Julia As Your First Programming Language: A Book for Scientists

(Version January 28, 2026)

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** and a pdf version **here**. I'll continue adding new content as I go. You can search for subjects below.

Search

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

WHY JULIA?

Scientific computing demands both an intuitive design and execution speed. Julia is built to deliver both.

SIMPLE AND EXPRESSIVE: Julia's syntax is concise and close to mathematical notation, so you can translate models into readable code that's easy to maintain.

INTERACTIVE PROTOTYPING: rapidly build and explore models in real-time, enabling quick iteration and immediate result inspection.

HIGH PERFORMANCE: Julia is engineered for speed. It's capable of achieving speeds comparable to C or Fortran, without forcing you to rewrite performance-critical parts in another language. Moreover, it offers seamless support for parallel computing, including native multithreading and GPU acceleration.

A UNIFIED COMPUTATIONAL PLATFORM: No need to switch between multiple languages. In Julia, you can preprocess data, perform statistical analysis, generate graphical representations, and implement numerical models. Furthermore, Julia interoperates with other languages and tools, allowing you to leverage the strengths of Python, R, or C++. Its ecosystem includes mature packages for data analysis, plotting, and computational modeling.

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**.

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

e. Slices: Copies vs Views
f. Array Indexing
g. In-Place Operations
h. In-Place Functions

6. Working with Collections
a. Overview and Goals
b. Named Tuples and Dictionaries
c. Chaining Operations
d. Useful Functions for Vectors
e. Illustration - Johnny, the YouTuber

PART II: HIGH PERFORMANCE

7. Introduction to Performance
a. Overview and Goals
b. When To Optimize Code?
c. Benchmarking Execution Time
d. Preliminaries on Types
e. Functions: Type Inference and Multiple Dispatch

8. Type Stability
a. Overview and Goals
b. Defining Type Stability
c. Type Stability with Scalars and Vectors
d. Type Stability with Global Variables
e. Barrier Functions
f. Type Stability with Tuples
g. Type Stability with Higher-Order Functions
h. Type-Stability Gotchas

9. Reducing Memory Allocations
a. Overview and Goals
b. Stack/CPU Registers vs Heap
c. Objects Allocating Memory
d. Slice Views to Decrease Allocations
e. Reductions
f. Lazy Operations
g. Lazy Broadcasting and Loop Fusion
h. Pre-Allocations
i. Static Vectors for Small Collections

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: Contiguous Access and Unit Strides
   f. SIMD: Branchless Code
   g. SIMD Packages

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

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. It 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, including its privacy-focused alternative VS Codium. To get started, you'll need to install the Julia extension of VS Code. 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, with code written in the area above it. To execute 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

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 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

Julia also allows you to work **non-interactively**, 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: the challenge here 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.

> **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 "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 the command `x = "Hello"`. When this is run, several actions take place. First, the computer reserves a memory location to store the object (e.g., at address `0x1234`). 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, so 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: **scalars** refer to single-element objects, and **collections** refer 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.

BOOLEAN VARIABLES

```
x = 2
y = 1

z = (x > y)        # is `x` greater than `y` ?
z = x > y          # equivalent (don't interpret 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 adhere to the syntax `x <symbol> y`, as indicated in our discussion on 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 treatment is necessary, as string variables are important for tasks like specifying paths, displaying messages, and documenting functions.

CHARACTERS

The `Char` type is employed to represent individual characters. Characters are written by enclosing them in single quotes, as in `'x'` for the character `x`. Given Julia's support for Unicode characters, `Char` encompasses not only numbers and letters, but also a wide range of symbols.

```
# 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, 🐒's value at that moment (not 🐒 itself)
z  = '🐒'    # z equals the character 🐒 (entirely independent of the 🐒 variable )
```

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, permitting the representation of 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 lets you embed Julia code directly inside a string. The embedded expression is then evaluated and replaced in the string with its value.

To interpolate an expression, the string must be prefixed with the `$` symbol. If the expression contains spaces, it must be enclosed in curly braces, as in `$()`. Both cases are exemplified below.

```
number_students = 10


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

```
julia> output_text
"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> output_text
"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 scalar variables, which store single-element objects. Now, we'll shift our focus to **collections**, which correspond to **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. Their distinctive feature is that all their elements share a **homogeneous type**. Formally, arrays are objects with type `Array{T,d}`, where `d` is the number of dimensions and `T` is the element type (e.g., `Int64` or `Float64`). Two special categories of arrays are:

- **Vectors**: 1-dimensional arrays. They're represented by the type `Vector{T}`, which is an alias for `Array{T,1}`.
- **Matrices**: 2-dimensional arrays. They're represented by 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 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
```

Note that, as previously indicated, arrays are defined so that all elements share a common type. This requirement, however, doesn't prevent arrays from holding elements of different kinds, such as numbers and strings. The reason is that the shared type can be an abstract type, effectively providing a unifying type for the collection.

For example, defining the vector `[1, 2.5, "Hello"]` is valid, because its elements will be assigned the abstract type `Any`. Recall that `Any` sits at the top of Julia's type hierarchy, encompassing all concrete types supported by the language.

Although arrays that mix element types are possible, they're strongly discouraged for several reasons, including performance.

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

It's also possible to access a subset of elements within `x`. 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
5-element Vector{Int64}:
 4
 5
 6
 7
 8
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 unit increments when omitting `<steps>`. To express the first and last index in a range, Julia provides the keywords `begin` and `end`.

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

```
julia> x
5-element Vector{Int64}:
 4
 5
 6
 7
 8


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

MATRICES (*OPTIONAL*)

Matrices can be defined as collections of row or column vectors. When constructing a matrix from multiple row vectors, each row must be separated by a semicolon `;`. Conversely, if created using multiple column vectors, their elements are separated by spaces.

Note that row vectors are treated as special cases of matrices, with their elements separated by a space. In this form, they're matrices consisting of a single row and multiple columns.

```
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 `X`, 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 MATRIX ELEMENTS

Given a matrix `X`, we can access the element at row `r` and column `c` by `X[r,c]`. Moreover, `X[r,:]` selects all elements across the row `r`, while `X[:,c]` selects all elements of column `c`. For a row vector `Y`, its *i*-th element can be accessed with `X[i]`. [1]

```
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 within a matrix, use the same methods employed for vectors. Their only difference is that these methods must be applied to 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.] Matrices also support linear indexing. For example, a 3×3 matrix `X` accepts indices from 1 to 9. However, unless you intend to iterate over all elements, the two-dimensional notation `X[r,c]` is generally clearer and 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*.

Tuples differ from arrays in several respects. Most notably, tuples can comprise elements with heterogeneous types, without any impact on performance. Moreover, they're characterized by their fixed size and immutability. Both aspects imply that, once a tuple is created, its elements cannot be added, removed, or modified.

Importantly, tuples are only suitable for **storing a small number of elements**. Large tuples will result in slow computations at best, or directly trigger fatal errors. This is why vectors must be opted for when it comes to large collections. While these restrictions might initially seem limiting, they also bring substantial performance gains.

At this stage, we'll limit our discussion to the basic features of tuples, introducing only what's necessary to develop the fundamentals of Julia. A more in-depth exploration will follow in Part II, after we've developed the necessary tools to appreciate their role in high-performance applications.

DEFINITION OF TUPLES

Tuples are defined by enclosing their elements within parentheses `()`, in contrast to the square brackets `[]` used in vectors. When a tuple contains more than one element, the parentheses are optional and often omitted.

Single-element tuples, instead, 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 distinguishes it from the expression `(10)`, which would simply be interpreted as the number `10`.

As with vectors, the syntax for accessing the *i*-th element of a tuple `x` is `x[i]`.

```
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 can be used to assign values to multiple variables at once. This operation is commonly referred to as **unpacking** or **destructuring**. It's accomplished by placing a tuple on the left-hand side of `=` and a collection on the right-hand side, which may be either another tuple or a vector.

```
(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
```

This technique is commonly employed when a function returns multiple values. It provides a convenient syntax to unpack outputs 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

Functions can be broadly categorized into three main categories:

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

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

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

PACKAGES

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

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

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

LOADING PACKAGES AND CALLING FUNCTIONS

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

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

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

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

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

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

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

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

BUILT-IN FUNCTIONS

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

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

Function in Julia	Meaning
`log(x)`	$\ln(x)$

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

Operators as Functions

Most of the symbols employed as operators in Julia are also available as functions. This is illustrated below for several arithmetic operators:

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

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

When functions share the same name across multiple packages, at least one of the packages must be loaded via `import` to prevent naming conflicts. For instance, if both the package `Statistics` and another one called `MyPackage` contain a function called `mean`, Julia will throw an error unless one of them is loaded with `import`. [3]

Beyond resolving naming conflicts, `import` can also enhance code clarity by reducing ambiguity in the meaning of a function. For instance, consider a function called `rank`. This name could reference a wide range of concepts depending on the context (e.g., the rank of a matrix, the order in a list). Explicitly specifying the package when the function is called could clarify its intended purpose.

Remark

`import` may also be beneficial when defining custom functions that will be reused across several projects. For example, imagine defining a function like `table_in_pdf` to export Julia tables to PDF format. While the function name clearly conveys its purpose, someone reading your code might reasonably wonder whether this function comes from a standard package. To avoid this ambiguity, you could place the function

> in a package called `userDefined` and load it with `import UserDefined`. This approach makes it immediately clear that `UserDefined.table_in_pdf` is a custom implementation, rather than part of a standard package.

APPROACHES TO LOADING PACKAGES AND CALLING FUNCTIONS

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

First, users have the possibility of loading only a subset of functions from a package. This is particularly valuable when working with heavy packages which may suffer from significant loading times. For instance, if we only need the function `mean` from `Statistics`, the following two approaches achieve the same result.

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

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

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

using Statistics: mean
mean(x)
```

Note that, in both cases, the function `mean` can be called without prefixing it with the package name. This holds even when the package is loaded via `import`.

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

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

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

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

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

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

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

MACROS

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

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

APPLYING MACROS

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

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

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

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

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

FOOTNOTES

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

[2.] Some built-in functions may require a prefix. For instance, the function `isgreater` must be called via `Base.isgreater`. Furthermore, some submodules are preloaded by default. For instance, the function `Base.Iterators.accumulate` belongs to the `Iterators` submodule of `Base`, and can be directly called using `Iterators.accumulate`.

[3.] Defining a function that shares the name of another package's function isn't necessarily an oversight. For instance, developers could implement their own version of `mean` within a package like `MyPackage`, 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 in creating your own functions is assigning them names. Function names adhere to similar rules as variable names. In particular, they accept Unicode characters, allowing us to specify functions like `∑(x)`. Once defined, functions can be directly called, without any prefix. Thus, a function named `foo` can be simply called by `foo(x)`. [1]

There are two approaches to defining functions. We'll refer to each as the **standard form** and the **compact form**. The standard form is the most general and allows you to write both short and long functions. On the other hand, the compact form is employed for single-line functions and is reminiscent of mathematical definitions.

To illustrate each form, consider a function `foo` that sums two variables `x` and `y`.

STANDARD FORM

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

COMPACT FORM

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

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

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

Below, we illustrate all this.

EXPLICIT OUTPUT

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

    return term2
end
```

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

IMPLICIT OUTPUT

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

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

MULTIPLE OUTPUTS

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

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

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

AN EXPRESSION AS OUTPUT

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

    return term1 + term2
end
```

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

> **Functions without Inputs**
>
> It's possible to define functions that don't require any input arguments, as shown below.

FUNCTION WITHOUT ARGUMENTS

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

The Order In Which Functions Are Defined is Irrelevant

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

CODE SNIPPET 1

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

foo2(x) = 1 + x
```

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

CODE SNIPPET 2

```
foo2(x) = 1 + x

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

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

FUNCTIONS AS OPERATORS

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

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

FUNCTIONS AS OPERATORS

```
x = 1
y = 1

⊕(x,y) = log(x) + log(y)
```

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

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

POSITIONAL AND KEYWORD ARGUMENTS

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

A major drawback of positional arguments is their susceptibility to silent errors: if we accidentally swap the positions of the arguments, the function may still execute and return a wrong result. As the number of arguments grows, the likelihood of introducing such bugs grows, while simultaneously noticing them becomes more difficult.

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

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

POSITIONAL ARGUMENTS

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

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

KEYWORD ARGUMENTS

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

```
julia> foo(x=1, y=1)
2
julia> foo(; x=1, y=1) # alternative notation (only for calling 'foo')
\output{./code/region06e}
```

POSITIONAL AND KEYWORD ARGUMENTS

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

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

KEYWORD ARGUMENTS WITH DEFAULT VALUES

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

POSITIONAL AND KEYWORD ARGUMENTS

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

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

OMITTING POSITIONAL ARGUMENTS

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

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

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

PRIOR ARGUMENTS AS DEFAULT VALUES

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

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

SPLATTING

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

TUPLE SPLATTING

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

z = (2,3)
```

```
julia> foo(z...)
5
```

VECTOR SPLATTING

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

z = [2,3]
```

```
julia> foo(z...)
5
```

ANONYMOUS FUNCTIONS

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

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

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

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

VIA COMPACT FUNCTION

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

output     = map(add_two, x)
```

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

VIA ANONYMOUS FUNCTION

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



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

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

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

VIA COMPACT FUNCTION

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

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

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

VIA ANONYMOUS FUNCTION

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


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

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

THE "DO-BLOCK" SYNTAX

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

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

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

DO-BLOCK SYNTAX

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

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

VIA COMPACT FUNCTION

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

output     = map(add_two, x)
```

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

VIA ANONYMOUS FUNCTION

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


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

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

VIA DO-BLOCK SYNTAX

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

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

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

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

VIA COMPACT FUNCTION

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

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

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

VIA ANONYMOUS FUNCTION

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


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

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

VIA DO-BLOCK SYNTAX

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

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

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

FUNCTION DOCUMENTATION

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

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

STANDARD FORM

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

COMPACT FORM

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

For further details about docstrings, see the official documentation.

FOOTNOTES

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

2. Anonymous functions are also known as *lambda functions* in other programming languages.

3. See here for an example in VS Code.

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, this `x` is considered *local* to that function. Moreover, any reference to `x` within the function refers to this local variable, with no connection to the `x` defined outside the function.
- If a function neither defines a new `x` nor `x` is a function argument, references to `x` inside the function point to the variable defined outside the function (i.e., the global `x`).

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

GLOBAL AND LOCAL VARIABLES

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

Variables local to a function encompass:

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

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

Recognizing whether a variable is local or global is crucial for predicting how a program behaves. Indeed, 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 in practice.

```
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 good programming practice, **functions** are best conceived **as self-contained mini-programs**, each designed to perform a specific and well-defined **task**. When a function successfully captures the implementation of that task, it becomes reusable: we can apply the same operation to different inputs or objects, thus avoiding code duplication. This reusability is one of the key reasons functions are such powerful abstractions. By defining a task once, we can rely on it consistently throughout a program, which not only reduces redundancy but also makes the overall structure easier to maintain and reason about.

To achieve this, a function must clearly express both the logic required to perform the task and the data necessary to accomplish it. Within this perspective, local variables simply act as temporary placeholders that help articulate the mechanics of that task. [1] Since their meaning is tied exclusively to

the function's internal process, it's natural that local variables can't be accessed from outside. Moreover, this ensures that the function only interacts with the rest of the program through its inputs and outputs, thereby preserving its integrity as a self-contained well-defined task.

To illustrate 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 a direct 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 simple 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 have a devastating impact on performance in Julia. 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, $\mathrm{d}t$ in $\int f(t)$ for some function f is simply 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 introduces operations on **iterable collections**, a class of data structures whose elements can be accessed sequentially. Common examples include vectors, tuples, and ranges.

A general way to process iterable collections is by using for-loops, which we'll study later. For now, our focus is on techniques that allow us to apply operations element-wise, without explicitly writing for-loops.

The first approach covered is the `map` function. This applies a given function to each element of a collection, producing a new collection with the transformed values. After this, we'll shift our focus to a fundamental technique in Julia known as **broadcasting**. Its distinctive syntax, which involves appending a dot `.` to the function/operator, makes it easily identifiable throughout the code.

Broadcasting enables the application of functions and operators element-wise. When two collections of the same size are involved, it pairs corresponding elements and applies the operation to each pair. Broadcasting also supports combinations of scalars and same-size collections, where the scalar is expanded to match the collection's size.

> **Broadcasting vs Vectorization**
>
> The terms **broadcasting** and **vectorization** will be used interchangeably throughout the website. Strictly speaking, they're not equivalent: the term vectorization applies when arrays have the same size, while broadcasting is an extension that allows for scalars.

> **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, for-loops are often the key to achieving high 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, although with different names. It's designed to transform collections by applying a function to each of their elements.

Depending on the number of inputs required, `map` can be applied in two different ways. In its simplest form, it takes a single-argument function `foo` and a collection `x`. The syntax is then `map(foo,x)`, returning a new collection with `foo(x[i])` as *i*-th element. A common practice is to use anonymous functions in place of `foo`, as illustrated below.

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

output     = map(log, x)
equivalent = [log(x[1]), log(x[2]), log(x[3])]
```

```
julia> output
3-element Vector{Float64}:
 0.0
 0.693147
 1.09861
julia> equivalent
3-element Vector{Float64}:
 0.0
 0.693147
 1.09861
```

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

output     = map(a -> 2 * a, x)
equivalent = [2*x[1], 2*x[2], 2*x[3]]
```

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

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

To demonstrate its use, let's consider the sum operation. Recall that `+` 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]

output     = map(+, x, y)        # `+` exists as both operator and function
equivalent = [+(x[1],y[1]), +(x[2],y[2]), +(x[3],y[3])]
```

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

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

output     = map((a,b) -> a+b, x, y)
equivalent = [x[1]+y[1], x[2]+y[2], x[3]+y[3]]
```

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

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

output     = map(+, x, y)        # `+` exists as both operator and function
equivalent = [+(x[1],y[1]), +(x[2],y[2])]
```

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

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

output     = map(+, x, y)        # `+` exists as both operator and function
equivalent = [+(x[1],y[1])]
```

```
julia> output
1-element Vector{Int64}:
 0
julia> equivalent
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 combinations of scalars and collections, despite not supporting operations with collections of different sizes. In such cases, 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**. This means that broadcasting is 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.

```julia
# `log(a)` applies to scalars `a`
x          = [1,2,3]

output     = log.(x)
equivalent = [log(x[1]), log(x[2]), log(x[3])]
```

```
julia> output
3-element Vector{Float64}:
 0.0
 0.693147
 1.09861
julia> equivalent
3-element Vector{Float64}:
 0.0
 0.693147
 1.09861
```

```julia
square(a)  = a^2     #user-defined function for a scalar 'a'
x          = [1,2,3]

output     = square.(x)
equivalent = [square(x[1]), square(x[2]), square(x[3])]
```

```
julia> output
3-element Vector{Int64}:
 1
 4
 9
julia> equivalent
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.

```julia
# 'max(a,b)' returns 'a' if 'a>b', and 'b' otherwise
x          = [0, 4, 0]
y          = [2, 0, 8]

output     = max.(x,y)
equivalent = [max(x[1],y[1]), max(x[2],y[2]), max(x[3],y[3])]
```

```
julia> output
3-element Vector{Int64}:
 2
 4
 8
julia> equivalent
3-element Vector{Int64}:
 2
 4
 8
```

```julia
foo(a,b)   = a + b        # user-defined function for scalars 'a' and 'b'
x          = [-2, -4, -10]
y          = [ 2,  4,  10]

output     = foo.(x,y)
equivalent = [foo(x[1],y[1]), foo(x[2],y[2]), foo(x[3],y[3])]
```

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

Broadcasting Applies to Any Function

Broadcasting can be used not only with numeric functions, but functions that take other types as inputs. To illustrate, 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"]

output  = string.(country, is_in, region)
```

```
julia> output
2-element Vector{String}:
 "France is in Europe"
 "Canada is in North America"
```

BROADCASTING OPERATORS

It's also possible to **broadcast operators**, in which case they apply them element-wise across collections. Its use requires prepending a dot ***before*** the operator.

Classifying operators by the number of operands helps apply broadcasting, since it directly determines its syntax. Specifically, recall that *unary operators* are written as `<symbol>x`. Thus, for example, `.√x` broadcasts the operator `√`. Likewise, the syntax for *binary operators* is `x <symbol> y`. For instance, `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]

output = x .+ y
```

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

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


output = .√x
```

```
julia> output
3-element Vector{Float64}:
 1.0
 1.41421
 1.73205
```

BROADCASTING OPERATORS WITH SCALARS

Broadcasting thus far was applied with inputs of the same size. This is because collections of dissimilar size, such as `x = [1,2]` and `y=[3,4,5]`, aren't in general allowed.

One exception to this rule is when broadcasting applies to vectors of equal size combined with scalars. In such 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

output = x .+ y
```

```
julia> output
3-element Vector{Int64}:
  5
 15
 25
```

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

output = x .+ y
```

```
julia> output
3-element Vector{Int64}:
  5
 15
 25
```

Broadcasting Can be Applied with Strings

The example based on strings presented above can be rewritten as follows.

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

output  = string.(country, is_in, region)
```

```
julia> output
2-element Vector{String}:
 "France is in Europe"
 "Canada is in North America"
```

ITERABLE OBJECTS AND BROADCASTING COMBINATION

Broadcasting isn't exclusive to vectors. Indeed, it can be applied to any iterable collection, including tuples and ranges.

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

```
julia> log.(x)
(0.0, 0.693147, 1.09861)
julia> x .+ x
(2, 4, 6)
```

```
x = 1:3
```

```
julia> log.(x)
3-element Vector{Float64}:
 0.0
 0.6931471805599453
 1.0986122886681098
julia> x .+ x
2:2:6
```

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

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

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

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

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

square(x) = x^2
output    = square.(x .+ y)
```

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

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

square(x) = x^2
output    = @. square(x + y)
```

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

BROADCASTING FUNCTIONS VS BROADCASTING OPERATORS

We've demonstrated that both functions and operators can be broadcast. 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 broadcast operation.

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

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

number_squared(a) = a ^ 2                # function for scalar 'a'
output            = number_squared.(x)
```

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

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

vector_squared(x) = x .^ 2               # function for a vector 'x'
output            = vector_squared(x)    # '.' not needed (it'd be redundant)
```

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

While both approaches yield the same output, **defining a function that operates on a scalar is generally the better choice**. There are two main reasons for this claim.

First, a scalar function such as `number_squared(a)` can be flexibly applied to both individual values and collections: we can simply call the function directly for scalars or rely on its broadcasted form for collections. This means the function itself remains general-purpose, without being tied to a specific type of input.

Second, the broadcasting syntax makes the programmer's intent explicit. Continuing with the example, writing `number_squared.(x)` clearly indicates that the operation is applied element-wise across the collection. By contrast, a function like `vector_squared(x)` hides this detail, leaving the reader to infer that the computation is performed element-wise.

BROADCASTING OVER ONLY ONE ARGUMENT

When we broadcast a function or operator over some vectors `x` and `y`, both objects are simultaneously iterated. Yet, there are instances where we want only one argument to vary while keeping the other fixed.

A typical scenario arises when checking whether elements from `x` match any values in a predefined list `y`. To illustrate this, let's first introduce the function `in(a, list)`, which determines whether the scalar `a` equals some element in the vector `list`. For instance, `in(2, [1,2,3])` evaluates to `true`, since `2` is contained in `[1,2,3]`.

Suppose now that, instead of a scalar `a`, we have a vector `x` and the goal is to verify whether *each* of the elements in `x` is present in `list = [1,2,3]`. Specifically, our aim is to verify if `1` belongs to `[1,2,3]`, if `2` belongs to `[1,2,3]`, and if `3` belongs to `[1,2,3]`. Below, we show that this operation can't be directly implemented via a simple broadcast version of `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 length or one of them be a scalar. In the second example, although the expression yields an output, it's not the intended result: it performs pairwise comparisons, thus checking whether `1==1`, `2==2`, and `4==3`.

Intuitively, we need a mechanism to inform Julia that `list` should be treated as a single fixed argument while iterating over `x`. This can be accomplished in two different ways. First, we can enclose `list` in a collection (e.g., a vector or tuple).

While it's possible to use any collection to wrap `list`, we'll see in Part II of the book that there's some performance penalty involved when creating vectors. Consequently, **you should prefer tuples when implementing this approach**. This requires inserting `(list,)` as the function argument, which defines a tuple whose only element is `list`. [1] Alternatively, we can rely on the function `Ref`. This requires expressing the function argument as `Ref(list)`, which makes `list` be treated as a single element.

Below, we demonstrate each approach. For completeness, we also show the case where `list` is wrapped in a vector.

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

output = in.(x, [list])
```

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

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

output = in.(x, (list,))
```

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

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

output = in.(x, Ref(list))
```

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

The vector returned in each case has a type `BitVector`, where `1` corresponds to `true` and `0` to `false`. Thus, the result could equivalently be expressed as `[true, false, false]`. This reflects that `x[1]` equals `2` and `2` belongs to `list`, whereas `x[2]` and `x[3]` don't equal any element in `list`.

While the previous example focused on functions, the same principle extends to broadcast operators. This can be illustrated through the `∈` operator, which serves a similar purpose to the `in` function: it determines whether a particular element exists within a collection. [2]

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

output = x .∈ (list,)       # only 'x[1]' equals an element in 'list'
```

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

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

output = x .∈ Ref(list)     # only 'x[1]' equals an element in 'list'
```

```
julia> output
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 a sequence of functions, each evaluated with a single argument. [3] For instance, the curried form of `f(x,y)` would be `f(x)(y)`, providing 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
curried1(x)    = (y -> addition(x,y))
curried2       = x -> (y -> addition(x,y))
```

```
julia> curried1(2)(1)
5
julia> curried2(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 broadcast 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> n.(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.] Recall that tuples with a single element must be written with a trailing comma, as in `(list,)`. Instead, `(list)` is interpreted as the variable `list`, and hence treated as a vector.

[2.] `∈` 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)`.

[3.] The term honors 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, a **condition** is any expression built from functions or operators that evaluates to either true or false. For instance, the comparison `x > y` is a condition that checks whether `x` is greater than `y`.

To get the most out of this section, it'll help 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`. Thes 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 `Boo` values are displayed as `true` and `false`, while `Bool` vectors use `1` and `0`. This is illustrated below

```
x = 2

#`y` equals `true` or `false`
y = (x > 0)
```

```
julia> y
true
```

```
x = 2

#'z' element equals 'true' or 'false', represented by 1 or 0
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, conditior 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 lower than zero nor equal to zero? (equivalent)
y2 = !(x ≤ 0)
```

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

PARENTHESES 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 must choose between alternative operations depending on how their execution unfolds. To handle these possibilities, programs rely on conditional statements, which enable the execution of specific code blocks only when certain conditions are satisfied.

Each code block within 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 case in depth. The presentation builds heavily on the logical operators introduced in the previous section. If you haven't read it, I strongly 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, rendering **for-loops** crucial 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-LOOP SYNTAX

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

```
hello
beautiful
world
```

> **Remark**
>
> The keyword `in` can be replaced by `∈` or `=`, where `∈` can be written through tab completion using the command `\in`. Considering this, 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)`, which shows the value of `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 that are 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 collections that are iterable and therefore 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. [1]

ITERATION 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 collection 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

```julia
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 be used to define vectors by combining them with the `collect` function.

CREATING A VECTOR FROM A RANGE

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

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

ITERATING OVER INDICES OF AN ARRAY

Ranges provide a straightforward mechanism for traversing the elements of a collection. When used in combination with a for-loop, they let you access each element of a vector by its index. There are several ways to construct such ranges.

A straightforward method is to write `1:length(x)`, which generates all valid indices for the vector `x`. Since `length(x)` returns the number of elements, this range covers every position from the first to the last. While this approach works, it relies on the assumption of linear indexing starting at 1, which isn't always guaranteed. As a result, it can be fragile when applied to collections with different indexing schemes.

A more robust practice is to use `eachindex(x)`. This function produces an iterator that's optimized to the specific structure of the collection. Furthermore, it ensures that you're iterating over the correct set of indices, regardless of the underlying data type. This makes your code more general, more efficient, and less error-prone, especially when working with other iterable objects that may not use standard indexing.

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
```

Julia provides other methods to iterate over all indices of a collection. Two constructions worth mentioning are `LinearIndices(x)` and `firstindex(x):lastindex(x)`. Just like the previous methods defined, they specify a range from the first to the last index of `x`.

All these approaches can 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
```

The multiplicity of methods to implement the same functionality is necessary to handle non-standard indices. For instance, the `OffsetArrays.jl` package sets the first index of arrays to 0, a common convention in many programming languages. When this package is loaded, using `1:length(x)` would break portability, while `firstindex(x):lastindex(x)` adapts seamlessly to the modified indexing scheme.

Unless you're developing a package for other users, you don't need to worry about which approach to implement.

RULES FOR VARIABLE SCOPE IN FOR-LOOPS

Like functions, for-loops introduce a new variable scope. The rules governing these scopes are largely the same, with one important distinction: **for-loops are allowed to modify global variables, whereas functions aren't**.

> **Warning!**
>
> The scoping rules presented for for-loops apply except in some edge cases, which primarily result from discouraged coding practices. Since these scenarios aren't widespread, we briefly outline them next but won't focus on them further.
>
> The issue appears only when three conditions occur simultaneously: *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). [2]

> Unless all three conditions are met, you don't need to worry about this scenario. And even if it does occur, Julia will issue a warning in the REPL, alerting you that the code may not behave as intended.

To make the rules of variable scope in for-loops precise, let's consider a variable named `x`. Its behavior is governed by the following principles:

- if `x` is the variable of iteration, then `x` is always local to the for-loop. This holds regardless of whether there's a variable `x` defined outside the for-loop.
- if there's a variable `x` defined within the for-loop body, this variable is local and won't be accessible outside the for-loop.
- if there's a variable named `x` outside the for-loop and no `x` is defined within the for-loop body, `x` refers to the global variable. Moreover, the for-loop can reassign or mutate the value of `x`.

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 beginner mistake**: defining a variable inside a for loop and then attempting to use it outside the loop, only to discover that it's no longer accessible.

ITERATION VARIABLE IS LOCAL

```
x = 2

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

```
hello
```

NEW VARIABLES ARE LOCAL

```
#no `x` defined 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 third rule. This affects the treatment of global variables.

REFERENCE TO GLOBAL X

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

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

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

REASSIGNING GLOBAL X

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

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

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

ARRAY COMPREHENSIONS

Julia provides **array comprehensions** as a concise and expressive way to generate new arrays via for-loops. The general syntax is `[<expression> for... if...]`, where `<expression>` denotes either an operation or a function.

To illustrate, suppose we want to define a new vector `y`, whose elements are the squares of the corresponding entries in `x`. Array comprehensions make this task straightforward. The following code snippets demonstrate the approach by using a direct expression and by calling a function.

ARRAY COMPREHENSION WITH OPERATION

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


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

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

ARRAY COMPREHENSION WITH FUNCTION CALL

```
x      = [1,2,3]
foo(a) = a^2

y      = [foo(a) for a in x]
z      = [foo(x[i]) for i in eachindex(x)]
```

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

Array comprehensions can also incorporate conditions, allowing you to filter elements as they're generated. In such cases, the condition must be placed at the end of the comprehension.

ARRAY COMPREHENSION WITH CONDITION

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

y = [a for a in x if a ≤ 2]
z = [x[i] for i in eachindex(x) if x[i] ≤ 2]
```

```
julia> y
2-element Vector{Int64}:
 1
 2
julia> z
2-element Vector{Int64}:
 1
 2
```

Array Comprehensions for Generating Matrices

Array comprehensions can also be used to construct matrices. In this case, the syntax requires a comma to separate the iteration variables for each dimension.

ARRAY COMPREHENSION FOR MATRICES

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

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

ITERATING OVER MULTIPLE OBJECTS

For-loops can handle more than just single-value iterations. They also support **simultaneous iteration over multiple values**.

We'll focus on two common scenarios: iterating over two iterable collections at the same time, and iterating over both the indices and values of a single collection. In each case, we'll explore how to implement the iteration using both plain for-loops and array comprehensions, highlighting the differences in syntax and use cases.

ITERATING OVER TWO COLLECTIONS

Consider two iterable objects `x` and `y`. There are two main ways to combine their elements during iteration, depending on the desired outcome.

First, the function `Iterators.product(x,y)` generates all the possible combinations of elements from `x` and `y`. Thus, it produces a pair `(x[i], y[j])`, where `i` and `j` aren't necessarily equal. The `Iterators.product` function is part of the package `Iterators`, imported by default in each Julia session.

Alternatively, it's possible to 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. Thus, compared to `Iterators.product`, it restricts the iterations `(x[i], y[i])`.

MULTIPLE ITERATORS (ALL COMBINATIONS)

```
list1 = ["A","B"]
list2 = [ 1 , 2 ]

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

```
("A", 1)
("B", 1)
("A", 2)
("B", 2)
```

MULTIPLE ITERATORS (PAIRS)

```
list1 = ["A","B"]
list2 = [ 1 , 2 ]

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

```
("A", 1)
("B", 2)
```

We can also iterate over multiple values to generate vectors via array comprehension. While `zip` can be employed to create a new vector by pairing elements, `Iterators.product` would return a matrix. Instead, to construct a vector that exhausts all combinations between two iterable collections, Julia offers a special syntax: simply repeat the `for` keyword to specify the iteration range of each variable.

MULTIPLE ITERATORS (ALL COMBINATIONS)

```
list1 = ["A","B"]
list2 = [ 1 , 2 ]

x     = [(i,j) for i in list1 for j in list2]
```

```
julia> x
4-element Vector{Tuple{String, Int64}}:
 ("A", 1)
 ("A", 2)
 ("B", 1)
 ("B", 2)
```

MULTIPLE ITERATORS (PAIRS)

```
list1 = ["A","B"]
list2 = [ 1 , 2 ]

x     = [(i,j) for (i,j) in zip(list1,list2)]
```

```
julia> x
2-element Vector{Tuple{String, Int64}}:
 ("A", 1)
 ("B", 2)
```

Note that the `for` clauses must be written sequentially, without commas. A comma changes the comprehension's behavior, instructing Julia to build a matrix instead of a vector, as demonstrated before.

SIMULTANEOUSLY ITERATING OVER INDICES AND VALUES

When iterating over a collection, there are scenarios where you need access to both the value and index of an element, rather than handling each separately. For example, when analyzing data, you may want to flag outliers not only by their magnitude, but by their position in the dataset.

Julia provides a direct way to achieve this through the `enumerate` function. This transforms a collection into an iterator that yields pairs of indices and values, thus providing access to both pieces of information during each iteration.

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)]
```

```
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> x
2-element Vector{Int64}:
 100
  10
```

FOOTNOTES

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

2. There are two methods to execute a script. The first method is the one used thus far, where we 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 containing 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 (scalars and collections) and core programming tools (functions, conditions, and for-loops). At this initial stage, the emphasis was primarily on familiarizing 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** permit modification of their internal state after creation. This means that their elements can be modified, appended, or removed at will, thus providing a high degree of flexibility. A prime example is vectors.

In contrast, **immutable objects** can't be altered after their creation: 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. Simultaneously, 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 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
'h': ASCII/Unicode U+0068 (category Ll: Letter, lowercase)
julia> x[1] = 'a'
ERROR: MethodError: no method matching setindex!(::String, ::Char, ::Int64)
```

The key characteristic of mutable objects is their ability to modify existing elements. Moreover, mutability 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, error too with 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 assignment and mutation,
- methods for initializing arrays to eventually mutate them, and
- techniques for vector indexing to select elements.

The current section in particular focuses on **the distinction between assignment and mutation of variables**. The difference between these operations can easily go unnoticed by new users, as both use the operator `=` despite being fundamentally different. Clearly delineating them is important not only in Julia, but also in 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 scalars (single-element objects such as integers and floating-point numbers) to collections (e.g. vectors).

NUMBER

$$\underbrace{2}_{\text{object}}$$

VECTOR

$$\underbrace{\overset{\text{elements}}{\swarrow\downarrow\searrow}\;[4,5,6]}_{\text{object}}$$

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 establish a binding between a variable and an object through the `=` operator. For instance, `x = 3` and `x = [1,2,3]` are examples of assignments, where the objects `3` and `[1,2,3]` are to `x`.

Mutations, by contrast, **modify the elements of an object, without creating a new one**. These operations also rely on the `=` operator. An example of mutation is given by `x[1] = 0`, which modifies the first element of `x`.

Despite sharing the same operator `=`, assignments and mutations are conceptually distinct. The difference becomes clearer when we **views objects as residing at specific memory addresses**. Thus, an assignment for `x` involves two steps: *i*) finding a memory location to store the object, and *ii*) labeling the memory address as `x` for easy access. A mutation, in contrast, modifies the data stored at an existing memory address, but leaving the address itself unchanged.

To illustrate, if `x = [6,7]` is run, `x` becomes associated with an object containing `[6,7]`. Thus, this constitutes an *assignment*. However, if `x[1] = 0` is executed afterwards, the operation modifies the original object `[6,7]`, which now becomes `[6,0]`. This operation constitutes a *mutation*: `x` continues to reference the same memory address, even though its content has changed.

Mutating All Elements vs Assignment

Mutating all the elements of `x` doesn't imply a new assignment. For example, this occurs by modifying `x` using `x[:]`.

```
x    = [4,5]

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

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

> The distinction is important for performance, as mutating existing objects is generally faster than creating new ones. This point will be particularly relevant in Part II of the website, where we discuss optimization strategies.

ALIAS VS COPY

So far, we've introduced the fundamental distinction between assignments and mutations. Because both operations employ the `=` operator, the natural question that arises is when `=` results in an assignment or a mutation.

In this section, we focus on cases where entire objects appear on both sides of `=`, as in `y = x`. Other scenarios are left for upcoming sections, after we introduce the concept of slices (i.e, subsets of elements from a vector).

When `y = x` is executed in Julia, `y` becomes another name for the object referenced by `x`. In other words, `x` and `y` become different labels for the same underlying object. Formally, we say that `y` is an **alias** of `x`.

It's important to stress that `y = x` doesn't bind `y` to `x` itself. Rather, `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 holding 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 visually illustrated below.

CONSEQUENCE **NOT THE CONSEQUENCE**

The conclusion can also be drawn from examining code execution.

```
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
```

Two variables may contain identical elements and yet refer to different objects

The claim can be demonstrated using the operators `==` and `===`, which capture two different notions of equality. The expression `x == y` checks whether `x` and `y` have the same *values*, regardless of whether they reference the same object. In contrast, `x === y` checks whether both `x` and `y` point to the same *object*, thus verifying if they point to 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
julia> x === y
true
```

```
x = [4,5]

y = [4,5]
```

```
julia> x == y
true
julia> x === y
false
```

GRAPHICAL REPRESENTATION

We've indicated that the operation `y = x` creates a new alias for `x`, turning `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 referenced by both**. The following diagram illustrates this.

GRAPHICAL REPRESENTATION

The corresponding code snippet captures this case.

```
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 aim to treat `x` and `y` as pointing to separate objects, you must use the `copy` function. This creates a *new object* with the same 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}:
 123134236261000
     94489280887
               0
```

```
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}:
 123134067398448
 123134030334464
 123134084626496
```

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 we haven't discussed yet**. As such, it's marked as optional.
>
> One such concept is that of **in-place functions**, identified by the symbol `!` appended to the function's name. The symbol is simply a convention chosen by developers to indicate that the function modifies at least one of its arguments. A detailed discussion of in-place functions will be provided later.
>
> Another concept introduced is that of **pairs**, which will be thoroughly examined in a future section too. For the purposes of this subsection, it's sufficient to know that pairs are written in the form `a => b`. In our applications, `a` will represent a given value and `b` denotes its corresponding replacement value.

Next, we demonstrate how to add, remove, and replace elements within a vector. Below, we begin by presenting methods for adding a single element.

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


push!(x, element_to_insert)                # add 0 as last element - 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 as first element - 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 as last elements
```

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

The function `push!` is particularly helpful to collect outputs in a vector. Since it doesn't require prior knowledge of how many elements will be stored, the vector can grow dynamically as new results are added. Notice that adding elements at the end with `push!` is generally faster than inserting them at the beginning with `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
```

By analogy with the behavior of `deleteat!`, it's also possible to specify 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, specific values can be replaced with new ones. This can be done by either creating a new copy using `replace` or by updating the original vector with `replace!`.

Both functions make use of pairs `a => b`, where `a` denotes a given value and `b` specifies its replacement. Note that these functions perform substitutions based on values, not on 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 our preparation for the study of mutations. The attention is now shifted to the concept of vector **slices**: subsets of elements drawn from a vector, written as `x[<indices>]`. In particular, the focus will be on the critical distinction between whether slices act as:

- **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.

Which of these two forms arises is subtle, since it depends on where the slice appears within an expression.

The difference is central to implementing mutations correctly, as they're only possible when the slice is a view. If a slice is instead a copy, the parent object and the slice become entirely independent entities, so that modifications to the slice have no impact on the original object.

In Part II of this website, we'll also see that the distinction between copies and views has implications for performance. Essentially, creating a copy requires allocating new memory, which introduces computational overhead. Views, by contrast, avoid this cost by reusing the memory of the original vector.

Because of its broad significance, we'll examine the topic of copies and views independently of mutations. In particular, we'll identify how slices behave in different cases. Moreover, we'll explain how to instruct Julia to treat slices as views or copies.

SLICES AND THE ASSIGNMENT OPERATOR

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 role, slices directly reference the original elements, enabling mutation of the parent object. In contrast, **slices on the right-hand side (RHS) of `=` create a copy**. Since copies point to a new object in memory, any modification to the slice won't affect the original object.

The following code snippet demonstrates these contrasting behaviors.

```
x    = [4,5]


x[1] = 0        # 'x[1]' is a view, line 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 Copying

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 that case, `y` and `x` will reference the same object, so that 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}:
 0
 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}:
 4
 5
```

THE FUNCTION 'VIEW'

Beyond assignments, we must also distinguish whether slices represent copies or views within other expressions. As a rule of thumb, **slices typically default to creating copies**. Such behavior arises when, for instance, a slice is passed as a function argument or incorporated into a computation. Several of the 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, if you instead want to work with views, you must indicate this explicitly. Several methods exist for achieving this, with the most straightforward being the function `view`. Its syntax is `view(x, <indices>)`, where `<indices>` specify the indices that define the slice. To demonstrate its usage, we revisit the previous examples.

```
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
```

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

The `@view` macro is equivalent to `view`, allowing you to write `@view x[1:2]` instead of `view(x, 1:2)`. Its benefits, however, are somewhat limited: it saves only a few characters and still requires parentheses when multiple slices appear in the same expression (e.g., `@view(x[1:2]) .+`

`@view(x[2:3])`). By contrast, the `@views` macro significantly streamlines notation, 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` arises in functions. When placed at the beginning of a function definition, `@views` ensures that every slice appearing in the function body or its arguments is treated as 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 itself 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, for example precluding 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 broadcast, 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)]
```

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
2-element Vector{Int64}:
 6
 7
```

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`.

FUNCTIONS '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
```

OPERATORS '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            = [-100, 2, 4, 100]
list         = [minimum(x), maximum(x)]

#logical indexing
bool_indices = 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.

FUNCTIONS '!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))`
y            = x[bool_indices]
```

```
julia> bool_indices
4-element BitVector:
 0
 1
 1
 0
julia> y
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)

y            = x[bool_indices]
```

```
julia> bool_indices
4-element BitVector:
 0
 1
 1
 0
julia> y
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**, where the contents of an existing object are directly modified. Unlike operations that generate new objects, in-place operations are characterized by the reuse of existing objects, giving rise to the expression *modifying values in place*.

Distinguishing between mutations and the creation of new copies is essential. If an operation mutates an object, any other variable that references the same object will also reflect the change. This can be intended if you seek to update data, but it can introduce subtle bugs if you expected the original to remain unchanged. In-place modifications are also relevant for performance, as they reduce the memory overhead introduced when new objects are created. This aspect will be explored in Part II of the website.

At the heart of in-place operations is the concept of slices introduced in a previous section. Before proceeding, I recommend reviewing that section before moving forward.

A **slice** of a vector `x` is defined as a subset of its elements, selected through the syntax `x[<indices>]`. Importantly, slices can behave in two distinct ways:

- As a **copy**, which creates a new object with its own memory address.
- As a **view**, which references the original memory of `x`.

This distinction determines whether changes to the slice affect the original vector.

In what follows, we'll explore three approaches to mutating vectors in place. First, we'll examine mutations by assigning new collections to slices. After this, we'll cover the traditional approach of using for-loops to modify elements one at a time. Finally, we'll introduce the broadcasting assignment operator `.=`, which provides a concise tool for in-place updates.

MUTATIONS VIA COLLECTIONS

The most straightforward approach to mutating a vector is to replace an entire slice with another collection. This is achieved through statements of the form `x[<indices>] = <expression>`, where `<expression>` must match the length of `x[<indices>]`. Because slices on the left-hand side of `=` act as views, the assignment effectively modifies the original vector `x`, rather than creating a new one.

```
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 use case is when `<expression>` depends on either elements of the original vector or on the slice being modified itself. This allows for self-referential updates, where new values are computed from old ones.

```
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, when the left-hand side is a single-element slice, the right-hand side of `=` accepts a scalar. This property will be particularly relevant when we present mutations via for-loops.

```
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 created, the type of its elements is implicitly defined. Consequently, attempting to replace elements with an incompatible type will result in an error. For instance, a vector of type `Int64` can only be mutated with other `Int64` values or `Float64` values that can be converted into it. This is shown below.

```
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

Previously, we indicated that single-element slices on the left-hand side of `=` permit seamless mutations with scalar values. Thus, statements like `x[i] = 0` directly updates the element at position `i`, without requiring a collection on the right-hand side. Extending this idea, multiple elements of a vector can be updated within a for-loop.

A common use case of this approach arises when populating a vector with values. Typically, this involves first initializing a vector, whose initial contents are irrelevant, and then iterating over its elements with a for-loop to assign the desired values. The strategy is especially prevalent when storing outputs generated during a computation.

```julia
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
```

```julia
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 presented above relies on `x[i]` on the left-hand side of `=`, ensuring each element is treated as a view. However, an alternative strategy is to leverage the function `view`. This function enables the creation of a variable that contains all the elements to be modified. In doing so, the mutation can be performed directly on the entire object created, rather than repeatedly accessing elements from the original object.

In the following, we illustrate the technique by mutating a vector initialized with zeros. Note that the function `zeros` defaults to zeros with type `Float64`, explaining why `1` is automatically converted to `1.0`.

```julia
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 Always be Wrapped in Functions
>
> In the example above, we left the for-loop outside a function to highlight the mutating strategy. In practice, however, placing for-loops in the global scope is highly discouraged: it not only severely hurts performance, but also introduces different variable-scoping rules. In fact, earlier versions of Julia completely disallowed mutations in the global scope through for-loops. To perform mutations via for-loops within functions, though, we first need to introduce the concept of mutating functions. This is done in the next section, where we'll return to this subject.

MUTATIONS VIA .=

Broadcasting provides a streamlined alternative to for-loops. This principle extends to mutations as well. The implementation is based on the broadcasting of the assignment operator `=`, denoted as `.=`. Specifically, the syntax is `x[<indices>] .= <expression>`, where `<expression>` can be either a *vector* or a *scalar*.

When in particular `x[<indices>]` appears on the left-hand side of `.=` and `<expression>` is a vector, the `.=` operator produces the same outcome as using `=`. In fact, using `=` rather than `.=` tends to be more performant in those cases.

```
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` is for expressions such as:

- `x[<indices>].= <scalar>`,
- `x .= <expression>`, and
- `y .= <expression>` where `y` is a view of `x`.

Next, we analyze each case separately.

SCALARS ON THE RIGHT-HAND SIDE OF .=

A common scenario with mutations is when multiple elements must be replaced with the *same* scalar value. Implementing this operation with `=` requires providing a collection on the right-hand side, whose length must match the number of elements on the left. This not only introduces unnecessary boilerplate, but also assumes prior knowledge of the elements being replaced.

The broadcasting assignment operator `.=` makes such operations much simpler, simply requiring the execution of `x[<indices>] .= <scalar>`. The following code snippet employs this strategy 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
```

OBJECT ITSELF ON THE LEFT-HAND SIDE OF .=

We've already shown that the inclusion of terms like `x[indices]` on the left-hand side of `=` results in mutations. Now, let's turn to cases where an entire object appears on the left-hand side. Here, the focus is on scenarios where the object is `x` itself. Instead, scenarios with slices constructed via `view` will be deferred until the next subsection.

When an object appears on the left-hand side, we need to carefully **distinguish between in-place operations and reassignments**. Whether one or the other operation is implemented depends on whether `.=` or `=` is employed. Specifically, it's only when `.=` is used that a mutation takes place.

Instead, `=` will perform a reassignment, creating a new object at a new memory address. While the distinction seems irrelevant since `x` will ultimately hold the new values in both cases, we'll see in Part II of the website that the distinction actually matters for performance.

To illustrate, suppose our goal is to modify *all* the elements of a vector `x`. All the following approaches determine that `x` ends up holding the new values, but only the last two achieve this by mutation of `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
```

This risk of mixing up `.=` and `=` becomes even greater when using the `@.` macro for broadcasting, rather than manually inserting dots into each operator. The placement of `@.` relative to `=` determines whether the operation is a reassignment or a mutation. Specifically:

- If `@.` appears *before* `=`, `x` is mutated since `.=` is being used.
- If `@.` instead appears after `=`, only the right-hand side is broadcast and the assignment is performed with `=`. This results in a reassignment, rather than 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 .=

Let's continue with our analysis of entire objects on the left-hand side of `.=`. Our focus now shifts to **view aliases**: variables such as `slices` defined by `slices = view(x[<indices>])`. They allow us to work directly with `slice` rather than `x[<indices>]`.

The introduction of view aliases is especially convenient when performing multiple operations on the same slice. It avoids repeated references to `x[<indices>]`, which would be inefficient, error-prone, and tedious.

As before, it's crucial to distinguish between using `.=` and `=`. In particular, only `.=` will perform a mutation, while `=` will result in a reassignment. With view aliases, however, additional care is required. The intended workflow involves first defining a slice (an assignment over a view) and then mutating that slice. This structure determines that there are now two possible wrong uses:

- the initial assignment is performed over a copy of `x[<indices>]`, rather than a view of `x[indices]`.
- the second step performs a reassignment (`=`), rather than a mutation (`.=`).

Below, we illustrate the correct usage, followed by these two incorrect patterns. The aim in the exercise is to replace all negative values in `x` with zero.

```
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 does NOT modify `x`
```

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

Note that mutations with view aliases also allow `slice` to be included on the right-hand side of `=`. Below, we provide again the correct implementation, along with the two incorrect ones.

```
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
```

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

slice  = x[x .≥ 2]          # 'slice' is a copy
slice  = slice .* 10        # this does NOT modify `x`
```

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

```julia
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
```

5h. In-Place Functions

Martin Alfaro
PhD in Economics

INTRODUCTION

This section continues exploring **approaches to mutating vectors**. The emphasis is now on **in-place functions**, defined as functions that mutate at least one of their arguments.

Many built-in functions in Julia have an in-place counterpart, which can easily recognized by the `!` suffix in their names. These versions store the output in one of the function arguments, thereby avoiding the creation of a new object. In practice, it means that variables can be updated immediately after executing the function. For example, given a vector `x`, the call `sort(x)` produces a new vector with ordered elements, but without altering the original `x`. In contrast, the in-place version `sort!(x)` overwrites the content of `x`.

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`.

```julia
y = [0,0]

function foo(x)
    x[1] = 1
end
```

```julia
julia> y
2-element Vector{Int64}:
 0
 0
julia> foo(y) #it mutates 'y'
julia> y
2-element Vector{Int64}:
 1
 0
```

Functions Can't Reassign Variables

While functions are capable of mutating values, they **can't reassign variables defined outside their scope**. Any attempt to redefine such variable 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, 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 global variables, 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**.

> **Appending ! To A Function Has No Impact**
>
> **Appending `!` to a function doesn't change the function's behavior**. 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)` directly updates the original vector `x`, overwriting its values with the sorted result.

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

output = sort(x)
```

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

```julia
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
```

```julia
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
```

FOR-LOOP MUTATION VIA IN-PLACE FUNCTION

In Julia, for-loops 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.

Considering this, 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 a for-loop. The examples below illustrate this application.

```julia
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
```

```julia
x = Vector{Int64}(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 understand 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. Moreover, although we introduced the concept of tuples, we haven't analyzed them depth. The current section fills this gap by offering a more comprehensive analysis of tuples.

Additionally, we introduce two new types of collections: **named tuples** and **dictionaries**. We'll also cover how to generally characterize collections through keys and values, methods for manipulating 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 for their elements and are paired with a corresponding **value**. [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: while indices are limited to integer identifiers, keys can be any valid Julia object (e.g., strings, numbers, or other objects).

Julia provides the functions `keys` and `values` to extract the keys and values of a collection. The following code snippets demonstrate their usage with vectors and tuples, whose keys are represented by indices. Note that neither `keys` nor `values` return a vector, requiring the `collect` function to obtain a vector representation of the keys or values.

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

x_keys   = collect(keys(x))
x_values = collect(values(x))
```

```
julia> x_keys
3-element Vector{Int64}:
 1
 2
 3
julia> x_values
3-element Vector{Int64}:
 4
 5
 6
```

```
x        = (4, 5, 6)

x_keys   = collect(keys(x))
x_values = collect(values(x))
```

```
julia> x_keys
3-element Vector{Int64}:
 1
 2
 3
julia> x_values
3-element Vector{Int64}:
 4
 5
 6
```

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 the key and `1` the corresponding value. Alternatively, pairs can be created using the function `Pair(<key>, <value>)`, where `Pair("a",1)` is equivalent to the previous example.

Given a pair `x`, its key can be accessed via either `x[1]` or `x.first`. Likewise, its value is retrieved using either `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 may vary across collections. A commonly one used for keys is `Symbol`, which offers an efficient way to represent string-based identifiers. A symbol named `x` is written as `:x` and can be constructed from a string using the function `Symbol(<string>)`. [2]

```
vector_symbols = [:x, :y]
```

```
julia> vector_symbols
2-element Vector{Symbol}:
 :x
 :y
```

```
vector_symbols = [Symbol("x"), Symbol("y")]
```

```
julia> vector_symbols
2-element Vector{Symbol}:
 :x
 :y
```

NAMED TUPLES

> **Warning!**
>
> **Tuples and named tuples are only suitable for small collections**. Using them with large collections can result in poor performance or even fatal errors (such as stack overflows). For large collections, arrays remain the preferred choice.
>
> Defining what qualifies as *small* is challenging, and unfortunately there's no definitive answer. We can only indicate that collections with fewer than 10 elements are certainly small, while those exceeding 100 elements clearly exceed the intended use.

Named tuples share several properties with regular tuples, including their **immutability**. However, they also exhibit some notable differences. One important distinction 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 single-element named tuple, the syntax 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, a pair `<key with Symbol type> => <value>` can be used, as in `:a => 10`. Once a named tuple `nt` is created, the element `a` can be accessed either by key lookup `nt[:a]` or by dot syntax `nt.a`.

The following code snippets illustrate these concepts.

```
# all 'nt' are equivalent
nt = (  a=10, b=20)
nt = (; a=10, b=20)
nt = (  :a => 10, :b => 20)
nt = (; :a => 10, :b => 20)
```

```
julia> nt
(a = 10, b = 20)
julia> nt.a
10
julia> nt[: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]
10
```

> **Remark**
>
> To see the list of keys and values, we can employ the functions `keys` and `values`.

```
nt        = (a=10, b=20)

nt_keys   = collect(keys(nt))
nt_values = collect(values(nt))
```

```
julia> nt_keys
2-element Vector{Symbol}:
 :a
 :b
julia> nt_values
2-element Vector{Int64}:
 10
 20
```

DISTINCTION BETWEEN THE CREATION OF TUPLES AND NAMED TUPLES

It's possible to create named tuples from existing variables. For instance, given variables `x = 10` and `y = 20`, one can define `nt = (; x, y)`. This creates a named tuple with keys `x` and `y`, and corresponding values `10` and `20`.

The semicolon `;` plays a crucial role 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:

- **Dictionary keys can be any object**: strings, numbers, and other objects are possible.
- **Dictionaries are mutable**: elements can be modified, added, and removed after creation.
- **Dictionaries are unordered**: keys have no inherent order.

The function `Dict` can be used to create dictionaries, where each argument is a key-value pair written in the form `<key> => <value>`.

```
some_dict = Dict(3 => 10, 4 => 20)
```

```
julia> some_dict
Dict{Int64, Int64} with 2 entries:
  4 => 20
  3 => 10
julia> some_dict[1]
10
```

```
some_dict = Dict("a" => 10, "b" => 20)
```

```
julia> some_dict
Dict{String, Int64} with 2 entries:
  "b" => 20
  "a" => 10
julia> some_dict["a"]
10
```

```
some_dict = Dict(:a => 10, :b => 20)
```

```
julia> some_dict
Dict{Symbol, Int64} with 2 entries:
  :a => 10
  :b => 20
julia> some_dict[:a]
10
```

```
some_dict = Dict((1,1) => 10, (1,2) => 20)
```

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

Note that regular dictionaries are inherently unordered, meaning that the access to their elements doesn't follow any pattern. The following example illustrates this, by collecting the dictionary keys into a vector. [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. The only requirement is that the source collection possesses a key-value structure.

To demonstrate this possibility, we begin by creating **dictionaries** from a variety of collections.

```
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 also 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 demonstrated how to create tuples and named tuples 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 separate variables**. This process is known as **destructuring**, enabbling users to "unpack" the values of a collection into distinct variables.

Destructuring involves the assignment operator `=` with either a tuple or named tuple on the left-hand side. The choice between one or the other determines what objects can be used on the right-hand side. Tuples on the left-hand side are quite flexible, allowing values to be unpacked from a variety of collections. Named tuples on the left-hand side, instead, necessarily require a named tuple on the right-hand side. Next, we develop each case separately.

DESTRUCTURING COLLECTIONS THROUGH TUPLES

Given a collection `list` with two elements, destructuring via tuples allows us to unpack its values into the variables `x` and `y`. The syntax for this is `<tuple> = <collection>`, as in `x,y = list`. In the following, we illustrate the process by considering different objects 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 unpacking all elements, destructuring can also be applied to only a subset of elements. The assignment is then performed in sequential order, following the collection's inherent order.

Importantly, this method excludes the possibility of skipping any specific value. When a value must be disregarded, the conventional approach is to bind this value to the special variable name `_`. This symbol serves purely as a placeholder to indicate that the value is unimportant and has no impact on execution.

For illustration, we'll use a vector as an example of `list`. Nonetheless, the same principle applies to any collection.

```
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        # _ or any symbol (it just signals we don't care about that value)
```

```
julia> z
5
```

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

x,_,z = list        # _ or any symbol (it just signals we don't care about that value)
```

```
julia> x
3
julia> y
5
```

DESTRUCTURING WITH NAMED TUPLES ON BOTH SIDES

Destructuring can also be applied with named tuples on the left-hand side. In this case, values are extracted by directly referencing field names, rather than relying on their positional order. The main advantage of this approach is that variables can be assigned in any order, provided their names correspond to some field in the named tuple.

```
nt             = (; key1 = 10, key2 = 20, key3 = 30)

(; key3, key1) = nt            # keys in any order
```

```
julia> key1
10
julia> key3
30
```

```
nt             = (; key1 = 10, key2 = 20, key3 = 30)

(; key2)       = nt            # only one key
```

```
julia> key2
20
```

Remark

When destructuring with a tuple on the left-hand side and a named tuple on the right-hand side, keep in mind that the assignment is carried out strictly by position. This means that variable names on the left don't influence the assignment operation. In other words, the keys of the named tuple are completely ignored during the process.

```
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> key2
20
julia> key1
10
```

The same caveat applies to assignments of single variables.

```
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 particularly valuable in scientific modelling, where numerous parameters are referenced repeatedly. By grouping all these parameters into a single named tuple, they can be passed to a function as *one* consolidated argument. When functions are defined following this procedure, the named tuple is then destructed at the beginning of the function body to extract the needed parameters.

```
β = 3
δ = 4
ϵ = 5

# function 'foo' uses β and δ, but not ϵ
function foo(x, δ, β)
    x * δ + exp(β) / β
end

output = foo(2, δ, β)
```

```
julia> output
14.6952
```

```julia
parameters_list = (; β = 3, δ = 4, ϵ = 5)



# function 'foo' uses β and δ, but not ϵ
function foo(x, parameters_list)
    x * parameters_list.δ + exp(parameters_list.β) / parameters_list.β
end

output = foo(2, parameters_list.β, parameters_list.δ)
```

```
julia> output
19.6495
```

```julia
parameters_list = (; β = 3, δ = 4, ϵ = 5)



# function 'foo' uses β and δ, but not ϵ
function foo(x, parameters_list)
    x * parameters_list.δ + exp(parameters_list.β) / parameters_list.β
end

output = foo(2, parameters_list)
```

```
julia> output
14.6952
```

Destructuring also provides a convenient solution for retrieving multiple outputs from a function. This makes it possible to unpack the returned outputs into separate variables. In the example below, the function `foo` returns tuple, which is then unpacked into variables `x`, `y`, and `z`.

```julia
function foo()
    out1 = 2
    out2 = 3
    out3 = 4

    out1, out2, out3
end

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

```julia
function foo()
    out1 = 2
    out2 = 3
    out3 = 4

    [out1, out2, out3]
end

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

Another common use of destructuring arises when only a subset of a function's outputs is needed. While both tuples and named tuples can be applied for this purpose, tuples offer greater flexibility as they can be combined with various types of collections. In contrast, named tuples are restricted to returning another named tuple as the function's output, thus requiring prior knowledge of the field names.

The following example illusates this functionality by extracting only the first and third output of `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

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 a key-value structure.

2. `Symbol` also enables the programmatic creation of variables. A typical use case arises in data analysis, where symbols are employed to generate new columns in tabular data structures.

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 presents two complementary strategies for expressing computations that involve multiple intermediate steps. The first relies on **let blocks**, which introduce a new scope and return the value of the final expression. Let blocks provide a compact way to group a sequence of operations, offering function-like structure with far 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 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 convenient when performing a series of operations but only the final result matters. To illustrate their utility, consider the task of computing the rounded logarithm of `a`'s absolute value. Formally, $\mathrm{round}\left(\ln\left(|a|\right)\right)$.

In Julia, this operation can be directly written as `round(log(abs(a)))`, where `round(a)` returns the integer nearest to `a`. However, the nested parentheses make the expression hard to read, with the issue potentially exacerbated if variables or functions had long names.

A straightforward way to improve clarity is to break the whole operation into multiple steps: *i)* compute the absolute value of `a`, *ii)* compute the logarithm of the result, and *iii)* round the resulting output. While this can be implemented through three intermediate variables that store the output in each step, such an 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 introduces a new scope delimited by the `let` and `end` keywords, enabling multiple calculations to be performed locally. The result of the last calculation is then returned as the output. Like functions, let-blocks also allow arguments to be passed by adding them after the `let` keyword.

To highlight the benefits of let-blocks, the following examples compare various approaches to computing `round(log(abs(a)))`.

```julia
a      = -2

output = round(log(abs(a)))
```

```
julia> output
1.0
```

```julia
a      = -2

temp1  = abs(a)
temp2  = log(temp1)
output = round(temp2)
```

```
julia> output
1.0
julia> temp1
2
julia> temp2
0.693147
```

```julia
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
```

```julia
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 follow the same rules as functions with respect to assignment and mutation. This means the values passed into a let block can be mutated, but global variables passed into it can't reassigned.

```
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
```

Given the possibility of mutating global variables within let-blocks, you should exercise caution to prevent unintended side effects.

PIPES

For operations consisting of multiple intermediate step, pipes constitutes an alternative to let-blocks. Unlike let-blocks, they're specifically designed to chain operations together, with each step receiving the output of the previous one as its input. Each individual step in the chain are separated via the `|>` keyword.

Pipes are particularly well-suited for sequential applications of single-argument functions. To illustrate their use, 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 let Blocks and pipes help create temporary aliases for variables with lengthy names. This lets users 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
1.65343
```

```
variable_with_a_long_name = 2

temp   = variable_with_a_long_name
output = temp - log(temp) / abs(temp)
```

```
julia> output
1.65343
```

```
variable_with_a_long_name = 2

output = variable_with_a_long_name  |>
         a -> a - log(a) / abs(a)
```

```
julia> output
1.65343
```

```
variable_with_a_long_name = 2

output = let x = variable_with_a_long_name
    x - log(x) / abs(x)
end
```

```
julia> output
1.65343
```

BROADCASTING PIPES

Just like any other operator, pipes can be broadcast by prefixing them with a dot `.`. In this form, `.|>` 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 this behavior, suppose we seek to transform each element of `x` with the logarithm of its absolute values, and then sum the results.

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

output = sum(log.(abs.(x)))
```

```
julia> output
1.79176
```

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

temp1  = abs.(x)
temp2  = log.(temp1)
output = sum(temp2)
```

```
julia> output
1.79176
```

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

output = x .|> abs .|> log |> sum
```

```
julia> output
1.79176
```

PIPES WITH MORE COMPLEX OPERATIONS

So far, our examples of pipes have followed a simple pattern, with each step consisting of a single-argument function. However, this form precludes the application of pipes to multiple-argument functions or even operations. For example, it prevents the inclusion of expressions like `foo(x,y)` or `2 * x`.

To address this limitation, we can **combine pipes with anonymous functions**. This enables users to specify how the output of the previous step is integrated into the subsequent operation. By doing so, the utility of pipes is significantly expanded, as demonstrated below.

```
a      = -2

output = round(2 * abs(a))
```

```
julia> output
4
```

```
a      = -2

temp1  = abs(a)
temp2  = 2 * temp1
output = round(temp2)
```

```
julia> output
4
```

```
a      = -2

output = a |> abs |> (x -> 2 * x) |> round

#equivalent and more readable
output = a              |>
         abs            |>
         x -> 2 * x     |>
         round
```

```
julia> output
4
```

PACKAGE PIPE

Combining pipes and anonymous functions can result in cumbersome code, defeating the very own purpose of using pipes in the first place.

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 `_`. Additionally, for single-argument operations that don't require anonymous functions, `@pipe` maintains the same syntax as built-in pipes.

To illustrate its convenience, below we reimplement the last code snippet.

```
#
a      = -2

output = a |> abs |> (x -> 2 * x) |> round

#equivalent and more readable
output =       a             |>
               abs           |>
               x -> 2 * x    |>
               round
```

```
julia> output
4
```

```
using Pipe
a = -2

output = @pipe a |> abs |> 2 * _ |> round

#equivalent and more readable
output = @pipe a             |>
               abs           |>
               2 * _         |>
               round
```

```
julia> output
4
```

FUNCTION COMPOSITION (*OPTIONAL*)

An alternative approach to nesting 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, given 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.

```julia
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
```

```julia
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
```

Importantly, the resulting function from function composition can be broadcast. To understand this notation more clearly, you should think of compositions as defining a new function: $h := f \circ g$. This entails that $h(x) := (f \circ g)(x)$, and therefore $h(x) := (f \circ g)(x) = f[g(x)]$. Given this, broadcasting `h` requires `h.(x)`, which is equivalent to `(f ∘ g).(x)` or `∘(f,g).(x)`.

```julia
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
 0.693147
 1.09861
```

```julia
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
julia> output
3-element Vector{Float64}:
 2.5
 3.0
 3.5
```

We can also broadcast the composition operator `∘` itself, enabling the simultaneous application of multiple functions to the same object. For instance, the following implementation 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
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 introduces a set of core functions for manipulating vectors. We focus on operations that arise frequently in data processing and numerical computing, such as sorting values, retrieving the indices that produce a sorted order, removing duplicates, counting occurrences, and computing rankings. The next section will show how these functions come together in a practical example.

SORTING VECTORS

The `sort` function arranges elements in ascending order by default, with the possibility of a descending order through the keyword argument `rev = true`. The function comes in two forms: `sort`, which returns a new sorted copy, and `sort!`, the in-place version that 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)` have the option of defining the sorting order based on transformations of `x`. Specifically, given a function `foo`, the elements can be ordered by the values of `foo(x)`. Its implementation requires the keyword argument `by`.

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(x)` returns the ordered *values* of `x`, it's also useful to obtain the *indices* of the sorted elements. This capability 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)` evaluates to `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
```

In the first two examples, the elements are already in ascending order, so `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 appears at index 1, the second smallest at index 4, the third smallest at index 2, and the largest at index 3.

Like `sort`, `sortperm` also supports retrieving indices in descending order. This requires including the keyword argument `rev = 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 accepts the keyword argument `by` to define a custom transformation.

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 one variable based on the values of another. 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 function `unique` removes duplicate entries from a vector, returning a new vector that contains each element exactly once. The function comes in two variants: `unique`, which produces a new copy, and `unique!`, which performs the operation in place and thus modifies 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 provides a related function called `countmap`, which counts the occurrences of each element in a vector. It returns a dictionary in which the unique elements act as keys, and their corresponding values represent the number of times each element appears.

By default, the keys in the resulting dictionary are unsorted. If instead sorted keys are preferred, you must apply the `sort` function to the result. This will automatically convert 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

```julia
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 for approximating numerical values to a specified precision:

- `round` approximates a number to its nearest integer.
- `floor` returns the greatest integer less than or equal to the given number.
- `ceil` returns the smallest integer greater than or equal to the given number.

Below, we show that these functions are quite flexible, allowing users to specify the output type (e.g., `Int64` or `Float64`), the number of decimals places via the keyword argument `digits`, and the number of significant digits.

ROUND

```julia
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`. Their main difference them lies in how they treat tied values: `competerank` assigns the same rank to all tied elements, while `ordinalrank` assigns consecutive ranks. In both cases, a rank of 1 corresponds to the smallest value. The keyword argument `rev = true` reverses this convention, assigning a rank of 1 to the largest value.

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
```

DESCENDING RANK (SAME RANK FOR TIES)

```julia
using StatsBase
x = [6, 6, 0, 5]

y = competerank(x, rev=true)
```

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

RANK (UNIQUE POSITIONS)

```julia
using StatsBase
x = [6, 6, 0, 5]

y = ordinalrank(x)
```

```julia
julia> y
4-element Vector{Int64}:
 3
 4
 1
 2
```

DESCENDING RANK (UNIQUE POSITIONS)

```julia
using StatsBase
x = [6, 6, 0, 5]

y = ordinalrank(x, rev=true)
```

```julia
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. Instead, `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 presenting a method for identifying both the indices and the values of extrema within a collection. The following examples are based on the maximum, with similar functions available for the minimum.

VALUE

```
x = [6, 6, 0, 5]

y = maximum(x)
```

```
julia> y
6
```

INDEX

```
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 their *arguments*. These functions will become particularly useful in procedures built on binary operations that we'll cover later (e.g., reductions).

'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 a practical example, this section will demonstrate 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 viewers per video, with a few viral videos exceeding 100,000 viewers. The information at our disposal is:

- `nr_videos`: 30 (one per day).
- `viewers`: viewers per video (in thousands).
- `payrates`: Dollars paid per video for 1,000 viewers. They range 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 were added below for the sake of completeness. Ultimately, what matters is that the mock data creates the variables `viewers` and `payrates`.

```julia
using StatsBase, Distributions
using Random; Random.seed!(1234)

function audience(nr_videos; median_target)
    shape   = log(4,5)
    scale   = median_target / 2^(1/shape)

    viewers = rand(Pareto(shape,scale),  nr_videos)

    return viewers
end

nr_videos = 30

viewers  = audience(nr_videos, median_target = 50)      # in thousands of viewers
payrates = rand(2:6, nr_videos)                         # per thousands of viewers
```

```
julia> viewers # in thousands
30-element Vector{Float64}:
 38.8086
 70.8113
   ⋮
 72.3673
 30.2565
julia> payrates # per thousand viewers
30-element Vector{Int64}:
 5
 3
 ⋮
 2
 4
```

The variables `viewers` and `payrates` enable us to calculate the total payment per video.

```julia
earnings = viewers .* payrates
```

```
julia> earnings
30-element Vector{Float64}:
 194.043
 212.434
    ⋮
 144.735
 121.026
```

SOME GENERAL INFORMATION

We begin by examining the per-view payments made by YouTube. We first show that Johnny's payments range 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 => 5
  3 => 6
  4 => 8
  5 => 5
  6 => 6
```

We can also provide some insights into Johnny's most profitable videos. By applying the `sort` function, we can isolate his top 3 highest-earning videos. Moreover, we can apply the `sortperm` function to identify their indices, allowing us to extract the payment per view and total viewers associated with each video.

```
top_earnings    = sort(earnings, rev=true)[1:3]
```

```
julia> top_earnings
3-element Vector{Float64}:
 7757.81
  693.813
  672.802
```

```
indices         = sortperm(earnings, rev=true)[1:3]

sorted_payrates = payrates[indices]
```

```
julia> sorted_payrates
3-element Vector{Int64}:
 6
 6
 6
```

```
indices         = sortperm(earnings, rev=true)[1:3]

sorted_viewers  = viewers[indices]
```

```
julia> sorted_viewers
3-element Vector{Float64}:
 1292.97
  115.636
  112.134
```

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 viewers. In particular, we'll determine the number of viewers and revenue generated by them.

To identify the viral videos, we'll create a `Bool` vector, where `true` identifies a viral video. This vector can then be employed as a logical index, allowing us to selectively extract data points from other variables. In the example below, we apply it to compute the total viewers and earnings attributable to the viral videos.

```
# characterization of viral videos
viral_threshold = 100
is_viral        = (viewers .≥ viral_threshold)

# stats
viral_nrvideos  = sum(is_viral)
viral_viewers   = sum(viewers[is_viral])
viral_revenue   = sum(earnings[is_viral])
```

```
julia> viral_nrvideos
4
julia> viral_viewers
1625.05
julia> viral_revenue
9750.3
```

Boolean indexing also enables subsetting data satisfying multiple conditions. For instance, we can apply this technique to calculate the proportion of viral videos for which YouTube paid more than $3 per thousand viewers.

```
# characterization
viral_threshold    = 100
payrates_above_avg = 3

is_viral           = (viewers .≥ viral_threshold)
is_viral_lucrative = (viewers .≥ viral_threshold) .&& (payrates .> payrates_above_avg)

# stat
proportion_viral_lucrative = sum(is_viral_lucrative) / sum(is_viral) * 100
```

```
julia> proportion_viral_lucrative
100.0
```

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, the number of decimal places in the approximation can be controlled via the `digits` keyword argument. Furthermore, it's possible to represent the number as an integer using either `Int` or `Int64` as an argument. [1]

```
rounded_proportion = round(proportion_viral_lucrative)
```

```
julia> rounded_proportion
100.0
```

```
rounded_proportion = round(proportion_viral_lucrative, digits=1)
```

```
julia> rounded_proportion
100.0
```

```
rounded_proportion = round(Int64, proportion_viral_lucrative)
```

```
julia> rounded_proportion
100
```

FUNCTIONS TO REPRESENT TASKS

The approach employed so far allows for a quick exploration of Johnny's viral videos. However, it lacks the structure needed for a systematic analysis across different subsets of the data. To address this limitation, we can capture the characterization of videos through a function.

Recall that a well-designed function should embody a single clearly defined task. In our case, the goal is to subset data and extract key statistics, including the number of videos, viewers, and revenue generated. Furthermore, the function should remain independent of any specific application, so it can be reused to analyze different groups of videos without rewriting code each time.

The function below implements this task taking three arguments: the raw data (`viewers` and `payrates`) and a condition that defines the subset of data (`condition`). By keeping the condition generic, the function is flexible enough to target any subset of videos. The example also showcases the convenience of pipes to compute intermediate temporary steps.

```
#
function stats_subset(viewers, payrates, condition)
    nrvideos = sum(condition)
    audience = sum(viewers[condition])

    earnings = viewers .* payrates
    revenue  = sum(earnings[condition])

    return (; nrvideos, audience, revenue)
end
```

```
using Pipe
function stats_subset(viewers, payrates, condition)
    nrvideos = sum(condition)
    audience = sum(viewers[condition])


    revenue  = @pipe (viewers .* payrates) |> x -> sum(x[condition])

    return (; nrvideos, audience, revenue)
end
```

```
using Pipe
function stats_subset(viewers, payrates, condition)
    nrvideos = sum(condition)
    audience = sum(viewers[condition])


    revenue  = @pipe (viewers .* payrates) |> sum(_[condition])

    return (; nrvideos, audience, revenue)
end
```

Below, we demonstrate the reusability of the function by characterizing various subsets of data.

```
viral_threshold  = 100
is_viral         = (viewers .≥ viral_threshold)
viral            = stats_subset(viewers, payrates, is_viral)
```

```
julia> viral
(nrvideos = 4, audience = 1625.05, revenue = 9750.3)
```

```
viral_threshold  = 100
is_notviral      = .!(is_viral)      # '!' is negating a boolean value and we broadcast it
notviral         = stats_subset(viewers, payrates, is_notviral)
```

```
julia> notviral
(nrvideos = 26, audience = 1497.02, revenue = 5687.67)
```

```
days_to_consider = (1, 10, 25)      # subset of days to be characterized
is_day           = in.(eachindex(viewers), Ref(days_to_consider))
specific_days    = stats_subset(viewers, payrates, is_day)
```

```
julia> specific_days
(nrvideos = 3, audience = 182.939, revenue = 1030.33)
```

VARIABLE MUTATION

Suppose Johnny is exploring ways to increase viewership through targeted 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 or 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 if he had run targeted ads.

The first step in this process involves creating a modified copy of `viewers`. This should now reflect the anticipated increase in viewership from running ads on the targeted videos (either viral or non-viral). With this updated audience data, Johnny can then apply the previously defined `stats_subset` function to estimate the potential earnings. By comparing the estimations for each group of targeted video, Johnny can determine which strategy offers the higher return on investment.

```
# 'temp' modifies 'new_viewers'
new_viewers     = copy(viewers)
temp            = @view new_viewers[new_viewers .≥ viral_threshold]
temp           .= 1.2 .* temp

allvideos       = trues(length(new_viewers))
targetViral     = stats_subset(new_viewers, payrates, allvideos)
```

```
julia> targetViral
(nrvideos = 30, audience = 3447.08, revenue = 17388.0)
```

```
# 'temp' modifies 'new_viewers'
new_viewers     = copy(viewers)
temp            = @view new_viewers[new_viewers .< viral_threshold]
temp           .= 1.2 .* temp

allvideos       = trues(length(new_viewers))
targetNonViral  = stats_subset(new_viewers, payrates, allvideos)
```

```
julia> targetNonViral
(nrvideos = 30, audience = 3421.47, revenue = 16575.5)
```

Given the results in each tab, promoting viral videos appears to be the more profitable option.

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 broadcast operator `.=`. Below, we present some implementations that fail to produce the intended result.

```
new_viewers = copy(viewers)


temp  = @view new_viewers[new_viewers .≥ viral_threshold]
temp .= temp .* 1.2
```

```
new_viewers = viewers      # it creates an alias, it's a view of
the original object!!!

# 'temp' modifies 'viewers' -> you lose the original info
temp  = @view new_viewers[new_viewers .≥ viral_threshold]
temp .= temp .* 1.2
```

```
new_viewers = copy(viewers)

# wrong -> not using `temp .= temp .* 1.2`
temp  = @view new_viewers[new_viewers .≥ viral_threshold]
temp  = temp .* 1.2      # it creates a new variable 'temp', it
does not modify 'new_viewers'
```

Use of "Let Blocks" To Avoid Bugs

In the code above, "Target Viral" and "Target Non-Viral" reference variables with identical names. This creates the risk of accidentally referring to a variable from the wrong scenario.

A practical way to mitigate this risk is by employing "let blocks". Since each let block introduces its own scope, this helps maintain a clean namespace and prevents variable collisions.

```
targetViral      = let viewers = viewers, payrates = payrates,
threshold = viral_threshold
    new_viewers = copy(viewers)
    temp        = @view new_viewers[new_viewers .≥ threshold]
    temp       .= 1.2 .* temp

    allvideos  = trues(length(new_viewers))
    stats_subset(new_viewers, payrates, allvideos)
end
```

```
julia> targetViral
(nrvideos = 30, audience = 3447.08, revenue = 17388.0)
```

```
targetNonViral  = let viewers = viewers, payrates = payrates,
threshold = viral_threshold
    new_viewers = copy(viewers)
    temp        = @view new_viewers[new_viewers .< threshold]
    temp       .= 1.2 .* temp

    allvideos  = trues(length(new_viewers))
    stats_subset(new_viewers, payrates, allvideos)
end
```

```
julia> targetNonViral
(nrvideos = 30, audience = 3421.47, revenue = 16575.5)
```

BROADCASTING OVER A LIST OF FUNCTIONS

A function like `stats_subset` is useful for computing a fixed set of summary statistics. However, since the choice of statistics is hard-coded into the function's definition, the output can't be changed without rewriting the code. This rigidity makes the function less reusable across different analytical contexts.

A more flexible approach consists of specifying which statistics to compute at the time of use. Julia makes this possible because functions are *first-class objects*, entailing that functions behave just like any other variable. This feature lets us define a list of statistical functions, eventually applying them element-wise to the variables we want to characterize.

Below, we apply this methodology to characterize the variable `viewers`.

```
list_functions = [sum, median, mean, maximum, minimum]

stats_viewers  = [fun(viewers) for fun in list_functions]
```

```
julia> stats_viewers
5-element Vector{Float64}:
 3447.08
   64.8765
  114.903
 1551.56
   28.2954
```

The same methodology can also be employed for characterizing multiple variables at once. In fact, broadcasting makes this straightforward to implement. For instance, below we simultaneously characterize `viewers` and `earnings`.

```
list_functions = [sum, median, mean, maximum, minimum]

stats_various  = [fun.([viewers, payrates]) for fun in list_functions]
```

```
julia> stats_various
5-element Vector{Vector{Float64}}:
 [3447.08, 121.0]
 [64.8765, 4.0]
 [114.903, 4.03333]
 [1551.56, 6.0]
 [28.2954, 2.0]
```

One major limitation of the current method is its inability to reflect each statistic's name. To address this, we can collect all statistics in a named tuple, enabling the access of each through its name. For instance, given a named tuple `stats_viewers`, it'll become possible to retrieve the average value of `viewers` by `stats_viewers.mean` or `stats_viewers[:mean]`.

To assign names to the statistics within the named tuple, we'll use the `Symbol` type. This translates strings into identifiers that can act as keys of a named tuple, enabling programmatic access to each statistic.

```
vector_of_tuples = [(Symbol(fun), fun(viewers)) for fun in list_functions]
stats_viewers    = NamedTuple(vector_of_tuples)
```

```
julia> stats_viewers
(sum = 3447.08, median = 64.8765, mean = 114.903, maximum = 1551.56, minimum = 28.2954)
julia> stats_viewers.mean
114.903
julia> stats_viewers[:median]
64.8765
```

FOOTNOTES

[1.] Recall that the type `Int` defaults to `Int64` on 64-bit systems and to `Int32` for 32-bit systems. Most modern computers fall into the former category, explaining why we usually employ `Int64`.

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. R and Python 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 without compilation latency, since the package discards the first run for the reported timing. Additionally, the package allows users to explicitly control variable scope: by prefixing function arguments with the `$` symbol, they're treated as local variables during a function call.

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. This means that you should always prefix the function arguments 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 comparing the performance of different methods, we must ensure that our measurements aren't skewed by variations in the input data. This implies each approach must be tested using *the exact same set of values*. This guarantees that differences in execution time can be attributed solely to the efficiency of the method itself, rather than to a change in the inputs.

Such consistency can be achieved by using random number generators. They rely on a **random seed**, which is an arbitrary starting point that dictates the entire sequence of values they produce. By setting the same seed before each test, we can generate identical deterministic sequences of random numbers across multiple runs. Importantly, **any arbitrary number can be used for the seed**. The only requirement is that the same number is employed, so that you 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. Note, though, that any other number could be 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`
```

For presentation purposes, code snippets on this website will omit the lines dedicated to setting 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. Yet, this doesn't mean that the choice of method is without practical consequence.

While it's true that operations in isolation may have an insignificant impact on a program's overall runtime, **the relevance of benchmarks emerges when 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 maximizing functions). 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` treated as a global variable. As we'll discuss in later sections, this approach is relatively inefficient. A more performant alternative is given by `sum(a -> 2 * a, x)`, where `x` is passed as a function argument. While we haven't explained why this implementation is better, it's sufficient to note that both methods produce the same result. The measured runtimes of each approach are 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, requiring less than 15% of the time taken by the slower method. However, even the "slow" approach is remarkably fast, taking less than 0.0001 seconds to execute.

This pattern will be recurring in our benchmarks, where absolute execution times are often negligible. In such cases, the relevance of our conclusions heavily depends on the context. 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 executed repeatedly, small differences in performance might accumulate and become meaningful, making the faster approach a more suitable choice.

To make this point concrete, let's revisit the functions from the previous example and call them inside a for-loop that runs 100,000 times. Since our sole goal is to repeat the operation, the iteration variable itself plays no role. In such cases, it's a common practice to employ a **throwaway variable**: placeholder that exists only to satisfy the loop's syntax, without ever being referenced. This convention signals to other programmers that the variable's value can be safely ignored. In our example, `_` serves this purpose, simply reflecting that each iteration performs 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 these functions 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. Such a stark contrast underscores the importance of optimizing functions that are executed repeatedly: even seemingly minor improvements can accumulate into pronounced performance gains.

7d. Preliminaries on Types

Martin Alfaro
PhD in Economics

INTRODUCTION

High performance in Julia is intimately related to 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 this property holds, Julia can specialize its computation method, resulting in fast code.

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. Moreover, **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 serve as mere labels for objects, with objects in turn holding values of specific 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 permits 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 the 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 under a common abstraction, making it possible to define operations generically. 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 particular supertype known as `Union` will be instrumental for our examples. It allows variables to hold values of any type specified in its arguments. Its syntax is `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 constituent types.

ABSTRACT AND CONCRETE TYPES

The hierarchical nature of types makes it possible to represent subtypes and supertypes as trees. Such 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` in Julia is a prime example. In contrast, a **concrete type** represents an irreducible unit that therefore lacks subtypes. Concrete types are considered final, in the sense that they can't be further specialized within the hierarchy.

The diagram below illustrates the difference between abstract and concrete types for scalars. In particular, we present the hierarchy of the type `Number`, where the labels included match the 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**. This means that variables can be instances of `Vector{T}`, where `Vector{T}` doesn't have subtypes. The consequence is that a vector like `Vector{Int64}` isn't a subtype of `Vector{Any}`, even though `Int64` is a subtype of `Any`. This behavior stands in stark contrast to scalars, where `Any` is an abstract type. However, it aligns perfectly with the concept of vectors as collections of *homogeneous elements*, meaning that all elements share the same type.

ONLY CONCRETE TYPES CAN BE INSTANTIATED, ABSTRACT TYPES CAN'T

Instantiation refers to the process of creating an object with a specific type. A key principle of Julia's type system is that **only concrete types can be instantiated**, implying that values can never be represented by abstract types. This distinction helps clarify the meaning of some widespread expressions used in Julia. For example, stating that a variable has type `Any` shouldn't be interpreted literally. Rather, it means the variable can hold values of any concrete type, considering that all concrete types are subtypes of `Any`.

This distinction will become crucial for what follows, particularly for type-annotating variables. It implies that declaring a variable with an abstract type restricts the set of possible concrete types it can 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 connection becomes clear when you consider how Julia performs computations.

High performance in Julia relies heavily on specializing the computation method. We'll see in the next section that this specialization is unattainable in the global scope, as Julia treats global variables as potentially holding values of any type. In contrast, when code is wrapped in a function, the execution process begins by determining the concrete types of each function argument. This information is then used to infer the concrete types of all the expressions within the function body.

When this inference succeeds and all expressions have unambiguous concrete types, the function is considered **type stable**. This enables Julia to specialize its computation method and generate optimized machine code. If, instead, expressions could potentially take on multiple concrete types, performance is substantially degraded, as Julia must consider a separate implementation for each possible type.

For scalars and vectors, type stability essentially requires that expressions ultimately operate on **primitive types**. Examples of numeric primitive types include integers and floating-point numbers, such as `Int64`, `Float64`, and `Bool`. Thus, applying functions like `sum` to a `Vector{Int64}` or `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 a 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 remainder of this section is dedicated to operators and functions for handling types. Specifically, we'll introduce the operator `<:`, which checks whether one type is a subtype of another. Then, we'll examine strategies for constraining variables to specific 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`, thus returning `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>`. This entails that `T` covers multiple possible types.

```
# 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 relying on parameters like `T` are called **parametric types**. In the example above, these types 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 the following equivalences hold.

```
# 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**.

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>`, where `T` can be replaced with other symbol.

Next, we illustrate both methods, separately considering type-annotations for assignments and function arguments.

ASSIGNMENTS

Let's start illustrating the approaches based on scalar assignments. Each tab below declares an identical type for `x` and for `y`.

```julia
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
```

```julia
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 a type for `x` has been assigned, the type can't be changed. The only way to fix this is by starting a new session.

The fact that `x` holds the same type across all tabs follows because `T <: Float64` can only represent `Float64`. More specifically, `Float64` is a concrete type, which by definition has no subtypes other than itself. Considering this, scalar types are usually asserted using `::` rather than `<:`.

While this behavior holds for scalars, it doesn't apply to vectors. Specifically, using `::` in combination with `Vector{Number}` establishes that the object will have `Vector{Number}` as its concrete type. Instead, `Vector{T} where T <: Number` indicates that the elements of the vector will adopt a concrete subtype of `Number`.

```julia
# `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})
```

```julia
# `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**. 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 variables listed below have their types constrained in a similar manner.

```julia
# all are equivalent
a      = 2
b::Any = 2
```

```julia
# all are equivalent
a                   = 2
b::T where T        = 2
c::T where T <: Any = 2
```

Once we recognize that variables default to the type `Any`, it becomes clear why they can be reassigned with values of different types. 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. This is illustrated below, where functions are restricted to 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 when both function arguments are annotated with the same type parameter `T`, they're constrained to share exactly the same type. Also notice that types like `Int64` preclude the use of `Float64`, even for numbers like `3.0`. Considering both facts, greater flexibility can be achieved by introducing separate type parameters, annotating them with a common abstract type like `Number`.

```julia
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)
```

```julia
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'll 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).

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

```julia
function foo(x::Any, y::Any)
    x + y
end
```

```julia
function foo(x::T, y::S) where {T <: Any, S <: Any}
    x + y
end
```

```julia
function foo(x::T, y::S) where {T, S}
    x + y
end
```

DEFINING VARIABLES WITH CERTAIN TYPE

To conclude this section, we present an approach for converting values into a specific type. The approach makes use of 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 via functions of the form `Type(x)`, where `Type` should be replaced with the name of the type (e.g., `Vector{Float64}`). Like any other function, constructors also support 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, abstract types may still serve as constructors, despite that they can't be instantiated. In such cases, Julia will attempt to convert the object to a specific concrete type. Nonetheless, 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
```

An alternative to transform `x`'s type into `T` is given by the function `convert(T,x)`. Note that this method only works when a valid conversion exists, such as when `Float64` can be translated into an equivalent `Int64` (e.g., `3.0`).

```
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, eventually selecting a corresponding method for computation. 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.

VARIABLE SCOPE IN FUNCTIONS

To fully appreciate why functions are essential for performance in Julia, we must first revisit the notion of variable scope. Recall that **local variables** in a function include its arguments and any variable created inside the function body. These variables exist only during the function's execution and are inaccessible from outside. **Global variables**, on the other hand, refer to any variable defined outside a function's scope and remain accessible throughout program execution.

Anticipating the conclusions we'll obtain this section, a key takeaway is that **wrapping code in functions is crucial for high performance in Julia**. Functions ensure that variables are local, which helps to avoid the significant performance penalty introduced by global variables.

Note that the use of global variables isn't limited to code written in the global scope. It also arises when a function references variables that aren't passed as arguments. As a result, the detrimental effect of global variables will appear in all the following cases.

GLOBAL SCOPE

```
x   = 2

y   = 3 * x
```

```
julia> y
6
```

FUNCTION USING A GLOBAL VARIABLE

```
x   = 2

f() = 3 * x
```

```
julia> f()
6
```

Recall that an expression like `x = 2` is shorthand for `x::Any = 2`, reflecting that global variables default to `Any` if they aren't 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's a subtype of `Any`. Since `Any` sits at the top of Julia's type hierarchy, this simply means that `x`'s types are unrestricted.

The unrestricted nature of global variables prevents specialization of operations. In our example, this means that a global variable like `x` prevents specialization of `*`. Because of this possibility, 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.

The reason for Julia treating global variables as potentially embodying any type is that, even if a variable holds some value at a specific moment, the user may reassign it at any point in the program. However, even if we had type-annotated `x` with a concrete type like `x::Int64 = 2`, the performance limitations wouldn't completely go away. Certain optimizations can only be implemented when both the scope of variables is clearly delimited and values are known. When this is the case, Julia can gain a comprehensive view of all the operations to be performed, creating opportunities for further optimizations.

Functions in Julia are designed precisely to address these considerations. By enforcing local scope and predictable variable behavior, they allow the compiler to specialize methods and apply powerful optimizations. In the next section, we'll explore the steps Julia takes to achieve this.

FUNCTIONS AND METHODS

A **function** is a name that can be associated with multiple implementations. These implementations are known as methods. A **method** specifies the function body for a particular combination of argument types and number of arguments. The list of methods associated with a function `foo` can be retrieved by executing `methods(foo)`.

To illustrate this, let's define several methods for some function `foo1`. Creating methods requires type-annotating `foo1`'s arguments with the operator `::` during its definition. This strategy makes it possible to set a distinct function body for each unique combination of argument types.

To keep matters simple, let's start considering a scenario where all the methods of `foo1` have the same number of arguments.

METHODS

```
foo1(a,b)                  = a + b
foo1(a::String, b::String) = "This is $a and this is $b"
```

```
julia> methods(foo1)
2 methods for generic function "foo1" from Main
julia> foo1(1,2)
3
julia> foo1("some text", "more text")
"This is some text and this is more text"
```

Since `foo1(a,b)` is equivalent to `foo1(a::Any,b::Any)`, the first method sets the behavior of `foo1` for any combination of input types. This behavior is subsequently overridden by defining the method `foo1(a::String, b::String)`. By doing this, we're providing a specific function body for `a` and `b` with type `String`. The existence of multiple methods explains the differences in outputs obtained: the first method of `foo1` is called with `foo1(1, 2)`, whereas `foo1("some text", "more text")` triggers the second method.

The example also reveals that **methods don't need to perform similar operations**. Although mixing drastically different operations under a single function name isn't recommended, allowing variation in function bodies is relevant for performance. In practice, it's commonly used for adapting computation algorithms to each specific type combination, thus optimizing the overall performance of a function.

Also note that **methods don't need to have the same number of arguments**. For instance, it's possible to define the following methods for a function `foo2`.

METHODS WITH DIFFERENT NUMBERS OF ARGUMENTS

```
foo2(x)       = x
foo2(x, y)    = x + y
foo2(x, y, z) = x + y + z
```

```
julia> methods(foo2)
3 methods for generic function "foo2" from Main
julia> foo2(1)
1
julia> foo2(1, 2)
3
julia> foo2(1, 2, 3)
6
```

Defining methods with different number of arguments is particularly useful for extending a function's behavior. A prime example 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 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 CALLS

Building upon our understanding of function definition and methods, let's now analyze the process triggered when a function is called. All our explanations will be based on the following function `foo`:

EXAMPLE

```
foo(a, b) = 2 + a * b
```

```
julia> foo(1, 2)
4
julia> foo(3, 2)
8
julia> foo(3.0, 2)
8.0
```

In Julia, defining a function like `foo(a, b)` is shorthand for creating a **method** with the signature `foo(a::Any, b::Any)`. Thus, the function body `foo(a,b)` holds for all possible type combinations of `a` and `b`.

When `foo(1, 2)` is called, Julia evaluates the expression `2 + a * b` by following a series of steps.

The process begins with what's known as **multiple dispatch**, where Julia selects which method of a function to execute. Importantly, this decision is based solely on the types of the arguments, not their values. Specifically, Julia begins by identifying the concrete types of the function arguments. In our example where `a = 1` and `b = 2`, both are identified as `Int64`. The information on types is then used to select a *method*, which defines the function body and hence the operations to be performed. This process involves searching through the available methods of `foo` to find the most specific one for the concrete types of `a` and `b`. In our example, `foo` has only one method `foo(a,b) = 2 + a * b`, which is defined for all argument types, including `a::Int64` and `b::Int64`. Therefore, the corresponding function body is `2 + a * b`.

The specific version of this method for the signature `foo(a::Int64, b::Int64)` is known as a **method instance**. If the code for the method instance `foo(a::Int64, b::Int64)` already exists, Julia will directly employ it to compute `foo(1,2)`. Otherwise, the compiler generates optimized code for that method instance, stores it in memory for future use, and Julia executes it.

The following diagram depicts the process unfolded when `foo(1,2)` is executed.

MULTIPLE DISPATCH

The process outlined has implications for how the language works. When a function is called with a particular combination of argument types for the first time, Julia incurs an additional cost because it must generate specialized code for those types. This initial delay is often referred to as **Time To First Plot**, a phrase that highlights how the compilation overhead becomes noticeable in interactive workflows such as plotting. Once the code has been compiled, however, Julia stores the resulting method instance, so that subsequent calls with the same argument types can reuse it. This eliminates the compilation step and leads to much faster execution.

The example with `foo` illustrates this process clearly. After evaluating `foo(1, 2)`, Julia has already compiled a method instance for the signature `foo(a::Int64, b::Int64)`. This is why the subsequent call `foo(3, 2)` can be executed immediately by invoking the cached method instance, without any need for recompilation. Instead, the execution of `foo(3.0, 2)` introduces a new combination of argument types, where `a::Float64` and `b::Int64`. Because no compiled method instance yet exists for this signature, Julia must generate one before executing the function.

TYPE INFERENCE

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.

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 behind **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 values 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, it must create generic code to accommodate multiple combinations of types. This forces Julia to perform type checks and conversions during runtime, significantly degrading performance.

REMARKS ON TYPE INFERENCE

Below, we provide various remarks about type inference that are worth keeping in mind for next sections.

FUNCTIONS DO NOT GUARANTEE IDENTIFICATION OF CONCRETE TYPES

Merely wrapping code in a function doesn't guarantee that the compiler will identify concrete types for all operations. The following example illustrates this.

TYPE-UNSTABLE FUNCTION

```
x       = [1, 2, "hello"]     # Vector{Any}

foo3(x) = x[1] + x[2]         # type unstable
```

```
julia> foo3(x)
3
```

The issue in the example is that the compiler assigns the type `Any` to both `x[1]` and `x[2]`, as they come from an object with type `Vector{Any}`. As a consequence, 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 generated code ignores the actual values `x[1] = 1` and `x[2] = 2`, which would otherwise reveal the type `Int64`.

GLOBAL VARIABLES INHERIT THEIR GLOBAL TYPE

Type inference is restricted to local variables, with any global variable having its type inherited from the global scope. For instance, consider the following example.

UNANNOTATED GLOBAL VARIABLE

```
a         = 2
b         = 1

foo4(a)   = a * b
```

```
julia> foo4(a)
2
```

TYPE-ANNOTATED GLOBAL VARIABLE

```
c         = 2
d::Number = 1

foo4(c)   = c * d
```

```
julia> foo4(c)
2
```

In the examples, `b` and `d` are a global variable. Consequently, the compiler infers `b`'s type as `Any` and `d` as `Number`.

TYPE-ANNOTATING FUNCTION ARGUMENTS DOES NOT IMPROVE PERFORMANCE

Identifying concrete types is crucial for achieving high performance. At first glance, this might suggest that explicitly annotating function arguments is necessary for performance, or at least beneficial. However, thanks to type inference, such annotations are redundant. In fact, adding them can be

counterproductive, as they unnecessarily restrict the types accepted by a function, thereby limiting its flexibility and potential applications.

To better appreciate this loss of flexibility, compare the following scripts.

UNANNOTATED FUNCTION

```
foo5(a, b) = a * b
```

```
julia> foo5(0.5, 2.0)
1.0
julia> foo5(1, 2)
2
```

TYPE-ANNOTATED FUNCTION

```
foo6(a::Float64, b::Float64) = a * b
```

```
julia> foo6(0.5, 2.0)
1.0
julia> foo6(1, 2)
ERROR: MethodError: no method matching foo6(::Int64, ::Int64)
```

The function on the first tab only accepts arguments of type `Float64`, implying that even integers are disallowed. By contrast, the function on the second tab mirrors the behavior with `Float64` inputs, but additionally allows for other types. This flexibility stems from the implicit type annotation `Any` on the function arguments.

Packages Commonly Type-Annotate Function Arguments

When inspecting 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 that computes the revenue of a theater via `nr_tickets * price`. Importantly, the operator `*` in Julia not only implements product of numbers, but also concatenates words when applied to expressions with type `String`. Without type-annotations, the function could potentially be misused, as exemplified in the first tab below. The second tab precludes 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

One of the primary reasons Julia achieves high performance is its ability to generate specialized machine code adapted to specific argument types. This capability relies heavily on type inference, the process by which the compiler deduces the types of variables and intermediate results before execution. When the compiler can successfully predict the concrete types of all operations within a function body, the resulting code is described as **type stable**.

Type stability isn't merely a theoretical ideal, but rather a practical requirement for writing efficient Julia programs. Without it, the compiler is forced to generate generic code, leading to significant performance penalties.

This section formally defines type stability and reviews the tools employed for its verification. In the next section, we'll start to examine how type stability applies in specific scenarios.

AN INTUITION

When a function is called, Julia follows a well-defined sequence of steps to determine how that call should be executed. We previously described this process. To briefly review it, consider a function `foo(x) = x + 2` and the execution of `foo(a)` for some variable `a`. We assume `a` has a specific value assigned and therefore a concrete type, although we omit explicitly specifying a value for `a`. This lets us highlight that the process unfolded depends on types, rather than values.

When evaluating `foo(a)`, Julia first determines the concrete type of `a`, which we'll denote as `T`. It then checks whether a compiled method instance of `foo` specialized for an argument of type `T` already exists. If such an instance exists, then `foo(a)` is executed immediately. Otherwise, Julia proceeds to compile one. This compilation step lverages type inference, wherein the compiler attempts to deduce concrete types for all terms within the function body. The resulting machine code is then stored, making it readily available for subsequent calls of `foo(b)` with `b` having 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 the compiler 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 concrete types for *each* expression within the function**, not only for the final output. This definition aligns with the output provided by `@code_warntype`. This is the built-in macro to detect type instabilities, which we'll present in the next subsection.

When type stability holds, 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, thus accommodating each possible combination of 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". Strictly speaking, however, type stability isn't a property of the function. Rather, it's a property of how the function behaves when called with arguments of particular concrete types. The distinction is crucial in practice, since a function may exhibit type stability for certain input types, but not for others.

AN EXAMPLE

To identify type stability in practice, let's consider the following example.

```
x = [1, 2, 3]                    # `x` has type `Vector{Int64}`

@ctime sum($x[1:2])              # type stable
```

```
 12.598 ns (2 allocations: 80 bytes)
```

```
x = [1, 2, "hello"]              # `x` has type `Vector{Any}`

@ctime sum($x[1:2])              # type UNSTABLE
```

```
 23.696 ns (2 allocations: 80 bytes)
```

The two operations may look identical at first glance, since both ultimately evaluate `1 + 2`. However, the way Julia compiles and executes each version differs significantly, and the first approach ends up being faster. The key distinction is that the first function is type stable.

In that first version, the expression `x[1] + x[2]` can be inferred to have type `Int64`. Because the compiler can determine that both `x[1]` and `x[2]` are `Int64`, it's able to generate efficient, specialized machine code for this concrete type. Importantly, this optimization isn't limited to the specific example shown: any call of the form `sum(y)` where `y` is a `Vector{Int64}` benefits from the same specialization.

The second version, however, introduces type instability. Here, `x` has type `Vector{Any}`, which prevents the compiler from deducing a single concrete type for the expression `x[1] + x[2]`. Since the elements of a `Vector{Any}` may hold values of any subtype of `Any`, the compiler can't predict whether `x[1]` or `x[2]` will be an `Int64`, a `Float64`, a `Float32`, or other type. This results in slow compiled code, and the penalty applies to every call of `sum(y)` where `y` is a `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(1.0)
```

```
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 inclusion 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 expression.
- If at least one line is **red**, the function is **type unstable**. It reflects that one expression 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.

> **Function Call Leverage Information on Types, Not Values**
> The conclusions regarding type stability wouldn't have changed if we had considered `foo(-2)` or `foo(-2.0)` in each tab, respectively. Type stability depends on whether `x` has type `Int64` or `Float64`, regardless of its actual value. Since the compiler analyzes type

> information exclusively, the actual numeric value plays no role in determining type stability.

YELLOW WARNINGS MAY TURN RED

Not all type instabilities carry the same performance cost, and Julia indicates their severity through yellow and red warnings. A yellow warning typically signals a mild issue: the compiler has detected a type instability, but it occurs in a small isolated computation that Julia can still optimize reasonably well. 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(1.0)
```

```
function foo(x)
    y = (x < 0) ?  0  :  x

    [y * i for i in 1:100]
end
```

```
julia> @code_warntype foo(1.0)
```

```
function foo(x)
    y = (x < 0) ?  0  :  x

    for i in 1:100
      y = y + i
    end

    return y
end
```

```
julia> @code_warntype foo(1.0)
```

In the first case, the yellow warning appears because the expression `y * 2` may produce either a `Float64` or an `Int64`. Since this computation happens only once and involves types the compiler can handle efficiently, the impact on performance is minimal. In contrast, the second tab performs repeated evaluations of `y * i` without a stable concrete type for `y`, resulting in a red warning.

Note, though, that a yellow warning doesn't necessarily escalate into a red warning when incorporated into a for-loop. The third tab demonstrates a situation where the compiler can still manage the instability effectively, even under repeated execution. This highlights that type instabilities vary in severity, and not all of them pose a meaningful threat to performance.

For-Loops and Yellow Warnings

When running for-loops, Julia will always emit a yellow warning, even if the operation is type stable. 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` (a value with type `Nothing`) when the sequence is exhausted.

```
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. 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. [1]

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

The notion of type stability applied to scalars is straightforward. It demands operations to be performed on variables with the same concrete type (e.g., `Float64`, `Int64`, `Bool`). Likewise, type stability for vectors requires that their *elements* have a concrete type.

The following table identifies types for 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 the CPU 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. Specifically, when a value is instantiated, we mean that it ultimately has a single concrete type. Abstract types, by contrast, never hold values directly. Their role is purely organizational: they describe sets of possible concrete types and help structure the type hierarchy.

This distinction explains why a type annotation such as `x::Number` shouldn't be read as `x` having type `Number`. Rather, it constrains `x` to values whose concrete types are subtypes of `Number`. At runtime, though, `x` must always have a concrete type. For instance, after evaluating `x::Number = 2`, the variable `x` contains the value 2, whose concrete type is `Int64`.

With this in mind, we can discuss how type instability arises. A common source of instability is mixing values of different types, such as combintations of `Int64` with `Float64`. However, mixing types doesn't automatically imply type instability. This leads us to define the concepts of type promotion and conversion.

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 conversion**. By transforming values to the respective type declared, this feature also prevents the mix of types.

Both mechanisms are illustrated below.

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

x1           = 2
y1           = 0.5

output        = foo(x1,y1)          # type stable: mixing `Int64` and `Float64` results in
`Float64`
```

```
julia> output
1.0
```

```julia
foo(x,y)     = x * y

x2::Float64 = 2                # this is converted to `2.0`
y2          = 0.5

output      = foo(x2,y2)         # type stable: `x` and `y` are `Float64`, so output type is predictable
```

```
julia> output
1.0
```

In the first tab, the output type depends on the argument type. However, in all cases the output 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 `x2` 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 value depends on a conditional statement, with each branch returning 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.

```julia
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)
  17.608 µs (3 allocations: 781.312 KiB)
```

```julia
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)
  45.474 µs (3 allocations: 781.312 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.

This is why it's important to distinguish between the type of the object and the type of its elements. In particular, note that 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 promotion and conversion applied to vectors.

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.

```
v1                 = [1, 2.0, 3.0]     # automatic conversion to `Vector{Float64}`

w1::Vector{Int64}  = v1                # conversion to `Vector{Int64}`
```

```
julia> w1
3-element Vector{Int64}:
 1
 2
 3
```

```
v2                 = [1, 2, 2.5]       # automatic conversion to `Vector{Float64}`

w2::Vector{Number} = v2                # `w2` is still `Vector{Number}`
```

```
julia> w2
3-element Vector{Number}:
 1.0
 2.0
 2.5
```

TYPE INSTABILITY

When evaluating type stability with vectors, two forms of operations must be considered. The first one involves operations that manipulate individual elements `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.

```
z1::Vector{Int}     = [1, 2, 3]

sum(z1)             # type stable
```

```
z2::Vector{Int64}   = [1, 2, 3]

sum(z2)             # type stable
```

```
z3::Vector{Float64} = [1, 2, 3]

sum(z3)             # type stable
```

```
z4::BitVector       = [true, false, true]

sum(z4)             # type stable
```

In contrast, the following vectors have elements with abstract types, which result in type instability.

```
z5::Vector{Number} = [1, 2, 3]

sum(z5)                 # type UNSTABLE -> `sum` must consider all possible subtypes of
`Number`
```

```
z6::Vector{Any}    = [1, 2, 3]

sum(z6)             # type UNSTABLE -> `sum` must consider all possible subtypes of `Any`
```

FOOTNOTES

[1.] 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.

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 throughout the entire codebase, while **local variables** only exist within a specific scope. For this section, the scope of interest is a function, so local variables will exclusively refer to function arguments and variables defined within the function body.

The distinction is especially relevant for this chapter, since **global variables are a frequent source of type instability**. The reason is that Julia doesn't assign concrete types to global variables. As a result, the compiler is forced to consider multiple potential types whenever these variables are used. Such behavior prevents specialization, leading to reduced performance.

The current section explores two approaches to reduce or even eliminate the detrimental effect of global variables: type-annotations and constants. Defining global variables as constants is a natural choice when values are truly constants, such as in the case of $\pi = 3.14159$. More broadly, constants are appropriate whenever a value remains unchanged throughout the program. Compared to type annotations, they 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 **constants in a function behave just like a literal value**. [1]

> **Warning!** - You Should Always 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 are unfeasible in the global scope.

WHEN ARE WE USING GLOBAL VARIABLES?

Let's begin by identifying operations that rely on global variables. To this end, we present two cases, each represented in a different tab. The first one considers the most direct use of global variables, where operations are performed directly in the global scope.

The second tab illustrates a more nuanced case, where a function accesses and manipulates 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 **promote clarity, predictability, testability, and reusability**. These benefits were briefly introduced in a previous section, where functions were framed as units that embody a specific task.

Specifically, self-contained functions are much easier to reason about, because all relevant information is local to the function. Due to this property, you don't need to track the state of variables scattered across the script. Instead, you can focus solely on the function's inputs and its internal logic. Self-contained functions also ensures that its behavior depends only on its arguments, not on the state of global variables. This makes its output more predictable and debugging more straightforward.Finally, a self-contained function behaves like a small, reusable program with a clearly defined purpose. Once written, it can be applied to similar tasks without modification, reducing code duplication and improving the overall maintainability of the codebase.

ACHIEVING TYPE STABILITY WITH GLOBAL VARIABLES

The advantages of self-contained functions provide compelling reasons to avoid global variables. Even so, there are situations where globals remain genuinely useful. A common example is when we work with true constants, defined as values that are fixed throughout the program.

With this in mind, the next section introduces two techniques that let us work with global variables, while mitigating their performance costs.

CONSTANT GLOBAL VARIABLES

Declaring a global variable as a constant simply requires prefixing its name with the `const` keyword, as in `const x = 3`. This mechanism works for any type of value, 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! - Avoid Reassignments to Global Variables

Global variables should be declared as constants only if their values will remain unchanged throughout the session. Although it's possible to redefine constants, doing so is highly discouraged. The feature was only introduced to facilitate testing in interactive sessions, eliminating the need to restart Julia after each modification of a constant's value.

Importantly, if a constant is reassigned, every function that depends on it must be redefined as well. Otherwise, those functions will continue to use the constant's original value. Because this requirement is easy to overlook, the most reliable practice is to rerun the entire script whenever a constant is modified.

To illustrate the potential consequences of ignoring this guideline, 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 x1 = 1
foo()    = x1
foo()             # it gives 1

x1       = 2

foo()             # it still gives 1
```

```
const x2 = 1
foo()    = x2
foo()             # it gives 1

x2       = 2
foo()    = x2
foo()             # it gives 2
```

TYPE-ANNOTATING A GLOBAL VARIABLE

The second approach to address type instability involves declaring *concrete* types for global variables. This is done by appending the operator `::` to the variable name, as in `x::Int64`. When working with vectors, ensure that their element type is also concrete. Otherwise, the variable will remain type-unstable despite the annotation.

```
x3::Int64           = 5
foo()               = 2 * x3

@code_warntype foo()     # type stable
```

```
x4::Vector{Float64} = [1, 2, 3]
foo()               = sum(x4)

@code_warntype foo()     # type stable
```

```
x5::Vector{Number}  = [1, 2, 3]
foo()               = sum(x5)

@code_warntype foo()     # type UNSTABLE
```

DIFFERENCES BETWEEN APPROACHES

The two approaches presented for handling global variables have different implications for both code behavior and performance. The key lies in that **type-annotations assert a variable's type, while constants fix both their types and values**. Next, we analyze the main differences between both approaches.

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 a variable.

```
x6::Int64 = 5
foo()     = 2 * x6
foo()               # output is 10

x6        = 2
foo()     = 2 * x6
foo()               # output is 4
```

DIFFERENCES IN PERFORMANCE

Type-annotated global variables are more flexible than constants: they require only a declaration of types, without binding to a specific value. This flexibility, however, comes at a performance cost. The reason is that constants not only convey type information, but also act as a promise of immutability throughout the program. As a result, constants behave like literal values, embedded directly in the code. ith this guarantee in place, the compiler can apply stronger optimizations. For instance, by replacing certain expressions with their precomputed results.

The following code demonstrates this behavior. It performs an operation that can be precomputed if the value of the global variable is known at compile time. Declaring the global variable as a constant allows the compiler to replace the operation with its result, effectively treating it as a hard-coded value. In contrast, merely type-annotating the global variable constrains only its type, without fixing its value. To make the performance difference more evident, we call this operation repeatedly inside a for-loop.

```
const k1  = 2

function foo()
    for _ in 1:100_000
       2^k1
    end
end
```

```
julia> @btime foo()
  0.791 ns (0 allocations: 0 bytes)
```

```
k2::Int64 = 2

function foo()
    for _ in 1:100_000
        2^k2
    end
end
```

```
julia> @btime foo()
  104.374 µs (0 allocations: 0 bytes)
```

Invariance of Operations

Even without declaring variables as constants, the compiler could still recognize the invariance of some operations across repeated calculations. In such cases, it computes the operation once and reuses the result whenever needed.

To illustrate, consider reexpressing each element of `x` as a proportion relative to the sum of elements. A naive implementation would involve a for-loop with `sum(x)` inside the for-loop body, causing `sum(x)` to be recomputed on every iteration. By contrast, when shares are computed through `x ./ sum(x)`, the compiler is smart enough to recognize the invariance of `sum(x)` across iterations. Therefore, it proceeds to its pre-computation, eliminating redundant work.

```
x           = rand(100_000)


foo(x) = x ./ sum(x)
```

```
julia> @btime foo($x)
  49.166 µs (3 allocations: 781.312 KiB)
```

```
x           = rand(100_000)
const sum_x = sum(x)

foo(x) = x ./ sum_x
```

```
julia> @btime foo($x)
  41.983 µs (3 allocations: 781.312 KiB)
```

```
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)
  830.068 ms (3 allocations: 781.312 KiB)
```

FOOTNOTES

[1.] A literal value is one written directly in the code (e.g., `1`, `"hello"`, or `true`) rather than provided through a variable.

8e. Barrier Functions

Martin Alfaro
PhD in Economics

INTRODUCTION

This section presents an approach to mitigating type instability through the use of barrier functions. A **barrier function** is a type-stable function that's called from within a type-unstable context, with the variables of uncertain type passed explicitly as arguments. Introducing barrier functions prompts the compiler to infer concrete types for those variables, effectively creating a "barrier" that prevents the propagation of type instability to subsequent operations.

A key benefit of barrier functions is that **they're agnostic to the underlying source of type instability**, making them widely applicable across scenarios.

> **Warning! -** Barrier Functions Should Be Considered as a Second Option
>
> Barrier functions are preferred for situations where type instability is either difficult to fix or directly unavoidable. Keep in mind that the original function will remain type unstable, entailing different consequences depending on the instability nature. For this reason, 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 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(1.0)     # type UNSTABLE
```

In the example, `0` is an `Int64`, whereas `x` could be either an `Int64` or `Float64`. This leads to two possibilities:

- if `x` is an `Int64`, then `y` will also be an `Int64`, making `foo(1)` type stable.

- if `x` is a `Float64`, the compiler then can't determine whether `y` will be an `Int64` or a `Float64`, rendering `foo(1.0)` type unstable.

A barrier function can address the type instability of the second case. It requires embedding a type-stable function into `foo`, passing `y` as an argument. The function will then attempt to deduce `y`'s type, allowing the compiler to use this information for subsequent operations. The example below 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(1.0)   # barrier function is type stable

@code_warntype foo(1)           # type stable
@code_warntype foo(1.0)         # barrier-function solution
```

With the introduction of `operation`, the variable `y` in `foo(1.0)` 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 with a method specialized for the specific type of `y`, ensuring type stability.

> **Warning!**
>
> Barrier Functions should address 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 the technique correctly: `y` can be either `Float64` or `Int64`, but `operation(y,i)` only identifies the type inside the for-loop. Thus, the compiler is forced to check `y`'s type at each iteration, which is the original problem we 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(1.0)        # type UNSTABLE
```

REMARKS ON @CODE_WARNTYPE

Functions that introduce barrier functions are capable of addressing type instability. However, this effect isn't necessarily reflected when `@code_warntype` is executed. The reason is that barrier functions typically mitigate type instability, rather than completely eliminating it. When this is the case, nonetheless, the impact of the remaining type instability may be negligible. And even if the barrier function successfully eliminates the type instability, a red warning may still be triggered.

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. Overall, 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[2])
```

The effectiveness of a barrier function ultimately hinges on how the function `foo` will be 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 for-loop, the type instability of `2 * x[2]` would be incurred multiple times. In such cases, simultaneously addressing the type instability in `2 * x[2]` could entail substantial performance benefits.

FOOTNOTES

[1.] In this particular example, there's an easier solution, where `0` is substituted with `zero(x)`. The function `zero(x)` has been designed to return the additive identity (i.e., the null element) of `x`'s type.

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 a single concrete type for each expression. This definition, while universal, takes on different forms when applied to specific objects. So far, we've exclusively dealt with scalars and vectors, whose conditions for type stability are relatively straightforward.

In this section, we begin our analysis of type stability for other data structures. In particular, we consider tuples, whose coverage automatically encompasses **named tuples**. Guaranteeing type stability with tuples is more nuanced compared to vectors, as their type characterization demands more information. In fact, 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 common forms of collections in Julia. While both fulfill a similar purpose, they differ significantly in their underlying implementation. In particular, tuples tend to outperform vectors when working with small objects, by avoiding the memory-allocation overhead incurred by vectors. This advantage will be explained in more depth when discussing static vectors, which are essentially tuples that can be manipulated as vectors.

Another key distinction is that tuples possess a more intricate type system in comparison to vectors. To see this, let's compare the information needed to describe each type.

Vectors represent collections of elements sharing a *homogeneous* type and exhibiting a variable size. 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 the 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,T}`, 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.

TUPLE SLICES WITH MIXED TYPES CAN STILL BE TYPE STABLE

One key difference between tuples and vectors in Julia lies in how they handle type information. While tuples explicitly define the type of each individual element, vectors require all elements to be of a uniform type.

Because vectors must maintain a consistent type throughout, attempting to store mixed concrete types within a single vector compels Julia to determine a common type that accommodates them all. For example, if you create a vector containing both `Int64` and `Float64`, Julia will infer the type of the vector as `Vector{Float64}`, the most general type encompassing both integer and float types.

However, when dealing with highly diverse element types within a vector, this process can lead to less efficient behavior. In extreme cases, Julia might resort to using the abstract type `Any`, resulting in a `Vector{Any}`. Working with vectors like this is extremely undesirable from a performance point of view.

This issue particularly affects vector slices, as they inherit the type information from their parent vector. Thus, if the parent vector has been widened to a more general type like `Vector{Any}`, operations performed on those slices will also be subject to that same type instability. The behavior contrasts sharply with **slices of tuples, where each element within the slice retains its concrete type.**

TUPLE

```julia
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
```

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 outcome of this conversion is predictable, stemming directly from our previous analysis: type stability will be preserved when the tuple contains all elements having the same type or when heterogeneous types can be promoted to a common concrete type.

For the examples, recall that **each type automatically generates a function** that transforms variables into the corresponding type. For instance, the function `Vector` converts variables to this type.

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
```

Likewise, **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, when attempting to construct a tuple from a vector, the compiler must account for the possibility of varying numbers of arguments. The result is that each potential number of elements corresponds to a distinct concrete type for the tuple.

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 introduces 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 A 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.

Note first that simply passing the vector's number of elements as a function 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.

DEFINING DISPATCH BY VALUE

Dispatch by value enables passing information about values to the compiler. Nonetheless, implementing this feature 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 the condition is `true` or `false`, and therefore have knowledge about `y`'s type. This makes it possible to specialize its operations.

TYPE UNSTABLE

```julia
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 VIA "VAL"

```julia
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 `{}`, as types define their parameters with `{}`. Instead, `Val` must be called with `()`, as with any other function.

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 VIA "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` with an abbreviation for the type `NamedTuple`. The "N" in the former case refers 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 handled just like any other variable: we can define vectors of functions, have functions whose outputs are other functions, and do many more sophisticated things that would be impossible if functions were treated as different entities.

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 worked with several of them, often in the form of anonymous functions passed as function arguments. A familiar example is `map(<function>, <collection>)`, which applies `<function>` to every element of `<collection>`.

In this section, the focus will be on conditions under which higher-order functions are type-stable. As we'll discover, these functions present some challenges in this regard.

> **Remark**
> Throughout the explanations, we'll often refer to the function passed as an argument as the *callback function*.

THE ISSUE

In Julia, *each function defines its own unique concrete type*. In turn, this concrete type is a subtype of an abstract type called `Function`. The type `Function` encompasses all possible functions defined in Julia. The design of the type system creates challenges when specializing the computation method of higher-order functions. Specifically, it can potentially lead to a combinatorial explosion of methods, where a unique method is generated for each callback function.

To address this issue, Julia takes a conservative stance, **often choosing not to specialize the methods of higher-order functions**. In particular, we'll see that Julia avoids specialization if the callback function isn't explicitly called. The performance in those cases can drop sharply, as the execution runtime would become similar to performing operations in the global scope.

Given this, it's important to pinpoint the scenarios where specialization is inhibited and monitor its consequences. If you notice that performance is severely impaired, there are still ways to enforce specialization. In this section, we'll explore these strategies.

AN EXAMPLE OF NO SPECIALIZATION

Let's illustrate the conditions under which higher-order functions fail to specialize. Consider a scenario where the goal is to sum the transformed elements of a vector `x`. The only requirement imposed is that the transforming function should be generic, allowing us to possibly apply different functions for the transformation.

We implement this construction via a higher-order function `foo`. The function applies broadcasting to transform `x` through some function `f`. To demonstrate how `foo` works, we call it with the function `abs` as the transformation function, which provides absolute values.

```
x          = rand(100)

foo(f, x) = f.(x)
```

```
julia> @code_warntype foo(abs,x)
```

Even when `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.

Type instability in this case arises because Julia **avoids specialization if a callback function isn't explicitly called within the function**. In the example, the function `f` only enters `foo` as an argument of broadcasting, but there's no explicit line calling `f`.

To obtain indirect evidence about the lack of specialization, we can compare the execution times of the original `foo` function with a version that explicitly calls `f`.

```julia
x = rand(100)

function foo(f, x)

    f.(x)
end
```

```
julia> foo(abs, x)
100-element Vector{Float64}:
 0.9063
 0.443494
 ⋮
 0.121148
 0.20453
julia> @btime foo(abs, $x)
  1.379 μs (12 allocations: 1.250 KiB)
```

```julia
x = rand(100)

function foo(f, x)
    f(1)                # irrelevant computation to force specialization
    f.(x)
end
```

```
julia> foo(abs, x)
100-element Vector{Float64}:
 0.9063
 0.443494
 ⋮
 0.121148
 0.20453
julia> @btime foo(abs, $x)
  41.000 ns (2 allocations: 928 bytes)
```

The comparison reveals a significant reduction in execution time when `f(1)` is added, along with a notable decrease in memory allocations. As we'll demonstrate in future sections, excessive allocations are often indicative of type instability.

FORCING SPECIALIZATION

> **Warning!**
>
> Exercise caution when forcing 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 higher-order function with many unique functions. [1]

Explicitly calling the callback function to circumvent the no-specialization issue isn't ideal, as it introduces an unnecessary computation. Fortunately, alternative solutions exist. One of them is to type-annotate `f`, which provides Julia with a hint to specialize the code for that type of function.

Another solution involves wrapping the function in a tuple before passing it as an argument. This ensures the identification of the function's type, as tuples define a concrete type for each of their elements.

Below, we outline both approaches.

```julia
x     = rand(100)

function foo(f, x)

    f.(x)
end
```

```
julia> foo(abs, x)
100-element Vector{Float64}:
 0.9063
 0.443494
 ⋮
 0.121148
 0.20453
julia> @btime foo(abs, $x)
  1.369 μs (12 allocations: 1.250 KiB)
```

```julia
x     = rand(100)


function foo(f::F, x) where F
    f.(x)
end
```

```
julia> foo(abs, x)
100-element Vector{Float64}:
 0.9063
 0.443494
 ⋮
 0.121148
 0.20453
julia> @btime foo(abs, $x)
  53.782 ns (2 allocations: 928 bytes)
```

```julia
x     = rand(100)
f_tup = (abs,)

function foo(f_tup, x)
    f_tup[1].(x)
end
```

```
julia> foo(f_tup, x)
100-element Vector{Float64}:
 0.9063
 0.443494
 ⋮
 0.121148
 0.20453
julia> @btime foo($f_tup, $x)
  70.496 ns (2 allocations: 928 bytes)
```

FOOTNOTES

[1.] For discussions about the issue of excessive specialization, see here and here.

8h. Type-Stability Gotchas

Martin Alfaro
PhD in Economics

INTRODUCTION

This section presents subtle scenarios where type instabilities arise. Since the root cause of the type instability isn't immediately obvious, we refer to these cases as "gotchas" and offer guidance on how to address them. To ensure self-containment, we revisit some examples previously discussed, but now providing recommendations for their mitigation.

GOTCHA 1: INTEGERS AND FLOATS

`Int64` and `Float64` are distinct types. Even though Julia promotes integers to floating-point numbers in many contexts, mixing them can still inadvertently introduce type instability.

To illustrate this, consider a function `foo` that takes a numeric variable `x` as its argument and performs two tasks: it first defines a variable `y` that replaces `x`'s negative values with zero, and then it executes an operation based on the resulting `y`.

In the following, we implement `foo` with an approach that suffers from type instability, and another one that addresses the issue.

```julia
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(1.0)    # type UNSTABLE
```

```julia
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(1.0)    # type stable
```

The first implementation uses the literal `0`, whose type is `Int64`. If `x` is also `Int64`, no type instability arises. However, if `x` is `Float64`, the compiler treats `y` as potentially `Int64` or `Float64`, thus causing type instability. [1]

Note, though, that Julia can generally handle combinations of `Int64` and `Float64` quite effectively. Thus, this type instability wouldn't be a significant problem if the operation calls `y` only once. Indeed, `@code_warntype` in this case would simply issue a yellow warning, hinting at potential for optimization but not a severe performance bottleneck. However, `foo` in our example repeatedly performs an operation involving `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** for this scenario. It introduces a function that returns the zero element of `x`'s type, instead of `0`. In this way, `y` is created ensuring that types won't be mixed.

This approach to solving type instability can be extended to values different from zero, by use of 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` 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(1.0)     # 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(1.0)     # 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(1.0)     # type stable
```

GOTCHA 2: COLLECTIONS OF COLLECTIONS

In data analysis, it's common to manipulate *collections of collections*, where one data structure nests others inside it. A well-known example in Julia is the `DataFrames` package, which organizes data into columns that represent multiple variables. In this case, each column is a collection itself, with the full set of columns acting as another collection. Since we haven't introduced this package, we'll consider a simpler analogous structure: a vector of vectors, whose type is `Vector{Vector}`.

The primary benefit of using `Vector{Vector}` is its inherent flexibility. Since it imposes no constraints on the element types stored in the inner vectors, each vector can hold whatever data you need: strings, integers, floating-point numbers, or even mixture of these. This makes it easy to represent heterogeneous datasets, without committing to a rigid schema.

That same flexibility, however, comes with a cost. The type system only knows that each element is some vector, with no information about the concrete element type inside each inner vector. Without that information, the compiler can't infer types when your code operates on these vectors, thus leading to type instability.

To see this more concretely, imagine a vector `data` whose elements are themselves vectors. Suppose we write a function `foo` that receives `data` and performs some computation on one of its inner vectors, `vec2`. As shown in the first tab, the compiler can determine that `vec2` is a vector, but it can't deduce the type of its elements. Consequently, calls to `foo` become type-unstable.

A simple and effective way to **address** this issue appears in the second tab. The solution is to introduce a barrier function that takes the inner vector `vec2` as its argument. The barrier function then rectifies the type instability by attempting to identify a concrete type for `vec2`.

```
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
```

Note that the second tab defines the barrier function in-place. This means that the function directly updates the contents of the inner vector `vec2`, rather than creating a new copy. Because `vec2` is part of the larger structure data, the outer structure `data` is updated as well. This approach is typical in data-analysis workflows, where the objective is to transform an existing dataset, rather than allocate a fresh one every time a change is applied.

GOTCHA 3: BARRIER FUNCTIONS

Barrier functions are an effective technique to mitigate type instabilities. However, keep in mind that **the parent function may remain type unstable**. When this occurs and instability isn't resolved before executing a repeated operation, the associated performance penalty 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
```

```julia
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 on the basis of types, without taking actual values into account. To demonstrate this, consider the following example.

```julia
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 `1.0`, 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 communicate information about values directly to the compiler. It's based on the type `Val` in conjunction 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 can then be identified using the `where` keyword within the function definition. Finally, when calling `foo`, we need pass `Val(a)` as its input.

Applied to our example, type instability in `foo` is caused because the value of `condition` isn't known by the compiler. Dispatching by value provides a mechanism to explicitly convey this information and hence solve the type instability.

```
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 both positional and keyword arguments. The possibility of keyword arguments in particular allows the user to assign default values. If these default values are set through variables rather than literal values, a type instability will be introduced. The reason is that such variables will be treated as global variables.

```
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
```

In case you necessarily need to set a variable as a default value, 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. hese 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, with the function attempting to infer a concrete type for the default value (*Solution 2*).

You can also solve the type instability by adopting a local approach, where type annotations are added to either the keyword argument (*Solution 3a*) or the default value itself (*Solution 3b*). Note that this isn't necessary when positional arguments are used as default values of keyword arguments (*Solution 4*).

All these scenarios are represented 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. They refer to functions that capture and retain access to variables from the scope in which they were defined. In practical terms, a closure arises when **one function is defined inside another**, including the case where anonymous functions are used inside a function.

Although closures provide a convenient way to write modular and self-contained code, they can sometimes introduce type instabilities. While Julia has made progress in mitigating these issues, they have persisted for years and remain a source of potential inefficiency. For this reason, it's essential to understand not only the consequences of using closures carelessly, but also to learn strategies for addressing their performance challenges.

CLOSURES ARE COMMON IN CODING

There are several scenarios where closures emerge naturally. One such scenario is when a task requires multiple steps, but you prefer to keep 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 how code implements a task with and without closures, we'll use generic code. This isn't intended to be executed, but rather to demonstrate the underlying structure.

```
function task()
        # <here you define parameters and initialize variables>

    function output()
        # <here you do computations with the parameters and variables>
    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 when one of these conditions hold:

- variables or arguments are redefined inside the function (e.g., when updating a variable)
- the order in which functions are defined is altered
- anonymous functions are introduced

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.

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 emerges when the reassignment involves the same object, 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
```

```julia
function foo()
    x::Int64     = 1
    x            = 1
    bar()::Int64 = x::Int64

    return bar()
end

@code_warntype foo()            # type UNSTABLE
```

```julia
function foo()
    x::Int64     = 1
    bar()::Int64 = x::Int64
    x            = 1

    return bar()
end

@code_warntype foo()            # type UNSTABLE
```

```julia
function foo()
    bar()::Int64 = x::Int64
    x::Int64     = 1
    x            = 1

    return bar()
end

@code_warntype foo()            # type UNSTABLE
```

```julia
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 closures as function arguments can sidestep the issue. However, such an approach is at odds with how code is generally written in Julia.

```julia
function foo(x)
    closure1(x) = x
    closure2(x) = closure1(x)

    return closure2(x)
end

@code_warntype foo(1)            # type stable
```

```julia
function foo(x)
    closure2(x) = closure1(x)
    closure1(x) = x

    return closure2(x)
end

@code_warntype foo(1)            # type UNSTABLE
```

```julia
function foo(x)
    closure2(x, closure1) = closure1(x)
    closure1(x)           = x

    return closure2(x, closure1)
end

@code_warntype foo(1)            # type stable
```

```julia
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

```julia
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
```

```julia
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
```

```julia
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
```

```julia
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 `0.0` 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/CPU Registers vs Heap

Martin Alfaro
PhD in Economics

INTRODUCTION

Memory allocations occur every time a new object is created. It involves setting aside a portion of the computer's Random Access Memory (RAM) to store the object's data. Alternatively, the compiler could optimize storage for extremely simple objects and directly opt to store them in CPU registers.

Conceptually, RAM is divided into two main areas: the stack and the heap. These areas aren't physical locations, but rather logical models that govern how memory is managed. When an object is created, its storage location is implicitly determined by its type. For example, collections defined as vectors are stored on the heap, whereas those defined as tuples can be allocated either on the stack or on registers, but never on the heap.

The choice between heap and non-heap allocations has significant implications for performance. Allocating memory on the heap is a comparatively expensive operation. It requires a systematic search for an available block of memory, bookkeeping to track its status, and an eventual deallocation process to reclaim the space once it's no longer needed. [1] Stack, by contrast, is simpler and therefore faster.

The performance gap between stack/registers and heap allocations can easily become a critical bottleneck when an operation is performed repeatedly. This disparity in performance explains the common convention in programming, including Julia, where **memory allocations exclusively refer to heap allocations**. In the following, we provide a brief overview of how each operates.

STACK AND CPU REGISTERS

The stack is typically reserved for holding objects with primitive types (e.g., integers, floating-point numbers, and characters) and fixed-size collections like tuples. These objects are characterized by their fixed size, precluding the possibility of dynamically growing or shrinking. Simultaneously, such constraint also makes the allocation and deallocation of memory extremely efficient. In some cases, the compiler can optimize the storage of these objects and directly store them in CPU registers.

The primary downside of the stack is its limited capacity, with CPU registers being even smaller than the stack. This makes them 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 will significantly degrade performance. [2]

HEAP ALLOCATIONS

Common objects stored on the heap include arrays (such as vectors and matrices) and strings. Unlike registers and the stack, the heap is designed to support more complex data structures. In particular, the heap enables us to handle objects as large as the available RAM permits, with the ability to grow or shrink dynamically.

While the heap offers greater flexibility, its more complex memory management comes at the cost of slower performance. [3] Due to its overhead, the following sections will discuss strategies for reducing heap allocations. These include computational techniques that translate vectorized operations into scalar operations, as well as programming practices that favor mutation of existing objects over the creation of new ones.

FOOTNOTES

1. This deallocation process is often automated by what's known as a garbage collector.

2. 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.

3. Heap memory is managed by what's known as the *garbage collector*, which automatically identifies and reclaims memory no longer in use. This process, while convenient, 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, noting that objects can be stored on either the heap or the stack/CPU registers. Furthermore, we introduced typical terminology, where **allocations exclusively refer to those on the heap**.

This convention underlies the common expression that an object "allocates" when it's stored on the heap. Such usage isn't merely to economize on words. Rather, it reflects a fundamental performance implication: heap allocations are the ones that significantly impact efficiency. They involve a more complex management process, thus potentially introducing significant overhead.

The close relationship between performance and heap allocations is even reflected in Julia's benchmarking tools. Macros like `@time` and `@btime` report not only the total runtime of an operation, but also the heap allocations involved.

Considering the importance of heap allocations, the current section categorizes objects into 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:

- Scalars (numbers)
- Tuples
- Named Tuples
- Ranges

As they don't allocate, neither does their creation, access, or manipulation. This is demonstrated below.

```julia
function foo()
    x = 1; y = 2

    x + y
end
```

```
julia> @btime foo()
  0.914 ns (0 allocations: 0 bytes)
```

```
function foo()
    tup = (1,2,3)

    tup[1] + tup[2] * tup[3]
end
```

```
julia> @btime foo()
  0.932 ns (0 allocations: 0 bytes)
```

```
function foo()
    nt = (a=1, b=2, c=3)

    nt.a + nt.b * nt.c
end
```

```
julia> @btime foo()
  0.835 ns (0 allocations: 0 bytes)
```

```
function foo()
    rang = 1:3

    sum(rang[1:2]) + rang[2] * rang[3]
end
```

```
julia> @btime foo()
  0.910 ns (0 allocations: 0 bytes)
```

ARRAYS AND THEIR SLICES DO ALLOCATE MEMORY

Arrays are among the most common objects that require memory allocation. The more straightforward cases where this allocation occurs is when an array is explicitly created and assigned to a variable, or when a computation returns a new array. The example below demonstrates this case.

```
foo() = [1,2,3]
```

```
julia> @btime foo()
  13.000 ns (2 allocations: 80 bytes)
```

```
foo() = sum([1,2,3])
```

```
julia> @btime foo()
  7.938 ns (1 allocations: 48 bytes)
```

Slicing is another operation that triggers memory allocation. The reason is the default behavior of slicing, which returns a new copy rather than a view of the original object. The sole exception is when a single element is accessed, in which case no allocations take place.

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

foo(x) = x[1:2]                    # allocations only from 'x[1:2]' itself (ranges don't
allocate)
```

```
julia> @btime foo($x)
  13.405 ns (2 allocations: 80 bytes)
```

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

foo(x) = x[[1,2]]              # allocations from both '[1,2]' and 'x[[1,2]]' itself
```

```
julia> @btime foo($x)
  24.094 ns (4 allocations: 160 bytes)
```

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

foo(x) = x[1] * x[2] + x[3]
```

```
julia> @btime foo($x)
  1.711 ns (0 allocations: 0 bytes)
```

Array comprehensions and broadcasting are two additional operations that result in the creation of new arrays. Notably, broadcasting even allocates for intermediate results computed on the fly that aren't explicitly returned. This behavior is demonstrated in the tab "Broadcasting 2" below.

```
foo()  = [a for a in 1:3]
```

```
julia> @btime foo()
  12.361 ns (2 allocations: 80 bytes)
```

```
x      = [1,2,3]
foo(x) = x .* x
```

```
julia> @btime foo($x)
  15.596 ns (2 allocations: 80 bytes)
```

```
x      = [1,2,3]
foo(x) = sum(x .* x)               # allocations from temporary vector 'x .* x'
```

```
julia> @btime foo($x)
  20.165 ns (2 allocations: 80 bytes)
```

9d. Slice Views to Decrease Allocations

Martin Alfaro
PhD in Economics

INTRODUCTION

Previously, we defined a slice as a subvector derived from a parent vector `x`. Common examples include expressions such as `x[1:2]`, which extracts elements at positions 1 and 2, or `x[x .> 0]`, which selects only those elements that are positive. By default, these operations create a copy of the data and therefore allocate memory. The only exception to this rule occurs when a slice comprises a single element.

In this section, we address the issue of memory allocations associated with slices. To do this, we highlight the role of **views**, which bypass the need for a copy by directly referencing the parent object. The strategy can be employed when slices are indexed through ranges, BUT it's not suitable for slices like `x[x .> 0]` that employ Boolean indexing, where memory allocation will still occur.

Interestingly, we'll show scenarios where **copying data could actually be faster than using views**, despite the additional memory allocation involved. This apparent paradox emerges because copied data is stored in a contiguous block of memory, which provides more efficient access patterns.

VIEWS OF SLICES

We begin by showing that views don't allocate memory *when a slice is indexed by a range*. This behavior can yield performance improvements over regular slices, which create a copy of the data by default.

SLICE AS A COPY

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

foo(x) = sum(x[1:2])           # allocations from the slice 'x[1:2]'
```

```
julia> @btime foo($x)
  13.147 ns (2 allocations: 80 bytes)
```

SLICE AS A VIEW

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

foo(x) = sum(@view(x[1:2]))    # it doesn't allocate
```

```
julia> @btime foo($x)
  1.863 ns (0 allocations: 0 bytes)
```

Keep in mind, though, that **views created through Boolean indexing neither reduce memory allocations nor are more performant**. This is why you shouldn't rely on views of these objects if your goal is 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)
  404.357 ns (6 allocations: 4.094 KiB)
```

BOOLEAN INDEX (VIEW)

```
x      = rand(1_000)

foo(x) = @views sum(x[x .> 0.5])
```

```
julia> @btime foo($x)
  591.800 ns (6 allocations: 4.094 KiB)
```

COPYING DATA MAY BE FASTER

Although views can reduce memory allocations, there are scenarios where copying data can result in faster performance. A detailed comparison of copies versus views will be provided in Part II. Here, we simply remark on this possibility.

Essentially, the choice between copies and views reflects a fundamental trade-off between memory allocation and data access patterns. On one hand, newly created vectors store data in contiguous blocks of memory, enabling more efficient CPU access and allowing for certain optimizations. On the other hand, views avoid allocation, but may also require accessing data scattered across non-contiguous memory regions.

Below, we illustrate a scenario in which the overhead of creating a copy is outweighed by the benefits of contiguous memory access, making copying the more efficient choice.

COPY

```
x      = rand(100_000)

foo(x) = max.(x[1:2:length(x)], 0.5)
```

```
julia> @btime foo($x)
  46.710 µs (6 allocations: 781.375 KiB)
```

VIEW

```julia
x      = rand(100_000)

foo(x) = max.(@view(x[1:2:length(x)]), 0.5)
```

```
julia> @btime foo($x)
  160.908 µs (3 allocations: 390.688 KiB)
```

9e. Reductions

Martin Alfaro
PhD in Economics

INTRODUCTION

Reductions are a computational technique for **operations taking a collection as input and returning a single element as output**. Such operations arise naturally in a wide range of contexts, such as when computing summary statistics (e.g., averages, variances, or maxima of collections).

The underlying technique involves iteratively applying an operation to pairs of elements, accumulating the results at each step until the final output is obtained. A classic example of reduction is the summation of all numeric elements in a vector. It arrives at the final result by applying the addition operator `+` to pairs of elements, iteratively updating the accumulated sum. This case is illustrated below.

```
x = rand(100)

foo(x) = sum(x)
```

```
julia> foo(x)
48.447
```

```
x = rand(100)

function foo(x)
    output = 0.0

    for i in eachindex(x)
        output = output + x[i]
    end

    return output
end
```

```
julia> foo(x)
48.447
```

```julia
x = rand(100)

function foo(x)
    output = 0.0

    for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
48.447
```

The last tab implements the reduction via an update operator. These operators are frequently used in reductions because they streamline notation, rewriting expressions like `x = x + a` in the more compact form `x += a`.

The main benefit of reductions is that, by operating on scalars, **they don't create memory allocations**. This is particularly convenient when a vector must be transformed prior to aggregating the result. For instance, if you have to compute `sum(log.(x))`, a reduction would avoid the allocations of the intermediate vector `log.(x)`.

IMPLEMENTING REDUCTIONS

Technically, reductions iteratively apply a *binary operation* to pairs of values, ultimately producing a scalar result. For a reduction to be valid, the binary operation must satisfy **two mathematical properties**:

- **Associativity**: the grouping of operations doesn't affect the outcome. For example, scalar addition is associative because $(a + b) + c = a + (b + c)$.
- **Existence of an identity element**: there exists an element that, when combined with any other element through a binary operation, leaves that element unchanged. For example, the identity element of scalar addition is 0 because $a + 0 = a$.

The identity element serves as the initial value of the accumulator variable. [1] Each operation has its own identity element, as shown in the table below.

Operation	Identity Element
Sum	0
Product	1
Maximum	-Inf
Minimum	Inf

Remarkably, binary operations can be expressed either as binary operators or as two-argument functions. For instance, the symbol `+` can be employed in either form, making `output = output + x[i]` equivalent to `output = +(output, x[i])`. The possibility of using functions for reductions expands their scope. For instance, it enables the use of the `max` function to find the maximum value in a vector `x`, where `max(a,b)` returns the larger of the two scalars `a` and `b`.

The figures below visually illustrate reductions implemented with a binary operator and with a two-argument function.

Manual implementations of reductions are done via for-loops. To illustrate its formulation, below we present `foo1` to identify the desired outcome, with `foo2` providing the same result through a reduction.

```
x       = rand(100)

foo1(x) = sum(x)

function foo2(x)
    output = 0.0

    for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
x       = rand(100)

foo1(x) = prod(x)

function foo2(x)
    output = 1.0

    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
```

AVOIDING MEMORY ALLOCATIONS THROUGH REDUCTIONS

One of the primary advantages of reductions is their avoidance of memory allocation, in particular for intermediate results.

To illustrate, consider the operation `sum(log.(x))` for a vector `x`. Its computation involves two steps: transforming `x` into `log.(x)`, and then summing the transformed elements. By default, broadcasting materializes its results, implying the internal creation of a new vector to store the values

of `log.(x)`. Consequently, the step results in memory allocations.

In many cases, however, only the scalar output matters, with the intermediate result being of no interest. In this context, computational strategies that obtain the final output while bypassing the allocation of `log.(x)` are preferred.

Reductions make this possible by defining a scalar `output`, which is iteratively updated by summing the transformed values of `x`. This means that each element of `x` is transformed by the logarithm, and the result is then immediately added to the accumulator. In this way, the storage of the intermediate vector `log.(x)` is entirely avoided. [2]

```
x       = rand(100)

foo1(x) = sum(log.(x))

function foo2(x)
    output = 0.0

    for i in eachindex(x)
        output += log(x[i])
    end

    return output
end
```

```
julia> @btime foo1($x)
  437.594 ns (2 allocations: 928 bytes)
julia> @btime foo2($x)
  427.652 ns (0 allocations: 0 bytes)
```

```
x       = rand(100)

foo1(x) = prod(log.(x))

function foo2(x)
    output = 1.0

    for i in eachindex(x)
        output *= log(x[i])
    end

    return output
end
```

```
julia> @btime foo1($x)
  436.706 ns (2 allocations: 928 bytes)
julia> @btime foo2($x)
  427.406 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($x)
  673.357 ns (2 allocations: 928 bytes)
julia> @btime foo2($x)
  538.000 ns (0 allocations: 0 bytes)
```

```julia
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($x)
  680.628 ns (2 allocations: 928 bytes)
julia> @btime foo2($x)
  526.982 ns (0 allocations: 0 bytes)
```

REDUCTIONS VIA BUILT-IN FUNCTIONS

So far, reductions with intermediate transformations have been implemented manually through explicit for-loops. While this approach makes the underlying mechanics transparent, it also introduces considerable verbosity.

To address this inconvenience, Julia provides several streamlined alternatives for expressing reductions. One such alternative is through specialized methods of common reduction functions, including `sum`, `prod`, `maximum`, and `minimum`. These function methods accept a transforming

function as their first argument, followed by the collection to be reduced. The general syntax is `foo(<transforming function>, x)`, where `foo` is the reduction function and `x` is the vector to be transformed and reduced.

The following examples illustrate this approach by applying a logarithmic transformation prior to the reduction.

```
x      = rand(100)

foo(x) = sum(log, x)        #same output as sum(log.(x))
```

```
julia> @btime foo($x)
  425.783 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = prod(log, x)       #same output as prod(log.(x))
```

```
julia> @btime foo($x)
  426.043 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = maximum(log, x)    #same output as maximum(log.(x))
```

```
julia> @btime foo($x)
  784.432 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = minimum(log, x)    #same output as minimum(log.(x))
```

```
julia> @btime foo($x)
  793.919 ns (0 allocations: 0 bytes)
```

These specialized function methods are commonly applied using anonymous functions, as shown below.

```
x      = rand(100)

foo(x) = sum(a -> 2 * a, x)       #same output as sum(2 .* x)
```

```
julia> @btime foo($x)
  9.750 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)
  12.222 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)
  237.689 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)
  235.365 ns (0 allocations: 0 bytes)
```

The methods also accept transforming functions with multiple arguments. In this case, the arguments must be paired using `zip`, with indices corresponding to each argument within the transforming function. This is illustrated below, with the transforming operation `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)
  40.502 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)
  64.310 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)
  224.798 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)
  231.638 ns (0 allocations: 0 bytes)
```

THE "REDUCE" AND "MAPREDUCE" FUNCTIONS

Beyond the specific cases discussed, reductions are applicable whenever a binary operation meets the necessary conditions for their implementation. To accommodate this generality, Julia provides the functions `reduce` and `mapreduce`.

The `reduce` function applies a binary operation directly to the elements of a collection, combining them into a single result. By contrast, `mapreduce` first transforms each element before applying the reduction.

It's worth remarking that reductions for sums, products, maximum, and minimum should still be implemented via their dedicated functions. This is because the methods in `sum`, `prod`, `maximum`, and `minimum` have been carefully optimized for their respective tasks, typically outperforming the general functions `reduce` and `mapreduce`. [3]

FUNCTION "REDUCE"

The function `reduce` uses the syntax `reduce(<function>,x)`, where `<function>` is a two-argument function.

```
x      = rand(100)

foo(x) = reduce(+, x)           #same output as sum(x)
```

```
julia> @btime foo($x)
  9.193 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = reduce(*, x)           #same output as prod(x)
```

```
julia> @btime foo($x)
  9.536 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = reduce(max, x)         #same output as maximum(x)
```

```
julia> @btime foo($x)
  229.969 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = reduce(min, x)          #same output as minimum(x)
```

```
julia> @btime foo($x)
  231.930 ns (0 allocations: 0 bytes)
```

Note that all the examples above could've been implemented as before, where we directly applied `sum`, `prod`, `maximum` and `minimum`.

FUNCTION "MAPREDUCE"

The `mapreduce` function integrates `map` and `reduce` into a unified operation (hence its name). Thus, it applies a transformation via the function `map` before doing the reduction. Its syntax is `mapreduce(<transformation>,<reduction>,x)`. 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)
  425.783 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = mapreduce(log, *, x)        #same output as prod(log.(x))
```

```
julia> @btime foo($x)
  425.942 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = mapreduce(log, max, x)      #same output as maximum(log.(x))
```

```
julia> @btime foo($x)
  784.676 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = mapreduce(log, min, x)      #same output as minimum(log.(x))
```

```
julia> @btime foo($x)
  793.946 ns (0 allocations: 0 bytes)
```

Note that, again, the examples could've been implemented directly through the functions `sum`, `prod`, `maximum`, and `minimum` as we did previously.

`mapreduce` can also be used with anonymous functions and functions with multiple arguments. Below, we illustrate these possibilities.

```
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)
  40.503 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)
  64.309 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)
  224.946 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)
  231.547 ns (0 allocations: 0 bytes)
```

REDUCE OR MAPREDUCE?

`mapreduce(<transformation>,<operator>,x)` yields the same result as `reduce(<operator>, map(<transformation>,x))`. Despite this, `mapreduce` is preferred if the vector input must be transformed beforehand. The reason is that `mapreduce` avoids the internal memory allocations of the transformed vector, while `map` doesn't. This aspect is demonstrated below, where `sum(2 .* x)` is computed through a reduction.

```
x      = rand(100)

foo(x) = mapreduce(a -> 2 * a, +, x)
```

```
julia> @btime foo($x)
  9.749 ns (0 allocations: 0 bytes)
```

```
x      = rand(100)

foo(x) = reduce(+, map(a -> 2 * a, x))
```

```
julia> @btime foo($x)
  43.327 ns (2 allocations: 928 bytes)
```

FOOTNOTES

[1.] Strictly speaking, an identity element isn't always required. In many cases, no natural identity exists, yet the first element of the collection can serve as the initial value. For simplicity, however, we assume the existence of an identity element as a requirement.

[2.] In the section Lazy Operations, we'll explore an alternative approach. This is based on broadcasting and also avoids materializing intermediate results.

[3.] The functions `reduce` and `mapreduce` are also convenient for packages that implement specialized versions of reductions. By simply defining these two functions, the package can then cover all possible reductions. For instance, the package `Folds` provides a multithreaded version of reductions via `map` and `mapreduce`.

9f. Lazy Operations

Martin Alfaro
PhD in Economics

INTRODUCTION

Computational approaches can be broadly classified as "lazy" or "eager". **Eager operations** are executed immediately upon definition, providing instant access to the results. This tends to be the default behavior when running an operation.

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 feed 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**. They're the lazy analogue of array comprehensions. After this, we'll introduce several functions from the package **Iterators**, which provides lazy variants of functions such as `map` and `filter`.

GENERATORS

Array comprehensions offer an expressive way to construct vectors, employing a syntax that mirrors for-loops. The elements defined by them are computed and stored right away, making array comprehensions an eager operation. **Generators** simply 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 that they're enclosed in parentheses `( )` instead of square brackets `[ ]`. Just like array comprehensions, generators also retain the ability to add conditions and simultaneously 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 and give immediate access to the elements of the vector. In contrast, generators formally define an object with type `Base.Generator`, where operations are described but no output is materialized.

This characteristic makes generators particularly useful for applying reductions to transformed values of a collection. 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 summing 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.

```julia
x = rand(100)

function foo(x)
    y = [a * 2 for a in x]       # 1 allocation

    sum(y)
end
```

```
julia> @btime foo($x)
  49.500 ns (2 allocations: 928 bytes)
```

```julia
x = rand(100)

function foo(x)
    y = (a * 2 for a in x)      # 0 allocations

    sum(y)
end
```

```
julia> @btime foo($x)
  26.460 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100)


foo(x) = sum(a * 2 for a in x)  # 0 allocations
```

```
julia> @btime foo($x)
  25.852 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 that offer more specialized functionality. They include `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 accessed 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)
  71.961 ns (3 allocations: 1.344 KiB)
```

```
x = collect(1:100)

function foo(x)
    y = Iterators.filter(a -> a > 50, x)    # 0 allocations

    sum(y)
end
```

```
julia> @btime foo($x)
  45.775 ns (0 allocations: 0 bytes)
```

```julia
x = rand(100)

function foo(x)
    y = map(a -> a * 2, x)                  # 1 allocation

    sum(y)
end
```

```
julia> @btime foo($x)
  68.528 ns (2 allocations: 928 bytes)
```

```julia
x = rand(100)

function foo(x)
    y = Iterators.map(a -> a * 2, x)        # 0 allocations

    sum(y)
end
```

```
julia> @btime foo($x)
  27.406 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.

9g. 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.

A key feature of broadcasting is its eager default behavior. This means that broadcast operations compute and materialize outputs immediately upon execution. Thus, it inevitably leads to memory allocation when applied to objects such as vectors. Such behavior becomes especially relevant in scenarios with intermediate broadcast operations, whose allocations are potentially avoidable.

To address the associated performance cost, we'll present two strategies. The first one highlights the notion of **loop fusion**. This enables the combination of multiple broadcasting operations into a single more efficient one. Its relevance lies in minimizing memory allocations, rather than their entire elimination. After this, we'll explore the `LazyArrays` package, which evaluates broadcasting operations lazily. When reductions require intermediate calculations, this technique can completely bypass memory allocations.

HOW DOES BROADCASTING WORK INTERNALLY?

Under the hood, broadcasting operations are converted into optimized for-loops. Indeed, broadcasting is essentially syntactic sugar for for-loops, sparing developers from writing them explicitly. This makes it possible to write code that's concise and expressive, without sacrificing performance.

Despite this, you'll often notice performance differences in practice. These discrepancies stem primarily from compiler optimizations, rather than inherent differences between broadcasting and for-loops. The reason is that an operation supporting a broadcast form is automatically revealing additional information about its underlying structure. Consequently, the compiler is allowed to apply more aggressive optimizations. In contrast, for-loops are conceived as a general-purpose construct, precluding the compiler from automatically making such assumptions.

It's worth noting, though, that carefully optimized for-loops always match or exceed the performance of broadcasting. The following code snippets demonstrate this point. The first tab outlines the operation being performed, while the second tab provides a rough internal translation of broadcasting into a for-loop.

The third tab demonstrates the equivalence more directly, using a for-loop that resembles the broadcast code more closely. Its implementation adds the `@inbounds` macro, which broadcasting always applies automatically. The definition of `@inbounds` will be studied in a later section. Its inclusion here is only to illustrate the internal implementation of broadcasting.

```
x      = rand(100)

foo(x) = 2 .* x
```

```
julia> @btime foo($x)
  25.512 ns (2 allocations: 928 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)
  26.306 ns (2 allocations: 928 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)
  27.366 ns (2 allocations: 928 bytes)
```

> **Warning!** - About `@inbounds`
>
> In the example provided, `@inbounds` **was solely added to demonstrate the internal implementation of broadcasting, not as a general recommended practice**. In fact, `@inbounds` **can cause serious issues** when used incorrectly.
>
> To understand the role of this macro, recall that Julia performs bounds checks in for-loops by default. This means Julia verifies the existence of elements accessed during every iteration. Bounds checking prevents

> out-of-range access, so that a vector `x` with 3 elements isn't accessed at index `x[4]`. Instead, placing `@inbounds` at the beginning of a for-loop instructs Julia to disable these checks.
>
> Removing bounds checking can improve performance, but it comes at the cost of safety: an out-of-range access may result in program termination or trigger more severe issues. Out-of-bounds access can't occur when operations support broadcasting, since the broadcast mechanism guarantees a valid index range. Consequently, Julia safely omits these checks by automatically applying `@inbounds`.

REMARK (OPTIONAL) Other Optimization Differences Between Broadcasting and For-Loops

CONSEQUENCES OF HOW BROADCASTING IS INTERNALLY COMPUTED

Once we understand how broadcasting is computed internally, we can also anticipate its consequences for memory allocations. First, broadcasting involves the creation of a collection like `output` to store the result. Therefore, the operation will necessarily allocate when broadcasting vectors, since `output` will inherit the type of its inputs.

Importantly, **memory allocations in broadcasting arise even when the result isn't explicitly stored**. For example, evaluating `sum(2 .* x)` involves storing internally the intermediate output `2 .* x`.

Second, Julia's broadcasting is eager by default. Continuing with the same example, this means that `2 .* x` in `sum(2 .* x)` is computed immediately. As we'll see, adopting a lazy strategy can be advantageous in such cases. By deferring the computation of `2 .* x` until `sum(2 .* x)` is executed, we let Julia know that the broadcast operation will be used as part of a reduction. Thus, the compiler can optimize the internal computation, by generating a for-loop that reduces a transformed version of `x`. This avoids materializing the intermediate result altogether, thereby eliminating the temporary allocation for `2 .* x`.

LOOP FUSION

When working with long broadcast operations, splitting them into smaller intermediate steps can significantly improve readability and reduce the likelihood of errors. However, this approach comes at the cost of separately allocating each broadcasting operation.

To mitigate unnecessary memory allocations, it's essential to preserve **loop fusion**: a compiler optimization that merges multiple element-wise operations into a single loop over the data. Thus, with loop fusion enabled, 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 unlocking further optimizations.

Loop fusion is applied automatically when all operations are expressed as a single broadcast operation. Yet, as indicated before, writing a single lengthy monolithic expression is inconvenient for complex computations. To overcome this limitation, we can rely on the lazy design of function definitions. Below, we show how this approach can break down an operation into partial calculations, while still preserving loop fusion.

```julia
x        = rand(100)

function foo(x)
    a      = x .* 2
    b      = x .* 3

    output = a .+ b
end
```

```
julia> @btime foo($x)
  82.627 ns (6 allocations: 2.719 KiB)
```

```julia
x        = rand(100)

foo(x)   = x .* 2 .+ x .* 3     # or @. x * 2 + x * 3
```

```
julia> @btime foo($x)
  26.787 ns (2 allocations: 928 bytes)
```

```julia
x        = rand(100)

term1(a) = a * 2
term2(a) = a * 3

foo(a)   = term1(a) + term2(a)
```

```
julia> @btime foo.($x)
  27.222 ns (2 allocations: 928 bytes)
```

Even with a single simple operation, certain coding patterns can inadvertently break loop fusion. When this occurs, Julia will internally fall back to evaluating sub-expressions in separate for-loops, which are eventually combined to present the final output. In the following, we present two of these cases. The key practical takeaway is that you should broadcast via the macro `@.` when possible, rather than manually adding dots to each operator and function within an expression.

MIXING BROADCASTING WITH VECTOR OPERATIONS BREAKS LOOP FUSION

To understand the issue, there are two facts to consider. The first one is that some vector operations produce an equivalent result to their broadcast counterparts. For instance, adding two vectors with `+` yields the same result as summing them element-wise with `.+`.

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

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

```
julia> foo(x,y)
3-element Vector{Int64}:
 5
 7
 9
```

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

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

```
julia> foo(x,y)
3-element Vector{Int64}:
 5
 7
 9
```

Another example is with product operations.

```
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
```

The second fact to consider is that vector operations don't participate in loop fusion. Thus, even if all these operations were combined into a single expression, Julia will evaluate each sub-expression separately, allocating temporary vectors at every step.

```
x = rand(100)

foo(x)  = x * 2 + x * 3
```

```
julia> @btime foo($x)
  84.459 ns (6 allocations: 2.719 KiB)
```

```
x = rand(100)

function foo(x)
    term1  = x * 2
    term2  = x * 3

    output = term1 + term2
end
```

```
julia> @btime foo($x)
  87.042 ns (6 allocations: 2.719 KiB)
```

Putting both facts together, mixing broadcasting with vector operations in the same expression may yield the correct result, but only achieve loop fusion partially. This means Julia will only fuse contiguous broadcast segments, internally partitioning the computation into separate for-loops when a vector operation is encountered. Each of these for-loops will produce a temporary intermediate vector.

```
x      = rand(100)

foo(x) = x * 2 .+ x .* 3
```

```
julia> @btime foo($x)
  67.221 ns (4 allocations: 1.812 KiB)
```

```
x      = rand(100)

function foo(x)
    term1  = x * 2

    output = term1 .+ x .*3
end
```

```
julia> @btime foo($x)
  64.923 ns (4 allocations: 1.812 KiB)
```

This behavior leads to a clear and actionable guideline: for full loop fusion, every operator and function in the expression must be explicitly broadcast. Yet manually adding dots throughout a large expression is both tedious and error-prone: one missing dot is enough to reintroduce unnecessary allocations. There are two coding practices that mitigate this risk.

One option is to broadcast via the macro `@.`, as shown below in the tab "Equivalent 1". By design, this adds a dot to *all* operators and functions within an expression, guaranteeing loop fusion. An alternative is to express the operation through a *scalar* function, which we then broadcast. This is presented below in the tab "Equivalent 2".

```
x      = rand(100)

foo(x) = x .* 2 .+ x .* 3
```

```
julia> @btime foo($x)
  28.810 ns (2 allocations: 928 bytes)
```

```
x      = rand(100)

foo(x) = @. x * 2 + x * 3
```

```
julia> @btime foo($x)
  28.215 ns (2 allocations: 928 bytes)
```

```
x      = rand(100)

foo(a) = a * 2 + a * 3
```

```
julia> @btime foo.($x)
  27.936 ns (2 allocations: 928 bytes)
```

LAZY BROADCASTING

Broadcasting results in memory allocations since the technique is eager by default. This property means that its output becomes readily available when a broadcast expression is evaluated. Considering this, another way to achieve loop fusion is by evaluating all broadcast sub-expressions lazily, until the final output is computed. The approach is similar in spirit to the use of helper functions to accomplish the same goal, but without cluttering the namespace with new functions.

The functionality is provided by the macro `@~` from the `LazyArrays` package. By prepending this macro to the broadcasting operation, its computation is deferred until its output is required.

```julia
x         = rand(100)

function foo(x)
    term1  = x .* 2
    term2  = x .* 3

    output = term1 .+ term2
end
```

```
julia> @btime foo($x)
  90.883 ns (6 allocations: 2.719 KiB)
```

```julia
x         = rand(100)

function foo(x)
    term1  = @~ x .* 2
    term2  = @~ x .* 3

    output = term1 .+ term2
end
```

```
julia> @btime foo($x)
  26.362 ns (2 allocations: 928 bytes)
```

```julia
x         = rand(100)

term1(a)  = a * 2
term2(a)  = a * 3

foo(a)    = term1(a) + term2(a)
```

```
julia> @btime foo($x)
  28.451 ns (2 allocations: 928 bytes)
```

Beyond this resemblance with an approach based on functions, lazy broadcasting additionally addresses certain scenarios that functions can't handle. To understand this, note that the operations explored thus far resulted in a *vector* output. In those cases, memory allocations could at best be reduced to a single unavoidable allocation, necessary for storing the final output.

When the final output is instead given by a scalar, as occurs with reductions, lazy broadcasting is capable of entirely eliminating memory allocations. The reason is that lazy broadcasting fuses with reduction operations, letting the compiler apply a non-allocating procedure. This is illustrated in the example below.

```
# eager broadcasting (default)
x      = rand(100)

foo(x) = sum(2 .* x)
```

```
julia> @btime foo($x)
  42.069 ns (2 allocations: 928 bytes)
```

```
using LazyArrays
x      = rand(100)

foo(x) = sum(@~ 2 .* x)
```

```
julia> @btime foo($x)
  7.360 ns (0 allocations: 0 bytes)
```

Lazy Broadcasting May Be Faster Than Other Lazy Alternatives

An additional advantage of `@~` is that it implements extra optimizations. This explains why `@~` tends to be faster than alternatives like a lazy map, even though neither allocates memory. This performance benefit can be noticed 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)
  10.940 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)
  29.170 ns (0 allocations: 0 bytes)
```

9h. Pre-Allocations

Martin Alfaro
PhD in Economics

INTRODUCTION

For-loops may involve the creation of new vectors during each iteration, resulting in repeated memory allocation. This dynamic allocation may be unnecessary, particularly if these vectors hold temporary intermediate results that don't need to be preserved for future use. In such situations, performance can be improved through the use of a technique known as pre-allocation.

Pre-allocation involves initializing a vector to store temporary results before the for-loop begins execution, which is then reused during each iteration. By allocating memory upfront and modifying it in place, the overhead associated with repeated vector creation is effectively bypassed.

The performance gains from pre-allocation can be substantial. Remarkably, this technique isn't exclusive to Julia, but rather represents an optimization strategy applicable across programming languages. Its effectiveness ultimately stems from prioritizing mutations over the creation of new objects, thereby minimizing assignments on the heap.

Our presentation begins with a review of methods for initializing vectors, which is a prerequisite for implementing a pre-allocation strategy. We then present two scenarios where pre-allocation proves advantageous, with special emphasis on its advantages within nested for-loops.

INITIALIZING VECTORS

> **Remark**
> The review of methods for vector initialization will be relatively brief and centered on performance considerations. For a more detailed review, see the section about vector initialization, as well as the sections on in-place assignments and in-place functions.

Vector initialization refers to the process of creating a vector that subsequently will be filled with values. The process typically involves two steps: reserving space in memory, and populating that space with some initial values. An efficient approach to initializing a vector involves performing only the first step, keeping whatever content is in the memory address. These values held in memory are referred to as `undef` (undefined). Although Julia will display them as numerical values, they're essentially arbitrary and meaningless.

There are two methods for initializing a vector with `undef` values. The first one is through a constructor, requiring the specification of length and element types. 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`. This approach is particularly useful when the output sought matches the structure of an input.

Below, we compare the performance of approaches to initializing a vector. In particular, we establish that working with `undef` values is faster than populating vectors with specific values. To starkly show the differences in execution time, we repeat the process of vector creation 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)
  2.005 ms (100000 allocations: 85.449 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)
  2.062 ms (100000 allocations: 85.449 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)
  9.002 ms (100000 allocations: 85.449 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)
  9.764 ms (100000 allocations: 85.449 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)
  8.967 ms (100000 allocations: 85.449 MiB)
```

> **Remark**
>
> Recall that `_` isn't a keyword, but rather a convention widely adopted by programmers to denote **dummy variables**. These are variables included solely to meet syntactical requirements, but are never actually used or referenced within the code. In our example, the inclusion of `_` is because for-loops must always include a variable to iterate over. While any other symbol could be used, `_` signals programmers that its value can be safely ignored.

INITIALIZING VECTORS IN FUNCTIONS

We can initialize a vector by passing it to the function as a keyword argument. This even enables the use of `similar(x)` with `x` being a previous function argument. Considering this, the following two implementations are equivalent.

```
function foo(x)
   output = similar(x)
   # <some calculations using 'output'>
end
```

```
function foo(x; output = similar(x))

   # <some calculations using 'output'>
end
```

When multiple variables of the same type need to be initialized, array comprehensions offer a concise solution. Nonetheless, a more efficient alternative is based on the so-called generators, which are the lazy counterpart of array comprehensions. This reduces memory allocations by initializing and assigning the vectors element-wise, while an array comprehension first creates a vector holding all the initialized vectors.

```
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)
  54.235 ns (8 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)
  21.595 ns (3 allocations: 144 bytes)
```

Although the example uses `similar(x)`, note that the same principle applies to other initialization methods, such as `Vector{Float64}(undef, length(x))`.

DESCRIBING THE TECHNIQUE

To describe how pre-allocation works, we'll consider a typical scenario where it proves to be advantageous. This setup involves a **nested for-loop**, in which the output of a for-loop serves as an intermediate input for another for-loop. The example will rely on explicit for-loops, but also on constructs that internally compute the operation via a for-loop (e.g., broadcasting).

Let's first describe the inner for-loop that will be eventually embedded in an outer for-loop. Suppose we're assessing a worker's performance over a 30-day period, with daily scores recorded on a scale from 0 to 1. Defining successful performance as any score above 0.5, the following code snippets generate vectors indicating the days on which the goal was achieved.

```
nr_days            = 30
score              = rand(nr_days)

performance(score) = score .> 0.5
```

```
julia> @btime performance($score)
  29.186 ns (3 allocations: 96 bytes)
```

```
nr_days            = 30
score              = rand(nr_days)

function performance(score)
    target = similar(score)

    for i in eachindex(score)
        target[i] = score[i] > 0.5
    end

    return target
end
```

```
julia> @btime performance($score)
  24.548 ns (2 allocations: 304 bytes)
```

```
nr_days            = 30
score              = rand(nr_days)

function performance(score; target=similar(score))


    for i in eachindex(score)
        target[i] = score[i] > 0.5
    end

    return target
end
```

```
julia> @btime performance($score)
  26.479 ns (2 allocations: 304 bytes)
```

Now consider that the output of this operation (namely, the vector of days in which the target was met) represents an intermediate step in another for-loop. For instance, suppose we have multiple workers, each with recorded performance scores. The goal is to summarize each worker's information through a summary statistic. In particular, we consider the ratio of the standard deviation to the mean, which expresses variability in units of the average.

In practice, this statistic requires computing both the mean and the standard deviation. Below, we show that computing each statistic via reductions isn't efficient. The reason is that, while reductions avoid memory allocations, they require computing the days on which the target was met twice.

Instead, it's more efficient to compute and store this vector, which we then reuse as an input for computing the mean and the standard deviation.

The result highlights a more general principle: when the same intermediate input is required for multiple outputs, the cost of allocating memory for the intermediate vector tends to be lower than its repeated computation. In the example, the reduction is implemented via lazy broadcasting, which internally relies on reduction techniques. The rest implement the result via explicit or implicit for-loops, with the intermediate input stored in a vector.

```
nr_days            = 30
scores             = [rand(nr_days), rand(nr_days), rand(nr_days)]  # 3 workers



function repeated_call(scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)

      stats[col] = std(@~ scores[col] .> 0.5) / mean(@~ scores[col] .> 0.5)
    end

    return stats
end
```

```
julia> @btime repeated_call($scores)
  364.949 ns (2 allocations: 80 bytes)
```

```
nr_days            = 30
scores             = [rand(nr_days), rand(nr_days), rand(nr_days)]  # 3 workers

performance(score) = score .> 0.5

function repeated_call(scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
      target     = performance(scores[col])
      stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```
julia> @btime repeated_call($scores)
  274.988 ns (11 allocations: 368 bytes)
```

```julia
nr_days            = 30
scores             = [rand(nr_days), rand(nr_days), rand(nr_days)]   # 3 workers

function performance(score)
    target = similar(score)

    for i in eachindex(score)
        target[i] = score[i] > 0.5
    end

    return target
end

function repeated_call(scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
      target     = performance(scores[col])
      stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```
julia> @btime repeated_call($scores)
  197.941 ns (8 allocations: 992 bytes)
```

```julia
nr_days            = 30
scores             = [rand(nr_days), rand(nr_days), rand(nr_days)]   # 3 workers

function performance(score; target = similar(score))


    for i in eachindex(score)
        target[i] = score[i] > 0.5
    end

    return target
end

function repeated_call(scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
      target     = performance(scores[col])
      stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```
julia> @btime repeated_call($scores)
  188.021 ns (8 allocations: 992 bytes)
```

Once we establish that storing the intermediate vector is more performant, our approach will necessarily involve memory allocation. However, the methods described above allocate more than required: during each iteration, a new vector is created for each worker to store the days in which the target was met. This intermediate vector doesn't need to be preserved for future use. Indeed, because it is stored in a local variable, it is discarded once the iteration concludes.

Such cases are strong candidates for pre-allocating intermediate results. By defining a single vector `target`, we can reuse it across iterations to compute the days when the target was met for each worker. This strategy incurs the overhead of creating the vector only once, after which it is reused for every worker.

The implementation requires defining the vector `target` before the execution of the outer for-loop. During each iteration, we then mutate target using an in-place function, which we call `performance!`. This function can update the contents either with a standard for-loop or by applying the broadcasting operator `.=`.

```
nr_days = 30
scores  = [rand(nr_days), rand(nr_days), rand(nr_days)]


performance(score) = score .> 0.5

function repeated_call(scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
        target     = performance(scores[col])
        stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```
julia> @btime repeated_call($scores)
  259.300 ns (11 allocations: 368 bytes)
```

```
nr_days = 30
scores  = [rand(nr_days), rand(nr_days), rand(nr_days)]
target  = similar(scores[1])

performance!(target, score) = (@. target = score > 0.5)

function repeated_call!(target, scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
        performance!(target, scores[col])
        stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```
julia> @btime repeated_call!($target, $scores)
  193.557 ns (2 allocations: 80 bytes)
```

```
nr_days = 30
scores  = [rand(nr_days), rand(nr_days), rand(nr_days)]
target  = similar(scores[1])

function performance!(target, score)
    for i in eachindex(score)
        target[i] = score[i] > 0.5
    end
end

function repeated_call!(target, scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
        performance!(target, scores[col])
        stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```
julia> @btime repeated_call!($target, $scores)
  171.647 ns (2 allocations: 80 bytes)
```

Warning! - Use of @. to update values

When your goal is to update the 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
```

Compared to a for-loop, the method using `.=` provides a simpler syntax. This is why, when the function `performance!` is simple enough as in our example, it's common to directly express the updates via broadcasting inside the inner for-loop. This possibility also enables implementing the update via a built-in in-place function. For instance, the function `map!` can be used with this purpose.

```
nr_days = 30
scores  = [rand(nr_days), rand(nr_days), rand(nr_days)]


function repeated_call(scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
        target     = @. score > 0.5
        stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```
julia> @btime repeated_call($scores)
  431.740 ns (23 allocations: 608 bytes)
```

```
nr_days = 30
scores  = [rand(nr_days), rand(nr_days), rand(nr_days)]
target  = similar(scores[1])

function repeated_call!(target, scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
        @. target  = scores[col] > 0.5
        stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```
julia> @btime repeated_call!($target, $scores)
  190.480 ns (2 allocations: 80 bytes)
```

```julia
nr_days = 30
scores  = [rand(nr_days), rand(nr_days), rand(nr_days)]
target  = similar(scores[1])

function repeated_call!(target, scores)
    stats = Vector{Float64}(undef, length(scores))

    for col in eachindex(scores)
        map!(a -> a > 0.5, target, scores[col])
        stats[col] = std(target) / mean(target)
    end

    return stats
end
```

```
julia> @btime repeated_call!($target, $scores)
  167.516 ns (2 allocations: 80 bytes)
```

9i. Static Vectors for Small Collections

Martin Alfaro
PhD in Economics

INTRODUCTION

The creation of new vectors can easily become a performance bottleneck due to its memory-allocation overhead. The issue has far-reaching implications, since vector creation doesn't just occur when a vector is explicitly defined. It also happens internally in various scenarios, such as when referencing a slice like `x[1:2]` in `sum(x[1:2])` or when computing intermediate results on the fly like `x .* y` in `sum(x .* y)`.

The current section introduces a strategy to address this issue, while retaining the convenience of vectors for expressing collections. It leverages the so-called **static vectors**, provided by the `StaticArrays` package. Unlike built-in vectors, which are allocated on the heap, static vectors are either stack-allocated or stored in CPU registers.

Under the hood, static vectors are built on top of tuples. This determines that **static vectors are only suitable for collections comprising a few elements**. As a rule of thumb, **you should use static vectors for collections with up to 75 elements**. Exceeding this threshold can lead to increased overhead, 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 typically degrade. Secondly, their manipulation is closer to regular vectors, making them more convenient to work with.

The `StaticArrays` package also supports other array types (including matrices) and have mutable variants. The latter make static vectors more flexible than tuples, which are only available in an immutable form. It's worth indicating though that, while the mutable version provides performance benefits relative to regular vectors, the immutable option still offers the best performance.

> **Warning!**
> To avoid repetition, **this entire section assumes all collections are small**. Given this, all the benefits highlighted are contingent upon this assumption. We also suppose that the `StaticArrays` package has been loaded by executing `using StaticArrays`, thus omitting this command from each code snippet.

CREATING STATIC VECTORS

The package `StaticArrays` includes several variants of static arrays. Our primary focus is in particular on the type `SVector`, whose objects will be referred to as SVectors for simplicity.

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 converts a standard vector into an SVector. The implementation of each approach is 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 that the use of splatting operator `...` is necessary when a regular vector must be turned into an SVector. This operator transforms a collection into a sequence of arguments. Thus, `foo(x...)` is equivalent to `foo(x[1], x[2], x[3])` given a vector or tuple `x` with 3 elements.

Regarding slices of SVectors, the **method used for indexing determines whether the resulting slice is a regular vector or an SVector**: a slice remains an SVector when indices are provided as SVectors, whereas the slice becomes a regular vector when indices are 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.

```julia
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
```

```julia
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 implemented on top of tuples. Consequently, SVectors don't allocate memory.

```julia
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)
```

```julia
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)
```

```julia
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)
```

```julia
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.

```julia
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 means that, even when operations on regular vectors don't allocate memory, SVectors could still deliver significant speedups. Below, we demonstrate this through the function `sum(f, <vector>)`, which adds all elements of `<vector>` after they're transformed via `f`. The example shows that the implementation with SVectors yields faster execution times, despite that 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 their elements can't be added, removed, or modified. To accommodate mutable collections, the package `StaticArrays` additionally provides a variant given by the `MVector` type. The creation of MVectors and their slices follow the same syntax as SVectors, but with the function `SVector` replaced 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.

To ensure type stability, you can employ approaches similar to those employed for tuples. Essentially, this entails 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, bear in mind that they're never more performant than SVectors. In fact, certain operations on MVectors may still trigger memory allocations. For this reason, the general guideline is to prefer SVectors when the collection doesn't need to be mutated, restricting MVectors when in-place mutation is necessary.

Below, we compare the performance of SVectors and MVectors. The examples demonstrate that they may exhibit similar performance, although SVectors are more performant in certain cases. Likewise, SVectors and MVectors consistently outperform regular vectors for small collections.

```
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, let's compare their performance to other strategies that decrease 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, the examples reveal that storing the final output in an MVector can lead to performance gains over using a regular vector.

For the illustration, 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 stark, we'll 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 the implementations below 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)
```

```julia
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)
```

```julia
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 a reference point for comparison with 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 the latter approach is more performant, as it avoids memory allocation and benefits from additional optimizations provided by SVectors.

As for the last two tabs, they continue to define `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, we observe that MVectors deliver additional performance gains.

FOOTNOTES

[1.] The recommended number of elements provided 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 `@SVector` requires some caveats. For instance, it doesn't support definition of SVectors based on local variables. In particular, it precludes the use of `eachindex(x)` within an array comprehension, unless `x` is a global variable.

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 general strategies, particularly type stability and memory allocations. This chapter transitions to **specialized optimization techniques**.

While the analysis will center on SIMD optimizations, there are two important lessons to carry forward from this chapter.

1. **Inherent Trade-offs**. Once type stability is ensured and memory allocations are reduced, further speed gains almost always require a compromise. These trade-offs typically involve sacrificing *precision* (accepting a less accurate result for a faster calculation), *safety* (bypassing safeguards that prevent errors), or *generality* (writing code that is highly specific to one problem). This is precisely why such techniques aren't applied by default in Julia, which prioritizes correctness and safety above all.
2. **Automated Optimization with Macros**. When the need for speed makes these trade-offs worthwhile, macros provide an elegant way to implement complex strategies. They allow developers to package sophisticated optimization algorithms into simple reusable tools. This makes advanced optimization highly accessible, as users can apply these techniques without the need to understand the underlying intricacies of their implementation.

10b. Macros as a Means for Optimizations

Martin Alfaro
PhD in Economics

INTRODUCTION

Customized 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 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 strategy. In this way, practitioners benefit from specialized algorithms, without having to write the implementation 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.

USES OF MACROS

Macros can be easily identified by the `@` symbol preceding their name. They resemble functions in that they take input and produce output. Their primary difference lies in what they operate on and what they can return. Specifically, *functions operate on values*: they take evaluated expressions as arguments and return a final computed value. In contrast, *macros operate on code*: they take expressions as input and return a new transformed expression, which is then compiled and executed in place of the original macro call. This unique feature enables macros to handle tasks that functions can't. Two notable applications are worth mentioning.

First, macros can be used **to simplify code**. By automating repetitive tasks and eliminating boilerplate, macros can make code significantly more readable and maintainable. 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. This is demonstrated below.

```julia
x = rand(1_000)

function foo(x)
    x1 = view(x, x .> 0.7)
    x2 = view(x, x .< 0.5)
    x3 = view(x, 1:500)
    x4 = view(x, 501:1_000)

    x1, x2, x3, x4
end
```

```julia
x = rand(1_000)

@views function foo(x)
    x1 = x[x .> 0.7]
    x2 = x[x .< 0.5]
    x3 = x[1:500]
    x4 = x[501:1_000]

    x1, x2, x3, x4
end
```

A second application of macros is **to modify how operations are computed**, which is the focus of the current section. This lets developers bundle sophisticated optimization techniques, making advanced solutions accessible. As a result, users unfamiliar with a method's underlying complexity can focus on choosing the most suitable computational approach, rather than grappling with the implementation details.

While macros are powerful tools, they're not without their limitations. Their black-box nature means that **misuse of macros can lead to unexpected results or compromise computational safety**. That's why it's crucial to identify the right scenarios for each macro. Although this requires some upfront work, it's considerably less demanding than implementing the functionality from scratch.

MACROS APPLIED IN FOR-LOOPS

One distinctive feature of Julia is its ability to execute for-loops with exceptional speed. In fact, carefully optimized for-loops can reach peak 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 customize how for-loops are computed.

To illustrate this application, let's consider the `@inbounds` macro. To understand what this macro accomplishes, we first need to understand how for-loops 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: the additional checks not only introduce computational overhead, but also limit the compiler's ability to implement certain optimizations. Nonetheless, in situations where iterations are guaranteed to stay within an array's bounds, these safety checks become redundant. Consequently, we can safely boost performance by disabling bounds checking with 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.
>
> Despite this, 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.

@INBOUNDS AS AN EXAMPLE

Using a macro that affects the whole structure of the for-loop requires its inclusion at the beginning. For instance, to disable bounds checking for every indexed element within a for-loop, we simply need to prepend the for-loop with `@inbounds`.

```julia
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)
  5.826 µs (0 allocations: 0 bytes)
```

> **Alternative Application of @inbounds**
>
> We can alternatively apply `@inbounds` individually to any specific line within the for-loop. Nonetheless, this possibility is specific to macros like `@inbounds`, which aren't exclusive to for-loops.

```julia
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)
  5.938 μs (0 allocations: 0 bytes)
```

The performance advantages of `@inbounds` come not only from eliminating bounds checking itself, but also from giving the compiler more freedom to implement additional optimizations.

To understand this, note that bounds checking is essentially a conditional statement, where the iteration is executed only if all indices are within range. As we'll see in the next sections, conditional statements limit the compiler's ability to apply the so-called SIMD instructions, which are a form of parallelism within a single core.

MACROS COULD BE APPLIED AUTOMATICALLY OR BE DISREGARDED BY THE COMPILER

The influence of a macro on code execution isn't always predictable. In many cases, it might have no impact at all because the compiler has the final say on the optimization strategy. Thus, it might already be applying the optimization suggested or determine that the macro's recommendation is unhelpful and simply ignore it. In either case, you can infer a macro is discarded when there's no significant change in execution time after applying it. [1]

Both scenarios can arise with `@inbounds` as we show below.

REDUNDANT MACROS

The compiler could prove on its own that a for-loop is safe and therefore disable bounds checking. In those cases, `@inbounds` becomes redundant. This behavior typically occurs in simple cases, such as when iterating over a single collection `x` and using `eachindex(x)`.

```
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)
  3.384 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)
  3.228 μs (0 allocations: 0 bytes)
```

DISREGARDED MACRO

A macro can also be treated as a mere hint that the compiler is free to disregard. In such cases, a macro signals that the necessary conditions for a particular optimization are satisfied, allowing the compiler to consider more aggressive strategies. The compiler will then carefully analyze the operations and decide if the suggested approach is actually beneficial. This fact highlights how macros can guide the compiler toward better performance, without imposing strict directives.

A prime example along these lines is the `@simd` macro. This suggests the application of SIMD instructions, a subject that will be explored in upcoming sections. When `@simd` is added to a for-loop, the compiler retains full autonomy in deciding whether to implement the suggested optimization. In the example below, the compiler concludes that SIMD would likely degrade performance, thus ignoring `@simd`. This explains why the execution time remains the same with and without the macro. [2]

```julia
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)
  881.929 μs (3 allocations: 15.259 MiB)
```

```julia
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)
  863.776 μs (3 allocations: 15.259 MiB)
```

FOOTNOTES

[1.] One can verify that macros are indeed ignored by examining the generated machine code. Because inspecting machine code is beyond the scope of this website, we'll instead use the runtime behavior as an indicator.

[2.] It's possible to confirm that the generated machine code is identical by inspecting the compiler's implemented code.

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, rather than sequentially processing them one by one. This parallel approach can yield substantial performance gains, especially for workloads involving simple identical calculations repeated across multiple data elements. [1]

The efficiency of SIMD lies in 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 points simultaneously. This is why the process of applying SIMD is often referred to as **vectorization**.

To illustrate, consider a computation involving four separate addition operations. Without SIMD, the computer would need to execute four distinct instructions, one for each addition. Instead, SIMD makes it possible to bundle the four additions into a single instruction, allowing the CPU to process them all at once. In an ideal scenario, the time required to complete four additions with SIMD would be the same as completing one addition without it.

Throughout the different sections, we'll cover two approaches for implementing SIMD instructions.

- Julia's native capabilities.
- The package `LoopVectorization`.

This section will exclusively concentrate on the built-in tools for applying SIMD. In particular, we'll explore the conditions that trigger automatic vectorization. The next section we'll introduce the `@simd` macro, which lets you manually apply it in for-loops. The discussion of `LoopVectorization` will be saved for later sections. Relative to Julia's built-in tools, this package often implements more aggressive optimizations, but can also introduce bugs if misused.

WHAT IS SIMD?

SIMD is a type of data-level parallelism that occurs within a single processor core, applying the same operation to multiple data elements at once. It's particularly effective for basic arithmetic operations, such as addition and multiplication, when the same operation must be applied to multiple data

elements. Given the nature of these operations, it's unsurprising that one of SIMD's primary applications is in linear algebra, where operations like matrix multiplication involve applying identical arithmetic steps to multiple elements.

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 operation, modern processors include specialized SIMD registers designed for this purpose. Today's processors typically feature 256-bit registers for vectorized operations, which are wide enough to hold four values of either `Float64` or `Int64`.

To illustrate the workings of SIMD, consider the task of adding two vectors, `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: `1+10`, `2+20`, `3+30`, `4+40` executed sequentially, thereby producing the result `[11, 22, 33, 44]` in four steps. In contrast, all four additions can be performed with a single instruction under SIMD in one step. The processor loads all four elements of `x` and `y` into the 256-bit register, and then runs a single "sum" instruction to compute all four additions simultaneously.

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 within the register's capacity. After this, the processor computes all the operations within each sub-vector simultaneously, repeatedly applying the same procedure for every sub-vector.

BROADCASTING AND FOR-LOOPS

Based on the previous discussion, we can identify **two types of operations that can potentially benefit from SIMD instructions: for-loops and broadcasting**. The latter operation is included since broadcasting is essentially syntactic sugar for for-loops.

The decision to apply SIMD instructions in its built-in form, nonetheless, is made entirely by the compiler. This relies on a set of heuristics to determine when their use will pay off. In the case of broadcasting, the compiler implements SIMD automatically, without any user intervention. Instead, the automatic application of SIMD in for-loops is only restricted to a few simple cases, delegating to the user the suggestion of whether SIMD should be implemented. This is why the upcoming sections

will identify conditions under which SIMD instructions should be suggested to the compiler. If these conditions aren't met, SIMD will substantially reduce its effectiveness or directly become infeasible. Given this, we'll also provide guidance on how to handle scenarios that aren't well-suited for SIMD.

SIMD IN BROADCASTING

To entirely shift our attention to for-loops in subsequent sections, we conclude by illustrating the automatic application of SIMD in broadcasting.

The following example demonstrates this. It compares the same computation implemented via a for loop and via broadcasting. While broadcasting automatically takes advantage of SIMD, this isn't necessarily true for for-loops (in fact it's not in this particular case).

```
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)
  788.201 μs (3 allocations: 7.629 MiB)
```

```
x      = rand(1_000_000)

foo(x) = 2 ./ x
```

```
julia> @btime foo($x)
  411.572 μs (3 allocations: 7.629 MiB)
```

FOOTNOTES

[1.] SIMD isn't exclusive to CPUs, with GPUs also taking advantage of it. The architecture of GPUs is a natural fit for SIMD, as they were designed 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 degrades performance or not applied when it would've been advantageous. To address this, Julia provides the `@simd` macro to suggest 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 slow 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. Macros tend to 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 beneficial.

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 `foo` to each element of a vector `x`.

In the following, we illustrate this case via a polynomial transformation of `x`. The transformation will be done via a for-loop 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.

```julia
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)
  783.380 μs (3 allocations: 7.629 MiB)
```

```julia
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)
  404.834 μs (3 allocations: 7.629 MiB)
```

```julia
x = rand(1_000_000)

foo(x) = @. x / 2 + x^2 / 3
```

```
julia> @btime foo($x)
  402.954 μs (3 allocations: 7.629 MiB)
```

A SPECIAL CASE OF DEPENDENCE: REDUCTIONS

SIMD requires independence of iterations. One exception to this rule is given by reductions, which have been carefully designed for their proper handling.

For reductions involving integers, Julia leverages SIMD automatically. Instead, reductions with floating-point numbers necessitate the explicit addition of the `@simd` macro. The following example demonstrates this behavior.

Starting from the case of integers, we can see 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)
  3.164 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)
  2.835 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)
  4.880 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)
  2.805 ms (0 allocations: 0 bytes)
```

Why Floating Points Are Treated Differently?

Addition of floating-point numbers doesn't obey associativity, unlike what occurs with integers: due to the inherent imprecision of floating-point arithmetic, `(x+y)+z` may differ from `x+(y+z)`. This loss of associativity illustrates a broader point: floating-point numbers are finite-precision approximations, and therefore they don't always satisfy the mathematical properties that hold for real numbers.

The following code snippets illustrate this point.

```
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 don't rely on a specific order of operations.

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: Contiguous Access and Unit Strides

Martin Alfaro
PhD in Economics

INTRODUCTION

This section contrasts the performance of copies and views, emphasizing that copies guarantee conditions favorable to SIMD execution.

To understand why this is, recall that 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 4 integers). These registers 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 must adhere to specific organizational and access rules in memory**. The two core requirements are contiguous memory layout and unit-stride access.

- **Contiguous Memory Layout**: this means that data elements reside in adjacent memory addresses with no gaps. Freshly allocated arrays meet this requirement, enabling entire segments to load directly into vector registers. In contrast, array views don't guarantee contiguity. By referencing the original data structure, views can result in highly irregular memory access patterns that jump between non-adjacent addresses.

- **Unit Strides**: strides refer to the step size between consecutive memory accesses. Unit strides means in particular that elements are accessed sequentially. For example, iterating through a freshly allocated vector `x` using `eachindex(x)` ensures unit stride: each access moves to the next adjacent address 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).

The fact that SIMD performs best when these two conditions hold creates a key trade-off for developers. On the one hand, using views reduces memory allocations, but makes SIMD less effective as it often violates contiguity or unit stride. Instead, copies create new contiguous arrays that ensure the effectiveness of SIMD, but at the expense of increase memory usage. The current section explores this trade-off.

REVIEW OF INDEXING APPROACHES

The performance trade-offs between copies and views depend on the slicing method employed. For this reason, we begin with a brief overview of the main slicing techniques.

```
x         = [10, 20, 30]

indices   = sortperm(x)
elements  = x[indices]    # equivalent to `sort(x)`
```

```
julia> indices
3-element Vector{Int64}:
 1
 2
 3
julia> 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> indices
3-element BitVector:
 1
 0
 1
julia> elements
2-element Vector{Int64}:
 20
 30
```

Vector Indexing works by referencing elements directly through their indices. **Ranges** are simply as a special form of vector indexing that lazily references consecutive elements according to a specified stride. The general syntax is `<first index>:<stride>:<last index>`, where omitting `<stride>` defaults to a step size of 1.

The function `sortperm` is incorporated as it'll starkly illustrate the difference between copies and views. Given some collection `x`, `sortperm(x)` returns the corresponding indices of `sort(x)`. Assuming that `x` hasn't been ordered previously, views that access elements via the indices of `sortperm(x)` will result in an irregular memory access pattern. For instance, given `x = [5, 4, 6]`, we get that `sort(x)` returns `[4, 5, 6]` and `sortperm(x)` provides `[2, 1, 3]`. Therefore, accessing elements of `x` via `sortperm(x)` will involve jumping around within the underlying data.

For its part, **Boolean indexing** returns a Boolean vector, with `1` indicating the element must be kept. This approach is primarily employed for the creation of slices based on broadcast conditions.

BENEFITS OF SEQUENTIAL ACCESS

In the section on reducing memory allocations, we highlighted the benefits of using views over copies when working with slices: by maintaining references to the original data, views avoid memory allocation. Yet, we briefly remarked that copies could outperform views in some scenarios. We're now in a position to explain in more depth why this occurs.

Creating a copy of some data structure involves allocating the information in a new contiguous block of memory. This layout ensures that elements are stored sequentially, thus offering two key advantages: faster element retrieval and a more effective use of SIMD instructions. Views can't guarantee this property. If the referenced elements are scattered in memory, views will necessarily introduce irregular access patterns.

An analogy may help clarify this. Imagine retrieving books from a library. If every book you need are neatly arranged 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. The analogy extends further: if you also have a cart that allows you to carry multiple books at once, the process becomes even more efficient. SIMD operations play the role of this cart.

ILLUSTRATING EACH BENEFIT IN ISOLATION

The following examples illustrate the potential advantages of copies compared to views. To ensure that the memory allocations of copies aren't distorting the results, slices will be created outside the benchmarked function,

We start with a scenario where views access elements following a pattern, but without this being characterized by unit stride.

```julia
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)
  192.755 µs (0 allocations: 0 bytes)
```

```julia
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)
  80.770 µs (0 allocations: 0 bytes)
```

```julia
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)
  200.414 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)
  42.108 μs (0 allocations: 0 bytes)
```

In some cases, the access of elements doesn't follow any predictable pattern. Such a behavior can be illustrated by accessing elements via `sortperm`.

```
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)
  20.596 ms (0 allocations: 0 bytes)
```

```julia
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)
  19.974 ms (0 allocations: 0 bytes)
```

```julia
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)
  2.250 ms (0 allocations: 0 bytes)
```

```julia
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)
  956.818 µs (0 allocations: 0 bytes)
```

A similar issue occurs with 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)
  2.230 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)
  1.791 ms (0 allocations: 0 bytes)
```

```julia
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)
  966.743 µs (0 allocations: 0 bytes)
```

```julia
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)
  204.331 µs (0 allocations: 0 bytes)
```

COPIES VS VIEWS: TOTAL EFFECTS

The previous examples defined slices outside the benchmarked functions, thus avoiding the memory allocations of copies. The goal was to emphasize the benefits of contiguous access and unit strides in isolation. However, the choice between copies and views requires incorporating the overhead of additional memory allocations. Overall, we must weigh these costs against the performance benefits of sequential memory accesses.

In general, benchmarking is the only way to decide whether copies or views deliver better performance. For instance, below we present a case where views are preferred, since operation executed is straightforward enough to benefit from SIMD.

```
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)
  22.736 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)
  34.126 ms (2 allocations: 38.148 MiB)
```

Instead, the following scenario features a more complex but SIMD-friendly operation, illustrating how copying can actually outperform using 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)
  249.382 ms (0 allocations: 0 bytes)
```

```julia
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)
  131.559 ms (2 allocations: 38.148 MiB)
```

SPECIAL CASES

Certain patterns allow us to predict the faster approach without exhaustive benchmarking.

One scenario where views always outperform copies is when the view is referencing sequential elements. These scenarios call for the use of view, as they provide the same benefits as copies but additionally avoiding memory allocations.

```julia
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)
  74.382 µs (0 allocations: 0 bytes)
```

```julia
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)
  81.146 μs (0 allocations: 0 bytes)
```

In contrast, a common scenario where copies tend to outpace views is when we perform multiple operations on the same slice. In this case, the cost of an additional memory allocation is usually offset by the speed gains from contiguous memory access. This is illustrated below.

```julia
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)
  252.034 ms (0 allocations: 0 bytes)
```

```julia
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)
  119.678 ms (2 allocations: 38.148 MiB)
```

10f. SIMD: Branchless Code

Martin Alfaro
PhD in Economics

INTRODUCTION

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 occurs because 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 in which the compiler introduces conditional statements as a byproduct 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. By this, we mean typical scenarios where conditions are explicitly introduced in the code, so that certain operations are only executed under specific circumstances. Here, we'll revisit the usual approaches to expressing conditions, focusing on their internal implementation. The goal is to outline the relative strengths and limitations of each approach, indicating which ones are more conducive to SIMD optimizations. Finally, we'll show that conditional statements can be equivalently recast as algebraic operations, effectively removing the branching logic that disrupts SIMD execution. In fact, the compiler may implement this strategy under the hood when it's likely beneficial.

TYPE INSTABILITY AND BOUNDS CHECKING AS CONDITIONS TO AVOID

Two patterns in Julia introduce internal branching: type-unstable functions and bounds checking during array indexing. This conditional operation arises from compiler decisions rather than explicit code, making them unnoticeable.

When a function is type-unstable, Julia generates multiple execution branches, one for each type. Those extra branches, while hidden from the user, still disrupt the uniform instruction flow required by SIMD. The remedies for this case are the same as those for fixing type instabilities, for which we devoted the whole chapter 8. Regardless of any SIMD consideration, it's worth remarking that type stability should always be a priority: any attempt to achieve high performance is nearly impossible without guaranteeing it.

A second source of hidden branching arises in for-loops, where Julia performs bounds checking by default. This operation represents a subtle form of conditional execution, where each iteration is executed only when indices remain within bounds. These checks interfere with the application of SIMD, since they prevent the compiler from treating the for-loop as a uniform sequence of operations.

The example below demonstrates two key insights regarding bounds checking. First, **adding `@inbounds` is a prerequisite for the application of SIMD**. Specifically, if bound checks remain in place, simply using `@simd` alone will rarely trigger the implementation of SIMD instructions. Second, merely adding `@inbounds` can be enough to induce the compiler to apply SIMD instructions, rendering `@simd` annotations redundant for performance improvements. This additionally explains why using `@inbounds @simd` may not speed up execution times. [1]

```
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)
  824.276 µs (3 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)
  1.109 ms (3 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)
  390.190 µs (3 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)
  393.380 µs (3 allocations: 7.629 MiB)
```

Overall, the main takeaway from the example is that when the goal is to exploit SIMD in a for-loop, the for-loop should be preceded by `@inbounds @simd`.

> **Broadcasting and For-Loops**
>
> Broadcasting in Julia automatically disables bounds checking and encourages the use of SIMD instructions. This often makes broadcasted expressions appear faster than a straightforward for-loop. However, broadcasting isn't fundamentally different regarding its mechanics. Essentially, broadcasting is a compact notation for implementing certain for-loops. For instance, a for-loop with `@inbounds` and `@simd` usually exhibits a similar performance to a broadcast variant. This is demonstrated below.
>
> ```
> x      = rand(1_000_000)
> foo(x) = 2 ./ x
> ```
>
> ```
> julia> @btime foo($x)
>  452.692 µs (3 allocations: 7.629 MiB)
> ```

```julia
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)
  802.428 μs (3 allocations: 7.629 MiB)
```

```julia
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)
  399.047 μs (3 allocations: 7.629 MiB)
```

APPROACHES TO CONDITIONAL STATEMENTS

When conditions are integral to a program's logical flow and can't be avoided, it becomes important to consider the most effective way to introduce them. The options in this regard must consider that 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. By contrast, an eager approach performs the computation immediately, regardless of whether `C` is satisfied. If the condition is eventually `false`, the result of the computation is then discarded.

When conditional statements are applied only once, a lazy approach is more performant as it avoids needless computations. However, when a condition is embedded inside a for-loop, there are multiple operations to be potentially computed. Since SIMD can compute more than one operation

simultaneously, 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 aspect 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` favors lazy computations, only evaluating the necessary components based on the truth value of the condition.

The following example demonstrates this computational difference between `if` and `ifelse`. It relies on a reduction operation, whose elements to be summed are contingent on a condition. Note that `ifelse` requires specifying an operation for both the true and false cases. For a sum reduction, the false case is handled by returning zero when the condition isn't met.

```julia
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)
  409.692 µs (0 allocations: 0 bytes)
```

```julia
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)
  406.976 μs (0 allocations: 0 bytes)
```

```julia
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)
  385.429 μs (0 allocations: 0 bytes)
```

```julia
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)
  89.666 μs (0 allocations: 0 bytes)
```

As the example reveals, the fact that `ifelse` is eager doesn't automatically trigger the application of SIMD. This is precisely why `@inbounds @simd` had to be included.

It's also worth remarking that applying SIMD instructions doesn't necessarily increase performance. The example below demonstrates this point.

```
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)
  18.720 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)
  16.518 ms (0 allocations: 0 bytes)
```

TERNARY OPERATORS

Ternary operators are an alternative approach for conditional statements. They have the form `<condition> ? <action if true> : <action if false>`. Unlike the previous methods, this form isn't committed to an eager or a lazy approach. Instead, it relies on heuristics to determine which approach should be implemented. The decision depends on the compiler's assessment about which strategy will likely be faster in the application considered.

For the illustrations, we'll consider examples where `@inbounds` and `@simd` are directly added in each approach. Based on the same example as above, the ternary operator could opt for 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)
  407.000 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)
  89.095 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)
  88.226 μs (0 allocations: 0 bytes)
```

Instead, the ternary operator opts for 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)
  393.501 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)
  3.609 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)
  413.706 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 example, the ternary operator actually implements the slower 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)
  26.299 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)
  17.272 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)
  25.943 ms (0 allocations: 0 bytes)
```

WHEN EACH APPROACH IS BETTER?

As a rule of thumb, conditional statements with **computational-demanding operations will more likely benefit from a lazy implementation**. In contrast, **an eager approach is potentially more performant when branches comprise simple algebraic computations**. In fact, these heuristics tend to drive the decision adopted by the ternary operator.

To demonstrate, the following example considers a conditional statement where there's only one branch with a computation and this is straightforward. An eager approach with SIMD is faster, and coincides with the approach chosen when a ternary operator is chosen.

```julia
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)
  399.915 μs (0 allocations: 0 bytes)
```

```julia
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)
  88.727 μs (0 allocations: 0 bytes)
```

```julia
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)
  89.035 μs (0 allocations: 0 bytes)
```

Instead, below we consider a branch with 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)
  12.655 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)
  12.023 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)
  13.124 ms (0 allocations: 0 bytes)
```

VECTORS WITH CONDITIONS

Next, we consider scenarios where you already have defined 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 previously 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 accounting for the type conversion.

```
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)
  3.098 ms (3 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)
  848.068 µs (5 allocations: 8.583 MiB)
```

No Vector With Conditions

The conclusions stated here assume the vector holding the conditions is already defined. If this isn't the case and you want to apply SIMD instructions, you should implement `ifelse` operating on scalars, without a vector of conditions. This allows you to avoid memory allocations, while still applying SIMD effectively. The following example illustrates this point. [2]

```
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)
  3.254 ms (7 allocations: 7.749 MiB)
```

```julia
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)
  704.903 μs (5 allocations: 8.583 MiB)
```

```julia
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)
  467.963 μs (3 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

$$1 - \prod_{i=1}^{n} (1 - c_i) = 1$$

Given two Boolean scalars `c1` and `c2`, these equivalences in Julia 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)
  2.063 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)
  865.019 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)
  2.917 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)
  868.655 µs (0 allocations: 0 bytes)
```

Instead, with broadcasting, these equivalences become:

```
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)
  5.223 ms (3 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)
  537.296 µs (3 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)
  3.248 ms (3 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)
  497.927 µs (3 allocations: 7.629 MiB)
```

FOOTNOTES

[1.] Recall that the compiler may automatically disable bounds checking in simple scenarios. For instance, this is the case when only `x` is indexed and `eachindex(x)` is employed as the iteration range. Instead, explicit use of `@inbounds` becomes necessary when we're indexing `x` and another vector, as in the examples below.

[2.] Note that the approach for `Vector{Bool}` is somewhat different to the examples 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. SIMD Packages

Martin Alfaro
PhD in Economics

INTRODUCTION

Up to this point, we've leaned on the built-in `@simd` macro to encourage vectorization in for-loops. This approach, nonetheless, exhibits several limitations.

The first limitation is control. `@simd` acts as a suggestion, rather than a strict command: it hints to the compiler that SIMD optimizations might improve performance, but the implementation decision is ultimately up to the compiler's discretion. Second, `@simd` is designed to be conservative, prioritizing code safety over speed. In practice, this means it won't implement many aggressive transformations that SIMD typically requires to unlock its full capabilities.

To overcome these shortcomings, we'll introduce the `@turbo` macro from the `LoopVectorization` package. Rather than merely suggesting vectorization, `@turbo` rewrites for-loops and broadcast expressions into a form that's explicitly structured for SIMD execution. That extra power comes with an important shift in responsibility: `@turbo` assumes that your code satisfies the conditions necessary for these transformations, placing the burden of safety and correctness on the user.

A further advantage of `LoopVectorization` is its integration with the `SLEEFPirates` package. This enables SIMD-accelerated implementations of common mathematical functions, including logarithms, exponentials, powers, and trigonometric operations.

CAVEATS ABOUT IMPROPER USE OF @TURBO

In contrast to `@simd`, applying `@turbo` demands particular care, as a misapplication can silently produce incorrect results. The risk stems from the additional assumptions that `@turbo` makes to enable more aggressive optimizations. In particular, `@turbo` operates under two key assumptions:

- **No out-of-bound access**: `@turbo` omits index bound checks, thus relying on that all indices are valid.
- **Iteration order invariance**: `@turbo` supposes that, aside from reductions, the outcome of the for-loop doesn't depend on the order in which iterations execute.

The second assumption is especially relevant. Even if your for-loop has no explicit dependency, floating-point arithmetic are still sensitive to reordering because it isn't associative. This causes results to depend on the iteration order. Integer operations, by contrast, are unaffected. The following

example illustrates the problem: because each iteration depends on the result of the previous one, applying `@turbo` violates the iteration-order invariance assumption and therefore yields incorrect results.

NO MACRO

```
x = [0.1, 0.2, 0.3]

function foo!(x)
    for i in 2:length(x)
        x[i] = x[i-1] + x[i]
    end
end
```

```
julia> foo!(x)

julia> x
3-element Vector{Float64}:
 0.1
 0.3
 0.6
```

@SIMD

```
x = [0.1, 0.2, 0.3]

function foo!(x)
    @inbounds @simd for i in 2:length(x)
        x[i] = x[i-1] + x[i]
    end
end
```

```
julia> foo!(x)

julia> x
3-element Vector{Float64}:
 0.1
 0.3
 0.6
```

@TURBO

```
x = [0.1, 0.2, 0.3]

function foo!(x)
    @turbo for i in 2:length(x)
        x[i] = x[i-1] + x[i]
    end
end
```

```
julia> foo!(x)

julia> x
3-element Vector{Float64}:
 0.1
 0.3
 0.5
```

Considering that `@turbo` isn't suitable for all operations, we next present cases where the macro can be safely applied.

SAFE APPLICATIONS OF @TURBO

Safe applications of `@turbo` fall into two broad categories: **embarrassingly parallel problems** and **reductions**.

In the first case, **iterations are completely independent**, making execution order irrelevant for the final outcome. The example below illustrates this case by performing an independent transformation on each element of a vector. We also show that the use of `@turbo` isn't restricted to for-loops, also allowing for broadcast operations.

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> @btime foo($x)
  3.840 ms (3 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> @btime foo($x)
  4.096 ms (3 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> @btime foo($x)
  271.104 μs (3 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> @btime foo($x)
  482.698 μs (3 allocations: 7.629 MiB)
```

The second safe application is **reductions**. While reductions introduce dependencies across 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> @btime foo($x)
  3.892 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> @btime foo($x)
  3.937 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> @btime foo($x)
  179.364 µs (0 allocations: 0 bytes)
```

SPECIAL FUNCTIONS

Another important application of `LoopVectorization` arises through its integration with the library *SLEEF* (an anocrym for "SIMD Library for Evaluating Elementary Functions"). This accelerates the evaluation of several mathematical functions, including the exponential, logarithmic, power, and trigonometric functions. SLEEF is exposed in `LoopVectorization` via the `SLEEFPirates` package,

Below, we illustrate the use of `@turbo` for each type of function. For a complete list of supported functions, see the SLEEFPirates documentation. All the examples rely on an element-wise transformation of `x` via the function `calculation`, which will take a different form depending on the function illustrated.

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> @btime foo($x)
  3.542 ms (3 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> @btime foo($x)
  3.546 ms (3 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> @btime foo($x)
  1.617 ms (3 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

```
x               = rand(1_000_000)
calculation(a) = log(a)

foo(x) = @turbo calculation.(x)
```

```
julia> @btime foo($x)
  1.618 ms (3 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> @btime foo($x)
  2.608 ms (3 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> @btime foo($x)
  2.639 ms (3 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> @btime foo($x)
  555.012 μs (3 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = exp(a)

foo(x) = @turbo calculation.(x)
```

```
julia> @btime foo($x)
  544.043 μs (3 allocations: 7.629 MiB)
```

POWER FUNCTIONS

This includes any operation comprising terms x^y.

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> @btime foo($x)
  3.517 ms (3 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> @btime foo($x)
  3.578 ms (3 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> @btime foo($x)
  371.218 µs (3 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = a^4

foo(x) = @turbo calculation.(x)
```

```
julia> @btime foo($x)
  302.605 µs (3 allocations: 7.629 MiB)
```

The implementation for power functions includes calls to other function, such as the one for 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> @btime foo($x)
  1.159 ms (3 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> @btime foo($x)
  1.200 ms (3 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> @btime foo($x)
  590.429 µs (3 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

```
x               = rand(1_000_000)
calculation(a) = sqrt(a)

foo(x) = @turbo calculation.(x)
```

```
julia> @btime foo($x)
  578.698 µs (3 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> @btime foo($x)
  3.915 ms (3 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> @btime foo($x)
  3.895 ms (3 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> @btime foo($x)
  1.341 ms (3 allocations: 7.629 MiB)
```

@TURBO (BROADCASTING)

```
x              = rand(1_000_000)
calculation(a) = sin(a)

foo(x)         = @turbo calculation.(x)
```

```
julia> @btime foo($x)
  1.315 ms (3 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 correct implementation of multithreading requires a basic understanding of how computers execute instructions. In particular, it's essential to understand the procedure in light of data dependencies, where one operation relies on the output of a previous one. If these dependencies aren't properly managed, multithreading can lead to incorrect results and introduce several forms of unsafe code.

This section introduces the preliminary concepts needed to reason about these issues, laying the foundation for the more practical discussions that follow. **The emphasis here is on explanation rather than implementation**. Accordingly, many of the macros and functions presented won't be reused elsewhere on this website.

NATURE OF COMPUTATIONS

Operations can be broadly classified based on their data dependency. 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. In contrast, an **independent operation** produces the same result, regardless of the order in which it's executed relative to other operations.

The following code gives rise to a dependent or independent operation, based on which values operation B sums.

```
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
```

Regardless of their dependency status, operations can be computed either sequentially or concurrently. A **sequential** procedure involves executing operations one after the other, ensuring each 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 design choice that prioritizes result correctness, as **concurrent execution of dependent operations can yield incorrect results if mishandled**. The issue arises because concurrency introduces non-determinism in the execution order, which can lead to timing inconsistencies when reading and writing data. A sequential approach precludes this possibility by guaranteeing a predictable order of execution.

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 the 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). Such misunderstandings can lead to flawed program design or incorrect handling of concurrent processes. To address this, we next revisit the 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. This is achieved through the concept of **tasks**. For a task to be computed, it must be assigned to **a computer thread**. Since a single task runs on exactly one thread at a time, **the number of threads available on your computer determines how many tasks can be computed simultaneously**.

Importantly, each session in Julia starts with a predefined pool of threads, defaulting to a single thread regardless of your computer's hardware. We'll begin by examining the single-threaded case, as it provides a clear basis 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. For its part, A's job consists of waiting for 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, simulated by calling the function `sleep(2)`.

The following code snippet defines functions capturing each worker's job.

```
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 merely create a blueprint for a set of operations. This implies that no computation is performed. It's only when we call these functions by lines like `job_A(2)` and `job_B(1)` that the operations are sent for execution.

To lay bare the internal steps Julia follows for computation, let's adopt a lower-level approach where the function calls `job_A(2)` and `job_B(1)` are defined as tasks. As shown below, tasks aren't abstractions to organize our discussion, but 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 a task is defined, the first step for its computation is **scheduling** it. This means the task is added to the queue of operations awaiting execution by the computer's processor. Then, as soon as a thread becomes available, the machine begins its computation.

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.

The implication of this statement is that true parallelism isn't feasible in single-threaded sessions. Nonetheless, concurrency can still improve efficiency through **task switching**. This is enabled by a **task-yielding** mechanism: when a task becomes idle (e.g., waiting for input or data), 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 run concurrently**. As a result, to enforce a sequential execution of tasks, we must instruct Julia to run them 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 with the rest of the program.

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 and after both tasks were scheduled. The purpose is to pause the main program flow until both scheduled tasks have finished, preventing subsequent code from executing prematurely.

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 task 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!

Notice, 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 chance 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
```

```julia
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
```

Overall, the key insight from this subsection is the underlying procedure outlined: **when a task is scheduled, the computer attempts to find an available thread to run it**. For concurrency, this implies that **starting a session with multiple threads enables parallel code execution**. This case is simply referred to as **multithreading** and explained next in more detail.

MULTITHREADING

Let's revisit the case where both workers A and B perform meaningful computations. The only change introduced is that Julia's session now starts with more than one thread available. For the concurrent approach, we also specify that the tasks are "non-sticky". This technical detail means a task can run on any available thread rather than the one it started on, allowing for more efficient resource allocation.

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 in the following implementation.

```julia
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
```

```julia
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
```

Previewing some of the approaches we'll introduce later, let's compare Julia's default implementation (sequential) 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. Essentially, it's equivalent to creating and scheduling a non-sticky task. The following code snippets demonstrate both the default and multithreaded approaches.

```julia
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
```

```julia
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 your program to wait for operations to complete before proceeding. 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 mutated state, but merely its value at the moment of reference.

For instance, suppose we seek to mutate the vector `x = [0,0,0]` into `x = [1,2,3]`. Let's begin considering Julia's default sequential execution. This ensures that the mutation completes before continuing with any other operation.

```julia
# 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, we define a task to perform a mutation as follows.

```
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 snippets show the consequences of waiting versus not waiting for the task 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)")
```

```julia
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)")
```

Without a `wait` call, the main program schedules the mutation task and immediately proceeds to the next line. Since the task contains a `sleep` instruction, the mutation hasn't yet begun when `x` is accessed, resulting in the use of its original value `[0,0,0]`. This demonstrates that properly synchronizing tasks is essential for correctness.

11c. Task-Based Parallelism: @spawn

Martin Alfaro
PhD in Economics

INTRODUCTION

The previous section introduced the basics of multithreading, highlighting that operations can be computed either sequentially (Julia's default) or concurrently. The latter enables the processing of multiple operations simultaneously, with each of them running as soon as a thread becomes available. This implies that, when Julia's session is initialized with more than one thread, computations can be executed on different CPU cores 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.

> **Warning!**
> 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. This is why, after grasping a basic understanding of parallelism techniques, we'll devote an entire section on thread-unsafe operations.

ENABLING MULTITHREADING

Each Julia session is initialized with a given pool of threads available. Since each thread can run a set of instructions independently, the total number of threads determines how many instructions can be processed simultaneously.

When we install Julia, this runs in single-threaded mode by default. To enable multithreading, you need to configure Julia to launch with more than one thread. In VSCode or VSCodium, this can be done by navigating to *File* > *Preferences* > *Settings*. Then, search for the keyword *threads*, which will display the following option:

After selecting *Edit in settings.json*, add the line `"julia.NumThreads": "auto"` in the JSON file that opens. This setting automatically detects the number of threads supported by your system, typically matching either the logical or physical cores of your CPU.

Notice the effects will take place after starting a new session. Moreover, the changes are permanent, so that every new Julia session will start with the number of threads specified. To check that the effects have taken place, run the command `Threads.nthreads()`, which displays the number of threads available in the session. Any number greater than one will indicate that multithreading is active.

Once a multithreaded session is started, you can perform parallel computations. While several packages exist for this, the current section will focus on the capabilities built directly into Julia. They're provided by the `Threads` package, which is automatically imported in every Julia session.

```julia
# package Threads automatically imported when you start Julia

Threads.nthreads()
```

```
2
```

```julia
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 for parallel execution we'll introduce is given by the macro `@spawn`. By prefixing an expression with `@spawn`, Julia creates a (non-sticky) task that's immediately scheduled for execution. If a thread is available, the task begins running right away.

Unlike other parallel programming approaches we'll examine later, `@spawn` requires explicit instructions to wait for task completion. The method for doing so depends on the nature of the task output.

The `fetch` function should be employed when tasks produce computational outputs. It serves a dual purpose of waiting for a task to complete and retrieving its return value. Since parallel computation consists of multiple tasks spawned, `fetch` should be broadcast over a collection containing all the

spawned tasks.

The following example demonstrates `fetch` with two spawned tasks, each returning a vector.

```
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
```

Note that `fetch` takes tasks as its input. Consequently, it's essential to distinguish between `task_a` and `a`:

- `task_a` denotes the task creating the vector `a` (i.e., the task itself),
- `a` refers to the vector created (i.e., the task's output).

For tasks that perform actions but don't return any output, we can use either the function `wait` or the macro `@sync`. The function `wait` works analogously to `fetch`, except that it doesn't return any value. Instead, the macro `@sync` requires enclosing all operations to be synchronized using the keywords `begin` and `end`.

To illustrate, consider a mutating function. Such functions are suitable as examples, since they modify the contents of a collection in place, without producing a return value.

```
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 calculate both the sum and the maximum of a vector `x`. Their computation will be implemented following a sequential and a simultaneous approach. To unveil the benefits of parallelization, we'll also include the execution time of each operation in isolation.

The results establish that the total runtime of the sequential procedure is essentially the sum of the individual runtimes. In contrast, the runtime under multithreading is roughly equivalent to the longer of the two computations, thanks to parallelism.

```
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)
```

```julia
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)
```

Although the multithreaded runtime is close to the maximum of the individual runtimes, the equivalence isn't exact. This is because **multithreading introduces overhead** due to the creation, scheduling, and synchronization of tasks. As a result, **multithreading isn't advantageous for small workloads**, where the overhead can outweigh any potential gains.

To illustrate this effect, 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.

```julia
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)
```

```julia
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 multiple threads to run simultaneously within a single process, enabling parallel execution of operations on the same machine. Unlike other forms of parallelization (e.g., multiprocessing), multithreading is characterized by **the sharing of a common memory space among all tasks**.

This shared-memory environment introduces additional complexity, since **parallel execution can result in side effects if not managed carefully**. The core issue arises when multiple threads access and modify shared data concurrently, which can cause unintended consequences in other threads. This possibility arises because threads are competing for the same resources, and their interactions must be coordinated to prevent errors.

Based on this potential problem, we distinguish between thread-safe and thread-unsafe operations. An operation is considered **thread-safe** if it can be executed in parallel without causing inconsistencies, crashes, or corrupted results. Conversely, **thread-unsafe** operations require explicit synchronization or code restructuring to avoid errors.

This section will focus on identifying key features that render certain operations unsafe. In particular, we'll see that common operations like reductions aren't thread-safe, leading 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 modifications to program structure.

THREAD-UNSAFE OPERATIONS

In multithreaded environments, unsafe operations correspond to those that can lead to incorrect behavior, data corruption, or program crashes when executed concurrently. Such issues typically arise when tasks exhibit some degree of dependency, either in terms of operations or shared resources.

WRITING ON A SHARED VARIABLE

One of the simplest examples of a thread-unsafe operation is writing to a shared variable. To illustrate, consider a scenario where a scalar variable `output` is initialized to zero. This value is then updated within a for-loop that iterates twice, with `output` set to `i` in the *i*-th iteration. The corresponding script is shown below.

To illustrate the challenges of concurrent execution, we deliberately introduce a decreasing delay before updating `output`. This is implemented with `sleep(1/i)`, causing the first iteration to pause for 1 second and the second iteration for half a second. Although this delay is artificially introduced via `sleep`, it represents the potential time gap caused by intermediate computations, which could preclude an immediate update of `output`.

```
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
```

The delay is inconsequential for a sequential procedure, where `output` takes on the values 0, 1, and 2 as the program progresses. However, when executed concurrently, it determines that the first iteration completes 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 in this simple example, it can manifest in more complex and subtle ways in real-world applications. The core issue is that the order of execution isn't guaranteed in a multithreaded environment. Thus, when multiple threads modify the same shared variable, the final value depends on which thread executes last.

In fact, the issue can be exacerbated when each iteration additionally involves reading a shared variable. Next, we consider such a scenario.

READING AND WRITING A SHARED VARIABLE

Reading and writing shared data doesn't necessarily yield incorrect results. For instance, a parallel for-loop could safely mutate a vector: even if multiple threads are simultaneously modifying the same shared object, each thread would be operating on distinct elements of the vector. Thus, no two threads would interfere with one another, making the updates remain independent.

Problems arise, however, **when the correctness of reading and writing shared data depends on the order in which threads execute**. This situation is known as a **race condition**. The term reflects the fact that the final output may change from one run to the next, depending on which thread finishes and updates the data last.

To illustrate the issue, let's modify our previous example by introducing a variable `temp`, whose value is updated in each iteration. This variable will be shared across threads and used to mutate the *i*-th entry of the vector `output`. By introducing a delay before writing each entry of `output`, a race condition is introduced, where all threads end up using the last value of `temp` (in this case, 2).

```julia
function foo()
    output = Vector{Int}(undef,2)
    temp   = 0

    for i in 1:2
        temp      = i; sleep(i)
        output[i] = temp
    end

    return output
end
```

```
julia> foo()
2-element Vector{Int64}:
 1
 2
```

```julia
function foo()
    output = Vector{Int}(undef,2)
    temp   = 0

    @threads for i in 1:2
        temp      = i; sleep(i)
        output[i] = temp
    end

    return output
end
```

```
julia> foo()
2-element Vector{Int64}:
 2
 2
```

```julia
function foo()
    output = Vector{Int}(undef,2)


    @threads for i in 1:2
        temp      = i; sleep(i)
        output[i] = temp
    end

    return output
end
```

```
julia> foo()
2-element Vector{Int64}:
 1
 2
```

In this specific scenario, the issue can be easily circumvented by defining `temp` as a variable local to the for-loop, rather than initializing it outside. This ensures that each thread works with its own private copy of `temp`, thereby eliminating the data race.

Beyond the solution proposed, the example highlights the subtleties of parallelizing operations. Even seemingly simple patterns can introduce hidden dependencies that lead to unsafe behavior when executed concurrently. To make this clearer, we now turn to a more common scenario where data races occur: reductions.

RACE CONDITIONS WITH REDUCTIONS

Reductions are a prime example of thread-unsafe operations. To illustrate, consider summing a collection in parallel. The problem arises because the variable accumulating the sum is shared across all threads. During each iteration, every thread attempts to read its current value, add its contribution, and write the result back. When several threads do this simultaneously, their updates can interleave unpredictably, causing some partial additions to be overwritten rather than combined. As a result, the final sum is nondeterministic and often incorrect, varying from run to run.

Below, we illustrate this issue when we sum the first two elements of a vector `x`.

The consequence is that when we execute reductions in parallel without considering the emerging race condition, the result becomes unpredictable. In particular, the total sum is lower than the correct result, given the lost updates.

```julia
x = rand(1_000_000)

function foo(x)
    output = 0.0

    for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
500658.01158503356
```

```julia
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @threads for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
21558.16818215786
```

```julia
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @threads for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
21933.164831088678
```

```julia
x = rand(1_000_000)

function foo(x)
    output = 0.0

    @threads for i in eachindex(x)
        output += x[i]
    end

    return output
end
```

```
julia> foo(x)
21402.970049988293
```

The key insight from this example isn't that reductions are incompatible with multithreading. Rather, it's that the strategy to apply multithreading needs to be adapted accordingly. While the upcoming sections will present these strategies, we conclude this section by turning to the opposite end of the spectrum: problems that naturally lend themselves to parallel execution

EMBARRASSINGLY PARALLEL PROBLEMS

Programs that can be paralleled seamlessly are referred to as **embarrassingly parallel problems**. They can be broken down into many completely independent tasks, each of which can be executed in parallel without any need for communication, synchronization, or shared state between them. This independence provides complete flexibility in the order of task execution.

In the context of for-loops, one of the simplest ways to parallelize these problems is with 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 balance the workload by dividing the iterations as evenly as possible.

The approach contrasts with `@spawn`, which implements task-based parallelism. With `@spawn`, the programmer explicitly defines tasks and synchronization must be handled manually. By contrast, `@threads` automatically schedules the tasks and waits for their completion before execution proceeds. The following example demonstrates this behavior.

```julia
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($x_small)
  3.350 µs (3 allocations: 7.883 KiB)
julia> @btime foo($x_medium)
  347.043 µs (3 allocations: 781.320 KiB)
julia> @btime foo($x_big)
  3.588 ms (3 allocations: 7.629 MiB)
```

```julia
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($x_small)
  13.529 μs (125 allocations: 20.508 KiB)
julia> @btime foo($x_medium)
  42.165 μs (125 allocations: 793.945 KiB)
julia> @btime foo($x_big)
  336.192 μs (125 allocations: 7.642 MiB)
```

In the next section, we'll present a detailed comparison of `@threads` and `@spawn`.

11e. Parallel For-Loops

Martin Alfaro
PhD in Economics

INTRODUCTION

For-Loops are fundamental building blocks in scientific computing, often representing the most computationally intensive parts of a program. To accelerate these computations, parallel processing is capable of distributing a for-loop's iterations across multiple processor threads. The central challenge, however, lies in how this work is divided, as the optimal strategy heavily depends on the nature of the workload.

This section introduces Julia's `@threads` macro as a straightforward approach to parallelizing for-loops. It operates by dividing the loop iterations evenly among the available threads. For the explanations, we'll contrast its underlying mechanism with the task-based parallelism offered by `@spawn`.

To clearly illustrate the differences between approaches, our examples will use `@spawn` in a simple but inefficient manner, where a separate task is created for every single iteration. While this is enough for the purpose of comparison with `@threads`, you should bear in mind that it's not representative of a typical `@spawn` usage. In the next section, we'll revisit the macro and show how to efficiently define tasks. It'll reveal that the inherent flexibility of `@spawn` enables the construction of sophisticated parallel strategies, turning the behavior of `@threads` into a particular application.

SOME PRELIMINARIES

Parallelism techniques target code that performs multiple operations, making it a natural fit for for-loops. Using the macro `@spawn` introduced in the previous sections, we can parallelize for-loops through a 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 code implementing this technique is shown below.

```julia
@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
```

```julia
@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 upcoming sections, offer more granular control, but at the expense of adding substantial complexity to the code.

In light of this complexity, Julia provides the `@threads` macro from the package `Threads`. The goal is to reduce the overhead of task creation, while keeping the parallelization simple. Specifically, `@threads` divides 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 with `@spawn`. The scenario considered is based on 4 iterations and 2 worker threads. We also display the thread on which each iteration is executed by using the `threadid()` function, which identifies the thread's ID that's computing the operation.

```julia
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
```

```julia
@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 the strategy for thread allocation. 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 previous example, a single thread ended up 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 potential calculations 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.

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 are two tasks 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)
```

```julia
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)
```

```julia
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. Parallelization in Practice

Martin Alfaro
PhD in Economics

INTRODUCTION

So far, we've explored two macros for parallelization: `@spawn` and `@threads`. The macro `@spawn` provides granular control over the parallelization process, letting users explicitly define the tasks to be executed concurrently. In contrast, `@threads` offers a simpler approach for parallelizing for-loops, where iterations are automatically partitioned into tasks, according to the number of available threads. Furthermore, we've pointed out that, due to inherent dependencies between computations, not all workloads are equally amenable to parallelization. In particular, a naive approach to parallelization can lead to severe issues.

Essentially, our discussions have largely focused on the *syntax* and *work distribution* of parallelization approaches. Yet, we have to address how to apply multithreading in real scenarios. Furthermore, given the possibility of dependencies between computations, *how* to parallelize is only part of the challenge: knowing *when* to parallelize is equally important.

This section and the next one aim to bridge this gap, providing practical guidance on implementing multithreading. We begin by highlighting the advantages of coarse-grained parallelization over fine-grained parallelization. By dividing the workload into a small number of large tasks, coarse-grained parallelization reduces the scheduling overhead from managing numerous lightweight tasks.

After this, we revisit the parallelization of for-loops, this time using `@spawn`. In particular, leveraging the additional control that `@spawn` provides over task creation, we'll demonstrate how to apply multithreading in the presence of a ubiquitous type of dependency: reductions.

We conclude by showing a performance issue arising with multithreading, known as false sharing. While this doesn't affect the correctness the result, it can significantly slow down computations if not addressed.

BETTER TO PARALLELIZE AT THE TOP

Given the overhead involved in multithreading, there's an inherent trade-off between creating new tasks and fully utilizing machine resources. This is why we must always consider whether parallelization is worthwhile in the first place. For instance, when it comes to operations over collections, multithreading is only justified if the collections have enough elements to offset the associated overhead. Otherwise, single-threaded approaches will consistently outperform parallelized ones.

In case multithreading is deemed beneficial, we immediately face another decision: at what level code should be parallelized. Next, we'll demonstrate that **parallelism at the highest possible level is preferable, compared to multithreading individual operations**. By adopting this strategy, we minimize the overhead of task creation.

Note that the level of parallelization is always constrained by the degree of dependency between operations. Hence, our qualification of highest **possible** level. For instance, in problems requiring strictly serial computation, the best we can achieve is parallelization within each individual step.

To illustrate, let's consider a for-loop where each iteration needs to sequentially compute three operations.

JULIA'S DEFAULT

```julia
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($x_small)
  5.206 μs (3 allocations: 7.883 KiB)
julia> @btime foo($x_large)
  537.663 μs (3 allocations: 781.320 KiB)
```

PARALLELIZATION AT THE HIGHEST LEVEL POSSIBLE

```julia
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($x_small)
  13.667 μs (125 allocations: 20.508 KiB)
julia> @btime foo($x_large)
  71.050 μs (125 allocations: 793.945 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($x_small)
  35.841 µs (375 allocations: 61.523 KiB)
julia> @btime foo($x_big)
  104.260 µs (375 allocations: 2.326 MiB)
```

The examples illustrate the two-step process outlined. First, it shows that parallelization is advantageous only with large collections. Otherwise, the question of whether to parallelize shouldn't even arise. Second, once multithreading proves to be advantageous, it demonstrates that grouping all operations into a single task is faster than parallelizing each operation individually.

IMPLICATIONS

The strategy of parallelizing code at the highest possible level has significant implications for program design. In particular, when the program will eventually be applied to multiple independent objects. It suggests a practical guideline: start with an implementation for a single object, without introducing parallelism. After thoroughly optimizing the single-case code, integrate parallel execution at the top level. The approach not only improves performance, but also simplifies the development by making debugging and testing more straightforward.

A common application of this strategy arises in scientific simulations. In those cases, independent executions of the same model are required. Thus, the most effective approach is to maintain a single-threaded implementation of the model, eventually launching multiple instances in parallel. This design ensures that each run remains efficient at the single-thread level, while taking advantage of full resource utilization.

THE IMPORTANCE OF WORK DISTRIBUTION

Multithreading performance is heavily influenced by how evenly the computational workload is distributed across iterations. The `@threads` macro spawns a task for every iteration, making it highly effective when each iteration requires roughly equal processing time. In contrast, scenarios with uneven computational effort are more challenging. In such cases, some threads may finish early and remain idle, while others continue processing heavier tasks. This imbalance undermines parallel efficiency, substantially diminishing the performance gains of multithreading.

To address this issue, we need greater control over how work is distributed among threads. This calls for the use of `@spawn`.

One strategy is to make each iteration a separate task. However, such approach is extremely inefficient if there's a large number of iterations: creating far more tasks than there are threads introduces substantial and unnecessary overhead. The following example illustrates this problem.

@THREADS

```julia
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)
  5.877 ms (125 allocations: 76.309 MiB)
```

@SPAWN

```julia
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)
  11.095 s (60005888 allocations: 5.277 GiB)
```

An alternative strategy, which lets users fine-tune the workload of each task, is to partition iterations into smaller subsets that can be processed in parallel. Before detailing the implementation, we'll first explore how to partition a collection along with its indices through the `ChunkSplitters` package.

PARTITIONING COLLECTIONS

The package `ChunkSplitters` provides two functions for lazy partitioning: `chunks` and `index_chunks`. These functions support `n` and `size` as keyword arguments, depending on the type of partition desired. Specifically, `n` sets the number of subsets to create, with each subset sized to distribute elements evenly. In contrast, `size` specifies the number of elements to be contained in each subset. Since an even distribution across all subsets can't be guaranteed in all cases, the package adjusts the number of elements in one of the subsets if necessary.

Below, we apply these functions to a variable `x` that contains the 26 letters of the alphabet. Note that the outputs provided require the use of `collect`, since `chunks` and `index_chunks` are lazy.

PARTITION BY NUMBER OF CHUNKS

```julia
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
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"]
```

For multithreading, a relevant partition is a number of chunks proportional to the number of worker threads. The example below implements this, generating both chunk indices and chunk values. Since this partition will eventually be used with for-loops, we also show how to use `enumerate` to pair each chunk with the values or subindices of its corresponding subset.

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)
```

```
julia> collect(chunk_indices)
24-element Vector{UnitRange{Int64}}:
 1:2
 3:4
 ⋮
 25:25
 26:26
julia> collect(chunk_values)
24-element Vector{SubArray{String, 1, Vector{String}, Tuple{UnitRange{Int64}}, true}}:
 ["a", "b"]
 ["c", "d"]
 ⋮
 ["y"]
 ["z"]
```

PARTITION BY NUMBER OF THREADS - ENUMERATE

```
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_iter1   = enumerate(chunk_indices)    # pairs (i_chunk, chunk_index)
chunk_iter2   = enumerate(chunk_values)     # pairs (i_chunk, chunk_value)
```

```
julia> collect(chunk_iter1)
24-element Vector{Tuple{Int64, UnitRange{Int64}}}:
 (1, 1:2)
 (2, 3:4)
 ⋮
 (23, 25:25)
 (24, 26:26)
julia> collect(chunk_iter2)
24-element Vector{Tuple{Int64, SubArray{String, 1, Vector{String},
Tuple{UnitRange{Int64}}, true}}}:
 (1, ["a", "b"])
 (2, ["c", "d"])
 ⋮
 (23, ["y"])
 (24, ["z"])
```

WORK DISTRIBUTION: DEFINING TASKS THROUGH CHUNKS

Leveraging the `ChunkSplitters` package together with `@spawn`, we can control how a for-loop parallelized. The process divides iterations into chunks, where each of them correspond to a independent task to be processed concurrently.

Below, we show how to replicate the behavior of `@threads` via `@spawn`. The two approaches becom equivalent when the number of chunks is set equal to the number of worker threads.

@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)
  5.877 ms (125 allocations: 76.309 MiB)
```

@SPAWN

```julia
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, nthreads())
  5.829 ms (157 allocations: 76.311 MiB)
```

@SPAWN (EQUIVALENT)

```julia
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, nthreads())
  5.841 ms (151 allocations: 76.310 MiB)
```

The flexibility of `@spawn` implies we're not constrained to following the approach of `@threads`. Fo instance, it's common to adopt a partitioning strategy where the number of chunks is proportional t the number of worker threads. This is shown below.

@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($x, 1 * nthreads())
  7.058 ms (157 allocations: 76.311 MiB)
julia> @btime foo($x, 2 * nthreads())
  5.492 ms (302 allocations: 76.325 MiB)
julia> @btime foo($x, 4 * nthreads())
  4.982 ms (590 allocations: 76.352 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($x, 1 * nthreads())
  4.379 ms (133 allocations: 76.310 MiB)
julia> @btime foo($x, 2 * nthreads())
  4.529 ms (254 allocations: 76.323 MiB)
julia> @btime foo($x, 4 * nthreads())
  4.080 ms (494 allocations: 76.347 MiB)
```

Having more chunks than number of threads can make sense when, for instance, the processor cores don't offer uniform performance. When this is the case, the strategy will ensure that there th fastest cores don't remain idle, so that there's full resource utilization of computational resources.

HANDLING DEPENDENCIES

So far, our discussion of parallelization has focused on embarrassingly parallel for-loops, where iterations are completely independent. This enables the execution of each iteration in isolation, making parallelization straightforward.

The situation becomes more complex when operations exhibit dependencies. Attempting to parallelize without first addressing these dependencies can lead not only to inefficiencies and wasted resources, but most critically to incorrect results.

There's no one-size-fits-all method for handling dependencies. The appropriate strategy depends on the structure of the specific program. In all cases, though, the approach will require adapting the parallelization technique to work on a reformulated version of the problem. This reformulation must ensure that the tasks to be parallelized are independent—ultimately, parallelization is only possible if independence between operations can be achieved.

Note that, once dependencies are present, some portion of the work will inevitably be non-parallelizable. In fact, no subset of independent tasks may exist at all, as when computations are inherently sequential.

HANDLING REDUCTIONS

A common technique that exhibits dependency between iterations is reductions. To still benefit from parallelization despite this dependency, its computation must be reorganized. The typical strategy is to partition the data into chunks, perform a partial reduction on each chunk in parallel, and then combine the partial results in a final reduction step. This restructuring removes the original dependency between iterations, since each partial reduction now operates on a disjoint subset of the data.

To illustrate, let's compute the sum of elements of a vector `x`. The implementation relies on the `ChunkSplitters` package to divide the data into independent segments.

JULIA'S DEFAULT (SEQUENTIAL)

```julia
x = rand(10_000_000)

function foo(x)
    output = 0.0

    for i in eachindex(x)
        output += x[i]
    end

    output
end
```

```
julia> @btime foo($x)
  5.203 ms (0 allocations: 0 bytes)
```

@THREADS

```julia
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)
  1.268 ms (124 allocations: 13.250 KiB)
```

@SPAWN

```julia
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)
  1.286 ms (156 allocations: 13.781 KiB)
```

FALSE SHARING

Addressing dependency between operations is necessary when parallelizing, since the program may otherwise yield incorrect results. However, even after removing dependencies and ensuring correctness, performance can still suffer due to hardware-level effects.

One common bottleneck is **cache contention**, where multiple processor cores compete for shared cache resources. A manifestation of this issue is what's known as **false sharing**, where multiple cores simultaneously access data stored in the same cache line. To understand why this degrades performance, it helps to first review how CPU caches work.

Processors rely on caches to hold copies of frequently accessed data. These caches are much smaller but significantly faster than main memory (RAM), and they're organized into fixed-size blocks called cache lines (typically 64 bytes). When the processor needs a piece of data, it first checks whether it's already present in the cache. If not, the data must be fetched from RAM and placed into a cache line, a process that's considerably slower.

Likewise, when multiple cores access data within the same cache line, the transfer of information is governed by a cache coherency protocol. Its goal is to ensure consistency across cores. However, the protocol can create inefficiencies: even if one core accesses data that remains unmodified, any modification to another value within the same cache line may cause the entire line to be invalidated. As a result, all cores are forced to reload the block, despite the absence of a logical need to do so. This phenomenon, known as **false sharing**, leads to unnecessary cache invalidations and refetches. The outcome is a notable degradation in program performance, particularly in workloads where threads frequently update their variables.

Below, we focus on the emergence of false sharing in reduction operations.

FALSE SHARING IN REDUCTIONS: AN ILLUSTRATION AND SOLUTIONS

Suppose that the elements of a vector are summed after applying a logarithmic transformation. We'll present two implementations. The first one acts as a baseline and consists of a sequential procedure. The second one is multithreaded but suffers from false sharing. The problem arises because each thread writes its partial sum into a different element of the `partial_outputs` vector. Although these elements are logically independent, they're stored in adjacent memory locations that fall within the same cache line. As a result, every update by one thread invalidates the cache line in other cores, forcing unnecessary reloads that degrade performance.

SEQUENTIAL

```julia
x = rand(10_000_000)

function foo(x)
    output = 0.0

    for i in eachindex(x)
        output += log(x[i])
    end

    output
end
```

```
julia> @btime foo($x)
  37.046 ms (0 allocations: 0 bytes)
```

FALSE SHARING

```julia
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)
        for j in chunk
            partial_outputs[i] += log(x[j])
        end
    end

    return sum(partial_outputs)
end
```

```
julia> @btime foo($x)
  17.434 ms (124 allocations: 13.250 KiB)
```

There are several strategies to address the issue. All of them ensure that threads don't write to data residing in the same cache line.

One common strategy is **vector padding**, where extra spacing is inserted between the elements of `partial_outputs`. This strategy ensures that each thread's accumulator is placed on a distinct cache line, so that concurrent writes no longer interfere with one another at the cache level. We implement this below by storing the partial outputs in a vector with sufficient separation between rows. In particular, a separation of 8 entries.

PADDING

```
x = rand(10_000_000)

function foo(x)
    chunk_ranges    = index_chunks(x, n=nthreads())
    nr_strides      = 8
    partial_outputs = Vector{Float64}(undef, length(chunk_ranges) * nr_strides)

    @threads for (i, chunk) in enumerate(chunk_ranges)
        for j in chunk
            partial_outputs[(i-1)*nr_strides + 1] += log(x[j])
        end
    end

    return sum(@view(partial_outputs[1:nr_strides:end]))
end
```

```
julia> @btime foo($x)
  6.243 ms (124 allocations: 14.625 KiB)
```

While intuitive, the solution lacks practical appeal. In fact, the exact number of rows to insert depends on the element type and the underlying computer architecture. For this reason, let's introduce more efficient general approaches to avoiding false sharing.

The first method introduces a thread-local variable `temp` to store partial results. In this way, each thread updates only its own local variable, finally writing to the shared array exactly once at the end. We implement this solution via `@threads` and `@spawn`. After that, we present an alternative solution where each partial reduction is computed inside a separate function. This addresses false sharing in a similar spirit, where accumulation is based on a local variable.

LOCAL VARIABLE (@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)
        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)
  4.820 ms (124 allocations: 13.250 KiB)
```

LOCAL VARIABLE (@SPAWN)

```julia
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 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)
  4.385 ms (156 allocations: 13.781 KiB)
```

FUNCTION

```julia
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 = Vector{Float64}(undef, 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)
  4.851 ms (124 allocations: 13.250 KiB)
```

11g. Multithreading Packages

Martin Alfaro
PhD in Economics

INTRODUCTION

Parallelizing code may seem straightforward at first glance. However, once we start delving into its subtleties, it's rapidly revealed that an effective implementation can be a daunting task. 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 we've mastered the necessary skills for a correct implementation, manual parallelization often impairs the readability and maintainability of code.

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 performance issues like false sharing. We'll then examine the `Polyester` package. Thanks to its reduced overhead, this package excels at parallelizing workloads involving small objects. After this, we revisit the `LoopVectorization` package and its `@tturbo` macro. This combines the benefits of SIMD instructions with multithreading.

PACKAGE "OHMYTHREADS"

`OhMyThreads` is designed as a user-friendly package for seamlessly applying multithreading. Consistent with its minimalist approach, it introduces only a handful of essential functionalities that could easily belong to Julia's `Base`. Despite its simplicity, the package covers a wide range of operations, including reductions.

The package's primary goal is to make parallelization accessible, even to users without deep expertise in multithreading. Indeed, its functions and macros automatically address common pitfalls, such as data races and performance issues like false sharing.

To achieve broad applicability, `OhMyThreads` extends familiar `Base` operations into the multithreaded domain through a set of higher-order functions. Each parallelized variant follows a simple naming convention: the original function name prefixed with `t`. For example, the parallel counterpart to `map` becomes `tmap` in the package. At the same time, the package remains flexible by integrating with `ChunkSplitters`, allowing users to fine-tune how work is distributed across tasks. This customization

is controlled through two keyword arguments: `nchunks` (or equivalently `ntasks`) to specify the number of subsets in the partition, and `chunksize` to define the number of elements in each partition.

> **Warning!**
> All the code snippets below assume you've already loaded the package with `using OhMyThreads`.

PARALLEL MAPPING

`OhMyThreads` defines `tmap` as the multithreaded counterpart to the `map` function. Its recommended usage is `tmap(foo, T, x)`, where `foo` is the function applied to each element of the collection `x`, and `T` specifies the *element type* of the output. For example, if the result is a `Vector{Float64}`, then `T` should be `Float64`. A more flexible alternative to explicitly declaring `T` is to use `eltype(x)`, ensuring that the output type matches the input collection.

While it's possible to omit `T` and write `tmap(foo, x)`, this comes at a performance cost. The issue stems from a technical nuance, given by the type instability of the `Task` object. Explicitly declaring the output type avoids this issue and restores full performance.

The package also provides an in-place version `tmap!`, whose syntax is `tmap(foo, output, x)`. Since `tmap!` requires the output vector to be specified directly, there's no need to declare `T` to prevent performance losses.

The examples below demonstrate the use of these functions. For comparison, we also include results from `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)
```

`tmap` allows us to control the work distribution among tasks through the keyword arguments `nchunks` and `chunksize`. They're 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 passing anonymous functions into `tmap`, the do-block syntax comes in handy for keeping code readable. It enables the creation of multi-line functions, without the need to introduce a new function before applying `tmap`.

```julia
x = rand(1_000_000)

function foo(x)

    output = tmap(a -> 2 * log(a), x)



    return output
end
```

```julia
x = rand(1_000_000)

function foo(x)

    output = tmap(x) do a
                2 * log(a)
             end

    return output
end
```

ARRAY COMPREHENSIONS

`OhMyThreads` provides an alternative to `tmap` that can be implemented via array comprehensions. Unlike the standard syntax of array comprehensions, the version from `OhMyThreads` combines a generator with a multithreaded variant of `collect` named `tcollect`.

As with `tmap`, specifying the output's element type is optional, although necessary to avoid performance losses. The recommended syntax is therefore `tcollect(T, <generator>)`, where `T` denotes the element type of the output. A common practice again is to use `eltype(x)` as `T`, with `x` being the variable iterated over in the generator. This ensures that the output type matches the input collection.

```julia
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` additionally provides multithreaded counterparts to `reduce` and `mapreduce`, respectively referred to as `treduce` and `tmapreduce`. These functions automatically manage the race conditions inherent in reductions, also addressing performance issues such as false sharing. Unlike `tmap`, they can achieve optimal performance without the need to explicitly specify an 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

`OhMyThreads` also offers a multithreaded version of `foreach` called `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.

```
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)
```

```julia
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)
```

```julia
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.

```julia
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)
```

```julia
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)
```

```julia
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)
```

```julia
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)
```

Similar to `tmap`, `tforeach` offers the keyword arguments `nchunks` and `chunksize` to control the workload distribution among worker threads. To illustrate, we use a work distribution analogous to the one used above for `tmap`.

```julia
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)
```

```julia
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 the overhead introduced by the creation and scheduling of tasks. For smaller computational workloads, this overhead can outweigh any potential performance gain, rendering parallelization detrimental. As a result, multithreading is typically reserved for objects large enough to justify the cost.

The `Polyester` package addresses this limitation by providing a low-overhead implementation of for-loops. Its approach makes it possible to parallelize operations on objects that, otherwise, would be deemed too small to benefit from multithreading. To use `Polyester`, we simply prefix the for-loop with the `@batch` macro.

To illustrate its benefits, let's consider a for-loop with 500 iterations, a relatively low number for applying multithreading. The first tab below shows that an approach based on `@threads` is slower than its single-threaded variant. In contrast, `Polyester` achieves a comparable performance to the single-threaded variant, even with such a small workload.

```
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 workloads, it's worth noting that `Polyester` may not consistently outperform or underperform other multithreading approaches. Ultimately, performance will depend on the specifics of the computation and data involved. In such cases, the best practice is to benchmark your particular application.

REDUCTIONS

`Polyester` also supports reduction operations. These can be implemented by prepending the for-loop with the expression `@batch reduce=(<tuple containing operation and variable reduced>)`. The implementation has been designed to avoid common pitfalls of reductions, such as data races and false sharing. This ensures both correctness and performance.

```
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)
```

It's possible to incorporate more than one reduction operation per iteration, as demonstrated below.

```julia
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)
```

```julia
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)
```

```julia
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

Unlike macros like `@threads`, `Polyester` treats variables as local per iteration.

```
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
```

```julia
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 parallelization in this package is achieved through integration with `Polyester`.

The primary way to implement it is via the `@tturbo` macro in a for-loop, 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` enforces its application.

To illustrate the benefits of `@tturbo`, let's consider an example that combines two features: SIMD isn't applied automatically by `@threads`, even when the operation is well-suited for this purpose.

```julia
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)
```

```julia
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)
```

```julia
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 also applies to broadcast operations. Offering this functionality is particularly valuable, as no built-in macro currently exists to parallelize broadcast expressions.

While applying `@tturbo` in a for-loop form often yields higher performance, the broadcast variant offers a much simpler and more concise syntax. The example below illustrates how this approach can significantly improve readability.

```
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 by executing `using FLoops`.

We conclude this section with a brief overview of the package `FLoops`. The presentation is labeled as optional since usage beyond simple applications may require some workarounds. In addition, the package doesn't appear to be actively maintained.

The primary macro offered by the package is `@floop`, exclusively designed for use with for-loops. An example of its application is provided below.

```julia
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)
```

```julia
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 placing `@reduce` at the beginning of the line containing the reduction operation. The macro addresses the inherent data race associated with and prevents false sharing.

```julia
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)
```

```julia
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)
```

```julia
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`.