involves reading a shared variable. Next, we consider a scenario like this.

READING AND WRITING A SHARED VARIABLE

Reading and writing shared data doesn't necessarily cause problems. For instance, we'll demonstrate that a parallel for-loop can safely mutate a vector, even though multiple threads are simultaneously modifying a shared object (the vector). However, in scenarios where **reading and writing shared data is sensitive to the specific order of thread execution**, it can give rise to a **data race** (also known as **race condition**). The name reflects that the final output will change in each execution, depending on which thread finishes and modifies the data last.

To illustrate the issue, let's keep using an example similar to the outlined above. We modify the example by introducing the variable `temp`, whose value is updated in each iteration. Moreover, this is a variable shared across threads, and is used to mutate the *i*-th entry of a vector `output`. By introducing a delay before writing each entry of `output`, the example shows that all threads end up using the last value of `temp`, which is 2.

```
function foo()
    out  = zeros(Int, 2)
    temp = 0

    for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo()
2-element Vector{Int64}:
 1
 2
```

```
function foo()
    out  = zeros(Int, 2)
    temp = 0

    @threads for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo()
2-element Vector{Int64}:
 1
 1
```

```
function foo()
    out  = zeros(Int, 2)


    @threads for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo()
2-element Vector{Int64}:
 1
 2
```

As the last tab shows, the issue can be easily circumvented in this case. The solution simply requires defining `temp` as a local variable, which is achieved by avoiding its initialization out of the for-loop. By doing so, each thread will refer to its own local copy of `temp`.

Beyond this specific solution, the example aims to highlight the subtleties of parallelizing operations. To further illustrate it, we next examine a more common scenario where data races occur: reductions.

RACE CONDITIONS WITH REDUCTIONS

To illustrate the issue with reductions, let's consider the sum operation. The gist of the problem lies in that the variable accumulating the sum is accessed and modified by all threads in each iteration.

```
x = rand(1_000_000)

function foo(x)
    output = 0.

    for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
500658.01158503356
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.

    @threads for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
21534.22602627773
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.

    @threads for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
21342.557817155746
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.

    @threads for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
21664.133622716112
```

The key insight from this example isn't that reductions are incompatible with multithreading. Rather, that the strategy to apply multithreading needs to be adapted accordingly.

In the following, we'll consider the simplest case to apply multithreading, which is referred to as embarrassingly parallel. Its distinctive feature is that multithreading can be applied without any transformation of the data. After that, we'll show scenarios that can handle dependent operations like reductions.

EMBARRASSINGLY PARALLEL PROBLEMS

The simplest scenario in which multithreading can be applied is known as an **embarrassingly parallel problem**. The term highlights the ease with which code can be divided for parallel execution. It comprises programs consisting of multiple independent and identical subtasks, not requiring interaction with one another to produce the final output. This independence allows for seamless parallelization, providing complete flexibility in the order of task execution.

In for-loops, one straightforward way to parallelize these problems is given by the macro `@threads`. This is a form of thread-based parallelism, where the distribution of work is based on the number of threads available. Specifically, `@threads` attempts to evenly distribute the iterations, in an effort to balance the workload. The approach contrasts with `@spawn`, which is a task-based parallelism where iterations are divided according how the user has manually defined tasks. Unlike `@spawn`, which requires a manual synchronization of the tasks, `@threads` automatically schedules the tasks and waits for their completion before proceeding with any further operations. This is demonstrated below.

```
x_small  = rand(    1_000)
x_medium = rand(  100_000)
x_big    = rand(1_000_000)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> @btime foo($x_small)
  3.043 µs (1 allocations: 7.938 KiB)
julia> @btime foo($x_medium)
  315.751 µs (2 allocations: 781.297 KiB)
julia> @btime foo($x_big)
  3.326 ms (2 allocations: 7.629 MiB)
```

```
x_small  = rand(    1_000)
x_medium = rand(  100_000)
x_big    = rand(1_000_000)

function foo(x)
    output = similar(x)

    @threads for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> @btime foo($x_small)
  10.139 μs (122 allocations: 20.547 KiB)
julia> @btime foo($x_medium)
  42.044 μs (123 allocations: 793.906 KiB)
julia> @btime foo($x_big)
  340.589 μs (123 allocations: 7.642 MiB)
```

In the next section, we provide a thorough analysis of the differences between `@threads` and `@spawn`.

11e. Parallel For-Loops: @threads

Martin Alfaro
PhD in Economics

INTRODUCTION

Parallelism techniques are aimed at code that performs multiple operations. This makes it a natural fit for for-loops. By using the macro `@spawn` introduced in the previous section, we can parallelize for-loops through task-based parallelism. In an upcoming section, we'll demonstrate that `@spawn` is flexible enough to split iterations into tasks in various ways. For now, we'll consider a simple (inefficient) case where each iteration defines a separate task. The coding implementing this technique is shown below.

```
@sync begin
    for i in 1:4
        @spawn println("Iteration $i is computed on Thread $(threadid())")
    end
end
```

```
Iteration 1 is computed on Thread 1
Iteration 2 is computed on Thread 2
Iteration 4 is computed on Thread 2
Iteration 3 is computed on Thread 2
```

```
@sync begin
    @spawn println("Iteration 1 is computed on Thread $(threadid())")
    @spawn println("Iteration 2 is computed on Thread $(threadid())")
    @spawn println("Iteration 3 is computed on Thread $(threadid())")
    @spawn println("Iteration 4 is computed on Thread $(threadid())")
end
```

```
Iteration 1 is computed on Thread 1
Iteration 2 is computed on Thread 2
Iteration 3 is computed on Thread 1
Iteration 4 is computed on Thread 2
```

When there are only a few iterations involved in a for-loop, creating one task per iteration can be a straightforward and effective way to parallelize the code. However, as the number of iterations increases, the approach becomes less efficient due to the overhead of task creation. To mitigate this issue, we need to consider alternative ways of parallelizing for-loops.

One such alternative is to create tasks that encompass multiple iterations, rather than just one iteration per task. The techniques to do this, which will be explored in following sections, offers more granular control, but at the expense of adding substantial complexity to the code.

In light of this, Julia provides the `@threads` macro from the package `Threads`, with the goal of reducing the overhead of task creation while keeping the parallelization simple. This is achieved by dividing the set of iterations evenly among threads, thereby restricting the creation of tasks to the number of threads available.

The following example demonstrates the implementation of `@threads`, highlighting its difference from the approach using `@spawn`. The scenario considered is based on 4 iterations and 2 worker threads, where we also display the thread on which each iteration is executed. This is achieved by the `threadid()` function, which identifies the ID of the thread computing the operation.

```
for i in 1:4
    println("Iteration $i is computed on Thread $(threadid())")
end
```

```
Iteration 1 is computed on Thread 1
Iteration 2 is computed on Thread 1
Iteration 3 is computed on Thread 1
Iteration 4 is computed on Thread 1
```

```
@threads for i in 1:4
    println("Iteration $i is computed on Thread $(threadid())")
end
```

```
Iteration 1 is computed on Thread 1
Iteration 2 is computed on Thread 1
Iteration 3 is computed on Thread 2
Iteration 4 is computed on Thread 2
```

```
@sync begin
    for i in 1:4
        @spawn println("Iteration $i is computed on Thread $(threadid())")
    end
end
```

```
Iteration 2 is computed on Thread 2
Iteration 1 is computed on Thread 1
Iteration 4 is computed on Thread 2
Iteration 3 is computed on Thread 2
```

The key distinction between `@threads` and `@spawn` lies in their thread allocation strategies. Thread assignments with `@threads` are predetermined: before the for-loop begins, the macro pre-allocates threads and distributes iterations evenly. Thus, each thread is assigned a fixed number of iterations upfront, creating a predictable workload distribution. In the example, the feature is reflected in the allocation of two iterations per thread. In contrast, `@spawn` creates a separate task for each iteration, dynamically scheduling them as soon as a thread becomes available. This method allows for more flexible thread utilization, with task assignments adapting in real-time to the current system load and available thread capacity. For instance, in the given example, one thread ended computing three out of the four iterations.

@SPAWN VS @THREADS

The macros `@threads` and `@spawn` embody two distinct approaches to work distribution, thus catering to different types of scenarios. By comparing the creation of one task per iteration relative to `@threads`, we can highlight the inherent trade-offs involved in parallelizing code.

`@threads` employs a coarse-grained approach, making it well-suited for workloads with similar computational requirements. By reducing the overhead associated with task creation, this approach excels in scenarios where tasks have comparable execution times. However, it's less effective in handling workloads with unbalanced execution times, where some iterations are computationally intensive while others are relatively lightweight.

In contrast, `@spawn` adopts a fine-grained approach, treating each iteration as a separate task that can be scheduled independently. This allows for more flexible work distribution, with tasks dynamically allocated to available threads as soon as they become available. As a result, `@spawn` is particularly well-suited for scenarios with varying computational efforts, where iteration completion times can differ significantly. While this approach has a bigger overhead due to the creation of numerous smaller tasks, it simultaneously enables more efficient resource utilization. This is because no thread remains idle while tasks await computation.

In the following, we demonstrate the efficiency of the approaches under each scenario. With this goal, consider a situation where the *i*-th iteration computes `job(i;time_working)`. This function represents some calculations that are performed during `time_working` seconds. It's formally defined as follows.

```
function job(i; time_working)
    println("Iteration $i is on Thread $(threadid())")

    start_time = time()

    while time() - start_time < time_working
        1 + 1                   # compute `1+1` repeatedly during `time_working` seconds
    end
end
```

Note that `job` additionally identifies the thread on which it's running and displays it on the REPL.

Based on a for-loop with four iterations and a session with two worker threads, we next consider two scenarios. They differ by the computational workload of the iterations.

SCENARIO 1: UNBALANCED WORKLOAD

The first scenario represents a situation with unbalanced work, where some iterations require more computational effort. The feature is captured by assuming that the *i*-th iteration has a duration of `i` seconds. A visual representation of the problem is as follows.

XXXXX

We start by presenting the coding implementing each approach, and then provide explanations for each.

```
function foo(nr_iterations)
    for i in 1:nr_iterations
      job(i; time_working = i)
    end
end
```

```
Iteration 1 is on Thread 1
Iteration 2 is on Thread 1
Iteration 3 is on Thread 1
Iteration 4 is on Thread 1
  10.000 s (40 allocations: 1.562 KiB)
```

```
function foo(nr_iterations)
    @threads for i in 1:nr_iterations
        job(i; time_working = i)
    end
end
```

```
Iteration 1 is on Thread 1
Iteration 3 is on Thread 2
Iteration 2 is on Thread 1
Iteration 4 is on Thread 2
  7.000 s (51 allocations: 2.625 KiB)
```

```
function foo(nr_iterations)
    @sync begin
        for i in 1:nr_iterations
            @spawn job(i; time_working = i)
        end
    end
end
```

```
Iteration 1 is on Thread 1
Iteration 2 is on Thread 2
Iteration 3 is on Thread 1
Iteration 4 is on Thread 2
  6.000 s (69 allocations: 3.922 KiB)
```

Given the execution times for each iteration, a sequential approach would take 10 seconds. As for parallel implementations, `@threads` ensures that there are as many tasks created as number of threads. In the example, this means that there two tasks are created, with the first task computing iterations 1 and 2, and the second task computing iterations 3 and 4. As a result, the overall execution time is reduced to 7 seconds.

In contrast, `@spawn` creates a separate task for each iteration, which increases the overhead of task creation. Although the overhead is negligible in this example, it can be appreciated in the increased memory allocation. Despite this disadvantage, the approach allows each iteration to be executed as soon as a thread becomes available. Given the varying execution times between iterations, this dynamic allocation becomes advantageous, enabling iterations 3 and 4 to run in parallel.

The example demonstrates this, where iterations 1 and 2 are now executed on different threads. Since the first iteration only requires one second, the thread becomes available to compute the third iteration immediately. The final distribution of tasks on threads is such that iterations 1 and 3 are executed on one thread, while iterations 2 and 4 are executed on the other thread. This results in a total execution time of 6 seconds.

SCENARIO 2: BALANCED WORKLOAD

Consider now a scenario where the execution of `job` requires exactly the same time regardless of the iteration considered. To make the overhead more apparent, we'll use a larger number of iterations. In this context, `@threads` ensures parallelization with a reduced overhead, explaining why it's faster than the approach relying on `@spawn`.

```
function foo(nr_iterations)
    fixed_time = 1 / 1_000_000

    for i in 1:nr_iterations
      job(i; time_working = fixed_time)
    end
end
```

```
julia> @btime foo(1_000_000)
  1.717 s (without a warmup) (0 allocations: 0 bytes)
```

```
function foo(nr_iterations)
    fixed_time = 1 / 1_000_000

    @threads for i in 1:nr_iterations
        job(i; time_working = fixed_time)
    end
end
```

```
julia> @btime foo(1_000_000)
  858.399 ms (11 allocations: 1.094 KiB, without a warmup)
```

```
function foo(nr_iterations)
    fixed_time = 1 / 1_000_000

    @sync begin
        for i in 1:nr_iterations
            @spawn job(i; time_working = fixed_time)
        end
    end
end
```

```
julia> @btime foo(1_000_000)
  1.270 s (5000021 allocations: 498.063 MiB, 19.16% gc time, without a warmup)
```

11f. Applying Parallelization

Martin Alfaro
PhD in Economics

INTRODUCTION

So far, we've explored two approaches for parallelizing code, tailored to different scenarios. The first one was `@spawn`. By allowing us to define the specific tasks to be processed, this provides granular control over the parallelization process. For its part, `@threads` represents a simplified approach to parallelizing for-loops, where the tasks spawned are automatically handled. In particular, they're defined based on the number of available threads.

Additionally, we've pointed out that, due to inherent dependencies between computations, not all tasks lend themselves equally to parallelization. Specifically, when tasks aren't embarrassingly parallel, a naive approach can lead to severe issues like race conditions.

All this implies that, to this point, our discussion has been exclusively focused on the syntax and work distribution of these approaches. Therefore, we have yet to address how to apply multithreading in real scenarios, including strategies to deal with dependencies.

This section and the next one aim to bridge this gap, providing practical guidance on implementing multithreading. With this goal, we begin by showing the advantages of parallelizing at a coarse level compared to parallelization at individual operations. After this, we introduce a more general method based on `@spawn` to parallelize for-loops. This enables us to have control over the partition of iterations that define tasks. Its main advantage is the provision of a flexible distribution of iterations among tasks, thus giving us further control over task creation. The technique also makes it possible to apply multithreading under a ubiquitous type of dependency: reductions.

BETTER TO PARALLELIZE AT THE TOP

Given the overhead involved in multithreading, there's an inherent trade off between creating new tasks and utilizing all our machine resources. Consequently, we must first carefully consider whether parallelizing our code is worthy. For instance, multithreading is only justified with collections if their sizes are substantial enough to make up for overhead involved. Otherwise, single-threaded approaches will consistently outperform parallelized ones.

In case multithreading is worthwhile, we immediately face another decision: at what level to parallelize code. In the following, we'll demonstrate that **parallelism at the highest level possible is preferable to multithreading individual operations**. The reason is that the former minimizes the overhead involved in creating tasks, thus resulting in faster execution times.

Note that the level of parallelization is limited by the degree of dependence between operations, explaining why we qualify the highest level as the one that's *possible*. For instance, in problems requiring serial computation, the best we can achieve is a parallelization at each individual step.

To illustrate all this, let's consider a for-loop where each iteration needs to sequentially compute three operations.

JULIA'S DEFAULT

```
step1(a) = a ^ 2
step2(a) = sqrt(a)
step3(a) = log(a + 1)

function all_steps(a)
   y      = step1(a)
   z      = step2(y)
   output = step3(z)

   return output
end

function foo(x)
   output = similar(x)

   for i in eachindex(output)
      output[i] = all_steps(x[i])
   end

   return output
end

x_small  = rand(  1_000)
x_large  = rand(100_000)
```

```
julia> @btime foo($x_small)
  4.923 µs (1 allocations: 7.938 KiB)
julia> @btime foo($x_large)
  507.797 µs (2 allocations: 781.297 KiB)
```

PARALLELIZATION AT THE HIGHEST LEVEL POSSIBLE

```
step1(a) = a ^ 2
step2(a) = sqrt(a)
step3(a) = log(a + 1)

function all_steps(a)
   y      = step1(a)
   z      = step2(y)
   output = step3(z)

   return output
end

function foo(x)
   output = similar(x)

   @threads for i in eachindex(output)
      output[i] = all_steps(x[i])
   end

   return output
end

x_small  = rand(  1_000)
x_large  = rand(100_000)
```

```
julia> @btime foo($x_small)
  9.963 µs (122 allocations: 20.547 KiB)
julia> @btime foo($x_large)
  59.551 µs (123 allocations: 793.906 KiB)
```

EACH OPERATION PARALLELIZED

```
step1(a) = a ^ 2
step2(a) = sqrt(a)
step3(a) = log(a + 1)

function parallel_step(f, x)
   output = similar(x)

   @threads for i in eachindex(output)
      output[i] = f(x[i])
   end

   return output
end

function foo(x)
   y      = parallel_step(step1, x)
   z      = parallel_step(step2, y)
   output = parallel_step(step3, z)

   return output
end

x_small  = rand(  1_000)
x_large  = rand(100_000)
```

```
julia> @btime foo($x_small)
  35.938 µs (366 allocations: 61.641 KiB)
julia> @btime foo($x_big)
  87.746 µs (369 allocations: 2.326 MiB)
```

The example clearly illustrates that parallelization is only advantageous when dealing with big collections. This is evidenced by the execution times, where multithreading is only faster when `x_large` is considered. Secondly, an approach where tasks comprise all operations is faster than multithreading each in isolation. This is in part reflected in the reduced memory allocations when all operations are encompassed.

IMPLICATIONS

The strategy of parallelizing code at the top level has significant implications for writing programs. This is especially the case when the code will eventually be applied to multiple objects. It suggests that we should start by writing code for a single object, without considering parallelization. Once the single-case code is thoroughly optimized, parallel execution can be seamlessly integrated at the highest level. The approach not only improves performance, but also simplifies the coding process by streamlining the debugging and testing of code.

A typical example where this strategy arises naturally is scientific simulations, where numerous independent runs of the same model are executed. In this case, the best strategy is to focus on a single-thread codebase for the model, subsequently processing the different runs of the model in parallel.

THE IMPORTANCE OF WORK DISTRIBUTION

Multithreading performance is influenced by the balance of computational workload across iterations. The `@threads` macro is highly effective when each iteration requires a roughly equal processing time. This is because the tasks spawned will comprise an equal amount of iterations. However, scenarios with varying computational effort can pose significant challenges. In those cases, some threads may remain idle, while others will be heavily loaded. This may dramatically reduce the potential performance gains of parallel processing.

To address this issue, we need to have more control over how to distribute work among threads. With the tools introduced so far, we could employ `@spawn` to make each iteration represent a different task. However, this approach becomes extremely inefficient if there is a substantial number of iterations. The reason is that defining many more tasks than the number of threads available results in an unnecessary substantial overhead. The following example reveals this feature, where the execution time of spawning one task per iteration is extremely slow.

@THREADS

```
x = rand(10_000_000)

function foo(x)
    output = similar(x)

    @threads for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> @btime foo($x)
  4.942 ms (123 allocations: 76.306 MiB)
```

@SPAWN

```
x = rand(10_000_000)

function foo(x)
    output = similar(x)

    @sync for i in eachindex(x)
        @spawn output[i] = log(x[i])
    end

    return output
end
```

```
julia> @btime foo($x)
  9.983 s (60001697 allocations: 5.136 GiB)
```

To have more control over the work distribution, we need to partition objects into smaller subsets that can be processed concurrently. Before explaining the implementation of the technique, we begin showing how to partition a collection and its indices. The procedure relies on the `ChunkSplitters` package.

PARTITIONING COLLECTIONS

The easiest way to partition a collection `x` and its indices `x` is through the package `ChunkSplitters`. The package provides two functions for *lazily* partitioning called `chunks` and `index_chunks`. The functions accept `n` and `size` as keyword arguments, depending on the partition to be implemented. Specifically, `n` specifies the number of subsets in which the collection should be divided, where each subset's size attempts to distribute elements evenly. In contrast, `size` provides the number of elements that each subset should contain. Since `size` can't guarantee an even distribution across all subsets, it'll adjust the number of elements in one of the subsets.

The following example considers a variable `x` that comprises the 26 letters of the alphabet. Note that the presentation of the outputs uses `collect`, since `chunks` and `index chunks` are lazy.

PARTITION BY NUMBER OF CHUNKS

```
x             = string.('a':'z')            # all letters from "a" to "z"

nr_chunks     = 5

chunk_indices = index_chunks(x, n = nr_chunks)
chunk_values  = chunks(x, n = nr_chunks)
```

```
julia> collect(chunk_indices)
5-element Vector{UnitRange{Int64}}:
 1:6
 7:11
 12:16
 17:21
 22:26
julia> collect(chunk_values)
5-element Vector{SubArray{String, 1, Vector{String}, Tuple{UnitRange{Int64}}, true}}:
 ["a", "b", "c", "d", "e", "f"]
 ["g", "h", "i", "j", "k"]
 ["l", "m", "n", "o", "p"]
 ["q", "r", "s", "t", "u"]
 ["v", "w", "x", "y", "z"]
```

PARTITION BY SIZE OF CHUNKS

```
x             = string.('a':'z')            # all letters from "a" to "z"

chunk_length  = 10

chunk_indices = index_chunks(x, size = chunk_length)
chunk_values  = chunks(x, size = chunk_length)
```

```
julia> collect(chunk_indices)
3-element Vector{UnitRange{Int64}}:
 1:10
 11:20
 21:26

julia> collect(chunk_values)
3-element Vector{SubArray{String, 1, Vector{String}, Tuple{UnitRange{Int64}}, true}}:
 ["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"]
 ["k", "l", "m", "n", "o", "p", "q", "r", "s", "t"]
 ["u", "v", "w", "x", "y", "z"]
```

One common way to apply partitions for multithreading is by considering a number of chunks that are proportional to the number of worker threads. Moreover, iterations can be based on `enumerate` to get pairs of chunk index and either the subcollection or subindices.

PARTITION BY NUMBER OF THREADS

```
x              = string.('a':'z')            # all letters from "a" to "z"

nr_chunks      = nthreads()

chunk_indices = index_chunks(x, n = nr_chunks)
chunk_values  = chunks(x, n = nr_chunks)

chunk_iter     = enumerate(chunk_indices)    # pairs (i_chunk, chunk_index)
```

```
julia> collect(chunk_indices)
24-element Vector{UnitRange{Int64}}:
 1:2
 3:4
 ⋮
 25:25
 26:26
julia> collect(chunk_iter)
24-element Vector{Tuple{Int64, UnitRange{Int64}}}:
 (1, 1:2)
 (2, 3:4)
 ⋮
 (23, 25:25)
 (24, 26:26)
julia> collect(chunk_values)
24-element Vector{SubArray{String, 1, Vector{String}, Tuple{UnitRange{Int64}}, true}}:
 ["a", "b"]
 ["c", "d"]
 ⋮
 ["y"]
 ["z"]
```

WORK DISTRIBUTION: DEFINING TASKS THROUGH CHUNKS

To define tasks through chunks, we need to partition the collection into smaller subsets that can be processed concurrently. We've already discussed how to apply these techniques using the `ChunkSplitters` package.

Let's consider a simple example where we want to parallelize a for-loop:

@THREADS

```
x = rand(10_000_000)

function foo(x)
    output = similar(x)

    @threads for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> @btime foo($x)
  4.942 ms (123 allocations: 76.306 MiB)
```

@SPAWN

```
x = rand(10_000_000)

function foo(x, nr_chunks)
    chunk_ranges = index_chunks(x, n=nr_chunks)
    output       = similar(x)

    @sync for chunk in chunk_ranges
        @spawn (@views @. output[chunk] = log(x[chunk]))
    end

    return output
end
```

```
julia> @btime foo($x)
  5.301 ms (156 allocations: 76.308 MiB)
```

@SPAWN (EQUIVALENT)

```
x = rand(10_000_000)

function foo(x, nr_chunks)
    chunk_ranges = index_chunks(x, n=nr_chunks)
    output       = similar(x)
    task_indices = Vector{Task}(undef, nr_chunks)

    for (i, chunk) in enumerate(chunk_ranges)
        task_indices[i] = @spawn (@views @. output[chunk] = log(x[chunk]))
    end

    return wait.(task_indices)
end
```

```
julia> @btime foo($x)
  4.893 ms (148 allocations: 76.307 MiB)
```

The approach provides more control over the allocation of tasks to threads. For instance, we could define the number of chunks as proportional to the number of worker threads.

@SPAWN

```
x = rand(10_000_000)

function foo(x, nr_chunks)
    chunk_ranges = index_chunks(x, n=nr_chunks)
    output       = similar(x)

    @sync for chunk in chunk_ranges
        @spawn (@views @. output[chunk] = log(x[chunk]))
    end

    return output
end
```

```
julia> @btime foo($x, 1 * nthreads())
  4.923 ms (156 allocations: 76.308 MiB)
julia> @btime foo($x, 2 * nthreads())
  4.898 ms (301 allocations: 76.323 MiB)
julia> @btime foo($x, 4 * nthreads())
  4.462 ms (589 allocations: 76.349 MiB)
```

@SPAWN

```
x = rand(10_000_000)

function compute!(output, x, chunk)
     @turbo for j in chunk
        output[j] = log(x[j])
     end
end

function foo(x, nr_chunks)
    chunk_ranges = index_chunks(x, n=nr_chunks)
    output       = similar(x)

    @sync for chunk in chunk_ranges
        @spawn compute!(output, x, chunk)
    end

    return output
end
```

```
julia> @btime foo($x, 1 * nthreads())
  4.418 ms (132 allocations: 76.307 MiB)
julia> @btime foo($x, 2 * nthreads())
  4.546 ms (253 allocations: 76.320 MiB)
julia> @btime foo($x, 4 * nthreads())
  3.659 ms (493 allocations: 76.344 MiB)
```

PARALLEL REDUCTIONS

So far, our exploration of parallelization has focused on cases with independent tasks. In particular, the iterations in for-loops were independent, thereby defining an embarrassingly parallel program. This was a deliberate choice, as not all tasks lend themselves to parallelization, due to inherent dependencies between computations. In particular, when tasks aren't embarrassingly parallel, a naive approach for their computation can not only lead to inefficiencies, but actually introduce critical issues, including incorrect results.

Nonetheless, depending on the nature of the dependency, we can adapt our parallelization strategy to still benefit from parallelization. One strategy involves dividing a large task into *smaller independent sub-tasks* that can be executed concurrently. By doing so, we can execute the subtasks in parallel without compromising the correctness of the results, as each sub-task remains independent of the others. Once all sub-tasks complete, their results are combined to generate the final output.

The approach is particularly suitable for reduction operations. Moreover, its technical implementation is a variant of the partition techniques previously presented. To illustrate it, we consider the simplest scenario possible, where we simply compute the sum of elements of a vector `x`.

JULIA'S DEFAULT (SEQUENTIAL)

```
x = rand(10_000_000)

function foo(x)
    output = 0.

    for i in eachindex(x)
        output += x[i]
    end

    output
end
```

```
julia> @btime foo($x)
  4.909 ms (0 allocations: 0 bytes)
```

@THREADS

```
x = rand(10_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    partial_outputs = Vector{Float64}(undef, length(chunk_ranges))

    @threads for (i,chunk) in enumerate(chunk_ranges)
        partial_outputs[i] = sum(@view(x[chunk]))
    end

    return sum(partial_outputs)
end
```

```
julia> @btime foo($x)
  1.163 ms (122 allocations: 13.234 KiB)
```

@SPAWN

```
x = rand(10_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    partial_outputs = Vector{Float64}(undef, length(chunk_ranges))

    @sync for (i, chunk) in enumerate(chunk_ranges)
        @spawn partial_outputs[i] = sum(@view(x[chunk]))
    end

    return sum(partial_outputs)
end
```

```
julia> @btime foo($x)
  1.163 ms (155 allocations: 13.750 KiB)
```

FALSE SHARING IN REDUCTIONS

Cache contention represents a performance challenge where multiple processor cores compete for shared cache resources. A particular manifestation of this issue known as **false sharing** arises when multiple cores access data stored in the same cache line. To understand this issue, it's essential to grasp how CPU caches function.

Processors use caches to store copies of frequently accessed data. They represent a smaller and faster memory unit than RAM, and are organized into fixed-size blocks called cache lines (typically 64 bytes). When data is needed, the processor first checks the cache. If the data isn't found, this must be retrieved from RAM and store a copy in the cache, a process that's significantly slower.

When multiple cores access data within the same cache line, the transfer of data follows a cache coherency protocol. This is designed to maintain data consistency across cores. This protocol can lead to situations where one core accesses data that isn't modified by another core, yet shares a cache

block with altered data. In such cases, the entire cache line may be invalidated, forcing the cores to reload the entire cache block, despite there being no logical necessity to do so. This phenomenon is known as false sharing, and can cause unnecessary cache invalidations and refetches. The consequence is a significant degradation of the program's performance, particularly if threads frequently modify their variables.

While false sharing can occur in various multithreading scenarios, it's particularly prevalent in reduction operations. This case will be our focus next.

AN ILLUSTRATION AND SOLUTIONS FOR REDUCTIONS

Let's consider a simple scenario where the elements of a vector are summed after applying a logarithmic transformation. We'll present two multithreaded implementations to illustrate the impact of false sharing on performance.

The first implementation is a naive approach that closely resembles a typical sequential implementation. Its goal is to illustrate false sharing. The issue arises because multiple threads are repeatedly reading and writing adjacent memory locations in the `partial_outputs` vector. Since CPU cache lines typically span several vector elements, this leads to cache invalidation and forced synchronization between cores.

In contrast, the second implementation avoids false sharing, and we'll analyze why this is so after presenting the code snippets.

SEQUENTIAL

```
x = rand(10_000_000)

function foo(x)
    output = 0.

    for i in eachindex(x)
        output += log(x[i])
    end

    output
end
```

```
julia> @btime foo($x)
  33.629 ms (0 allocations: 0 bytes)
```

FALSE SHARING

```
x = rand(10_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    partial_outputs = zeros(length(chunk_ranges))

    @threads for (i,chunk) in enumerate(chunk_ranges)
        for j in chunk
            partial_outputs[i] += log(x[j])
        end
    end

    return sum(partial_outputs)
end
```

```
julia> @btime foo($x)
  12.579 ms (122 allocations: 13.234 KiB)
```

LOCAL VARIABLE (@THREADS)

```
x = rand(10_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    partial_outputs = zeros(length(chunk_ranges))

    @threads for (i,chunk) in enumerate(chunk_ranges)
        temp = 0.0
        for j in chunk
            temp += log(x[j])
        end
        partial_outputs[i] = temp
    end

    return sum(partial_outputs)
end
```

```
julia> @btime foo($x)
  3.579 ms (122 allocations: 13.234 KiB)
```

LOCAL VARIABLE (@SPAWN)

```
x = rand(10_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    partial_outputs = zeros(length(chunk_ranges))

    @sync for (i,chunk) in enumerate(chunk_ranges)
        @spawn begin
            temp = 0.0
            for j in chunk
                temp += log(x[j])
            end
            partial_outputs[i] = temp
        end
    end

    return sum(partial_outputs)
end
```

```
julia> @btime foo($x)
  3.379 ms (155 allocations: 13.750 KiB)
```

To address false sharing in parallel reductions, there are several strategies that can be employed. All of them aim to prevent threads from repeatedly accessing the same cache line.

The previous example already presented one solution. It involves introducing a thread-local variable called `temp` to accumulate results. In this way, each thread maintains its own accumulator, writing to the shared array only once at the end.

Two additional solutions are presented below. The first one entails computing the reduction through a separate function. This address false sharing by the same logic as before, where the accumulation is done through a variable local to a function. The second solution involves defining `partial_outputs` as a matrix with extra rows (seven in particular), a technique known as vector padding. This approach guarantees that each thread's accumulator is allocated on a different cache line, so that that concurrent updates don't interfere with each other at the cache level.

FUNCTION

```
x = rand(10_000_000)

function compute(x, chunk)
    temp = 0.0

    for j in chunk
        temp += log(x[j])
    end

    return temp
end

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    partial_outputs = zeros(length(chunk_ranges))

    @threads for (i,chunk) in enumerate(chunk_ranges)
        partial_outputs[i] = compute(x, chunk)
    end

    return sum(partial_outputs)
end
```

```
julia> @btime foo($x)
  3.504 ms (122 allocations: 13.234 KiB)
```

PADDING

```
x = rand(10_000_000)

function foo(x)
    chunk_ranges     = index_chunks(x, n=nthreads())
    partial_outputs  = zeros(7, length(chunk_ranges))

    @threads for (i,chunk) in enumerate(chunk_ranges)
        for j in chunk
            partial_outputs[1,i] += log(x[j])
        end
    end

    return sum(@view(partial_outputs[:,1]))
end
```

```
julia> @btime foo($x)
  3.729 ms (122 allocations: 14.438 KiB)
```

11g. Packages for Multithreading

Martin Alfaro
PhD in Economics

INTRODUCTION

Parallelizing code may seem straightforward at a first glance. However, once we start delving into its implementation, it's rapidly revealed that an effective implementation can be a daunting task. As we've discussed, naive implementations can lead to various issues, including performance problems like suboptimal load balancing or false sharing, and more severe concerns such as data races. Furthermore, even if the necessary skills for a correct implementation were mastered, the added complexity can severely impair the code's readability and maintainability.

To assist users in overcoming these obstacles, several packages for parallelization have emerged. These tools aim to simplify the implementation of multithreading, allowing users to leverage its benefits without grappling with low-level intricacies. In this section, we'll present a few of these packages. In particular, the focus will be on those that facilitate the application of multithreading to embarrassingly parallel problems and reductions.

The first package we explore is `OhMyThreads`. This offers a collection of high-level functions and macros that help developers parallelize operations with minimal effort. For instance, it eliminates the need to manually partition tasks and tackles subtle performance issues like false sharing. We'll then examine the `Polyester` package. Thanks to its reduced overhead, this package is capable of streamlining parallelization for small objects. After this, we revisit the `LoopVectorization` package. In particular, we introduce the macro `@tturbo`, which combines the benefits of SIMD instructions with multithreading.

PACKAGE "OHMYTHREADS"

As the package's documentation claims, `OhMyThreads` strives to be a user-friendly package to seamlessly apply multithreading. Consistent with its minimalist approach, the package only introduces a handful of essential functionalities that could easily be part of Julia's `Base`. The goal is to allow users to implement code parallelization, even if they don't possess deep expertise in the subject.

Specifically, the package provides various higher-order functions and macros that internally handle data-race conditions and performance issues like false sharing. Despite its simplicity, `OhMyThreads` still covers a significant range of scenarios, even supporting reduction operations.

In the following, we present the main higher-functions provided by the package. Before introducing them, let's indicate several features that these functions share. Firstly, the names of these functions in `OhMyThreads` mirror those in `Base`, but adding a prefix of `t`. For instance, the counterpart to `map` is `tmap`. Secondly, `OhMyThreads` offers the option of customizing the parallelization process. This is

achieved through an integration with `ChunkSplitters`, enabling customized work distributions among tasks via two keyword arguments: `nchunks` (or equivalently `ntasks`) to define the number of subsets in the partition, and `chunksize` to specify the number of elements in each subset.

> **Warning!**
> All the code snippets below assume you've already loaded the package with `using OhMyThreads`.

PARALLEL MAPPING

The `tmap` function serves as the multithreaded counterpart to `map`. Its syntax is `tmap(foo, x)`, where `foo` is the transforming function applied to each element of the collection `x`. Unfortunately, applying `tmap` in this form results in a performance loss. This is due to a technical matter arising from the type instability of the object `Task`.

To circumvent this issue and regain the lost performance, we must then explicitly indicate the output's type. To do this, we need the function method `tmap(foo, T , x)`, where `T` represents the *element type* of the output. Thus, if for instance the output is a `Vector{Float64}`, `T` would be `Float64`. Instead of directly declaring `T`, a more flexible alternative is to use `eltype(x)`, making the output's type mirror that of `x`.

The package also provides an in-place version, `tmap!`. In this case, since `tmap!` requires specifying the output vector, there's no need to do any extra work to avoid performance losses.

Below, we illustrate the application of these functions. To provide a basis for comparison, we include results of `map` and `map!` as single-threaded baselines.

```
x                       = rand(1_000_000)


foo(x)                  =  map(log, x)
foo_parallel1(x)        = tmap(log, x)
foo_parallel2(x)        = tmap(log, eltype(x), x)
```

```
julia> foo($x)
  3.254 ms (2 allocations: 7.629 MiB)
julia> foo_parallel1($x)
  1.494 ms (568 allocations: 16.958 MiB)
julia> foo_parallel2($x)
  337.724 μs (155 allocations: 7.642 MiB)
```

```
x                       = rand(1_000_000)
output                  = similar(x)

foo!(output,x)          =  map!(log, output, x)
foo_parallel!(output,x) = tmap!(log, output, x)
```

```
julia> foo($x)
  3.303 ms (0 allocations: 0 bytes)
julia> foo_parallel!($x)
  334.747 μs (150 allocations: 13.188 KiB)
```

`OhMyThreads` additionally provides the option to control the work distribution among tasks. This is done through the keyword arguments `nchunks` and `chunksize`, which are internally implemented via the package `ChunkSplitters`. Specifically, `nchunks` controls the number of subsets in the partition, while `chunksize` sets the number of elements per task. Note that `nchunks` and `chunksize` are mutually exclusive options, so that only one of them can be used at a time.

To illustrate the use `nchunks`, we'll set its value equal to `nthreads()`. By setting a number of chunks equal to the number of worker threads, we're adopting an even distribution among tasks, similar to how `@threads` operates. To set the same number with `chunksize`, we'll make use of the floor division operator `÷`. This is a binary operator that rounds a division down to the nearest integer towards zero. [1]

```
x      = rand(1_000_000)

foo(x) = tmap(log, eltype(x), x; nchunks = nthreads())
```

```
julia> @btime foo($x)
  339.006 μs (155 allocations: 7.642 MiB)
```

```
x      = rand(1_000_000)

foo(x) = tmap(log, eltype(x), x; chunksize = length(x) ÷ nthreads())
```

```
julia> @btime foo($x)
  355.825 μs (164 allocations: 7.643 MiB)
```

> **Do-Block Syntax**
>
> When `tmap` requires passing more complex functions, we can still use an anonymous function. In this case, the do-block syntax comes in handy. It enables the creation of multi-line functions, making code more readable. Below, we show an example.

```
x = rand(1_000_000)

function foo(x)

    output = tmap(a -> 2 * log(a), x)



    return output
end
```

```
x = rand(1_000_000)

function foo(x)

    output = tmap(x) do a
                2 * log(a)
             end

    return output
end
```

ARRAY COMPREHENSIONS

`OhMyThreads` also provides an alternative to `tmap` via array comprehensions. Unlike the standard implementation in `Base`, the version from `OhMyThreads` combines a multithreaded variant of `collect` with a generator. Similarly to `tmap`, specifying the output's element type is necessary to prevent performance losses.

```
x               = rand(1_000_000)
output          = similar(x)

foo(x)          = [log(a) for a in x]
foo_parallel1(x) = tcollect(log(a) for a in x)
foo_parallel2(x) = tcollect(eltype(x), log(a) for a in x)
```

```
julia> foo($x)
  3.231 ms (2 allocations: 7.629 MiB)
julia> foo_parallel1($x)
  1.489 ms (568 allocations: 16.958 MiB)
julia> foo_parallel2($x)
  336.948 μs (155 allocations: 7.642 MiB)
```

REDUCTIONS AND MAP-REDUCTIONS

`OhMyThreads` also offers multithreaded versions of `reduce` and `mapreduce`. They're respectively referred to as `treduce` and `tmapreduce`. These functions internally handle the race conditions inherent in reductions and address performance issues like false sharing. Notably, unlike `map`, these

functions can achieve optimal performance without requiring a specified output type.

```
x               = rand(1_000_000)

foo(x)          =  reduce(+, x)
foo_parallel(x) = treduce(+, x)
```

```
julia> foo($x)
  86.102 µs (0 allocations: 0 bytes)
julia> foo_parallel($x)
  29.542 µs (513 allocations: 43.047 KiB)
```

```
x               = rand(1_000_000)

foo(x)          =  mapreduce(log, +, x)
foo_parallel(x) = tmapreduce(log, +, x)
```

```
julia> foo($x)
  3.385 ms (0 allocations: 0 bytes)
julia> foo_parallel($x)
  389.624 µs (511 allocations: 43.000 KiB)
```

FOREACH AS A FASTER OPTION FOR MAPPINGS

The package also offers an implementation similar to for-loops through the function `tforeach`. Since we haven't covered the single-threaded version `foreach`, we begin by presenting it. The function follows a syntax identical to `map`, and is usually implemented using a do-block syntax, as shown below.

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  3.329 ms (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    foreach(i -> output[i] = log(x[i]), eachindex(x))



    return output
end
```

```
julia> foo($x)
  3.251 ms (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    foreach(eachindex(x)) do i
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  3.265 ms (2 allocations: 7.629 MiB)
```

Despite the similarities of `tforeach` and `tmap`, `tforeach` is more performant. Furthermore, it doesn't incur a performance penalty when the output type isn't specified.

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  3.281 ms (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    tmap(eachindex(x)) do i
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  1.868 ms (571 allocations: 24.589 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    tmap(eltype(x), eachindex(x)) do i
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  582.144 μs (158 allocations: 15.272 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    tmap(eltype(x), eachindex(x)) do i
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  582.144 μs (158 allocations: 15.272 MiB)
```

Just like `tmap`, `tforeach` offers the keyword arguments `nchunks` and `chunksize` to control the workload distribution among worker threads. For the illustration, we use a distribution analogous to the one used above for `tmap`.

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    tforeach(eachindex(x); nchunks = nthreads()) do i
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  340.708 μs (154 allocations: 7.642 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    tforeach(eachindex(x); chunksize = length(x) ÷ nthreads()) do i
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  358.567 μs (161 allocations: 7.643 MiB)
```

POLYESTER: PARALLELIZATION FOR SMALL OBJECTS

> **Warning!**
> All the code snippets below assume you executed `using Polyester` to load the package.

One key limitation of multithreading is its overhead due to the creation and scheduling of tasks. This issue can render parallelization impractical for smaller computational tasks, as the cost of thread management would outweigh any potential performance gain. Considering this, the application of multithreading is commonly reserved for objects sufficiently large to justify the cost.

The `Polyester` package addresses this limitation by implementing techniques that reduce the overhead. In this way, it becomes possible to parallelize objects that, otherwise, would be deemed too small to benefit from multithreading. Importantly, the package requires expressing the code to be parallelized as a for-loop.

To illustrate the benefits of the package, let's compare its performance to traditional methods. The following example considers a for-loop with 500 iterations, a relatively low number for applying multithreading. Indeed, the first tab shows that an approach based on `@threads` is slower than its single-threaded variant. In contrast, `Polyester` achieves comparable performance to the single-threaded variant, despite the low number of iterations. To use `Polyseter`, we simply need to prefix the for-loop with the `@batch` macro.

```
x = rand(500)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  1.552 µs (1 allocations: 4.062 KiB)
```

```
x = rand(500)

function foo(x)
    output = similar(x)

    @threads for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  9.362 µs (122 allocations: 16.672 KiB)
```

```
x = rand(500)

function foo(x)
    output = similar(x)

    @batch for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  992.000 ns (1 allocations: 4.062 KiB)
```

For larger objects, it's worth noting that `Polyester` may not necessarily outperform (or underperform) alternative methods. In such cases, it's recommend benchmarking your particular application.

REDUCTIONS

`Polyester` also supports reduction operations. These can be implemented by prepending the for-loop with the expression `@batch reduce=(<tuple with operation and variable reduced>)`. Notably, Polyester's implementation has been designed to avoid common pitfalls of reductions, such as data races and false sharing, ensuring both correctness and performance. The following example illustrates its application.

```
x = rand(250)

function foo(x)
    output = 0.0

    for i in eachindex(x)
        output += log(x[i])
    end

    return output
end
```

```
julia> @btime foo($x)
  745.289 ns (0 allocations: 0 bytes)
```

```
x = rand(250)

function foo(x)
    output = 0.0

    @batch reduction=( (+, output) ) for i in eachindex(x)
        output += log(x[i])
    end

    return output
end
```

```
julia> @btime foo($x)
  543.889 ns (0 allocations: 0 bytes)
```

We can also incorporate more than one reduction operation per iteration, as demonstrated below.

```
x = rand(250)

function foo(x)
    output1 = 1.0
    output2 = 0.0

    for i in eachindex(x)
        output1 *= log(x[i])
        output2 += exp(x[i])
    end

    return output1, output2
end
```

```
julia> @btime foo($x)
  1.241 µs (0 allocations: 0 bytes)
```

```
x = rand(250)

function foo(x)
    output1 = 1.0
    output2 = 0.0

    @batch reduction=( (*, output1), (+, output2) ) for i in eachindex(x)
        output1 *= log(x[i])
        output2 += exp(x[i])
    end

    return output1, output2
end
```

```
julia> @btime foo($x)
  630.302 ns (0 allocations: 0 bytes)
```

```
x = rand(250)

function foo(x)
    output1 = 1.0
    output2 = 0.0

    @batch reduction=( (*, output1), (+, output2) ) for i in eachindex(x)
        output1  = output1 * log(x[i])
        output2  = output2 + exp(x[i])
    end

    return output1, output2
end
```

```
julia> @btime foo($x)
  641.075 ns (0 allocations: 0 bytes)
```

LOCAL VARIABLES

`Polyester` also treats variables as local per iteration, unlike `@threads`.

```
function foo()
    out  = zeros(Int, 2)
    temp = 0

    for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo($x)
2-element Vector{Int64}:
 1
 2
```

```
function foo()
    out  = zeros(Int, 2)


    @threads for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo($x)
2-element Vector{Int64}:
 1
 2
```

```
function foo()
    out  = zeros(Int, 2)
    temp = 0

    @threads for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo($x)
2-element Vector{Int64}:
 2
 2
```

```
function foo()
    out  = zeros(Int, 2)
    temp = 0

    @batch for i in 1:2
        temp   = i; sleep(i)
        out[i] = temp
    end

    return out
end
```

```
julia> foo($x)
2-element Vector{Int64}:
 1
 2
```

SIMD + MULTITHREADING

> **Warning!**
> All the code snippets below assume you've already loaded the package with `using LoopVectorization`.

We've already covered the package `LoopVectorization` in the section about SIMD instructions. We now revisit this package to demonstrate its ability to combine SIMD with multithreading. The feature is achieved through integration with the `Polyester` package.

The primary approach to implementing the functionality involves the `@tturbo` macro, which provides a parallelized version of `@turbo`. Unlike the @`threads` macro, where the application of SIMD optimizations is left to the compiler's discretion, `@tturbo` automatically applies SIMD.

To illustrate the benefits of `@tturbo`, let's consider an example scenario where SIMD isn't applied automatically by `@threads`, despite that the operation is well-suited for this purpose.

```
x = BitVector(rand(Bool, 100_000))
y = rand(100_000)

function foo(x,y)
    output = similar(y)

    for i in eachindex(x)
        output[i] = ifelse(x[i], log(y[i]), y[i] * 2)
    end

    output
end
```

```
julia> foo($x)
  587.694 μs (2 allocations: 781.297 KiB)
```

```
x = BitVector(rand(Bool, 100_000))
y = rand(100_000)

function foo(x,y)
    output = similar(y)

    @threads for i in eachindex(x)
        output[i] = ifelse(x[i], log(y[i]), y[i] * 2)
    end

    output
end
```

```
julia> foo($x)
  80.625 μs (123 allocations: 793.906 KiB)
```

```
x = BitVector(rand(Bool, 100_000))
y = rand(100_000)

function foo(x,y)
    output = similar(y)

    @tturbo for i in eachindex(x)
        output[i] = ifelse(x[i], log(y[i]), y[i] * 2)
    end

    output
end
```

```
julia> foo($x)
  57.225 μs (2 allocations: 781.297 KiB)
```

The `@tturbo` macro is also available as a broadcasting version. Although the for-loop implementation could be more performant in some scenarios, the broadcasting variant significantly simplifies the syntax. Furthermore, it's particularly useful for parallelizing broadcasting operations,

since no built-in macro currently exists for this purpose. Below, we provide a simple example that demonstrates the improvement in readability achieved by using this variant.

```
x      = rand(1_000_000)

function foo(x)
    output = similar(x)

    @tturbo for i in eachindex(x)
        output[i] = log(x[i]) / x[i]
    end

    return output
end
```

```
julia> foo($x)
  525.304 μs (2 allocations: 7.629 MiB)
```

```
x      = rand(1_000_000)

foo(x) = @tturbo log.(x) ./ x
```

```
julia> foo($x)
  524.273 μs (2 allocations: 7.629 MiB)
```

FLOOPS: PARALLEL FOR-LOOPS (*OPTIONAL*)

> **Warning!**
>
> All the code snippets below assume you've already loaded the package with `using FLoops`.

We conclude this section with a brief overview of the package `FLoops`. The presentation is labeled as optional since its use beyond simple applications could require some workarounds. Moreover, it appears not to be actively maintained.

The primary macro provided by the package is `@floop`, exclusively designed to parallelize for-loops. An example of its usage is provided below.

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  3.353 ms (2 allocations: 7.629 MiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = similar(x)

    @floop for i in eachindex(x)
        output[i] = log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  388.563 µs (157 allocations: 7.645 MiB)
```

`@floop` can also be used for reductions by including `@reduce` at the beginning of the line with a reduction operation. The macro addresses the inherent data race of reductions and avoids false sharing issues.

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    for i in eachindex(x)
        output += log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  3.396 ms (0 allocations: 0 bytes)
```

```
x = rand(1_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    partial_outputs = zeros(length(chunk_ranges))

    @threads for (i,chunk) in enumerate(chunk_ranges)
        for j in chunk
            partial_outputs[i] += log(x[j])
        end
    end

    return sum(partial_outputs)
end
```

```
julia> foo($x)
  1.314 ms (122 allocations: 13.234 KiB)
```

```
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @floop for i in eachindex(x)
        @reduce output += log(x[i])
    end

    return output
end
```

```
julia> foo($x)
  370.835 μs (252 allocations: 17.516 KiB)
```

FOOTNOTES

[1] For example, `5 ÷ 3` would return `1`.