
Julia Data Science

Jose Storopoli, Rik Huijzer and Lazaro Alonso

Contents

1 Preface	7
1.1 Why Data Science?	7
1.1.1 Data is Everywhere	7
1.1.2 Data Literacy	8
1.2 Software engineering and data science	8
1.3 Acknowledgements	9
2 Why Julia?	11
2.1 For non-programmers	11
2.2 For programmers	11
2.3 What Julia tries to accomplish?	13
2.3.1 Julia versus other programming languages	13
2.3.2 The Two-Language Problem	15
2.3.3 Multiple Dispatch	16
2.4 Julia in the Wild	19
3 Julia 101	21
3.1 Language Syntax	21
3.1.1 Variables	21
3.1.2 User-defined Types	23
3.1.3 Boolean Operators an Numeric Comparison	24
3.1.4 Functions	25
3.1.5 Conditional If-Else-Elseif	29
3.1.6 For Loop	30
3.1.7 While Loop	30
3.2 Native Data Structures	31
3.2.1 Broadcasting Operators and Functions	31
3.2.2 String	32
3.2.3 Tuple	36
3.2.4 Named Tuple	37
3.2.5 Ranges	37
3.2.6 Array	38
3.2.7 Pair	48
3.2.8 Dict	49
3.2.9 Symbol	51
3.2.10 Splat Operator	51
3.3 Filesystem	52

3.4	Julia Standard Library	53
3.4.1	Dates	53
3.4.2	Random Numbers	59
3.4.3	Downloads	62
4	DataFrames.jl	65
4.1	Load and Save Files	70
4.1.1	CSV	70
4.1.2	Excel	73
4.2	Index and Summarize	74
4.3	Filter and Subset	77
4.3.1	Filter	77
4.3.2	Subset	79
4.4	Select	81
4.5	Types and Missing Data	84
4.6	Join	87
4.6.1	innerjoin	88
4.6.2	outerjoin	88
4.6.3	crossjoin	89
4.6.4	leftjoin and rightjoin	90
4.6.5	semijoin and antijoin	91
4.7	Variable Transformations	92
4.7.1	Multiple Transformations	94
4.8	Groupby and combine	95
4.8.1	Multiple Source Columns	97
5	Performance	99
6	Data Vis with Plots	101
6.1	Layouts	102
6.2	Animations	102
6.3	Other backends	102
6.4	StatsPlots.jl	103
7	Data Vis with Makie.jl	105
7.1	CairoMakie.jl	105
7.2	Attributes	106
7.3	Themes	113
7.4	using LaTeXStrings.jl	118
7.5	Colors and Colormaps	121
7.5.1	Custom cycle	127
7.6	Layouts	129
7.6.1	Nested Axis (<i>subplots</i>)	137
7.6.2	Nested GridLayout	138
7.6.3	Inset plots	139

7.7	GLMakie.jl	142
7.7.1	Scatters and Lines	143
7.7.2	Surfaces, wireframe, contour, contourf and contour3d	144
7.7.3	Arrows and Streamplots	148
7.7.4	Meshes and Volumes	149
7.7.5	Filled Line and Band	152
7.8	Animations	153
8	Appendix	155
8.1	Notation	155
8.1.1	Julia Style Guide	155
8.1.2	BlueStyle	156
8.1.3	Our additions	156
8.2	Project Management in Julia	157
References		159

1 Preface

Here, we discuss two important aspects of data analysis. Actually, there is a third one, Julia, but it deserves a whole chapter (see Section 2).

First, we emphasize the “**data**” **part of data science** and why data skills are, and will remain, in **high demand** in industry as well as in academia.

Second, we make an argument for **incorporate software engineering practices into data science** which should reduce friction in updating and sharing code with collaborators. Most of data analyses are a collaborative endeavor, these practices should come in handy.

1.1 Why Data Science?

Data science is not only machine learning and statistics, also it is not all about prediction, alas, it is not even a discipline fully contained within STEM (Science, Technology, Engineering, and Mathematics) fields (Meng, 2019). But one thing we can assure with high confidence is that **data is always involved**. Our aim in this book is two-fold:

- First, we want to focus on the backbone of data science: data.
- Second, we want you to use Julia.

We will cover why we think you should use Julia in Section 2. For now, let’s turn our attention towards data.

1.1.1 Data is Everywhere

Data is abundant and will be even more in the near future. A report from late 2012 concluded that from 2005 to 2020, the amount of data stored digitally will **grow by a factor of 300, from 130 exabytes¹ to a whooping 40,000 exabytes** (Gantz & Reinsel, 2012). This is equal to 40 trillion gigabytes and, to put it into perspective, more than **5,200 gigabytes for every living human currently in this planet!** In 2020, on average every person created **1.7 MB of data per second** (Domo, 2018). A recent report predicted that almost **two thirds (65%) of national GDPs will have undergone digitization by 2022** (Fitzgerald et al., 2020).

Every profession will be impacted not only by the increasing availability of data but also by data’s increased importance (Chen et al., 2014; Khan et al., 2014). This is why data skills are important. In order to become the researcher or professional that the world is demanding right now you need to feel comfortable around data. You need to become a **data literate**.

¹1 exabyte (EB) = 1,000,000 terabyte (TB).

1.1.2 Data Literacy

According to Wikipedia, the formal definition of **data literacy** is “**the ability to read, understand, create, and communicate data as information.**”. We also like the informal idea that, as a data literate, you won’t feel overwhelmed by data, quite the contrary, you would feel empowered by it.

Data literacy can be seen as a highly competitive skill to possess. In this book we’ll cover two aspects of data literacy:

1. **Data Manipulation** with `DataFrames.jl` (Section 4). In this chapter you will learn how to:

- 1. Read CSV and Excel data from disk and from the internet
- 2. ...
- 3. ...

2. **Data Visualization** with `Plots.jl` and `Makie.jl` (Section 6 and Section 7). In these chapter you will learn how to:

- 1. ...
- 2. ...
- 3. ...

1.2 Software engineering and data science

Unlike most books on data science, this book lays more emphasis on **properly structuring code**. The reason for this is that we noticed that many data scientists simply place their code into one large file and run all the statements sequentially. You can think of this as writing a book for someone and forcing that person to read it from beginning to end. This works great for small and simple projects, but the bigger or more complex the project becomes, more problems will start to arise. For example, with a well written book: the book is split into distinctly named chapters and sections which contains several references to other parts in the book. The software equivalent of this is **splitting code into functions**. Each function has a name and some contents. With functions, you can basically tell the computer at any point in your code to jump to some other place and continue from there. This allows you to more easily re-use code between projects, update code, share code, collaborate and see the big picture. Hence, with functions, you can **save time**.

So, while reading this book you will eventually get accustomed to reading and using functions. Also, another benefit of having good software engineering skills is that it will allow you to more easily read the packages’s source code that you’re using, which could be greatly beneficial when you are debugging your code or wonder how exactly the code works. Finally, you can rest assured that we did not at all invented ourselves this emphasis on functions. In industry, it is common practice to encourage developers to use “**functions instead of comments**”. That means that, instead of writing a comment for humans and some code for the computer, the developers write a function which is universally read by humans and computers.

1.3 Acknowledgements

Many people have contributed directly and indirectly to this book.

Jose Storopoli would like to thank his colleagues, specially Fernando Serra, Wonder Alexandre Luz Alves and André Librantz, for their encouragement and support.

Rik Huijzer would like to thank his PhD supervisors, Peter de Jonge, Ruud den Hartigh and Frank Blaauw, for their support.

Lazaro Alonso would like to thank his wife and daughters for their encouragement to get involved in this project.

2 Why Julia?

The world of data science is already filled with different open source programming languages.

Industry has mostly adopted Python and academia have adopted R. **Why bother learning another language?** To answer this question, we will address two common backgrounds:

1. **Did not program before** – see Section 2.1.
2. **Did program before** – see Section 2.2.

2.1 For non-programmers

In the first background, the common underlying story is the following.

Data science has captivated you, making you interested in learning what is it all about and how can you use it to build your career in academia or industry. Then, you try to find resources to learn this new craft and you stumble into a world of intricate acronyms: `pandas`, `dplyr`, `data.table`, `numpy`, `matplotlib`, `ggplot2`, `bokeh`, and the list goes on and on...

Out of the blue you hear a name: “Julia.” What is this? How is it any different from other stuff that I’ve been told that I need to do data science?

Why should I dedicate my precious time into learning a language that is almost never mentioned in any job listing, lab position, postdoc offer or academic job description?. The answer is that **Julia is a fresh approach** both to programming and data science. Everything that you do in Python or in R you can do it in Julia with the advantage of using readable¹, fast, and powerful code syntax. Therfore, we see that the Julia language is gaining traction, and for good reasons.

So, if you don’t have any programming background knowledge, we highly encourage you to take up Julia as a first programming language and data science framework.

2.2 For programmers

In the second background, the common underlying story changes a little bit. You are someone who knows how to program and probably does this for a living. You are well-verses (or even fluent) in Python or R. You’ve heard about this new flashy thing called “data science” and you wanna jump on the band-wagon. You begin to learn how to do stuff in `numpy`, how to manipulate `DataFrames` in `pandas` and how

¹no C++ or FORTRAN API calls.

to plot things in `matplotlib`. Or maybe you've learned all that in R by using the `tidyverse` and `tibbles`, `data.frames`, `%>%` (pipes) and `geom_*`...

Then, from someone or somewhere you become aware about this new language called "Julia." Why bother? You are already proficient in Python or R and you can do everything that you need. Well, let us contemplate some plausible scenarios.

Have you ever in Python or R:

1. Done something and had to wait minutes for it to finish? Well, **in Julia Python or R minutes can be translated to seconds**². We reserved Section 2.4 for displaying successful Julia use cases in both academia and industry.
2. Tried to do something different than `numpy/dplyr` conventions and discovered that your code is slow and you'll probably have to learn dark magic³ to make it faster? **In Julia you can do your custom different stuff without loss of performance**.
3. Had to debug code and somehow you see yourself reading a Fortran or C/C++ source code and having no idea what you are trying to accomplish? **In Julia you only read Julia code**⁴, **no need to learn another language to make your original language fast**. This is called the "two-language problem" (see Section 2.3.2). It also covers the use case for when "you had an interesting idea and wanted to contribute to an open-source package and gave up because almost everything is not in Python or R but in C/C++ or Fortran"⁵.
4. Wanted to use a data structure defined in a package in another package and found that doesn't work and you'll probably need to build an interface⁶. **Julia allows users to easily share and reuse code from different packages**. Most of Julia user-defined types and functions work right out of the bat⁷ and some users are even marvelled upon discovering how their packages are being used by other libraries in ways that they could not have imagined. We have some examples in Section 2.3.3.
5. Needed to have a better project management, with dependecies and version control tightly controlled, manageable and replicable? **Julia has an amazing project management solution allied to a great package manager**. Unlike traditional package managers, which install and manage a single global set of packages. Julia package manager is designed around "environments": independent sets of packages that can be local to an individual project or shared. Each project maintains its own independent set of package versions. We'll talk more about how to manage your projects and packages in the Appendix 8.2.

If we got your attention by exposing somewhat familiar or plausible situations, you might be interested to learn more about this newcomer called Julia.

Let's proceed then!

²and sometimes milliseconds.

³numba, or even Rcpp or cython?

⁴no C++ or FORTRAN API calls.

⁵have a look at some deep learning libraries in GitHub and you'll be surprised that Python is only 25%-33% of the codebase.

⁶this is most a Python ecosystem problem, while R doesn't suffer heavily from this is not blue skies either.

⁷or with little effort necessary.

2.3 What Julia tries to accomplish?

NOTE: In this section we will explain the bells and whistles about what makes Julia shine as a programming language. If it becomes too technical for you, you can skip and go straight to Section 4 to learn about tabular data with `DataFrames.jl`.

Julia (Bezanson et al., 2017) is a relatively new language, first released in 2012, aims to be **both easy and fast**. It “runs like C⁸ but reads like Python” (Perkel, 2019). It was made for scientific computing, capable of handling **large amounts of data and computation** while still being fairly **easy to manipulate, create and prototype code**.

The creators of julia explained why they created Julia in a 2012 blogpost. Their motivations show how Julia was designed to be:

We are greedy: we want more. We want a language that’s open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that’s homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like Matlab. We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as Matlab, as good at gluing programs together as the shell. Something that is dirt simple to learn, yet keeps the most serious hackers happy. We want it interactive and we want it compiled.

Most users are attracted to Julia because of the **superior speed**. After all, Julia is a member of a prestigious and exclusive club. The **petaflop club** is comprised of languages who can exceed speeds of **one petaflop⁹ per second at peak performance**. Currently only C, C++, Fortran and Julia belong to the petaflop club.

But speed is not all that Julia can deliver. The **ease of use, unicode support** and a language that makes **code sharing effortless** is some of Julia’s features. We’ll address all those features in this section, but we want to focus on the Julia code sharing feature for now.

Julia ecosystem of packages is something unique. It enables not only code sharing but also allows sharing of user-created types. For example, Python’s `pandas` uses its own `Datetime` type to handle dates. The same with R tidyverse’s `lubridate` package, which also defines its own `datetime` type to handle dates. Julia doesn’t need any of this, it has all the date stuff already baked onto its standard library. This means that other packages don’t have to worry about dates. They just have to extend dates type to new functionalities by defining new functions and not having to touch a hair on creating new types. Julia’s `Dates` can do amazing stuff, but we are getting ahead of ourselves now. Let’s talk about the other Julia’s features.

2.3.1 Julia versus other programming languages

In Figure 2.1 a highly opinionated representation it is shown, that divides the main open-source¹⁰ scientific computing languages in a 2x2 diagram with two axes: Slow-Fast and Easy-Hard.

⁸sometimes even faster than C.

⁹a petaflop is one thousand trillion, or one quadrillion, operations per second.

¹⁰we have a very strong bias towards open-source and we won’t discuss paid programming languages such as Matlab.

We've put C++ and FORTRAN in the hard and fast quadrant. Being static languages that needs compilation, type checking and other professional care and attention, they are really hard to learn and slow to prototype. The advantage is that they are **really fast** languages.

R and Python goes into the easy and slow quadrant. They are dynamic languages that are not compiled and executes in runtime. Because of this they are really easy to learn and fast to prototype. Of course, this come with a disadvantage: they are **really slow** languages.

Julia is the only language in the easy and fast quadrant. We don't know any other serious language that would want to be hard and slow, so this quadrant is left empty.

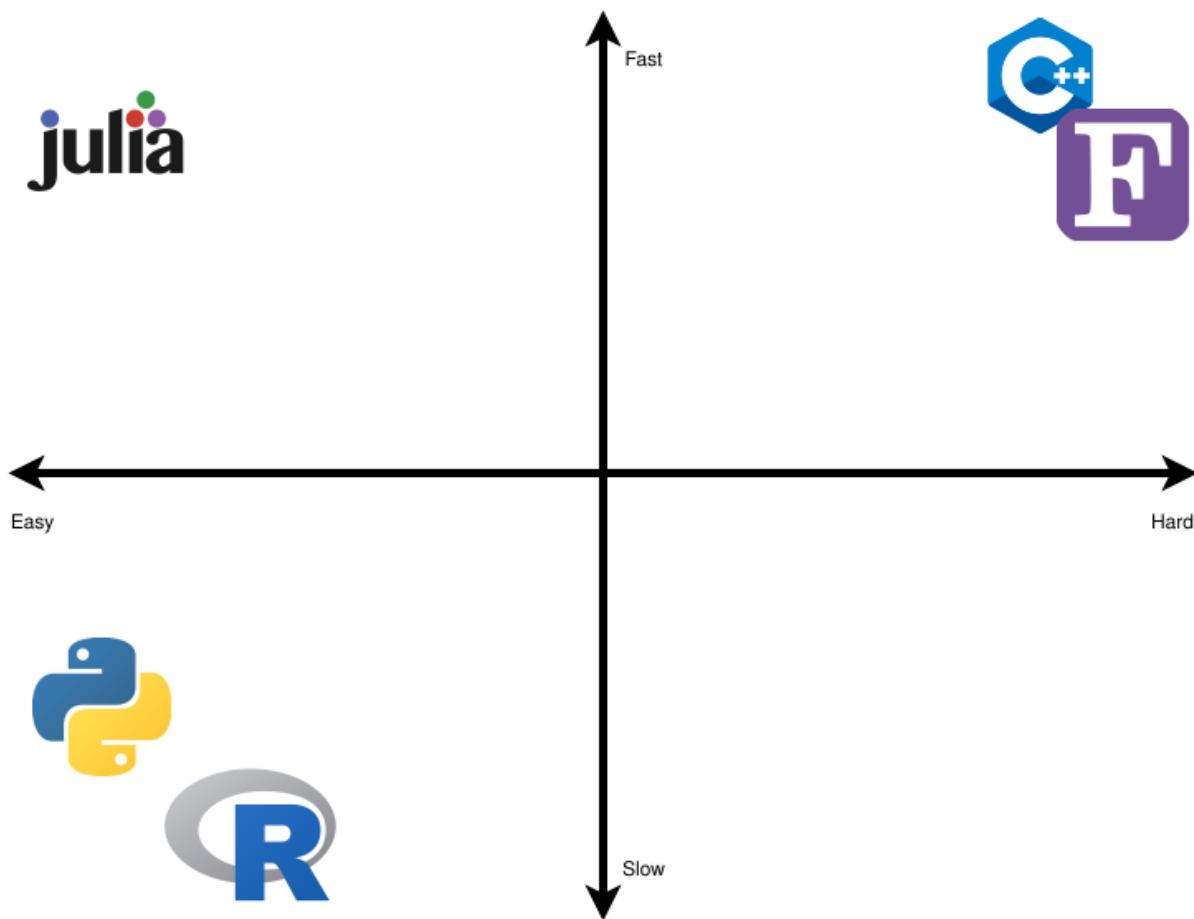


Figure 2.1: Scientific Computing Language Comparisons: logos for FORTRAN, C++, Python, R and Julia.

Julia is fast! Very fast! It was made for speed from the drawing board. It accomplishes this by exposing native machine code to the compiler. The compiler in turns does what it does best: optimizes machine code¹¹. You can find benchmarks for Julia and several other languages here. Figure 2.2 was taken from Julia's website benchmarks section¹². As you can see Julia is **indeed** fast.

¹¹if you like to learn more about how Julia is designed you should definitely check Bezanson et al. (2017).

¹²please note that the Julia results depicted above do not include compile time.

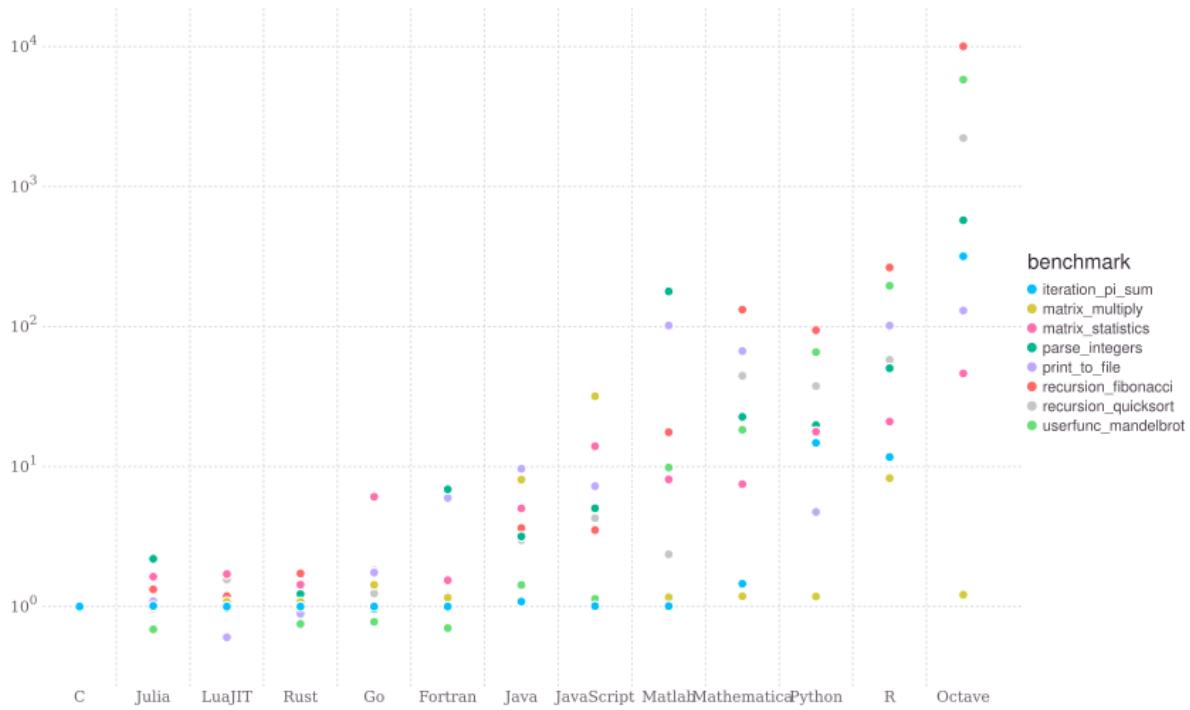


Figure 2.2: Julia versus other programming languages.

We really believe in Julia. (Otherwise, we wouldn't be writing this book.) We think that Julia is the **future in scientific computing and scientific data analysis**. It enables the user to develop rapid and powerful code with a simple syntax. Usually, researchers develop code by prototyping using a very easy, but slow, language. Once the code is assured to run correct and fulfill its goal, then begins the process of converting the code to a fast, but hard, language. This is known as the "Two-Language Problem" and we discuss next.

2.3.2 The Two-Language Problem

The "Two-Language Problem" is a very typical situation in scientific computing where a researcher devises an algorithm or a solution to tackle a desired problem or analysis at hand. Then, the solution is prototyped in an easy to code language (like Python or R). If the prototype works the researcher would code in a fast language that would not be easy to prototype (C++ or FORTRAN). Thus, we have two languages involved in the process of developing a new solution. One which is easy to prototype but is not suited for implementation (mostly due to being slow). And another one which is not so easy to code (and consequently not easy to prototype) but suited for implementation (mostly because it is fast). Julia comes to eliminate such situations by being the **same language that you prototype (ease of use) and implement the solution (speed)**.

Also, Julia lets you use **unicode characters as variables or parameters**. This means no more using `sigma` or `sigma_i`, and instead just use σ or σ_i as you would in mathematical notation. When you see code for an algorithm or for a mathematical equation you see almost the same notation and idiom. We call this feature "**One-To-One Code and Math Relation**", while usually downplayed can be a powerful

feature.

We think that the “Two-Language problem” and the “One-To-One Code and Math Relation” are best described by one of the creators of Julia, Alan Edelman, in a TEDx Talk (TEDx Talks, 2020) (if you are reading the printed book or a static PDF please click on the link to go the video or check the citation):

2.3.3 Multiple Dispatch

To explain multiple dispatch, we’ll give an illustrative example in **Python**. Suppose that you want to have different types of researcher that will inherit from a “base” class `Researcher`. The base class `Researcher` would define the initial common values for every derived class, namely `name` and `age`. These would go inside the default constructor method `__init__`:

```
class Researcher:
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age
```

Now let us define a `Linguist` class that will inherit from the `Researcher` class. We will also define a method `citation` that returns the citation style that is mostly used in the linguistics research field.

```
class Linguist(Researcher):
    def citation(self):
        return "APA"
```

We do the same for the `ComputerScientist` class, but with different citation style:

```
class ComputerScientist(Researcher):
    def citation(self):
        return "IEEE"
```

Finally, let’s instantiate our two researchers, Noam Chomsky a linguist and Judea Pearl a computer scientist with their respective ages:

```
noam = Linguist("Noam Chomsky", 92)
judea = ComputerScientist("Judea Pearl", 84)
```

Now suppose you want to define a function that will have *different behaviors* depending on the *argument’s types*. For example, a linguist researcher might approach a computer scientist researcher and ask him to collaborate on a new paper. The approach would be different if the situation were the opposite: a computer scientist researcher approaching the linguist researcher. Below, you’ll see that we defined two functions that should behave differently in both approaches:

```
def approaches(li: Linguist, cs: ComputerScientist):
    print(f"Hey {cs.name}, wanna do a paper together? We need to use {li.citation()} style.")

def approaches(cs: ComputerScientist, li: Linguist):
    print(f"Hey {li.name}, wanna do a paper together? We need to use {cs.citation()} style.")
```

Now lets say Noam Chomsky approaches Judea Pearl with a paper idea:

```
approaches(noam, judea)
```

```
Hey Judea Pearl, wanna do a paper? We need to use APA style.
```

That was not what `judea` as a `Linguist` type would say to `noam`, a `ComputerScientist` type. This is **single dispatch** and is the default feature available on most object-oriented languages, like Python or C++. Single dispatch just act on the first argument of a function. Since both of our researchers `noam` and `judea` are instantiate as types inherited from the same base type `Researcher` we cannot implement what we are trying to do in Python. You would need to change your approach with a substancial loss of simplicity (you would probably need to create different functions with different names).

Now let's do this in Julia.

In Julia, we don't have classes but we have structures (`struct`) that are meant to be “structured data”: they define the kind of information that is embedded in the structure, that is a set of fields (i.e. “properties” or “attributes” in other languages), and then individual instances (or “objects”) can be produced each with its own specific values for the fields defined by the structure. They differ from the primitive types (e.g. integer and floats) that are by default defined already inside the core of the Julia language. Thus, they are known as user-defined types. The user can only create new abstract types or structs (which are also known as composite types). In Julia, all structs are final and may only have abstract types as their supertypes.

First we'll create an abstract type named `Researcher`.

```
abstract type Researcher end
```

We proceed, similar as before, by creating two derived structs from the `Researcher` abstract type. Note that the `<:` operator is the subtype operator to assign that a struct or type is a subtype of another struct or type, which in turns would become a supertype (and we have the analogous `>:` operator). We also create two field names, one for the researcher `name` and other for `age`. They are represented as strings and 64-bit integers, respectively:

```
struct Linguist <: Researcher
    name::AbstractString
    age::Int64
end
```

```
struct ComputerScientist <: Researcher
    name::AbstractString
    age::Int64
end
```

The final step is to define two new functions that will behave differently depending on which derived struct of `Researcher` are the first or second argument. We also use `$` for string interpolation of the researcher's name:

```
approaches(li::Linguist, cs::ComputerScientist) = "Hey $(cs.name), wanna do a paper? We need to
use APA style."
```

```
approaches(cs::ComputerScientist, li::Linguist) = "Hey $(li.name), wanna do a paper? We need to
use IEEE style."
```

```
approaches (generic function with 2 methods)
```

Finally, let's instantiate our two researchers, `noam` and `judea`, as we did before in the Python case:

```
noam = Linguist("Noam Chomsky", 92)
judea = ComputerScientist("Judea Pearl", 84)
```

Again, let's see what Noam Chomsky will say when he approaches Judea Pearl with a paper idea:

```
approaches(noam, judea)
```

Hey Judea Pearl, wanna do a paper? We need to use APA style.

Perfect! It behaves just as we wanted! This is **multiple dispatch** and it is an important feature in Julia. Multiple dispatch acts on all arguments of a function and defines function behavior based on all argument's types.

Multiple dispatch is a powerful feature that allows us also to extend existing functions or to define custom and complex behavior to new types. To show how this works, we'll use another example. Suppose you want to define two new `structs` for two different animals. For simplicity, we won't be adding fields for the `structs`:

```
struct fox end
struct chicken end
```

Next, we want to define addition for both `fox` and `chicken` new types. We proceed by defining a new function signature of the `+` operator from the `Base` module of Julia¹³:

```
import Base: +
+(F::fox, C::chicken) = "trouble"
+(C1::chicken, C2::chicken) = "safe"
```

```
+ (generic function with 361 methods)
```

Now let us call addition with the `+` sign on instantiated `fox` and `chicken` objects:

```
my_fox = fox()
my_chicken = chicken()
my_fox + my_chicken
```

```
trouble
```

And, as expected, adding two `chicken` objects together signals that they are safe:

¹³this is just an example for teaching purposes. Doing something similar as this example will probably result in a method invalidation.

```
chicken_1 = chicken()
chicken_2 = chicken()
chicken_1 + chicken_2
```

```
safe
```

This is the power of multiple dispatch: **we don't need everything from scratch for our custom-defined user types.** If you are excited as much as we are by multiple dispatch, here are two more in-depth examples. The first is a fast and elegant implementation of a one-hot vector by Storopoli (2021). The second is an interview of Christopher Rackauckas at Tanmay Bakshi YouTube's Channel (see from time 35:07 onwards) (tanmay bakshi, 2021). Chris mentions that, while using `DifferentialEquations.jl`, a package that he developed and currently maintains, a user reported that Julia has the best GPU-based quaternion ODE Solver and he said that he had no idea because he didn't even implemented it. Most of the merit is due to multiple dispatch and high user code/type sharing.

To conclude, we think that multiple dispatch is best explained by one of the creators of Julia, Stefan Karpinski, at JuliaCon 2019 (The Julia Programming Language, 2019) (if you are reading the printed book or a static PDF please click on the link to go the video or check the citation):

2.4 Julia in the Wild

In Section 2.3, we exposed why we think Julia is such a unique programming language. We showed simple examples about the main features of Julia. If you would like to have a deep dive on how Julia is being used, we have some **interesting use cases:**

1. NASA uses Julia in a supercomputer to analyze the “Largest Batch of Earth-Sized Planets Ever Found” and achieve a whopping **1,000x speedup** to catalog 188 million astronomical objects in 15 minutes.
2. The Climate Modeling Alliance (CliMa) is using mostly Julia to **model climate in the GPU and CPU**. Launched in 2018 in collaboration with researchers at Caltech, the NASA Jet Propulsion Laboratory, and the Naval Postgraduate School, CliMA is utilizing recent progress in computational science to develop an Earth system model that can predict droughts, heat waves, and rainfall with unprecedented precision.
3. US Federal Aviation Administration (FAA) is developing an **Airborne Collision Avoidance System (ACAS-X)** using Julia. This is a nice example of the “Two-Language Problem” (see Section 2.3). Previous solutions used Matlab to develop the algorithms and C++ for a fast implementation. Now, FAA is using one language to do all this: Julia.
4. **175x speedup** for Pfizer’s pharmacology models using GPUs in Julia. It was presented as a poster in the 11th American Conference of Pharmacometrics (ACoP11) and won a quality award.
5. The Attitude and Orbit Control Subsystem (AOCS) of the Brazilian satellite Amazonia-1 is **written 100% in Julia** by Ronan Arraes Jardim Chagas

6. Brazil's national development bank (BNDES) ditched a paid solution and opted for open-source Julia modeling and gained a **10x speedup**.

If this is not enough, there are more case studies in Julia Computing website.

3 Julia 101

NOTE: In this chapter we will cover the basics of Julia as a programming language. Please note that this is not *strictly necessary* for you to use Julia as a tool of data manipulation and data visualization. Having a basic understanding of Julia will definitely make you more *effective* and *efficient* in using Julia. However, if you prefer to get started straight away, you can jump to Section 4 to learn about tabular data with `DataFrames.jl`.

This is going to be a very brief and *not* in-depth overview of Julia language. If you are already familiar and comfortable with other programming languages, we highly encourage you to read Julia's documentation. It is a thoroughly deep dive into Julia. It covers all the basics and corner cases, but it can be cumbersome. Especially, if you are aren't familiar with open-source software documentation.

We'll cover the basics of Julia. Imagine that Julia is a fancy feature-loaded car, such as a brand-new Tesla. We'll just explain to you how to "drive the car, park it and how to navigate in traffic." If you want to know what all "buttons in the steering wheel and dashboard do," this is not the resource you are looking for.

3.1 Language Syntax

Julia is a **dynamic-typed language**. This means that you don't need to compile it before you run code, like you would do in C++ or FORTRAN. The main differences from Julia to other dynamic languages such as R and Python are the following.

First, Julia, contrary to most dynamic-typed languages, **enables the user to specify type declaration**. You already saw some types declarations in *Why Julia?* (Section 2): they are those double colon `::` that sometimes comes after variables. But if you don't want to specify what type are your variables or functions, Julia will gladly infer it for you.

Second, Julia also allows users to define function behavior across many combinations of argument types via multiple dispatch. We also covered multiple dispatch in Section 2.3. We defined a different type behavior by defining new function signatures for different type's argument while using the same function name.

3.1.1 Variables

Variables are values that you tell the computer to store with an specific name, so that you can later recover or change its value. Julia have several type of variables but what we most use in data science are:

- Integers: `Int64`
- Real Numbers: `Float64`
- Boolean: `Bool`
- Strings: `String`

Integers and real numbers have by default 64 bits, that's why they have the `64` suffix in their type. If you need more or less precision, there are `Int8` or `Int128` for example. Most of the time this won't be an issue and can be overwhelming having to deal with different types.

We create new variables by writing the variable name on the left and its value in the right, and in the middle we use the `=` assign operator. For example:

```
name = "Julia"
age = 9
```

```
9
```

Note that the return output of the last statement (`age`) was printed to the console. Here, we are defining two new variables: `name` and `age`. We can recover their values by typing its name given in the assignment:

```
name
```

```
Julia
```

If you want to define new values for an existing variable, you can repeat the steps in the assignment. Note that Julia will now override the previous variable's value with the new one. Suppose `jl` `name`'s birthday has passed and now it has turned `jl age+1`:

```
age = 10
```

```
10
```

We can do the same with its `name`, suppose that Julia has earned some titles due to its blazing speed. We would change the variable `name` to the new value:

```
name = "Julia Rapidus"
```

```
Julia Rapidus
```

We can also do operations on variables such as addition or division. Let's see how much months of age `jl name` has by multiplying `age` by 12:

```
age * 12
```

```
120
```

We can inspect variables types by using `typeof` function:

```
typeof(age)
```

```
Int64
```

The next question then becomes: “What else can I do with integers?” There is a nice handy function `methodswith` that spits out every function available, along with its signature, for a certain type. Here I will restrict the output to the first 5 rows:

```
first(methodswith(Int64), 5)
```

```
[1] fftfreq(n::Int64) in AbstractFFTs at /home/runner/.julia/packages/AbstractFFTs/JebmH/src/
    definitions.jl:459
[2] fftfreq(n::Int64, fs::Number) in AbstractFFTs at /home/runner/.julia/packages/AbstractFFTs/
    JebmH/src/definitions.jl:459
[3] rfftfreq(n::Int64) in AbstractFFTs at /home/runner/.julia/packages/AbstractFFTs/JebmH/src/
    definitions.jl:494
[4] rfftfreq(n::Int64, fs::Number) in AbstractFFTs at /home/runner/.julia/packages/AbstractFFTs
    /JebmH/src/definitions.jl:494
[5] to_subscript(x::Union{Int64, AbstractString}) in UnicodeFun at /home/runner/.julia/packages
    /UnicodeFun/QjvWn/src/sub_super_scripts.jl:1
```

3.1.2 User-defined Types

Having variables around without any sort of hierarchy and relationships are not ideal. In Julia we can define that kind of structured data with a `struct` (also known as composite types). Inside each `struct` there is an optional set of fields. They differ from the primitive types (e.g. integer and floats) that are by default defined already inside the core of Julia language. Since most `structs` are user-defined they are known as user-defined types.

For example lets create a `struct` to represent scientific open-source programming languages. We'll also define a set of fields along with the corresponding types inside the `struct`:

```
struct Language
    name::String
    title::String
    year_of_birth::Int64
    fast::Bool
end
```

To inspect the field names you can use the `fieldnames` and passing the desired `struct` as an argument:

```
fieldnames(Language)
```

```
(:name, :title, :year_of_birth, :fast)
```

To use `structs`, we must instantiate individual instances (or “objects”), each with its own specific values for the fields defined inside the `struct`. Let's instantiate two instances, one for Julia and one for Python with the appropriate types as fields inside the `Language` constructor:

```
julia = Language("Julia", "Rapidus", 2012, true)
python = Language("Python", "Letargicus", 1991, false)
```

```
Language("Python", "Letargicus", 1991, false)
```

One thing to note with `structs` is that we cannot change their values once they are instantiated. We can solve this with a `mutable struct`. Also, note that mutable objects will, generally, be slower and more error prone. Whenever possible make everything *immutable*. Let's create a `mutable struct`.

```
mutable struct MutableLanguage
    name::String
    title::String
    year_of_birth::Int64
    fast::Bool
end

julia MutableLanguage("Julia", "Rapidus", 2012, true)
```

```
MutableLanguage("Julia", "Rapidus", 2012, true)
```

Suppose that we want to change `julia MutableLanguage`'s title. Now we can do this, since `julia MutableLanguage` is an instantiated `mutable struct`:

```
julia MutableLanguage.title = "Python Obliteratus"

julia MutableLanguage
```

```
MutableLanguage("Julia", "Python Obliteratus", 2012, true)
```

3.1.3 Boolean Operators and Numeric Comparison

Now that we've covered types, we can move to boolean operators and numeric comparison.

We have three boolean operators in Julia:

- `!:` **NOT**
- `&&:` **AND**
- `||:` **OR**

Here are a few examples with some of them:

```
!true
```

```
false
```

```
(false && true) || (!false)
```

```
true
```

```
(6 isa Int64) && (6 isa Real)
```

```
true
```

Regarding numeric comparison, Julia has three major types of comparisons:

1. **Equality**: either something is *equal* or *not equal* another
 - == “equal”
 - != or ≠ “not equal”
2. **Less than**: either something is *less than* or *less than or equal to*
 - < “less than”
 - <= or ≤ “less than or equal to”
3. **Greater than**: either something is *greater than* or *greater than or equal to*
 - > “greater than”
 - >= or ≥ “greater than or equal to”

Here are some examples:

```
1 == 1
```

```
true
```

```
1 >= 10
```

```
false
```

It even works between different types:

```
1 == 1.0
```

```
true
```

We can also mix and match boolean operators with numeric comparisons:

```
(1 != 10) || (3.14 <= 2.71)
```

```
true
```

3.1.4 Functions

Now that we already know how to define variables and custom types as `structs`, let's turn our attention to **functions**. In Julia, a function is an **object that maps argument's values to a return value**. The basic syntax goes something like this:

```
function f_name(arg1, arg2)
    stuff_done = stuff with the arg1 and arg2
    return stuff_done
end
```

Every function declaration begins with the keyword `function` followed by the function name. Then, inside parentheses `()`, we define the arguments separated by a comma `,`. Inside the function, we specify

what we want Julia to do with the parameters that we supplied. After all the operations that we want the function to do has been performed, we ask Julia to return the result of those operations with the `return` statement. Finally, we let Julia know that our function is well-defined and finished with the `end` keyword.

There is also the compact **assignment form**:

```
f_name(arg1, arg2) = stuff with the arg1 and arg2
```

It is the **same function** as before but with a different, more compact, form. Let's dive into some examples.

3.1.4.1 Creating new Functions

Let's create a new function that adds number together:

```
function add_numbers(x, y)
    return x + y
end
```

```
add_numbers (generic function with 1 method)
```

Now, we can use our `add_numbers` function:

```
add_numbers(17, 29)
```

```
46
```

And it works also with floats:

```
add_numbers(3.14, 2.72)
```

```
5.86
```

We can also define custom behavior by specifying types declarations. Suppose we want to have a `round_number` function that behaves differently if its argument is either a `Float64` or `Int64`:

```
function round_number(x::Float64)
    return round(x)
end

function round_number(x::Int64)
    return x
end
```

```
round_number (generic function with 2 methods)
```

We can see that it is a function with multiple methods:

```
methods(round_number)
```

```
# 2 methods for generic function "round_number":  
[1] round_number(x::Float64) in Main at none:1  
[2] round_number(x::Int64) in Main at none:5
```

There is one issue: what happens if we want to round a 32-bit float `Float32`? Or a 8-bit integer `Int8`?

If you want something to function on all float's and integer's types you can use an abstract type as type signature, such as `AbstractFloat` OR `Integer`:

```
function round_number(x::AbstractFloat)  
    return round(x)  
end
```

```
round_number (generic function with 3 methods)
```

Now, it works as expected with any float type:

```
x_32 = Float32(1.1)  
round_number(x_32)
```

```
1.0
```

NOTE: We can inspect types with the `supertypes` and `subtypes` functions.

Let's go back to our `Language` struct that we defined above. This is an example of multiple dispatch. We will extend the `Base.show` function that prints the output of instantiated types and `structs`.

By default a `struct` has a basic output, which you saw above in the `python` case. We can define `Base.show` function to our `Language` type, so that we have some nice printing for our programming languages instances. We want to clearly communicate programming languages' names, titles and ages in years of old. The function `Base.show` accepts as arguments a `IO` type named `io` followed by the type you want to define custom behavior:

```
Base.show(io::IO, l::Language) = print(  
    io, l.name, " ",  
    2021 - l.year_of_birth, ", years old, ",  
    "has the following titles: ", l.title  
)
```

Now let's see how `python` will output:

```
python
```

```
Python 30, years old, has the following titles: Letargicus
```

3.1.4.2 Multiple Return Values

A function can, also, return two or more values. See the new function `add_multiply` below:

```
function add_multiply(x, y)
```

```

addition = x + y
multiplication = x * y
return addition, multiplication
end

```

```
add_multiply (generic function with 1 method)
```

In that case we can do two things:

1. We can, analogously as the return values, define two variables to hold the function return values, one for each return value:

```

return_1, return_2 = add_multiply(1, 2)
return_2

```

```
2
```

2. Or we can define just one variable to hold the function return values and access them with either `first` or `last`:

```

all_returns = add_multiply(1, 2)
last(all_returns)

```

```
2
```

3.1.4.3 Keyword Arguments

Some functions can accept keywords arguments instead of positional arguments. These arguments are just like regular arguments, except that they are defined after the regular function's arguments and separated by a semicolon ;. Another difference is that we must supply a **default value** for every keyword argument. For example, let's define a `logarithm` function that by default uses base e (2.718281828459045) as a keyword argument. Note that here we are using the abstract type `Real` so that we cover all types derived from `Integer` and `AbstractFloat`, being both themselves subtypes of `Real`:

```
AbstractFloat <: Real && Integer <: Real
```

```
true
```

```

function logarithm(x::Real; base::Real=2.7182818284590)
    return log(base, x)
end

```

```
logarithm (generic function with 1 method)
```

It works without specifying the `base` argument:

```
logarithm(10)
```

```
2.3025850929940845
```

And also with the keyword argument `base` different from its default value:

```
logarithm(10; base=2)
```

```
3.3219280948873626
```

3.1.4.4 Anonymous Functions

A lot of times, we don't care about the name of the function and want to quickly make one. What we need are **anonymous functions**. They come a lot in Julia's data science workflow. For example, when using `DataFrames.jl` (Section 4) or `Plots.jl` (Section 6), sometimes we need a quick and dirty function to filter data or format plot labels. That's when we reach out for anonymous functions. They are specially useful when we don't want to create a function and a simple in-place statement would be enough.

The syntax is simple. We use the `->` operator. On the left of `->` we define the parameter name. And on the right of `->` we define what operations we want to perform on the parameter we defined on the left of `->`. Here's an example, suppose we want to undo the log transformation by using an exponentiation:

```
map(x -> 2.7182818284590^x, logarithm(2))
```

```
2.0
```

Here, we are using the `map` function to conveniently map the anonymous function (first argument) to `logarithm(2)` (the second argument). As a result, we get back the same number, because `logarithm` and `exponentiation` are inverse (at least in the base that we've chosen – 2.7182818284590)

3.1.5 Conditional If-Else-Elseif

In most programming languages, the user is allowed to control the computer's flow of execution. Depending on the situation, we want the computer to do one thing or another. In Julia we can control the flow of execution with `if`, `elseif` and `else` keywords. There are known as conditional syntax.

`if` keyword prompts Julia to evaluate an expression and depending whether `true` or `false` certain portions of code will be executed. We can compound several `if` conditions with the `elseif` keyword, for complex and nuanced control flow. Finally we can define an alternative portion to be executed if anything inside the `if` or `elseifs` are evaluated as `true`. This is the purpose of the `else` keyword. Finally, like all the previous keyword operators that we saw, we must tell Julia when the conditional statement is finished with the `end` keyword.

Here's an example with all the `if-elseif-else` keywords:

```
a = 1
b = 2

if a < b
    "a is less than b"
elseif a > b
    "a is greater than b"
```

```
else
    "a is equal to b"
end
```

```
a is less than b
```

We can even wrap this in a function called `compare`:

```
function compare(a, b)
    if a < b
        "a is less than b"
    elseif a > b
        "a is greater than b"
    else
        "a is equal to b"
    end
end

compare(3.14, 3.14)
```

```
a is equal to b
```

3.1.6 For Loop

The classical for loop in Julia follow a similar syntax as the conditional statements. You begin with a keyword, in this case `for`. Then you specify what Julia should “loop” for, i.e. a sequence. Also, like everything else, you must finish with the `end` keyword.

So, to make Julia print every number from 1 to 10, you’ll need the following for loop:

```
for i in 1:10
    println(i)
end
```

3.1.7 While Loop

The while loop is a mix of the previous conditional statements and for loops. Here the loop is executed everytime the condition is `true`. The syntax follows the same fashion as the previous one. We begin with the keyword `while`, followed by the statement to evaluated as either `true`. Like previously, you must end with the `end` keyword.

Here’s an example:

```
n = 0

while n < 3
    global n += 1
end

n
```

```
3
```

As you can see we have to use the `global` keyword. This is because of **variable scope**. Variables defined inside conditional statements, loops and functions exist only inside it. This is known as the *scope* of the variable. Here we had to tell Julia that the `n` inside `while` loop is in the global scope with the `global` keyword.

Finally, we used also the `+=` operator which is a nice short hand for `n = n + 1`.

3.2 Native Data Structures

Julia have several native data structures. They are abstractions of data that represent somehow structured data. We will cover the most used ones. They hold homogeneous or heterogeneous data. Since they are collections, they can be *looped* over with the `for` loops.

We will cover `String`, `Tuple`, `NamedTuple`, `UnitRange`, `Arrays`, `Pair`, `Dict`, `Symbol`.

When you stumble into a data structure in Julia, you can find functions/methods that accept it as an argument with the `methodswith` function. Just like we did before with types. This is a nice thing to have in your bag of tricks. We personally use it often. Let's see what we can do with an `String` for example:

```
first(methodswith(String), 5)
```

```
[1] crc32c(s::String) in CRC32c at /opt/hostedtoolcache/julia/1.6.2/x64/share/julia/stdlib/v1
    .6/CRC32c/src/CRC32c.jl:39
[2] crc32c(s::String, crc::UInt32) in CRC32c at /opt/hostedtoolcache/julia/1.6.2/x64/share/
    julia/stdlib/v1.6/CRC32c/src/CRC32c.jl:39
[3] getindex(p::Base.BinaryPlatforms.AbstractPlatform, k::String) in Base.BinaryPlatforms at
    binaryplatforms.jl:137
[4] getindex(s::String, i::Int64) in Base at strings/string.jl:222
[5] getindex(s::String, r::UnitRange{Int64}) in Base at strings/string.jl:257
```

3.2.1 Broadcasting Operators and Functions

Before we dive into data structures, we need to talk about broadcasting (also known as *vectorization*) and the “dot” operator ..

For mathematical operation, like `*` (multiplication) or `+` (addition), we can broadcast it using the dot operator. For example, broadcasted addition would imply in changing the `+` to `.+`:

```
[1, 2, 3] .+ 1
```

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

It also works with functions automatically. Remember our `logarithm` function?

```
logarithm.([1, 2, 3])
```

```
[0.0, 0.6931471805599569, 1.0986122886681282]
```

3.2.1.1 Functions with a bang !

In Julia syntax it is common to append ! to names of functions that modify their arguments. This is a convention that warns the user that the function is **not pure**, i.e. it has *side effects*. A function with side effects is useful when you want to update a large data structure or variable container without having all the overhead from creating a new instance.

Most of the ! functions' arguments are data structures.

For example, we can create a function that adds 1 to its argument:

```
function add_one!(x)
    for i in 1:length(x)
        x[i] += 1
    end
    return nothing
end
```

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

add_one!(my_data)

my_data
```

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

3.2.2 String

Strings are represented delimited by double quotes:

```
typeof("This is a string")
```

```
String
```

We can also write a multiline string:

```
s = "
This is a big multiline string.
As you can see.
It is still a String to Julia.
"
```

```
This is a big multiline string.
As you can see.
It is still a String to Julia.
```

But, it is typically more clear to use triple-backticks:

```
s = """  
    This is a big multiline string.  
    As you can see.  
    It is still a String to Julia.  
"""
```

```
This is a big multiline string.  
As you can see.  
It is still a String to Julia.
```

When using triple-backticks, the indentation and newline at the start is ignored by Julia. This helps code readability.

3.2.2.1 String Concatenation

A common string operation is **string concatenation**. Suppose you want to construct a new string that is the concatenation of two or more strings. This is accomplished in Julia either with the `*` operator (this is where Julia departs from other scientific open-source programming languages) or the `join` function:

```
hello = "Hello"  
goodbye = "Goodbye"  
  
hello * goodbye
```

```
HelloGoodbye
```

As you can see we are missing a space between `hello` and `goodbye`. We could concatenate an additional `" "` string with the `*`, but that would be cumbersome for more than two strings. That's when the `join` function comes up. We just pass as arguments the strings inside the brackets `[]` and the separator:

```
join([hello, goodbye], " ")
```

```
Hello Goodbye
```

3.2.2.2 String Interpolation

Concatenating strings can be convoluted. We can be much more expressive with **string interpolation**. It works like this: you specify whatever you want to be included in your string with the dollar sign `$`. Here's the example before but now using interpolation:

```
"$hello $goodbye"
```

```
Hello Goodbye
```

It works even inside functions. Let's revisit our `test` function from Section 3.1.5:

```
function test_interpolated(a, b)  
    if a < b
```

```

    "$a is less than $b"
elseif a > b
    "$a is greater than $b"
else
    "$a is equal to $b"
end
end

test_interpolated(3.14, 3.14)

```

3.14 is equal to 3.14

3.2.2.3 String Manipulations

There are several functions to manipulate strings in Julia. We will demonstrate the most common ones. Also, note that most of these functions accepts a Regular Expression (RegEx) as arguments. We won't cover RegEx in this book, but you are encouraged to learn about them, specially if most of your work uses textual data.

First, let us define a string for us to play around with:

```
julia_string = "Julia is an amazing opensource programming language"
```

Julia is an amazing opensource programming language

1. `occursin`, `startswith` and `endswith`: A conditional (returns either `true` or `false`) if the first argument is a:

- **substring** of the second argument

```
occursin("Julia", julia_string)
```

true

- **prefix** of the second argument

```
startswith("Julia", julia_string)
```

false

- **suffix** of the second argument

```
endswith("Julia", julia_string)
```

false

2. `lowercase`, `uppercase`, `titlecase` and `lowercasefirst`:

```
lowercase(julia_string)
```

```
julia is an amazing opensource programming language
```

```
uppercase(julia_string)
```

```
JULIA IS AN AMAZING OPENSOURCE PROGRAMMING LANGUAGE
```

```
titlecase(julia_string)
```

```
Julia Is An Amazing Opensource Programming Language
```

```
lowercasefirst(julia_string)
```

```
julia is an amazing opensource programming language
```

3. `replace`: introduces a new syntax, called the Pair

```
replace(julia_string, "amazing" => "awesome")
```

```
Julia is an awesome opensource programming language
```

4. `split`: breaks up a string by a delimiter:

```
split(julia_string, " ")
```

```
SubString{String}["Julia", "is", "an", "amazing", "opensource", "programming", "language"]
```

3.2.2.4 String Conversions

Often, we need to convert between types in Julia. We can use the `string` function:

```
my_number = 123  
typeof(string(my_number))
```

```
String
```

Sometimes, we want the opposite: convert a string to a number. Julia has a handy function for that: `parse`

```
typeof(parse(Int64, "123"))
```

```
Int64
```

Sometimes, we want to play safe with these conversions. That's when `tryparse` function steps in. It has the same functionality as `parse` but returns either a value of the requested type, or `nothing`. That makes `tryparse` handy when we want to avoid errors. Of course, you would need to deal with all those `nothing` values afterwards.

```
tryparse(Int64, "A very non-numeric string")
```

```
nothing
```

3.2.3 Tuple

Julia has a data structure called **tuple**. They are really *special* in Julia because they are *closely related* to functions. Since functions are an important feature in Julia, every Julia user should know the basics of tuples.

A tuple is a **fixed-length container that can hold any type of value**. A tuple is an **immutable object**, meaning that it cannot be modified after instantiation. To construct a tuple you use parentheses () to delimitate the beginning and end, along with commas , as value's delimiters:

```
my_tuple = (1, 3.14, "Julia")
```

```
(1, 3.14, "Julia")
```

Here we are creating a tuple with three values. Each one of the values is a different type. We can access them via indexing. Like this:

```
my_tuple[2]
```

```
3.14
```

We can also loop over tuples with the `for` keyword. And even apply functions to tuples. But we can **never change any value of a tuple**, since they are **immutable**.

The relationship between tuples and functions is a very important one. Remember functions that return multiple values back in Section 3.1.4.2? Let's inspect what our `add_multiply` function returns:

```
return_multiple = add_multiply(1, 2)
typeof(return_multiple)
```

```
Tuple{Int64, Int64}
```

So, now you can see how they are related. **Functions that return multiple arguments do so by returning a Tuple** with the types inside the {} brackets.

One more thing about tuples. **When you want to pass more than one variable to an anonymous function, guess what you would need to use? Once again: tuples!**

```
map((x, y) -> x^y, 2, 3)
```

```
8
```

Or even more than two arguments:

```
map((x, y, z) -> x^y + z, 2, 3, 1)
```

```
9
```

3.2.4 Named Tuple

Sometimes you want to name the values in tuples. That's when **named tuples** comes in. Their functionality is pretty much same the same as tuples: they are **immutable** and can hold **any type of value**.

Named tuple's construction are slightly different from tuples. You have the familiar parentheses () and comma , value separator. But now you must **name the values**:

```
my_namedtuple = (i=1, f=3.14, s="Julia")
```

```
(i = 1, f = 3.14, s = "Julia")
```

We can access a named tuple's values via indexing like regular tuples or, alternatively, **access by their names** with the .:

```
my_namedtuple.s
```

```
Julia
```

To finish named tuples, there is one important *quick* syntax that you'll see a lot in Julia code. Often Julia users create a named tuple by using the familiar parenthesis () and commas , but without naming the values. To do so you **begin the named tuple construction by specifying first a semicolon ; before the values**. This is specially useful when the values that would compose the named tuple are already defined in variables:

```
i = 1  
f = 3.14  
s = "Julia"
```

```
my_quick_namedtuple = (; i, f, s)
```

```
(i = 1, f = 3.14, s = "Julia")
```

3.2.5 Ranges

A range in Julia represents an interval between a start and stop boundaries. The syntax is `start:stop`:

```
1:10
```

```
1:10
```

As you can see our instantiated range is of type `UnitRange{T}` where `T` is the type inside the `UnitRange`:

```
typeof(1:10)
```

```
UnitRange{Int64}
```

And, if we gather all the values, we get:

```
[x for x in 1:10]
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We can construct ranges also for other types:

```
typeof(1.0:10.0)
```

```
StepRangeLen{Float64, Base.TwicePrecision{Float64}, Base.TwicePrecision{Float64}}
```

Sometimes, we want to change the default interval stepsize behavior. We can do that by adding a step-size in the range syntax `start:step:stop`. For example, suppose we want a range of `Float64` from 0 to 1 with steps of size 0.2:

```
0.0:0.2:1.0
```

```
0.0:0.2:1.0
```

If you want to “materialize” a `UnitRange` into a collection you can use the function `collect`:

```
collect(1:10)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We have an array of the type specified in the `UnitRange` between the boundaries that we’ve set. Speaking in arrays, let’s talk about them.

3.2.6 Array

Arrays are a **systematic arrangement of similar objects, usually in rows and columns**. Most of the time you would want **arrays of a single type for performance issues**, but note that they can also hold objects of different types. They are the “bread and butter” of data scientist, because arrays are what constitutes most of **data manipulation** and **data visualization** workflows.

Arrays are a powerful data structure. They are one of the main features that makes Julia blazing fast.

3.2.6.1 Array Types

Let’s start with arrays types. There are several, but we will focus on two the most used in data science:

- `Vector{T}`: **one-dimensional** array. Alias for `Array{T, 1}`.
- `Matrix{T}`: **two-dimensional** array. Alias for `Array{T, 2}`.

Note here that `T` is the type of the underlying array. So, for example, `Vector{Int64}` is a vector which all elements are `Int64`s and `Matrix{AbstractFloat}` is a matrix which all elements are subtypes of `AbstractFloat`.

Most of the time, specially when dealing with tabular data, we are using either one- or two-dimensional arrays. They are both `Array` types for Julia. But we can use the handy aliases `Vector` and `Matrix` for clear and concise syntax.

3.2.6.2 Array Construction

How do we construct an array? The simplest answer is to use the **default constructor**. It accepts the element type as the type parameter inside the {} brackets and inside the constructor you'll pass the element type followed by the desired dimensions. It is common to initialize vector and matrices with undefined elements by using the `undef` argument for type. A vector of 10 `undef` `Float64` elements can be constructed as:

```
my_vector = Vector{Float64}(undef, 10)
```

```
[6.94515436342974e-310, 0.0, 6.94515436750954e-310, 6.94515436342974e-310, 0.0,
 6.94515436750954e-310, 6.94515436342974e-310, 0.0, 6.94515436750954e-310, 6.94515436342974
e-310]
```

For matrices, since we are dealing with two-dimensional objects, we need to pass two dimensions arguments inside the constructor: one for **rows** and another for **columns**. For example, a matrix with 10 rows, 2 columns and `undef` elements can be instantiate as:

```
my_matrix = Matrix{Float64}(undef, 10, 2)
```

```
10×2 Matrix{Float64}:
 6.94515e-310 0.0
 0.0 6.94515e-310
 6.94515e-310 6.94516e-310
 6.94515e-310 0.0
 0.0 6.94515e-310
 6.94515e-310 6.94515e-310
 6.94515e-310 0.0
 0.0 6.94515e-310
 6.94516e-310 6.94515e-310
 6.94516e-310 0.0
```

We also have some **syntax aliases** for the most common elements in array construction:

- `zeros` for all elements being initialized to value zero. Note that the default type is `Float64` which can be changed if necessary:

```
my_vector_zeros = zeros(10)
```

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
my_matrix_zeros = zeros(Int64, 10, 2)
```

```
10×2 Matrix{Int64}:
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
```

```
0 0
0 0
0 0
```

- `ones` for all elements being initialized to value one:

```
my_vector_ones = ones(Int64, 10)
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

```
my_matrix_ones = ones(10, 2)
```

```
10×2 Matrix{Float64}:
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
```

For other elements we can first intantiate an array with `undef` elements and use the `fill!` function to fill all elements of an array with the desired element. Here's an example with π :

```
π
my_matrix_ = Matrix{Float64}(undef, 2, 2)
fill!(πmy_matrix_, π)
```

```
2×2 Matrix{Float64}:
 3.14  3.14
 3.14  3.14
```

We can also create arrays with **arrays literals**. For example a 2x2 matrix of integers:

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

```
2×2 Matrix{Int64}:
 1  2
 3  4
```

Array literals also accept a type specification before the `[]` brackets. So, if we want the same 2x2 array as before but now as floats, we can do so:

```
Float64[[1 2]
 [3 4]]
```

```
2×2 Matrix{Float64}:
 1.0  2.0
```

```
3.0 4.0
```

It also works for vectors:

```
Bool[0, 1, 0, 1]
```

```
Bool[0, 1, 0, 1]
```

You can even **mix and match** array literals with the constructors:

```
[ones(Int, 2, 2) zeros(Int, 2, 2)]
```

```
2×4 Matrix{Int64}:
```

```
1 1 0 0  
1 1 0 0
```

```
[zeros(Int, 2, 2)  
 ones(Int, 2, 2)]
```

```
4×2 Matrix{Int64}:
```

```
0 0  
0 0  
1 1  
1 1
```

```
[ones(Int, 2, 2) [1; 2]  
 [3 4] 5]
```

```
3×3 Matrix{Int64}:
```

```
1 1 1  
1 1 2  
3 4 5
```

Another powerful way to create arrays are **array comprehensions**. This way of creating arrays is our *preferred* way, it avoids loops, indexing and other error-prone operations. You specify what you want to do inside the [] brackets. For example, say we want to create a vector of squares from 1 to 100:

```
[x^2 for x in 1:10]
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

They also support multiple inputs:

```
[x*y for x in 1:10 for y in 1:2]
```

```
[1, 2, 2, 4, 3, 6, 4, 8, 5, 10, 6, 12, 7, 14, 8, 16, 9, 18, 10, 20]
```

And conditionals:

```
[x^2 for x in 1:10 if isodd(x)]
```

```
[1, 9, 25, 49, 81]
```

As with array literals you can specify your desired type before the [] brackets:

```
Float64[x^2 for x in 1:10 if isodd(x)]
```

```
[1.0, 9.0, 25.0, 49.0, 81.0]
```

Finally, we can also create arrays with **concatenation functions**:

- `cat`: concatenate input arrays along a specific dimension `dims`

```
cat(ones(2), zeros(2), dims=1)
```

```
[1.0, 1.0, 0.0, 0.0]
```

```
cat(ones(2), zeros(2), dims=2)
```

```
2×2 Matrix{Float64}:
```

```
1.0 0.0
1.0 0.0
```

- `vcat`: vertical concatenation, a shorthand for `cat(...; dims=1)`

```
vcat(ones(2), zeros(2))
```

```
[1.0, 1.0, 0.0, 0.0]
```

- `hcat`: horizontal concatenation, a shorthand for `cat(...; dims=2)`

```
hcat(ones(2), zeros(2))
```

```
2×2 Matrix{Float64}:
```

```
1.0 0.0
1.0 0.0
```

3.2.6.3 Array Inspection

Once we have arrays, the next logical step is to inspect them. There are a lot of handy functions that allows the user to have an inner insight into any array.

It is most useful to know what **type of elements** are inside an array. We can do this with `eltype`:

```
eltype(πmy_matrix_)
```

```
Float64
```

After knowing its types, one might be interested in array dimensions. Julia has several functions to inspect array dimensions:

- `length`: total number of elements

```
length(πmy_matrix_)
```

```
4
```

- `ndims`: number of dimensions

```
ndims(πmy_matrix_)
```

```
2
```

- `size`: this one is a little tricky. By default it will return a tuple containing the array's dimensions.

```
size(πmy_matrix_)
```

```
(2, 2)
```

You can get a specific dimension with a second argument to `size`

```
size(πmy_matrix_, 2) # columns
```

```
2
```

3.2.6.4 Array Indexing and Slicing

Sometimes we want to only inspect certain parts of an array. This is called **indexing** and **slicing**. If you want a particular observation of a vector, or a row or column of a matrix; you'll probably need to **index an array**.

First, I will create an example vector and matrix to play around:

```
my_example_vector = [1, 2, 3, 4, 5]

my_example_matrix = [[1 2 3]
                     [4 5 6]
                     [7 8 9]]
```

Let's see first an example with vectors. Suppose you want the second element of a vector. You append `[]` brackets with the desired **index** inside:

```
my_example_vector[2]
```

```
2
```

The same syntax follows with matrices. But, since matrices are 2-dimensional arrays, we have to specify *both* rows and columns. Let's retrieve the element from the second row (first dimension) and first column (second dimension):

```
my_example_matrix[2, 1]
```

```
4
```

Julia also have conventional keywords for the first and last elements of an array: `begin` and `end`. For example, the second to last element of a vector can be retrieved as:

```
my_example_vector[end-1]
```

```
4
```

It also work for matrices. Let's retrieve the element of the last row and second column:

```
my_example_matrix[end, begin+1]
```

```
8
```

Often, we are not only interested in just one array element, but in a whole **subset of array elements**. We can accomplish this by **slicing** an array. It uses the same index syntax, but with the added colon `:` to denote the boundaries that we are slicing through the array. For example, suppose we want to get the 2nd to 4th element of a vector:

```
my_example_vector[2:4]
```

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

We could do the same with matrices. Particularly with matrices if we want to select all elements in a following dimension we can do so with just a colon `..`. For example, all elements in the second row:

```
my_example_matrix[2, :]
```

```
[4, 5, 6]
```

You can interpret this with something like “take 2nd row and all columns.”

It also supports `begin` and `end`:

```
my_example_matrix[begin+1:end, end]
```

```
[6, 9]
```

3.2.6.5 Array Manipulations

There are several ways we could manipulate an array. The first would be to manipulate a **singular element of the array**. We just index the array by the desired element and proceed with an assignment `=`:

```
my_example_matrix[2, 2] = 42
```

```
my_example_matrix
```

```
3x3 Matrix{Int64}:
```

```
1 2 3
```

```
4 42 6
```

```
7 8 9
```

Or you can manipulate a certain **subset of elements of the array**. In this case, we need to slice the array and then assign with =:

```
my_example_matrix[3, :] = [17, 16, 15]
my_example_matrix
```

```
3x3 Matrix{Int64}:
 1   2   3
 4  42   6
17  16  15
```

Note that we had to assign a vector because our sliced array is of type `Vector`:

```
typeof(my_example_matrix[3, :])
```

```
Vector{Int64} (alias for Array{Int64, 1})
```

The second way we could manipulate an array is to **alter its shape**. Suppose you have a 6-element vector and you want to make it a 3x2 matrix. You can do so with `reshape`, by using the array as first argument and a tuple of dimensions as second argument:

```
six_vector = [1, 2, 3, 4, 5, 6]
tree_two_matrix = reshape(six_vector, (3, 2))
tree_two_matrix
```

```
3x2 Matrix{Int64}:
 1   4
 2   5
 3   6
```

You can do the reverse, convert it back to a vector, by specifying a tuple with only one dimension as second argument:

```
reshape(tree_two_matrix, (6, ))
```

```
[1, 2, 3, 4, 5, 6]
```

The third way we could manipulate an array is to **apply a function over every array element**. This is where the familiar broadcasting “dot” operator . comes in.

```
logarithm.(my_example_matrix)
```

```
3x3 Matrix{Float64}:
 0.0      0.693147  1.09861
 1.38629  3.73767  1.79176
 2.83321  2.77259  2.70805
```

We also broadcast operators:

```
my_example_matrix .+ 100
```

```
3x3 Matrix{Int64}:
```

```
101 102 103
104 142 106
117 116 115
```

We can use also `map` to apply a function to every element of an array:

```
map(logarithm, my_example_matrix)
```

```
3x3 Matrix{Float64}:
 0.0      0.693147  1.09861
 1.38629  3.73767   1.79176
 2.83321  2.77259   2.70805
```

It also accepts an anonymous function:

```
map(x -> x*3, my_example_matrix)
```

```
3x3 Matrix{Int64}:
 3     6     9
 12    126   18
 51    48    45
```

It also works with slicing:

```
map(x -> x + 100, my_example_matrix[:, 3])
```

```
[103, 106, 115]
```

Finally, sometimes, and specially when dealing with tabular data, we want to apply a **function over all elements in a specific array dimension**. This can be done with the `mapslices` function. Similar to `map`, the first argument is the function and the second argument is the array. The only change is that we need to specify the `dims` argument to flag what dimension we want to transform the elements.

For example let's use `mapslice` with the `sum` function on both rows (`dims=1`) and columns (`dims=2`):

```
# rows
mapslices(sum, my_example_matrix; dims=1)
```

```
1x3 Matrix{Int64}:
 22  60  24
```

```
# columns
mapslices(sum, my_example_matrix; dims=2)
```

```
3x1 Matrix{Int64}:
 6
 52
 48
```

3.2.6.6 Array Iteration

One common operation is to **iterate over an array with a for loop**. The **regular for loop over an array returns each element**.

The simplest example is with a vector.

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

empty_vector = Int64[]

for i in simple_vector
    push!(empty_vector, i + 1)
end

empty_vector
```

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

Sometimes you don't want to loop over each element, but actually over each array index. **We can eachindex function combined with a for loop to iterate over each array index.**

Again, let's show an example with a vector:

```
forty_two_vector = [42, 42, 42]

empty_vector = Int64[]

for i in eachindex(forty_two_vector)
    push!(empty_vector, i)
end

empty_vector
```

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

In this example the `eachindex(forty_two_vector)` iterator inside the `for` loop returns not `forty_two_vector`'s values but its indices: `[1, 2, 3]`.

Iterating over matrices involves more details. The standard `for` loop goes first over columns then over rows. It will first traverse all elements in column 1, from the first row to the last row, then it will move to column 2 in a similar fashion until it has covered all columns.

Those familiar with other programming languages, Julia, like most scientific programming languages, is “column-major.” This means that arrays are stored contiguously using a column orientation. If any time you are seeing problems of performance and there is an array `for` loop involved, chances are that you are mismatching Julia’s native column-major storage orientation.

Ok, let's show this in an example:

```
column_major = [[1 2]
                [3 4]]
```

```
row_major = [[1 3]
             [2 4]]
```

```
empty_vector = Int64[]

for i in column_major
    push!(empty_vector, i)
end

empty_vector
```

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

```
empty_vector = Int64[]

for i in row_major
    push!(empty_vector, i)
end

empty_vector
```

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

There are some handy functions to iterate over matrices.

- `eachcol`: iterates over an array column first

```
first(eachcol(column_major))
```

```
[1, 3]
```

- `eachrow`: iterates over an array row first

```
first(eachrow(column_major))
```

```
[1, 2]
```

3.2.7 Pair

Compared to the huge section on arrays, this section on pairs will be brief. **Pair is a data structure that holds two types.** How we construct a pair in Julia is using the following syntax:

```
my_pair = Pair("Julia", 42)
```

```
"Julia" => 42
```

Alternatively, we can create a pair by specifying both values and in between we use the `=>` operator:

```
my_pair = "Julia" => 42
```

```
"Julia" => 42
```

The elements are stored in the fields `first` and `second`.

```
my_pair.first
```

```
Julia
```

```
my_pair.second
```

```
42
```

Pairs will be used a lot in data manipulation and data visualization since both `DataFrames.jl` (Section 4) or `Plots.jl` (Section 6) main functions depends on `Pair` as type arguments.

3.2.8 Dict

If you understood what a `Pair` is, then `Dict` won't be a problem. **Dict in Julia is just a “hash table” with pairs of key and value.** keys and values can be of any type, but generally you'll see keys as strings.

There are two ways to construct `Dicts` in Julia. The first is using the **default constructor Dict and passing a vector of tuples composed of (key, value)**:

```
my_dict = Dict("one" => 1, "two" => 2)
```

```
Dict{String, Int64} with 2 entries:  
"two" => 2  
"one" => 1
```

We prefer the second way of constructing `Dicts`. It offers a much elegant and expressive syntax. You use the same **default constructor Dict**, but now you pass **pairs of key and value**:

```
my_dict = Dict("one" => 1, "two" => 2)
```

```
Dict{String, Int64} with 2 entries:  
"two" => 2  
"one" => 1
```

You can retrieve a `Dicts` value by indexing it by the corresponding key:

```
my_dict["one"]
```

```
1
```

Similarly, to add a new entry you index the `Dict` by the desired key and assign a value with the assignment `=` operator:

```
my_dict["three"] = 3
```

```
3
```

If you want to check if a `Dict` has a certain key you can use the `haskey` function:

```
haskey(my_dict, "two")
```

```
true
```

To delete a key you can use either the `delete!` function:

```
delete!(my_dict, "three")
```

```
Dict{String, Int64} with 2 entries:
"two" => 2
"one" => 1
```

Or to delete a key while retuning its value you can use the `pop!` function:

```
popped_value = pop!(my_dict, "two")
```

```
2
```

Now our `my_dict` has only one key:

```
length(my_dict)
```

```
1
```

```
my_dict
```

```
Dict{String, Int64} with 1 entry:
"one" => 1
```

`Dicts` are also used in data manipulations by `DataFrames.jl` (Section 4) and data visualization by `Plots.jl` (Section 6). So it is important to know their basic functionality.

There is one useful `Dict` constructor that we use a lot. Suppose you have two vectors and you want to construct a `Dict` with one of them as keys and the other as values. You can do that with the `zip` function which “glues” together two objects just like a zipper:

```
A = ["one", "two", "three"]
B = [1, 2, 3]

dic = Dict(zip(A, B))
```

```
Dict{String, Int64} with 3 entries:
"two" => 2
"one" => 1
"three" => 3
```

For instance, we can now get the number 3 via:

```
dic["three"]
```

```
3
```

3.2.9 Symbol

`Symbol` is actually *not* a data structure. It is a type and behaves a lot like a string. Instead of surrounding the text by quotation marks, a symbol starts with a colon (:) and can contain underscores:

```
sym = :some_text
```

```
:some_text
```

Since symbols and strings are so similar, we can easily convert a symbol to string and vice versa:

```
s = string(sym)
```

```
some_text
```

```
sym = Symbol(s)
```

```
:some_text
```

One simple benefit of symbols is that you have to type one character less, that is, `:some_text` versus `"some text"`. We use symbols a lot in data manipulations with the `DataFrames.jl` package (Section 4) and data visualizations (Section 6 and 7).

3.2.10 Splat Operator

In Julia we have the “splat” operator ... which is mainly used in function calls as a **sequence of arguments**. We will occasionally use splatting in some function calls in **data manipulation** and **data visualization** chapters.

The most intuitive way to learn about splatting is with an example. The `add_elements` function below takes three arguments to be added together:

```
add_elements(a, b, c) = a + b + c
```

```
add_elements (generic function with 1 method)
```

Now suppose that I have a collection with three elements. The naïve way to this would be to supply the function with all three elements as function arguments like this:

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

```
add_elements(my_collection[1], my_collection[2], my_collection[3])
```

```
6
```

Here is where we use the “splat” operator ... which takes a collection (often an array, vector, tuple or range) and converts into a sequence of arguments:

```
add_elements(my_collection...) # and splat!
```

```
6
```

The ... is included after the collection that we want to “splat” into a sequence of arguments. In the example above, syntactically speaking, the following are the same:

```
collection = [x, y, z]

function(collection...) = function(x, y, z)
```

Anytime Julia sees a splatting operator inside a function call, it will be converted on a sequence of arguments for all elements of the collection separated by commas.

It also works for ranges:

```
add_elements(1:3...) # and splat!
```

```
6
```

3.3 Filesystem

In data science, most projects are undertaken in a collaborative effort. We share code, data, tables, figures and so on. Behind everything, there is the **operational system (OS) filesystem**. In an ideal work, code would run the *same* in *different* OS. But that is not what actually happens. One instance of this is the difference of Windows paths, such as `C:\\\\user\\\\john\\\\`, and Linux paths, such as `/home/john`. This is why is important to discuss **filesystem best practices**.

Julia has native filesystem capabilities that can **handle all different OS demands**. They are located in the `Filesystem` module from the core `Base` Julia library. This means that Julia provides everything you need to make your code perform flawlessly in any OS that you want to.

Whenever you are dealing with files such as CSV, Excel files or other Julia scripts, make sure that your code is compliant with all different OS filesystems. This is easily accomplished with the `joinpath` and `pwd` functions.

The `pwd` function is an acronym for **p**rint **w**orking **d**irectory and it returns a string containing the current working directory. One nice thing about `pwd` is that it is robust to OS, i.e. it will return the correct string in Linux, MacOS, Windows or any other OS. For example, let's see what are our current directory and record it in a variable `root`:

```
root = pwd()
```

```
/home/runner/work/JuliaDataScience/JuliaDataScience
```

The next step would be to include the relative path from `root` to our desired file. Since different OS have different ways to construct relative paths with subfolders, some use forward slash / while other might use backslashes \, we cannot simply concatenate the our file's relative path with the `root` string.

For that, we have the `joinpath` function, which will join different relative paths and filenames into your specific OS filesystem implementation.

Suppose you have a script named `my_script.jl` inside your project's directory. You can have a robust representation of the filepath to `my_script.jl` as:

```
joinpath(root, "my_script.jl")
```

```
/home/runner/work/JuliaDataScience/JuliaDataScience/my_script.jl
```

`joinpath` also handles subfolders. Let's now imagine a common situation where you have a folder named `data/` in your project's directory. Inside this folder there is a CSV file named `my_data.csv`. You can have the same robust representation of the filepath to `my_data.csv` as:

```
joinpath(root, "data", "my_data.csv")
```

```
/home/runner/work/JuliaDataScience/JuliaDataScience/data/my_data.csv
```

Always make sure that your code can run anywhere. It's a good habit to pick up, because it's very likely to save problems later.

3.4 Julia Standard Library

Julia has a rich standard library that ships with every Julia installation. Contrary to everything that we have seen so far, e.g. types, data structures and filesystem; you **must import standard library modules into your environment** to use a particular module or function.

This is done with the `using` keyword:

```
using ModuleName
```

Now you can access all functions and types inside `ModuleName`.

3.4.1 Dates

How to handle dates and timestamps is something quite important in data science. Like we said in *Why Julia?* (Section 2) section, Python's `pandas` uses its own `Datetime` type to handle dates. The same with R tidyverse's `lubridate` package, which also defines its own `datetime` type to handle dates. Julia doesn't need any of this, it has all the **date stuff already baked onto its standard library, in a module named `Dates`.**

To begin, let's import the `Dates` module:

```
using Dates
```

3.4.1.1 Date and DateTime Types

The `Dates` Standard library module has **two types for working with dates**:

1. `Date`: representing time in days; and
2. `DateTime`: representing time in millisecond precision.

We can construct `Date` and `DateTime` with the default constructor either by specifying an integer to represent year, month, day, hours and so on:

```
Date(1987) # year
```

```
1987-01-01
```

```
Date(1987, 9) # month
```

```
1987-09-01
```

```
Date(1987, 9, 13) # day
```

```
1987-09-13
```

```
DateTime(1987, 9, 13, 21) # hour
```

```
1987-09-13T21:00:00
```

```
DateTime(1987, 9, 13, 21, 21) # minute
```

```
1987-09-13T21:21:00
```

For the curious, September 13th 1987, 21:21 is the official time of birth of the first author, Jose.

We can also pass `Period` types to the default constructor. **Period types are the human-equivalent representation of time** for the computer. Julia's `Dates` have the following `Period` abstract subtypes:

```
subtypes(Period)
```

```
DatePeriod
```

```
TimePeriod
```

Which divide into the following concrete types, they are pretty much self-explanatory:

```
subtypes(DatePeriod)
```

```
Day
```

```
Month
```

```
Quarter
```

```
Week
```

```
Year
```

```
subtypes(TimePeriod)
```

```
Hour
```

```
Microsecond
```

```
Millisecond
```

```
Minute
```

```
Nanosecond
```

```
Second
```

So we could alternatively construct Jose's official time of birth as:

```
DateTime(Year(1987), Month(9), Day(13), Hour(21), Minute(21))
```

```
1987-09-13T21:21:00
```

3.4.1.2 Parsing Dates

Most of the time we won't be constructing `Date` or `DateTime` instances from scratch. Actually, we probably will be **parsing strings as Date or DateTime types**.

The `Date` and `DateTime` constructors can be fed a string and a format string. For example, the string "19870913" representing September 13th 1987 can be parsed with:

```
Date("19870913", "yyyyymmdd")
```

```
1987-09-13
```

Notice that the second argument is a string representation of the format. We have the first four digits representing year `y`, followed by two digits for month `m` and finally two digits for day `d`.

It also works for timestamps with `DateTime`:

```
DateTime("1987-09-13T21:21:00", "yyyy-mm-ddTHH:MM:SS")
```

```
1987-09-13T21:21:00
```

You can find more on how to specify different format as strings in the Julia `Dates`' documentation. Don't worry if you have to revisit it all the time, we ourselves have to do it all the time when working with dates and timestamps.

According to Julia `Dates`' documentation, using the `Date(date_string, format_string)` method is fine if only called a few times. If there are many similarly formatted date strings to parse, however, it is much more efficient to first create a `DateFormat` type, and then pass it instead of a raw format string. So our previous example would become:

```
format = DateFormat("yyyymmdd")
Date("19870913", format)
```

```
1987-09-13
```

Alternatively, without loss of performance, you can use the string literal prefix `dateformat"..."`:

```
Date("19870913", dateformat"yyyymmdd")
```

```
1987-09-13
```

3.4.1.3 Extracting Date Information

It is easy to **extract desired information from Date and DateTime objects**. First, let's create an instance of a very special date:

```
my_birthday = Date("1987-09-13")
```

```
1987-09-13
```

We can extract anything we want from `my_birthday`:

```
year(my_birthday)
```

```
1987
```

```
month(my_birthday)
```

```
9
```

```
day(my_birthday)
```

```
13
```

Julia's Dates module also have **compound functions that returns a tuple of values**:

```
yearday(my_birthday)
```

```
(1987, 9)
```

```
monthday(my_birthday)
```

```
(9, 13)
```

```
yeardaymonthday(my_birthday)
```

```
(1987, 9, 13)
```

We can also see day of the week and other handy stuff:

```
dayofweek(my_birthday)
```

```
7
```

```
dayname(my_birthday)
```

```
Sunday
```

```
dayofweekofmonth(my_birthday) # second sunday
```

```
2
```

Yep, Jose was born on the second sunday of September.

NOTE: Here's a handy tip to just recover weekdays from `Dates` instances. Just use a `filter` on `dayofweek(your_date) <= 5`. For business day you can check the package `BusinessDays.jl`.

3.4.1.4 Date Operations

We can perform **operations** in `Dates` instances. For example, we can add days to a `Date` or `DateTime` instance. Notice that Julia's `Dates` will automatically perform the adjustments necessary for leapyears, of months with 30 or 31 days (this is known as *calendrical arithmetic*).

```
my_birthday + Day(90)
```

```
1987-12-12
```

We can add as many as we like:

```
my_birthday + Day(90) + Month(2) + Year(1)
```

```
1989-02-11
```

To get **date duration**, we just use the **subtraction** - operator. Let's see how many days Jose is old:

```
today() - my_birthday
```

```
12405 days
```

The **default duration** of `Date` types is a `Day` instance. For the `DateTime`, the default duration is `Millisecond` instance:

```
DateTime(today()) - DateTime(my_birthday)
```

```
1071792000000 milliseconds
```

3.4.1.5 Date Intervals

One nice thing about `Dates` module is that we can also easily construct date and time intervals. Julia is clever enough to not have to define the whole interval types and operations that we covered in Section 3.2.5. It just extends the functions and operations defined for `UnitRange` to `Date`'s types. This is known as multiple dispatch and we already covered in *Why Julia?*(Section 2).

For example suppose you want to create a `Day` interval. This is easy done with the colon `:` operator:

```
Date("2021-01-01"):Day(1):Date("2021-01-07")
```

```
2021-01-01
```

```
2021-01-02
```

```
2021-01-03
```

```
2021-01-04
```

```
2021-01-05
```

```
2021-01-06
```

```
2021-01-07
```

There is nothing special in using `Day(1)` as interval, we can **use whatever Period type** as interval. For example using 3 days as intervals:

```
Date("2021-01-01"):Day(3):Date("2021-01-07")
```

```
2021-01-01
```

```
2021-01-04
```

```
2021-01-07
```

Or even months:

```
Date("2021-01-01"):Month(1):Date("2021-03-01")
```

```
2021-01-01
```

```
2021-02-01
```

```
2021-03-01
```

Note that the **type of this interval is a stepRange with the Date and concrete Period type** we used as interval inside the colon `:` operator:

```
my_date_interval = Date("2021-01-01"):Month(1):Date("2021-03-01")
typeof(my_date_interval)
```

```
StepRange{Date, Month}
```

We can convert this to a **vector** with the `collect` function:

```
my_date_interval_vector = collect(my_date_interval)
```

```
2021-01-01
```

```
2021-02-01
```

```
2021-03-01
```

And have all the **array functionalities available**, like, for example, indexing:

```
my_date_interval_vector[end]
```

```
2021-03-01
```

We can also **broadcast date operations** to our vector of `Date`s:

```
my_date_interval_vector .+ Day(10)
```

```
2021-01-11
```

```
2021-02-11
```

```
2021-03-11
```

All we've done with `Date` types can be extended to `DateTime` types in the same manner.

3.4.2 Random Numbers

Another important module in Julia's standard library is the `Random` module. This module deals with **random number generation**. `Random` is a rich library and, if you interested in it, you should consult Julia's `Random` documentation. We will cover *only* three functions: `seed!`, `rand` and `randn`.

To begin we first import the `Random` module:

```
using Random
```

We have **two main functions that generate random numbers**:

- `rand`: samples a **random element** of a data structure or type.
- `randn`: generates a random number that follows a **standard normal distribution** (mean 0 and standard deviation 1) of a specific type.

NOTE: Note that those two functions are already in the Julia `Base` module. So you don't need to import `Random` if you planning to use them

3.4.2.1 `rand`

By default if you call `rand` without arguments it will return a `Float64` in the interval $[0, 1)$, which means between 0 inclusive to 1 exclusive:

```
rand()
```

```
0.12406452856290495
```

You can modify `rand` arguments in several ways. For example, suppose you want more than 1 random number:

```
rand(3)
```

```
[0.8333794984256451, 0.656629460673362, 0.36027087512495437]
```

Or you want a different interval:

```
rand(1.0:10.0)
```

```
7.0
```

You can also specify a different step size inside the interval and a different type. Here we are using number without the `.` so Julia will interpret them as `Int64`:

```
rand(2:2:20)
```

```
14
```

You can also mix and match arguments:

```
rand(2:2:20, 3)
```

```
[6, 4, 20]
```

It also supports a collection of elements as a tuple:

```
rand((42, "Julia", 3.14))
```

```
3.14
```

And also arrays:

```
rand([1, 2, 3])
```

```
2
```

DictS:

```
rand(Dict("one"=>1, "two"=>2))
```

```
"one" => 1
```

To finish off all the `rand` arguments options, you can specify the desired random number dimensions in a tuple. If you do this, the returned type will be an array. For example, a 2x2 matrix of `Float64` between 1.0 and 3.0:

```
rand(1.0:3.0, (2, 2))
```

```
2×2 Matrix{Float64}:
 3.0  3.0
 3.0  2.0
```

3.4.2.2 `randn`

`randn` follows the same general principle from `rand` but now it only returns numbers generated from standard normal distribution. The standard normal distribution is the normal distribution with mean 0 and standard deviation 1. The default type is `Float64` and it only allows for subtypes of `AbstractFloat` or `Complex`:

```
randn()
```

```
-1.1014775344404644
```

We can only specify the size:

```
randn((2, 2))
```

```
2×2 Matrix{Float64}:
 0.920475   0.668393
 -0.0627252  0.499883
```

3.4.2.3 `seed!`

To finish off the `Random` overview, let's talk about **reproducibility**. Often, we want to make something replicable. Meaning that, we want the random number generator to generate the same random sequence of numbers, despite paradoxical that might sound... We can do so with the `seed!` function.

Let me show you an example of a `rand` that generates the same three numbers given a certain seed:

```
Random.seed!(123)
rand(3)
```

```
[0.7684476751965699, 0.940515000715187, 0.6739586945680673]
```

```
Random.seed!(123)
rand(3)
```

```
[0.7684476751965699, 0.940515000715187, 0.6739586945680673]
```

Note that `seed!` is not automatically exported by the `Random` module. We have to call it with the `Module.function` syntax.

In order to avoid tedious and inefficient repetition of `seed!` all over the place we can instead define an instance of a `seed!` and pass it as a first argument of **either `rand` or `randn`**.

```
my_seed = Random.seed!(123)
```

```
MersenneTwister(123)
```

```
rand(my_seed, 3)
```

```
[0.3954531123351086, 0.3132439558075186, 0.6625548164736534]
```

```
rand(my_seed, 3)
```

```
[0.3954531123351086, 0.3132439558075186, 0.6625548164736534]
```

NOTE: If you want your code to be reproducible you can just call `Random.seed!` in the beginning of your script. This will take care of reproducibility in sequential `Random` operations. No need to use it all `rand` and `randn` usage.

3.4.3 Downloads

One last thing from Julia's standard library for us to cover is the `Download` module. It will be really brief because we will only be covering a single function named `download`.

Suppose you want to **download a file from the internet to your local storage**. You can accomplish this with the `download` function. The first and only required argument is the file's url. You can also specify as a second argument the desired output path for the downloaded file (don't forget the filesystem best practices!). If you don't specify a second argument, Julia will, by default, create a temporary file with the `tempfile` function.

Let's import the `Download` module:

```
using Download
```

For example let's download our `JuliaDataScience` GitHub repository `Project.toml` file. Note that `download` function is not exported by `Downloads` module, so we have to use the `Module.function` syntax. By default it returns a string that holds the file path for the downloaded file:

```
url = "https://raw.githubusercontent.com/JuliaDataScience/JuliaDataScience/main/Project.toml"
my_file = Downloads.download(url) # tempfile() being created
```

```
/tmp/jl_QLTMym
```

Let's just show the first 4 lines of our downloaded file with the `readlines` function:

```
readlines(my_file)[1:4]
```

```
4-element Vector{String}:
"name = \"JDS\""
"uuid = \"6c596d62-2771-44f8-8373-3ec4b616ee9d\""
"authors = [\"Jose Storopoli\", \"Rik Huijzer\", \"Lazaro Alonso\"]"
"version = \"0.1.0\""
```

NOTE: If you want to interact with web requests or web APIs, you would probably need to use the `HTTP.jl` package.

4 DataFrames.jl

Data comes mostly in a tabular format. By tabular, we mean that the data consists of a table containing rows and columns. Columns are usually of the same data type, whereas rows have different types. The rows, in practice, denote observations while columns denote variables. For example, we can have a table of TV shows containing in which country it was produced and our personal rating, see Table 4.1.

Table 4.1: TV shows.

	name	country	rating
	Game of Thrones	United States	8.2
	The Crown	England	7.3
	Friends	United States	7.8

Here, the dots mean that this could be a very long table and we only show a few rows. While analyzing data, often we come up with interesting questions about the data, also known as *data queries*. For large tables, computers would be able to answer these kinds of questions much quicker than you could do it by hand. Examples of, so called *queries*, for this data could be:

- Which TV show has the highest rating?
- Which TV shows were produced in the United States?
- Which TV shows were produced in the same country?

But, as a researcher, real science often starts with having multiple tables or data sources. For example, if we also have data from someone else's ratings for the TV shows (Table 4.2):

Table 4.2: Ratings.

	name	rating
	Game of Thrones	7
	Friends	6.4

Now, questions that we could ask ourselves could be:

- What is Game of Thrones' average rating?
- Who gave the highest rating for Friends?
- What TV shows were rated by you but not by the other person?

In the rest of this chapter, we will show you how you can easily answer these questions in Julia. To do so, we first show why we need a Julia package called `DataFrames.jl`. In the next sections, we show how you can use this package and, in the end, we show how to write fast data transformations (Section 5).

Let's look at a table of grades like the one in Table 4.3:

Table 4.3: Grades for 2020.

	name	age	grade_2020
	Bob	17	5.0
	Sally	18	1.0
	Alice	20	8.5
	Hank	19	4.0

Here, the column `name` has type `string`, `age` has type `integer`, and `grade` has type `float`.

So far, this book has only handled Julia's basics. These basics are great for many things, but not for tables. To show that we need more, lets try to store the tabular data in arrays:

```
function grades_array()
    name = ["Bob", "Sally", "Alice", "Hank"]
    age = [17, 18, 20, 19]
    grade_2020 = [5.0, 1.0, 8.5, 4.0]
    (; name, age, grade_2020)
end
```

Now, the data is stored in, so called, column-major, which is cumbersome when we want to get data from a row:

```
function second_row()
    name, age, grade_2020 = grades_array()
    i = 2
    row = (name[i], age[i], grade_2020[i])
end
second_row()
```

```
("Sally", 18, 1.0)
```

Or, if you want to have the grade for Alice, you first need to figure out in what row Alice is:

```
function row_alice()
    names = grades_array().name
```

```
i = findfirst(names .== "Alice")
end
row_alice()
```

```
3
```

and, then, we can get the value:

```
function value_alice()
    grades = grades_array().grade_2020
    i = row_alice()
    grades[i]
end
value_alice()
```

```
8.5
```

DataFrames.jl can easily solve these kinds of issues. You can start by loading `DataFrames.jl` with using :

```
using DataFrames
```

With `DataFrames.jl`, we can define a `DataFrame` to hold our tabular data:

```
names = ["Sally", "Bob", "Alice", "Hank"]
grades = [1, 5, 8.5, 4]
df = DataFrame(; name=names, grade_2020=grades)
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

which gives us a variable `df` containing our data in table format.

NOTE: This works, but there is one thing that we need to change straight away. In this example, we defined the variables `name`, `grade_2020` and `df` in global scope. This means that these variables can be accessed and edited from anywhere. If we would continue writing the book like this, we would have a few hundred variables at the end of the book even though the data that we put into the variable `name` should only be accessed via `DataFrame`! The variables `name` and `grade_2020` where never meant to be kept for long! Now, imagine that we would change the contents of `grade_2020` a few times in this book. Given only the book as PDF, it would be near impossible to figure out the contents of the variable by the end.

We can solve this very easily by using functions.

Let's do the same thing as before but now in a function:

```
function grades_2020()
    name = ["Sally", "Bob", "Alice", "Hank"]
    grade_2020 = [1, 5, 8.5, 4]
    DataFrame(; name, grade_2020)
end
grades_2020()
```

Table 4.5: Grades 2020.

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

Note that `name` and `grade_2020` are destroyed after the function returns, that is, they are only available in the function. There are two other benefits of doing this. First, it is now clear to the reader where `name` and `grade_2020` belong to: they belong to the grades of 2020. Second, it is easy to determine what the output of `grades_2020()` would be at any point in the book. For example, we can now assign the data to a variable `df`:

```
df = grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

Change the contents of `df`:

```
df = DataFrame(name = ["Malice"], grade_2020 = ["10"])
```

name	grade_2020
Malice	10

And still recover the original data back without any problem:

```
df = grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

Of course, this assumes that the function is not re-defined. We promise to not do that in this book, because it is a bad idea exactly for this reason. Instead of “changing” a function, we will make a new one and give it a clear name.

So, back to the `DataFrames` constructor. As you might have seen, the way to create one is simply to pass vectors as arguments into the `DataFrame` constructor. You can come up with any valid Julia vector and it will work **as long as the vectors have the same length**. Duplicates, Unicode symbols and any sort of numbers are fine:

```
DataFrame(σ = ["a", "a", "a"], δ = π[, π/2, π/3])
```

σ	δ
a	3.141592653589793
a	1.5707963267948966
a	1.0471975511965976

Typically, in your code, you would create a function which wraps around one or more `DataFrames`’ functions. For example, we can make a function to get the grades for one or more names:

```
function grades_2020(names::Vector{Int})
    df = grades_2020()
    df[names, :]
end
grades_2020([3, 4])
```

name	grade_2020
Alice	8.5
Hank	4.0

This way of using functions to wrap around basic functionality in programming languages and packages

is quite common. Basically, you can think of Julia and `DataFrames.jl` as providers of building blocks. They provide very **generic** building blocks which allow you to build things for your **specific** use case like this grades example. By using the blocks, you can make a data analysis script, control a robot or whatever you like to build.

So far, the examples were quite cumbersome, because we had to use indexes. In the next sections, we will show how to load and save data, and many powerful building blocks provided by `DataFrames.jl`.

4.1 Load and Save Files

Having only data inside Julia programs and not being able to load or save it would be very limiting. Therefore, we start by mentioning how to store files to and load files from disk. We focus on CSV, see Section 4.1.1, and Excel, see Section 4.1.2, file formats since those are the most common data storage formats for tabular data.

4.1.1 CSV

Comma-separated values (CSV) files are are very effective way to store tables. CSV files have two advantages over other data storage files. First, it does exactly what the name indicates it does, namely storing values by separating them using commas ,. This acronym is also used as the file extension. So, be sure that you save your files using the “.csv” extension such as “myfile.csv.” To demonstrate how a CSV file looks, we can install the `csv.jl` package:

```
julia> ]
pkg> add CSV
```

and load it via:

```
using CSV
```

We can now use our previous data:

```
grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

and read it from a file after writing it:

```
function write_grades_csv()
```

```
path = "grades.csv"
CSV.write(path, grades_2020())
end
```

```
path = write_grades_csv()
read(path, String)
```

```
name,grade_2020
Sally,1.0
Bob,5.0
Alice,8.5
Hank,4.0
```

Here, we also see the second benefit of CSV data format: the data can be read by using a simple text editor. This differs from many alternative data formats which require proprietary software, e.g. Excel.

This works wonders, but what if our data **contains commas**, as values? If we would naively write data with commas, it would make the files very hard to convert back to a table. Luckily, `csv.jl` handles this for us automatically. Consider the following data with commas ,:

```
function grades_with_commas()
    df = grades_2020()
    df[3, :name] = "Alice,"
    df
end
grades_with_commas()
```

Table 4.12: Grades with commas.

name	grade_2020
Sally	1.0
Bob	5.0
Alice,	8.5
Hank	4.0

If we write this, we get:

```
function write_comma_csv()
    path = "grades-commas.csv"
    CSV.write(path, grades_with_commas())
end
path = write_comma_csv()
read(path, String)
```

```
name,grade_2020
Sally,1.0
Bob,5.0
```

```
"Alice,",8.5
Hank,4.0
```

So, `csv.jl` adds quotation marks " around the comma-containing values. Another common way to solve this problem, is to write the data to a **tab-separated values** (TSV) file format. This assumes that the data doesn't contain tabs, which holds in most cases.

Also, note that TSV files can be also read using a simple text editor and uses the ".tsv" extension.

```
function write_comma_tsv()
    path = "grades-comma.tsv"
    CSV.write(path, grades_with_commas(); delim='\t')
end
read(write_comma_tsv(), String)
```

```
name      grade_2020
Sally    1.0
Bob      5.0
Alice,   8.5
Hank     4.0
```

Text file formats. like CSV and TSV files, can also come up with other delimiters, such as semicolons ";" spaces ", " or even something as unusual as "π."

```
function write_space_separated()
    path = "grades-space-separated.csv"
    CSV.write(path, grades_2020(); delim=' ')
end
read(write_space_separated(), String)
```

```
name grade_2020
Sally 1.0
Bob 5.0
Alice 8.5
Hank 4.0
```

By convention, its still best to give files with special delimiters, such as ";" the ".csv" extension.

Loading CSV files using `csv.jl` is done in a similar way. You can use `CSV.read` and specify in what kind of format you want the output. We specify a `DataFrame`.

```
path = write_grades_csv()
CSV.read(path, DataFrame)
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

Conveniently, `csv.jl` will automatically infer column types for us:

```
path = write_grades_csv()
df = CSV.read(path, DataFrame)
```

4x2 DataFrame		
Row	name	grade_2020
	String	Float64
1	Sally	1.0
2	Bob	5.0
3	Alice	8.5
4	Hank	4.0

It works even for far more complex data:

```
my_data = """
a,b,c,d,e
Kim,2018-02-03,3,4.0,2018-02-03T10:00
"""

path = "my_data.csv"
write(path, my_data)
df = CSV.read(path, DataFrame)
```

1x5 DataFrame				
Row	a	b	c	d
	String	Date	Int64	Float64
1	Kim	2018-02-03	3	4.0

1x5 DataFrame				
Row	a	b	c	d
	String	Date	Int64	Float64
1	Kim	2018-02-03	3	4.0

1x5 DataFrame				
Row	a	b	c	e
	String	Date	Int64	DateTime
1	Kim	2018-02-03	3	2018-02-03T10:00:00

These CSV basics should cover most use cases. For more information, see the `csv.jl` documentation{target="_blank"} and especially the `CSV.File` constructor docstring.

4.1.2 Excel

There are multiple Julia packages to read Excel files. In this book, we will only look at `XLSX.jl`, because it is the most actively maintained package in the Julia's ecosystem that deals with Excel data. As a second benefit, `XLSX.jl` is written in pure Julia, which makes it easy for us to inspect and understand what's going on under the hood.

Install the `XLSX.jl` package via:

```
julia> ]
pkg> add XLSX
```

and load it with:

```
import XLSX
```

To write files, we define a little helper function for data and column names:

```
function write_xlsx(name, df::DataFrame)
    path = "$name.xlsx"
    data = collect(eachcol(df))
    cols = names(df)
    XLSX.writetable(path, data, cols)
end
```

Now, we can easily write the grades to an Excel file:

```
function write_grades_xlsx()
    path = "grades"
    write_xlsx(path, grades_2020())
    "$path.xlsx"
end
```

When reading it back, we will see that `xlsx.jl` puts the data in a `XLSXFile` type which we can access the desired `sheet` much like a `Dict`:

```
path = write_grades_xlsx()
xf = XLSX.readxlsx(path)
```

```
XLSXFile("grades.xlsx") containing 1 Worksheet
  sheetname size      range
-----
  Sheet1  5x2      A1:B5
```

```
xf = XLSX.readxlsx(write_grades_xlsx())
sheet = xf["Sheet1"]
XLSX.eachtablerow(sheet) |> DataFrame
```

	name	grade_2020
	Sally	1.0
	Bob	5.0
	Alice	8.5
	Hank	4.0

Notice that we cover just the basics of `xlsx.jl` but more powerful usage and customizations are available. For more information and options, see the `xlsx.jl` documentation.

4.2 Index and Summarize

Let's go back to the example `grades_2020()` data defined before:

```
grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

To retrieve a **vector** for `name`, we can access the `DataFrame` with the `.`, as we did previously with `structs` in Section 3:

```
function names_grades1()
    df = grades_2020()
    df.name
end
names_grades1()
```

```
["Sally", "Bob", "Alice", "Hank"]
```

or we can index a `DataFrame` much like an `Array` with symbols and special characters. The **second index is the column indexing**:

```
function names_grades2()
    df = grades_2020()
    df[!, :name]
end
names_grades2()
```

```
["Sally", "Bob", "Alice", "Hank"]
```

Note that `df.name` is exactly the same as `df[!, :name]`, which you can verify yourself by doing:

```
julia> df = DataFrame(id = [1]);
julia> @edit df.name
```

In both cases, it gives you the column `:name`. There also exists `df[:, :name]` which copies the column `:name`. In most cases, `df[!, :name]` is the best bet since it is more versatile and does an in-place modification.

For any **row**, say the second row, we can use the **first index as row indexing**:

```
df = grades_2020()
df[2, :]
```

name	grade_2020
Bob	5.0

or create a function to give us any row `i` we want:

```
function grade_2020(i::Int)
    df = grades_2020()
    df[i, :]
end
grade_2020(2)
```

name	grade_2020
Bob	5.0

We can also get only `names` for the first 2 rows using **slicing** (again similar to an `Array`):

```
grades_indexing(df) = df[1:2, :name]
grades_indexing(grades_2020())
```

```
["Sally", "Bob"]
```

If we assume that all names in the table are unique, we can also write a function to obtain the grade for a person via their `name`. To do so, we convert the table back to one of Julia's basic data structures (see Section 3.2) which is capable of creating mappings, namely `Dicts`:

```
function grade_2020(name::String)
    df = grades_2020()
    dic = Dict(zip(df.name, df.grade_2020))
    dic[name]
end
grade_2020("Bob")
```

```
5.0
```

which works because `zip` loops through `df.name` and `df.grade_2020` at the same time like a “zipper”:

```
df = grades_2020()
collect(zip(df.name, df.grade_2020))
```

```
("Sally", 1.0)
```

```
("Bob", 5.0)
```

```
("Alice", 8.5)
```

```
("Hank", 4.0)
```

However, converting a `DataFrame` to a `Dict` is only useful when the elements are unique. Generally that is not the case and that's why we need to learn how to filter a `DataFrame`.

4.3 Filter and Subset

There are two ways to remove rows from a `DataFrame`, one is `filter` (Section 4.3.1) and the other is `subset` (Section 4.3.2). `filter` was added earlier to `DataFrames.jl`, is more powerful and more consistent with syntax from Julia base, so that is why we start discussing `filter` first. `subset` is newer and often more convenient.

4.3.1 Filter

From this point on, we start to get into the more powerful features of `DataFrames.jl`. To do this, we need to learn some functions, such as `select` and `filter`. But don't worry! It might be a relieve to know that the **general design goal of `DataFrames.jl` is to keep the number of functions that a user has to learn to a minimum¹**.

Like before, we resume from the `grades_2020`:

```
grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

We can filter rows by using `filter(source => f::Function, df)`. Note how this function is very similar to the function `filter(f::Function, V::Vector)` from `Julia Base` module. This is because `DataFrames.jl` uses **multiple dispatch** (see Section 2.3.3) to define a new method of `filter` that accepts a `DataFrame` as argument.

At first sight, defining and working with a function `f` for filtering can be a bit difficult to use in practice. Hold tight, that the effort is well-paid, since **it is a very powerful way of filtering data**. As a simple example, we can create a function `equals_alice` that checks whether its input equals "Alice":

```
equals_alice(name::String) = name == "Alice"  
equals_alice("Bob")
```

```
false
```

```
equals_alice("Alice")
```

```
true
```

¹According to Bogumił Kamiński (lead developer and maintainer of `DataFrames.jl`) on Discourse.

Equipped with such a function, we can use it as our function `f` to filter out all the rows for which `name` equals “Alice”:

```
filter(:name => equals_alice, grades_2020())
```

name	grade_2020
Alice	8.5

Note that this doesn’t only work for `DataFrames`, but also for vectors:

```
filter(equals_alice, ["Alice", "Bob", "Dave"])
```

```
["Alice"]
```

We can make it a bit less verbose by using an **anonymous function** (see Section 3.1.4.4):

```
filter(n -> n == "Alice", ["Alice", "Bob", "Dave"])
```

```
["Alice"]
```

which we can also use on `grades_2020`:

```
filter(:name => n -> n == "Alice", grades_2020())
```

name	grade_2020
Alice	8.5

To recap, this function call can be read as “for each element in the column `:name`, let’s call the element `n`, check whether `n` equals Alice.” For some people, this is still too verbose. Luckily, Julia has added a *partial function application* of `==`. The details of these words are not important, only that you can use it just like any other function:

```
filter(:name => ==(Alice), grades_2020())
```

name	grade_2020
Alice	8.5

To get all the rows which are *not* Alice, `==` (equality) can be replaced by `!=` (inequality) in all previous examples:

```
filter(:name => !=("Alice"), grades_2020())
```

name	grade_2020
Sally	1.0
Bob	5.0
Hank	4.0

Now, to show **why anonymous functions are so powerful**, we can come up with a slightly more complex filter. In this filter, we want to have the people whose name start with A or B **and** have a grade above 6:

```
function complex_filter(name, grade)::Bool
    interesting_name = startswith(name, 'A') || startswith(name, 'B')
    interesting_grade = 6 < grade
    interesting_name && interesting_grade
end
```

```
filter([:name, :grade_2020] => complex_filter, grades_2020())
```

name	grade_2020
Alice	8.5

4.3.2 Subset

The `subset` function was added to make it easier to work with missing values (Section 4.5). In contrast to `filter`, `subset` works on complete columns instead of rows or single values. If we want to use our earlier defined functions, we should wrap it inside `ByRow`:

```
subset(grades_2020(), :name => ByRow(equals_alice))
```

name	grade_2020
Alice	8.5

Also note that the `DataFrame` is now the first argument `subset(df, args...)`, whereas in `filter` it was the second one `filter(f, df)`. The reason for this is that Julia defines `filter` as `filter(f, V::Vector)` and `DataFrames.jl` chose to maintain consistency with existing Julia functions that were extended to `DataFrames` types by multiple dispatch.

NOTE: Most of native `DataFrames.jl` functions, which `subset` belongs to, have a **consistent function signature that always takes a DataFrame as first argument**.

Just like with `filter`, we can also use anonymous functions inside `subset`:

```
subset(grades_2020(), :name => ByRow(name -> name == "Alice"))
```

name	grade_2020
Alice	8.5

Or, the partial function application for ==:

```
subset(grades_2020(), :name => ByRow==(("Alice")))
```

name	grade_2020
Alice	8.5

Ultimately, let's show the real power of `subset`. First, we create a dataset with some missing values:

```
function salaries()
    names = ["John", "Hank", "Karen", "Zed"]
    salary = [1_900, 2_800, 2_800, missing]
    DataFrame(; names, salary)
end
salaries()
```

Table 4.27: Salaries.

names	salary
John	1900
Hank	2800
Karen	2800
Zed	missing

This data is about a plausible situation where you want to figure out your colleagues' salaries, and haven't figured it out for Zed yet. Even though we don't want to encourage these practices, we suspect it is an interesting example. Suppose we want to know who earns more than 2000. If we use `filter`, without taking the `missing` values into account, it will fail:

```
filter(:salary => >(2_000), salaries())
```

```
TypeError: non-boolean (Missing) used in boolean context
Stacktrace:
 [1] (::DataFrames.var"#89#90"{Base.Fix2{typeof(>), Int64}})(x::Missing)
 @ DataFrames ~/.julia/packages/DataFrames/vuMM8/src/abstractdataframe/abstractdataframe.jl
```

```
:1043  
...
```

subset will also fail, but it will fortunately point us towards an easy solution:

```
subset(salaries(), :salary => ByRow(>(2_000)))
```

```
ArgumentError: missing was returned in condition number 1 but only true or false are allowed;  
    pass skipmissing=true to skip missing values  
Stacktrace:  
[1] _and(x::Missing)  
@ DataFrames ~/julia/packages/DataFrames/vuMM8/src/abstractdataframe/subset.jl:11  
...
```

So, we just need to pass the keyword argument skipmissing=true:

```
subset(salaries(), :salary => ByRow(>(2_000)); skipmissing=true)
```

names	salary
Hank	2800
Karen	2800

4.4 Select

Whereas **filter removes rows, select removes columns**. However, select is much more versatile than just removing columns, as we will discuss in this section. First, let's create a dataset with multiple columns:

```
function responses()  
    id = [1, 2]  
    q1 = [28, 61]  
    q2 = [:us, :fr]  
    q3 = ["F", "B"]  
    q4 = ["B", "C"]  
    q5 = ["A", "E"]  
    DataFrame(; id, q1, q2, q3, q4, q5)  
end  
responses()
```

Table 4.29: Responses.

id	q1	q2	q3	q4	q5
1	28	us	F	B	A
2	61	fr	B	C	E

Here, the data represents answers for five questions (q_1, q_2, \dots, q_5) in a given questionnaire. We will start by “selecting” a few columns from this dataset. As usual, we use symbols to specify columns:

```
select(responses(), :id, :q1)
```

	id	q1
1	28	
2	61	

We can also use strings if we want:

```
select(responses(), "id", "q1", "q2")
```

	id	q1	q2
1	28	us	
2	61	fr	

To select everything except one or more columns, use `Not` with either a single column:

```
select(responses(), Not(:q5))
```

	id	q1	q2	q3	q4
1	28	us	F	B	
2	61	fr	B	C	

Or, with multiple columns:

```
select(responses(), Not([:q4, :q5]))
```

	id	q1	q2	q3
1	28	us	F	
2	61	fr	B	

It's also fine to mix and match columns that we want to preserve with columns that we do `Not` want to select:

```
select(responses(), :q5, Not(:id))
```

	q5	q1	q2	q3	q4
A	28	us	F	B	
E	61	fr	B	C	

Note how `q5` is now the first column in the `DataFrame` returned by `select`. There is a more clever way to achieve the same using `:.` The colon `:` can be thought of as “all the columns that we didn’t include yet.” For example:

```
select(responses(), :q5, :)
```

	q5	id	q1	q2	q3	q4
A	1	28	us	F	B	
E	2	61	fr	B	C	

Or, to put `q5` at the second position²:

```
select(responses(), 1, :q5, :)
```

	id	q5	q1	q2	q3	q4
1	A	28	us	F	B	
2	E	61	fr	B	C	

NOTE: As you might have observed there are several ways to select a column. These are known as *column selectors*.

We can use:

- `Symbol`: `select(df, :col)`
- `String`: `select(df, "col")`
- `Integer`: `select(df, 1)`

Even renaming columns is possible via `select` using the `source => target` pair syntax:

```
select(responses(), 1 => "participant", :q1 => "age", :q2 => "nationality")
```

²Thanks to Sudete on Discourse for this suggestion.

	participant	age	nationality
1	28	us	
2	61	fr	

Additionally, thanks to the “splat” operator . . . (see Section 3.2.10) operator, we can also write:

```
renames = (1 => "participant", :q1 => "age", :q2 => "nationality")
select(responses(), renames...)
```

	participant	age	nationality
1	28	us	
2	61	fr	

4.5 Types and Missing Data

As discussed in Section 4.1, `csv.jl` will do its best to guess what kind of types your data have as columns. However, this won’t always work perfectly. In this section, we show why suitable types are important and we fix wrong data types. To be more clear about the types, we show the text output for `DataFrames` instead of a pretty-formatted table. In this section, we work with the following dataset:

```
function wrong_types()
    id = 1:4
    date = ["28-01-2018", "03-04-2019", "01-08-2018", "22-11-2020"]
    age = ["adolescent", "adult", "infant", "adult"]
    DataFrame(; id, date, age)
end
wrong_types()
```

Row	id	date	age
	Int64	String	String
1	1	28-01-2018	adolescent
2	2	03-04-2019	adult
3	3	01-08-2018	infant
4	4	22-11-2020	adult

Because the date column has the wrong type, sorting won’t work correctly:

```
sort(wrong_types(), :date)
```

Row	id	date	age
	Int64	String	String
1	1	28-01-2018	adolescent
2	2	03-04-2019	adult
3	3	01-08-2018	infant
4	4	22-11-2020	adult

```
1 |      3 01-08-2018 infant
2 |      2 03-04-2019 adult
3 |      4 22-11-2020 adult
4 |      1 28-01-2018 adolescent
```

To fix the sorting, we can use the `Date` module from Julia's standard library as described in Section 3.4.1:

```
function fix_date_column(df::DataFrame)
    strings2dates(dates::Vector) = Date.(dates, DateFormat("dd-mm-yyyy"))
    dates = strings2dates(df[:, :date])
    df[:, :date] = dates
    df
end
fix_date_column(wrong_types())
```

4x3 DataFrame

Row	id	date	age
	Int64	Date	String
1	1	2018-01-28	adolescent
2	2	2019-04-03	adult
3	3	2018-08-01	infant
4	4	2020-11-22	adult

Now, sorting will work as intended:

```
df = fix_date_column(wrong_types())
sort(df, :date)
```

4x3 DataFrame

Row	id	date	age
	Int64	Date	String
1	1	2018-01-28	adolescent
2	3	2018-08-01	infant
3	2	2019-04-03	adult
4	4	2020-11-22	adult

For the age column, we have a similar problem:

```
sort(wrong_types(), :age)
```

4x3 DataFrame

Row	id	date	age
	Int64	String	String
1	1	28-01-2018	adolescent
2	2	03-04-2019	adult
3	4	22-11-2020	adult
4	3	01-08-2018	infant

This isn't right, because an infant is younger than adults and adolescents. The solution for this issue and any sort of categorical data is to use `CategoricalArrays.jl`:

```
using CategoricalArrays
```

with the `CategoricalArrays.jl` package, we can add levels that represents the ordering of our categorical variable to our data:

```
function fix_age_column(df)
    levels = ["infant", "adolescent", "adult"]
    ages = categorical(df[!, :age]; levels, ordered=true)
    df[!, :age] = ages
    df
end
fix_age_column(wrong_types())
```

4x3 DataFrame			
Row	id	date	age
	Int64	String ...	Cat
1	1	28-01-2018	adolescent
2	2	03-04-2019	adult
3	3	01-08-2018	infant
4	4	22-11-2020	adult

NOTE: Also note that we are passing the argument `ordered=true` which tells `CategoricalArrays.jl`'s `categorical` function that our categorical data is “ordered.” Without this any type of sorting or bigger/smaller comparissons would not be possible.

Now, we can sort the data correctly on the age column:

```
df = fix_age_column(wrong_types())
sort(df, :age)
```

4x3 DataFrame			
Row	id	date	age
	Int64	String ...	Cat
1	3	01-08-2018	infant
2	1	28-01-2018	adolescent
3	2	03-04-2019	adult
4	4	22-11-2020	adult

Because we have defined convenient functions, we can now define our fixed data by just performing the function calls:

```
function correct_types()
    df = wrong_types()
    df = fix_date_column(df)
    df = fix_age_column(df)
end
correct_types()
```

4x3 DataFrame			
Row	id	date	age
	Int64	Date	Cat
1	1	2018-01-28	adolescent
2	2	2019-04-03	adult
3	3	2018-08-01	infant
4	4	2020-11-22	adult

Since age in our data is ordinal (`ordered=true`), we can properly compare categories of age:

```
df = correct_types()  
a = df[1, :age]  
b = df[2, :age]  
a < b
```

```
true
```

which would give wrong comparisons if the element type were strings:

```
"infant" < "adult"
```

```
false
```

4.6 Join

At the start of this chapter, we showed multiple tables and raised questions also related to multiple tables. However, we haven't talked about combining tables yet, which we will do in this section. In `DataFrames.jl`, combining multiple tables is done via *joins*. Joins are extremely powerful, but it might take a while to wrap your head around. It is not necessary to know the joins below by heart, the `DataFrames.jl` documentation along with this book will list them for you. But, it's essential to know that joins exists. If you ever find yourself looping over rows in a `DataFrame` and comparing it with other data, then you probably need one of the joins below.

In Section 4, we've introduced the grades for 2020 with `grades_2020`:

```
grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

Now, we're going to combine `grades_2020` with grades from 2021:

```
grades_2021()
```

name	grade_2021
Bob	9.5
Sally	9.5
Hank	6.0

To do this, we are going to use joins. `DataFrames.jl` lists no less than seven kinds of joins. This might seem daunting at first, but hang on because they are all useful and we will showcase them all.

4.6.1 innerjoin

This first is `innerjoin`. Suppose that we have two datasets `A` and `B` with respectively columns `A_1`, `A_2`, ..., `A_n` and `B_1`, `B_2`, ..., `B_m` **and** one of the columns has the same name, say `A_1` and `B_1` are both called `:id`. Then, the inner join on `:id` will go through all the elements in `A_1` and compare it to the elements in `B_1`. If the elements are **the same**, then it will add all the information from `A_2`, ..., `A_n` and `B_2`, ..., `B_m` after the `:id` column.

Okay, so no worries if you didn't get this description. The result on the grades datasets looks like this:

```
innerjoin(grades_2020(), grades_2021(); on=:name)
```

name	grade_2020	grade_2021
Sally	1.0	9.5
Hank	4.0	6.0

Note that only “Sally” and “Hank” are in both datasets. The name *inner* join makes sense since, in mathematics, the *set intersection* is defined by “all elements in A , that are also in B , or all elements in B that are also in A .”

4.6.2 outerjoin

Maybe, you're now thinking “aha, if we have an *inner*, then we probably also have an *outer*.” Yes, you've guessed right!

The `outerjoin` is much less strict than the `innerjoin` and just takes any row it can find which contains a name in **at least one of the dataset**:

```
outerjoin(grades_2020(), grades_2021(); on=:name)
```

	name	grade_2020	grade_2021
	Sally	1.0	9.5
	Hank	4.0	6.0
	Bob	5.0	missing
	Alice	8.5	missing
	Bob 2	missing	9.5

So, this method can create `missing` data even though none of the original datasets had missing values.

4.6.3 crossjoin

We can get even more `missing` data if we use the `crossjoin`. This gives the **Cartesian product of the rows**, which is basically multiplication of rows, that is, for every row create a combination with any other row:

```
crossjoin(grades_2020(), grades_2021(); on=:id)
```

```
MethodError: no method matching crossjoin(::DataFrame, ::DataFrame; on=:id)
Closest candidates are:
  crossjoin(::AbstractDataFrame, ::AbstractDataFrame; makeunique) at /home/runner/.julia/
    packages/DataFrames/vuMM8/src/join/composer.jl:1332 got unsupported keyword argument "on"
  crossjoin(::AbstractDataFrame, ::AbstractDataFrame, !Matched::AbstractDataFrame...;
    makeunique) at /home/runner/.julia/packages/DataFrames/vuMM8/src/join/composer.jl:1343
    got unsupported keyword argument "on"
...
...
```

Oops. Since `crossjoin` doesn't take the elements in the row into account, we don't need to specify the `on` argument for what we want to join:

```
crossjoin(grades_2020(), grades_2021())
```

```
ArgumentError: Duplicate variable names: :name. Pass makeunique=true to make them unique using
  a suffix automatically.
Stacktrace:
 [1] add_names(ind::DataFrames.Index, add_ind::DataFrames.Index; makeunique::Bool)
   @ DataFrames ~/.julia/packages/DataFrames/vuMM8/src/other/index.jl:323
 [2] merge!(x::DataFrames.Index, y::DataFrames.Index; makeunique::Bool)
...
...
```

Oops again. This is a very common error with `DataFrames` and `joins`. The tables for the 2020 and 2021 grades have a duplicate column name, namely `:name`. Like before, the error that `DataFrames.jl` outputs shows a simple suggestion that might fix the issue. We can just pass `makeunique=true` to solve this:

```
crossjoin(grades_2020(), grades_2021(); makeunique=true)
```

name	grade_2020	name_1	grade_2021
Sally	1.0	Bob 2	9.5
Sally	1.0	Sally	9.5
Sally	1.0	Hank	6.0
Bob	5.0	Bob 2	9.5
Bob	5.0	Sally	9.5
Bob	5.0	Hank	6.0
Alice	8.5	Bob 2	9.5
Alice	8.5	Sally	9.5
Alice	8.5	Hank	6.0
Hank	4.0	Bob 2	9.5
Hank	4.0	Sally	9.5
Hank	4.0	Hank	6.0

So, now, we have one row for each grade from everyone in grades 2020 and grades 2021 datasets. For direct queries, such as “who has the highest grade?” the Cartesian product is usually not so useful, but for “statistical” queries, it can be.

4.6.4 leftjoin and rightjoin

More useful for scientific data projects are the `leftjoin` and `rightjoin`. The left join gives all the elements in the *left* DataFrame:

```
leftjoin(grades_2020(), grades_2021(); on=:name)
```

name	grade_2020	grade_2021
Sally	1.0	9.5
Hank	4.0	6.0
Bob	5.0	missing
Alice	8.5	missing

Here, grades for “Bob” and “Alice” were missing in the grades 2021 table, so that’s why there are also missing elements. The right join does sort of the opposite:

```
rightjoin(grades_2020(), grades_2021(); on=:name)
```

name	grade_2020	grade_2021
Sally	1.0	9.5
Hank	4.0	6.0
Bob 2	missing	9.5

Now, grades in 2020 are missing.

Note that `leftjoin(A, B) != rightjoin(B, A)`, because the order of the columns will differ. For example, compare the output below to the previous output:

```
leftjoin(grades_2021(), grades_2020(); on=:name)
```

name	grade_2021	grade_2020
Sally	9.5	1.0
Hank	6.0	4.0
Bob 2	9.5	missing

4.6.5 semijoin and antijoin

Lastly, we have the `semijoin` and `antijoin`.

The semi join is even more restrictive than the inner join. It returns **only the elements from the left DataFrame which are in both DataFrames**. This is like a combination of the left join with the inner join.

```
semijoin(grades_2020(), grades_2021(); on=:name)
```

name	grade_2020
Sally	1.0
Hank	4.0

The opposite of the semi join is the anti join. It returns **only the elements from the left DataFrame which are not in the right DataFrame**:

```
antijoin(grades_2020(), grades_2021(); on=:name)
```

name	grade_2020
Bob	5.0
Alice	8.5

4.7 Variable Transformations

In Section 4.3.1, we saw that `filter` works by taking one or more source columns and filtering it by applying a “filtering” function. To recap here’s an example of filter using the `source => f::Function` syntax: `filter(:name => name -> name == "Alice", df)`.

In Section 4.4, we saw that `select` can take one or more source columns and put it into one or more target columns `source => target`. Also to recap here’s an example: `select(df, :name => :people_names)`.

In this section, we discuss how to **transform** variables, that is, how to **modify data**. In `DataFrames.jl`, the syntax is `source => transformation => target`.

Like before, we use the `grades_2020` dataset:

```
function grades_2020()
    name = ["Sally", "Bob", "Alice", "Hank"]
    grade_2020 = [1, 5, 8.5, 4]
    DataFrame(; name, grade_2020)
end
grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

Suppose we would want to increase all the grades in `grades_2020` by 1. First, we define a function that take as argument a vector of data and return all of its elements increased by 1. Then we use the `transform` function from `DataFrames.jl` that, as all native `DataFrames.jl`’s functions, takes a `DataFrame` as first argument followed by the transformation syntax:

```
plus_one(grades) = grades .+ 1
transform(grades_2020(), :grade_2020 => plus_one)
```

name	grade_2020	grade_2020_plus_one
Sally	1.0	2.0
Bob	5.0	6.0
Alice	8.5	9.5
Hank	4.0	5.0

Here, the `plus_one` function receives the whole `:grade_2020` column. That is the reason why we've added the broadcasting "dot" `.` before the plus `+` operator. For a recap on broadcasting please see Section 3.2.1.

Like said above, the `DataFrames.jl` minilanguage is always `source => transformation => target`. So, if we want to keep the naming of the `target` column in the output, we can do:

```
transform(grades_2020(), :grade_2020 => plus_one => :grade_2020)
```

name	grade_2020
Sally	2.0
Bob	6.0
Alice	9.5
Hank	5.0

We can also use the keyword argument `renamecols=false`:

```
transform(grades_2020(), :grade_2020 => plus_one; renamecols=false)
```

name	grade_2020
Sally	2.0
Bob	6.0
Alice	9.5
Hank	5.0

The same transformation can also be written with `select` as follows:

```
select(grades_2020(), :, :grade_2020 => plus_one => :grade_2020)
```

name	grade_2020
Sally	2.0
Bob	6.0
Alice	9.5
Hank	5.0

Where the `:` means "select all the columns" as described in Section 4.4. Alternatively, you can also use Julia's broadcasting and modify the column `grade_2020` by accessing it with `df.grade_2020`:

```
df = grades_2020()
```

```
df.grade_2020 = plus_one.(df.grade_2020)
df
```

name	grade_2020
Sally	2.0
Bob	6.0
Alice	9.5
Hank	5.0

But, although the last example is easier since it builds on more native Julia operations, **we strongly advise to use the functions provided by `DataFrames.jl` in most cases because they are more capable and easier to work with.**

4.7.1 Multiple Transformations

To show how to transform two columns at the same time, we use the left joined data from Section 4.6:

```
leftjoined = leftjoin(grades_2020(), grades_2021(); on=:name)
```

name	grade_2020	grade_2021
Sally	1.0	9.5
Hank	4.0	6.0
Bob	5.0	missing
Alice	8.5	missing

With this, we can add a column saying whether someone was approved by the criterion that all of his grades were above 5.5:

```
pass(A, B) = [5.5 < a || 5.5 < b for (a, b) in zip(A, B)]
transform(leftjoined, [:grade_2020, :grade_2021] => pass; renamecols=false)
```

name	grade_2020	grade_2021	grade_2020_grade_2021
Sally	1.0	9.5	true
Hank	4.0	6.0	true
Bob	5.0	missing	missing
Alice	8.5	missing	true

We can clean up the outcome and put the logic in a function to get a list of all the approved students:

```
function only_pass()
    leftjoined = leftjoin(grades_2020(), grades_2021(); on=:name)
    pass(A, B) = [5.5 < a || 5.5 < b for (a, b) in zip(A, B)]
    leftjoined = transform(leftjoined, [:grade_2020, :grade_2021] => pass => :pass)
    passed = subset(leftjoined, :pass; skipmissing=true)
    return passed.name
end
only_pass()
```

```
["Sally", "Hank", "Alice"]
```

4.8 Groupby and combine

In the R programming language, Wickham (2011) has popularized the, so called, split-apply-combine strategy for data transformations. In essence, this strategy **splits** a dataset into distinct groups, **applies** one or more functions to each group, and then **combines** the result. `DataFrames.jl` fully supports split-apply-combine. We will use student grades example like before. Suppose that we want to know each student's mean grade:

```
function all_grades()
    df1 = grades_2020()
    df1 = select(df1, :name, :grade_2020 => :grade)
    df2 = grades_2021()
    df2 = select(df2, :name, :grade_2021 => :grade)
    rename_bob2(data_col) = replace.(data_col, "Bob 2" => "Bob")
    df2 = transform(df2, :name => rename_bob2 => :name)
    return vcat(df1, df2)
end
all_grades()
```

name	grade
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0
Bob	9.5
Sally	9.5
Hank	6.0

The strategy is to **split** the dataset into distinct students, **apply** the mean function to each student, and **combine** the result.

The split is called `groupby` and we give as second argument the column ID that we want to split the dataset into:

```
groupby(all_grades(), :name)
```

```
GroupedDataFrame with 4 groups based on key: name
Group 1 (2 rows): name = "Sally"
Row | name      grade|
String  Float64-----|_____
1 | Sally      1.0
2 | Sally      9.5
Group 2 (2 rows): name = "Bob"
Row | name      grade|
String  Float64-----|_____
1 | Bob       5.0
2 | Bob       9.5
Group 3 (1 row): name = "Alice"
Row | name      grade|
String  Float64-----|_____
1 | Alice     8.5
Group 4 (2 rows): name = "Hank"
Row | name      grade|
String  Float64-----|_____
1 | Hank     4.0
2 | Hank     6.0
```

We apply the `mean` function from Julia's standard library `Statistics` module:

```
using Statistics
```

Applying this function is done by using the `combine` function:

```
gdf = groupby(all_grades(), :name)
combine(gdf, :grade => mean)
```

	name	grade_mean
	Sally	5.25
	Bob	7.25
	Alice	8.5
	Hank	5.0

Imagine having to do this without the `groupby` and `combine` functions. We would need to loop over our data to split it up into groups, then loop over each split to apply a function, **and** finally loop over each group to gather the final result. Therefore, the split-apply-combine technique is a great one to know.

4.8.1 Multiple Source Columns

But what if we want to apply a function on multiple columns of our dataset?

```
group = [:A, :A, :B, :B]
X = 1:4
Y = 5:8
df = DataFrame(; group, X, Y)
```

group	X	Y
A	1	5
A	2	6
B	3	7
B	4	8

This is accomplished in a similar manner:

```
gdf = groupby(df, :group)
combine(gdf, [:X, :Y] .-> mean; renamecols=false)
```

group	X	Y
A	1.5	5.5
B	3.5	7.5

Note that we've used the dot . operator before the right arrow => to indicate that the `mean` has to be applied to multiple source columns `[:X, :Y]`.

To use composable functions, a simple way is to create a function that does the intended composable transformations. For instance, for a series of values, let's first take the `mean` followed by `round` to a whole number (also known as an integer `Int`):

```
gdf = groupby(df, :group)
rounded_mean(data_col) = round(Int, mean(data_col))
combine(gdf, [:X, :Y] .-> rounded_mean; renamecols=false)
```

group	X	Y
A	2	6
B	4	8

5 Performance

So far, we didn't think about making our `DataFrames.jl` code **fast**. Like everything in Julia, `DataFrames.jl` can be really fast. In this section, we will give some performance tips and tricks.

6 Data Vis with Plots

Data visualization it is a vital part of almost any data analysis process. Here, in this chapter, we will consider several packages that can be used in Julia, namely `Plots.jl`, `StatsPlots.jl`, and `Makie.jl`

- Overview of the JuliaPlots ecosystem

Plots is

The default backend is GR.

```
using Plots, LaTeXStrings

function test_plots_layout()
    x = LinRangeπ(0, 2, 50)
    plt1 = Plots.plot(x, sin.(x), st = :scatter, label = "sin(x)",
        m = (3, :black, Plots.stroke(0)), leg = :bottomleft,
        fg_legend = :black, bg_legend = nothing)
    plt2 = Plots.plot(x, sin.(x), c = :black, m=(3,:d,:black,Plots.stroke(0)),
        label = "sin(x)", leg =:bottomleft, fg_legend = :black,
        bg_legend = nothing)
    plt3 = Plots.plot(x, [sin.(x), cos.(x)], c = :black, leg =:topright,
        m = (3, [:d :o], [:black :orangered], Plots.stroke(0)),
        label = ["sin(x)" "cos(x)"], fg_legend = nothing,
        bg_legend = :white, xlab =L"x")
    plt4 = Plots.plot(x, [sin.(x), cos.(x), -sin.(x), -cos.(x)], lw = 1.5,
        c = [:viridis :plasma :magma :inferno], linez = x,
        colorbar = false, legend =:false, xlab =L"x")
    Plots.plot(plt1, plt2, plt3, plt4, layout = (2,2), legendfont=(8,))
end
test_plots_layout()
```

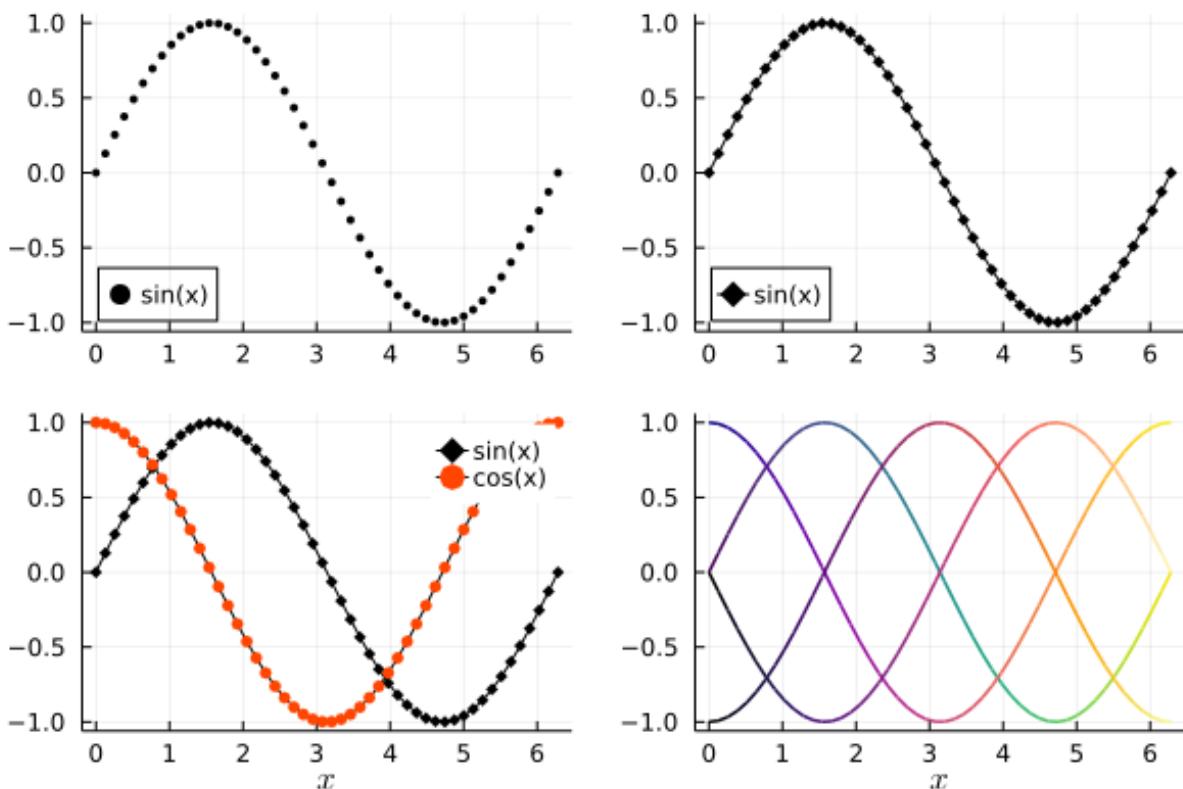


Figure 6.1: Test plots layout.

6.1 Layouts

- Overview on several ways to do layouts
- the layout argument, also cover the grid
- the `@layout` macro
- specific measures with the `Plots.PlotMeasures` submodule (additionally a footnote to say that we have it in `StatsPlots.PlotMeasures` also)
- adding subplots incrementally. Define `p1, p2, p3`; then do a `plot(p1, p2, p3; layout=l)`

6.2 Animations

Doing an animation with Plots is quite easy.

6.3 Other backends

Plotly, PGFPlotsX, PyPlot?

6.4 StatsPlots.jl

1. A brief intro and show that is is really just “syntactic sugar” for Plots.jl. Also note that works on all Plots.jl stuff:
 - Backends
 - Attributes
 - Colours
 - Layouts
 - Animations
 - Themes
2. Works on Types of Distributions and DataFrames, but we will only focus on DataFrames
3. the `@df` macro and the Symbols (`:col1, :col2`) instead of other input data (vectors, arrays, tuples etc.)
4. Recipes (I would cover almost all of them, except for andrewsplot MDS plot, Dendograms, QQ-Plot etc., since they are specific for modeling or simulation):
 - histogram, histogram2d and ea_histogram
 - groupedhist and groupedbar
 - boxplot
 - dotplot
 - violin
 - marginalhist, marginalkde and marginalsscatter
 - corrplot and cornerplot

7 Data Vis with Makie.jl

From the Japanese word Maki-e, which is a technique to sprinkle lacquer with gold and silver powder. Data is the gold and silver of our age, so let's spread it out beautifully on the screen!

Simon Danisch, Creator of Makie.jl

Makie.jl is a high-performance, extendable, and multi-platform plotting ecosystem for the Julia programming language. In our opinion, it is the prettiest and most versatile plotting package.

Makie.jl is the frontend package that defines all plotting functions. It is reexported by every backend (*the machinery behind-the-scenes making the figure*), so you don't have to specifically install or import it. There are three main backends which concretely implement all abstract rendering capabilities defined in Makie. One for non-interactive 2D publication-quality vector graphics: CairoMakie.jl. Another for interactive 2D and 3D plotting in standalone GLFW.jl windows (also GPU-powered), GLMakie.jl. And the third one, a WebGL-based interactive 2D and 3D plotting that runs within browsers, WGLMakie.jl. See Makie's documentation for more.

In this book we will only show examples for CairoMakie.jl and GLMakie.jl.

You can activate any backend by using the appropriate package and calling its `activate!` function. For example:

```
using GLMakie  
GLMakie.activate!()
```

Now, we will start with publication-quality plots.

7.1 CairoMakie.jl

Let's start with our first plot, a not so boring example but still simple enough, e.g. a line and some scatter points:

```
using CairoMakie  
CairoMakie.activate!()
```

```
scatterlines(1:10, 1:10)
```

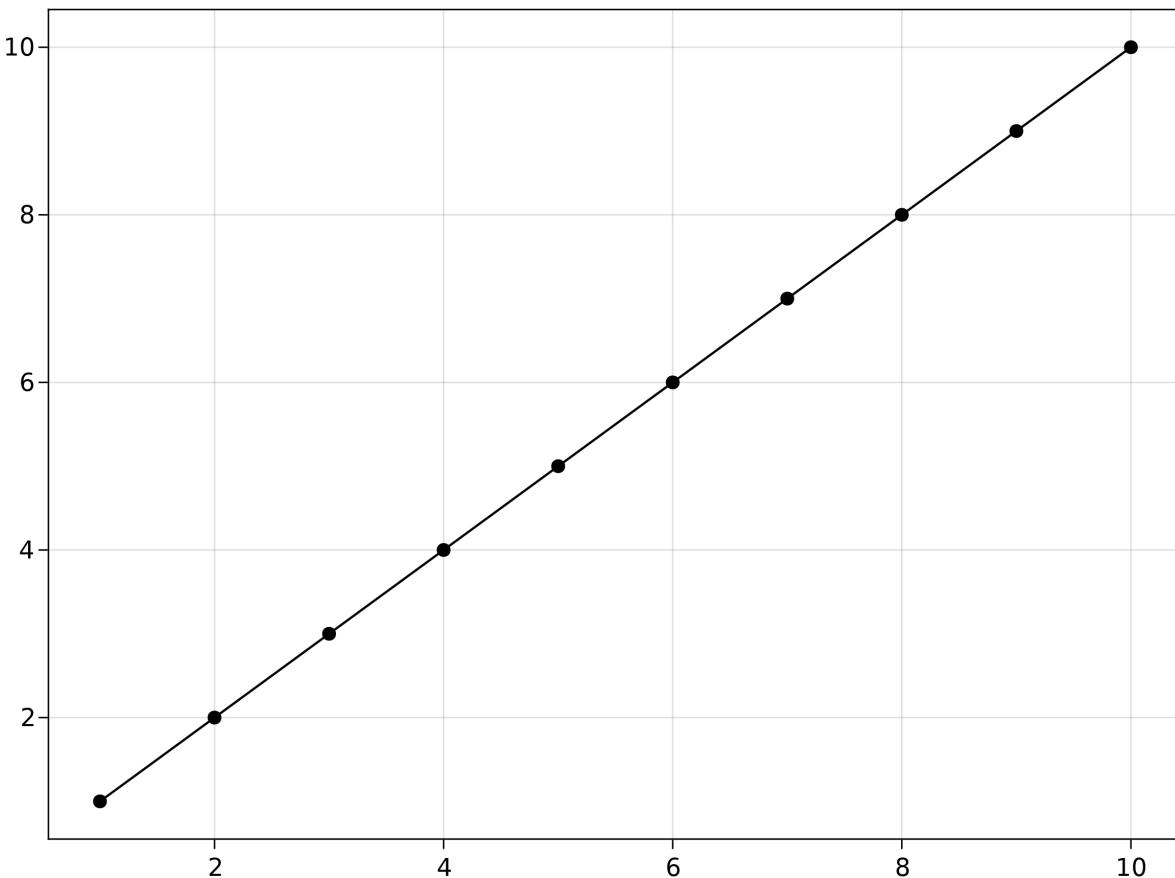


Figure 7.1: First plot.

The previous plot is the default output, which we probably need to tweak by using axis names and labels.

Also note that every plotting function like `scatterlines` creates and returns a new `Figure`, `Axis` and `plot` object in a collection called `FigureAxisPlot`. These are known as the `non-mutating` methods. On the other hand, the `mutating` methods (e.g. `scatterlines!`, note the `!`) just return a `plot` object which can be appended into a given `axis` or the `current_figure()`.

The next question that one might have is: how do I change the color or the marker type? This can be done via `attributes`, which we do in the next section.

7.2 Attributes

A custom plot can be created by using `attributes`. The attributes can be set through keyword arguments. A list of `attributes` for every plotting object can be viewed via:

```
fig, ax, pltobj = scatterlines(1:10)
pltobj.attributes
```

```
Attributes with 14 entries:
  color => black
```

```
colormap => viridis
colorrange => Automatic()
inspectable => true
linestyle => nothing
linewidth => 1.5
marker => Circle{T} where T
markercolor => black
markercolormap => viridis
markercolorange => Automatic()
markersize => 9
model => Float32[1.0 0.0 0.0 0.0; 0.0 1.0 0.0 0.0; 0.0 0.0 1.0 0.0; 0.0 0.0 0.0 1.0]
strokecolor => black
strokewidth => 0
```

or as a Dict calling `pltobject.attributes.attributes`.

Asking for help in the REPL as `?lines` or `help(lines)` for any given plotting function will show you their corresponding attributes plus a short description on how to use that specific function, e.g.:

```
help(lines)
```

```
lines(positions)
lines(x, y)
lines(x, y, z)
```

Creates a connected line plot for each element in (x, y, z), (x, y) or positions. |

Tip ||

You can separate segments by inserting NaNs.

`lines` has the following function signatures:

```
(Vector, Vector)
(Vector, Vector, Vector)
(Matrix)
```

Available attributes for `Lines{T}` where `T` are:

```
ambient
color
colormap
colorrange
cycle
diffuse
inspectable
lightposition
linestyle
linewidth
nan_color
overdraw
shininess
```

```
specular
ssao
transparency
visible
```

But not only the plot objects have attributes, also the `Axis` and `Figure` objects do. For example, for `Figure`, we have `backgroundcolor`, `resolution`, `font` and `fontsize` and the `figure_padding` which changes the amount of space around the figure content, see the grey area in the plot. It can take one number for all sides, or a tuple of four numbers for left, right, bottom and top.

`Axis` has a lot more, some of them are `backgroundcolor`, `xgridcolor` and `title`. For a full list just type `help(Axis)`.

Hence, for our next plot we will call several attributes at once as follows:

```
figure = (; figure_padding=5, resolution=(600,400),
           backgroundcolor=:grey90, fontsize=16, font="sans")

axis = (; xlabel="x", ylabel="x^2", title="title",
         backgroundcolor=:white, xgridstyle=:dash, ygridstyle=:dash)

lines(1:10, (1:10).^2; color=:black, linewidth=2, linestyle=:dash, figure, axis)
```

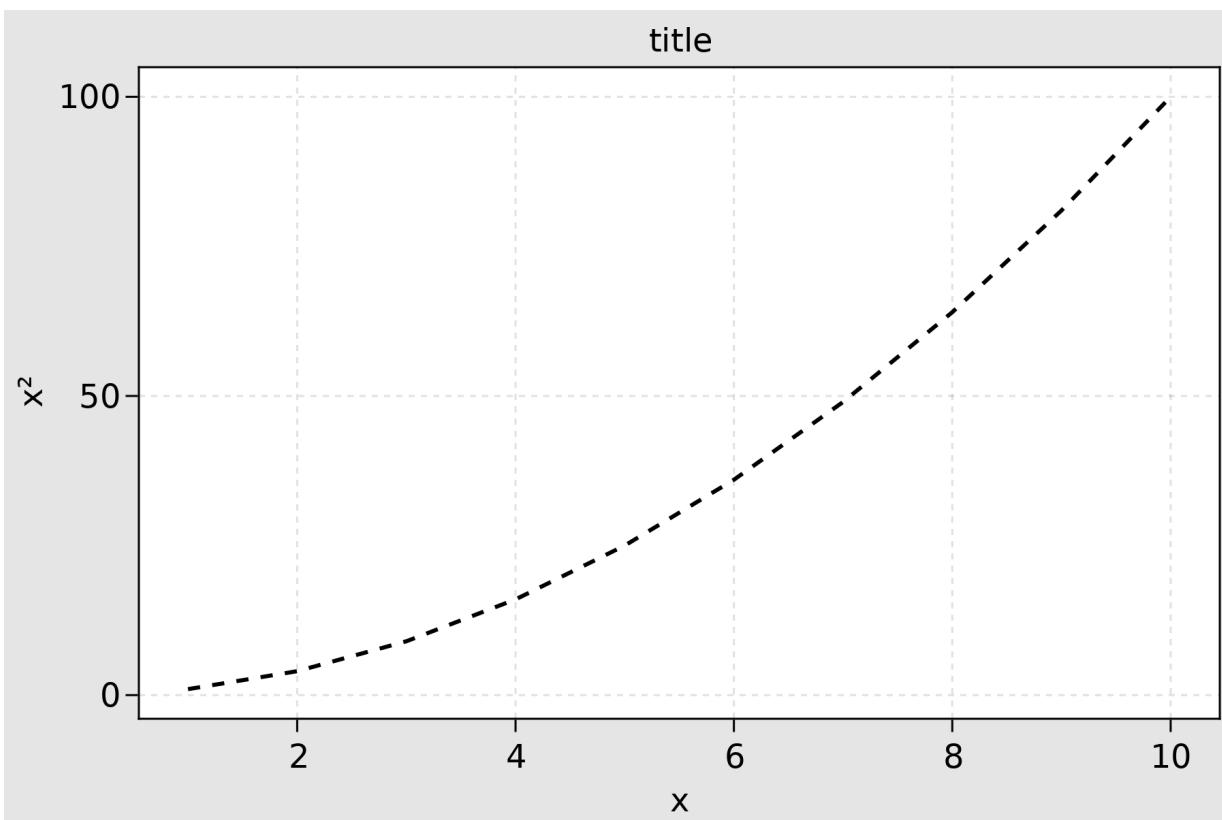


Figure 7.2: Custom plot.

This example has already most of the attributes that most users will normally use. Probably, a `Legend` will also be good to have. Which for more than one function will make more sense. So, let's append an-

other mutation `plot` object and add the corresponding legends by calling `axislegend`. This will collect all the labels you might have passed to your plotting functions (Figure 7.3):

```
lines(1:10, (1:10).^2; label = "x²", linewidth = 2, linestyle = nothing,
    figure = (; figure_padding = 5, resolution = (600,400),
        backgroundcolor = :grey90, fontsize = 16, font = "sans"),
    axis = (; xlabel = "x", title = "title", backgroundcolor = :white,
        xgridstyle=:dash, ygridstyle=:dash))
scatterlines!(1:10, (10:-1:1).^2; label = "Reverse(x)²")
axislegend("legend"; position = :ct)
current_figure()
```

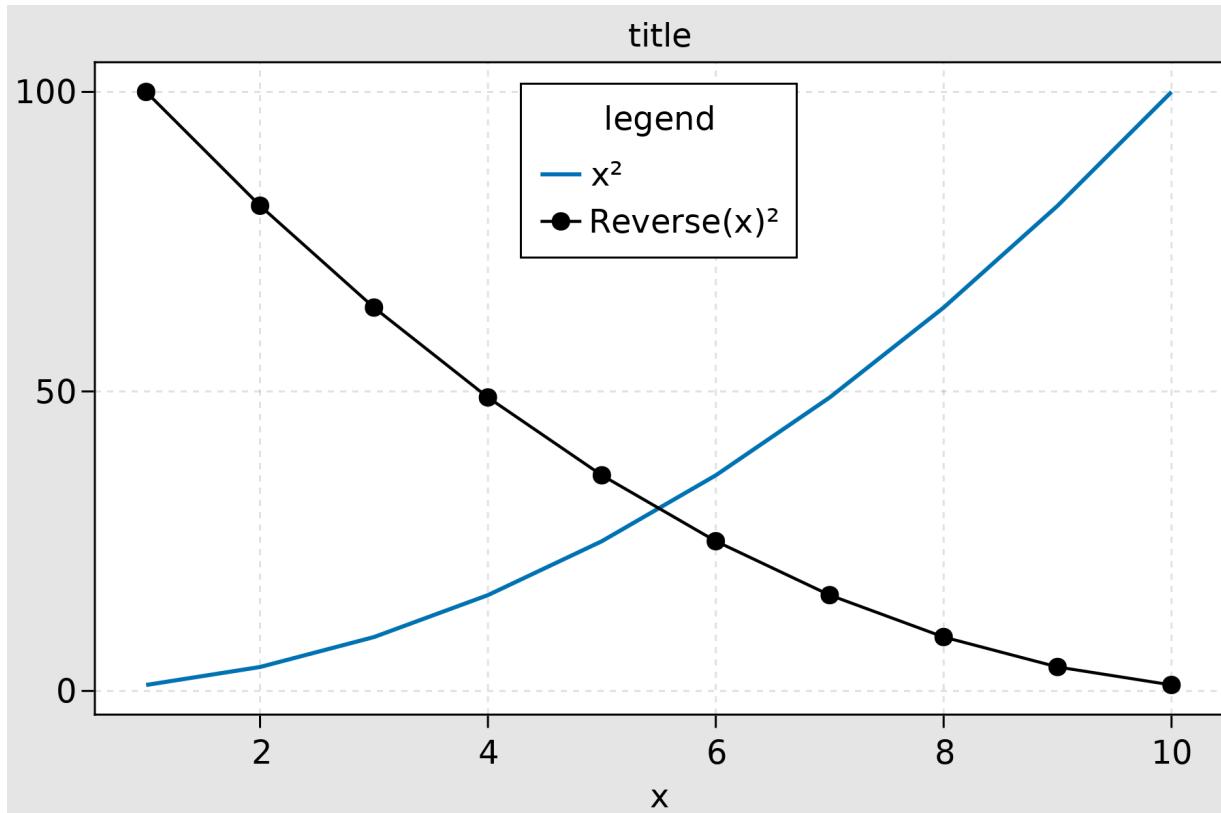


Figure 7.3: Custom plot legend.

However, having to write this so much code just for two lines can become cumbersome and tired. So if you plan on doing a lot of plots with the same general aesthetics then setting a theme will be better. We can do this with `set_theme!()` as the following example illustrates, not a particular good set of attributes but you'll get the idea.

```
set_theme!(resolution = (600,400), backgroundcolor = (:orange, 0.5),
    fontsize = 16, font = "sans",
    Axis = (backgroundcolor = :white, xgridstyle=:dash, ygridstyle=:dash),
    Legend = (bgcolor = (:red,0.2), framecolor = :dodgerblue)
)

lines(1:10, (1:10).^2; label = "x²", linewidth = 2, linestyle = nothing,
    axis = (; xlabel = "x", title = "title"))
scatterlines!(1:10, (10:-1:1).^2; label = "Reverse(x)²")
```

```
axislegend("legend"; position = :ct)
current_figure()
set_theme!() # in order to go back to the default settings.
```

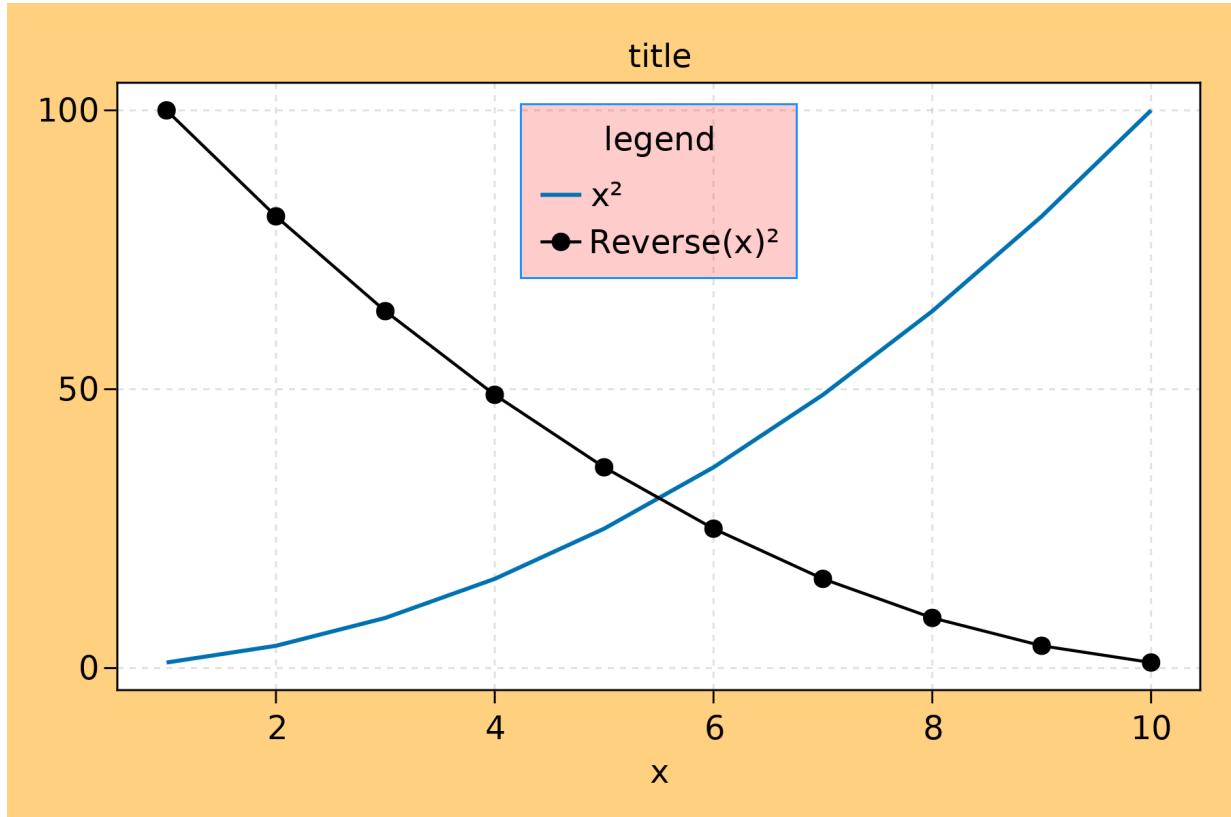


Figure 7.4: set theme

For more on themes please go to Section 7.3.

Before moving on into the next section, it's worthwhile to see an example where an array of attributes are passed at once to a plotting function. For this example, we will use the `scatter` plotting function to do a bubble plot.

```
let
    Random.seed!(123)
    n = 100
    fig, ax, pltobj = scatter(randn(n), randn(n); color = randn(n),
        label = "Bubbles", colormap = :plasma, markersize = 25*rand(n),
        figure = (; resolution = (550,400)), axis = (; aspect= DataAspect()))
    limits!(-3, 3, -3, 3)
    Legend(fig[1,2], ax, valign = :top)
    Colorbar(fig[1,2], pltobj, height = Relative(3/4))
    fig
end
```

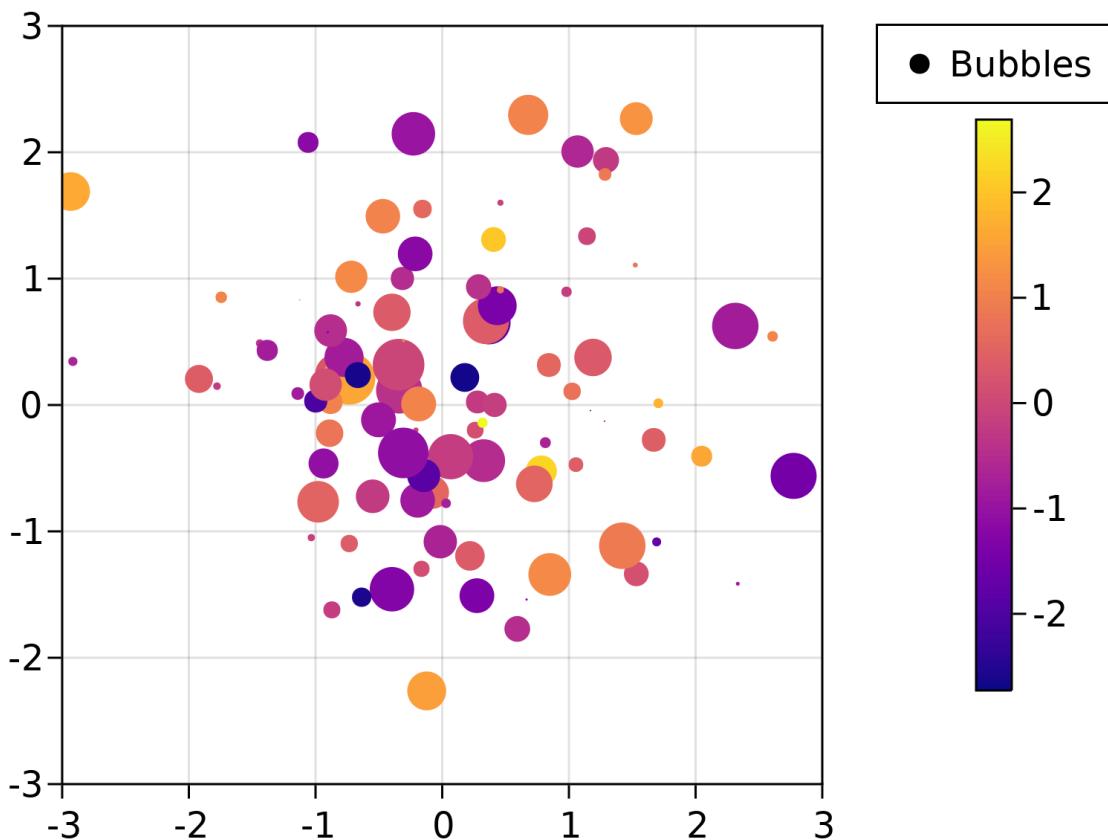


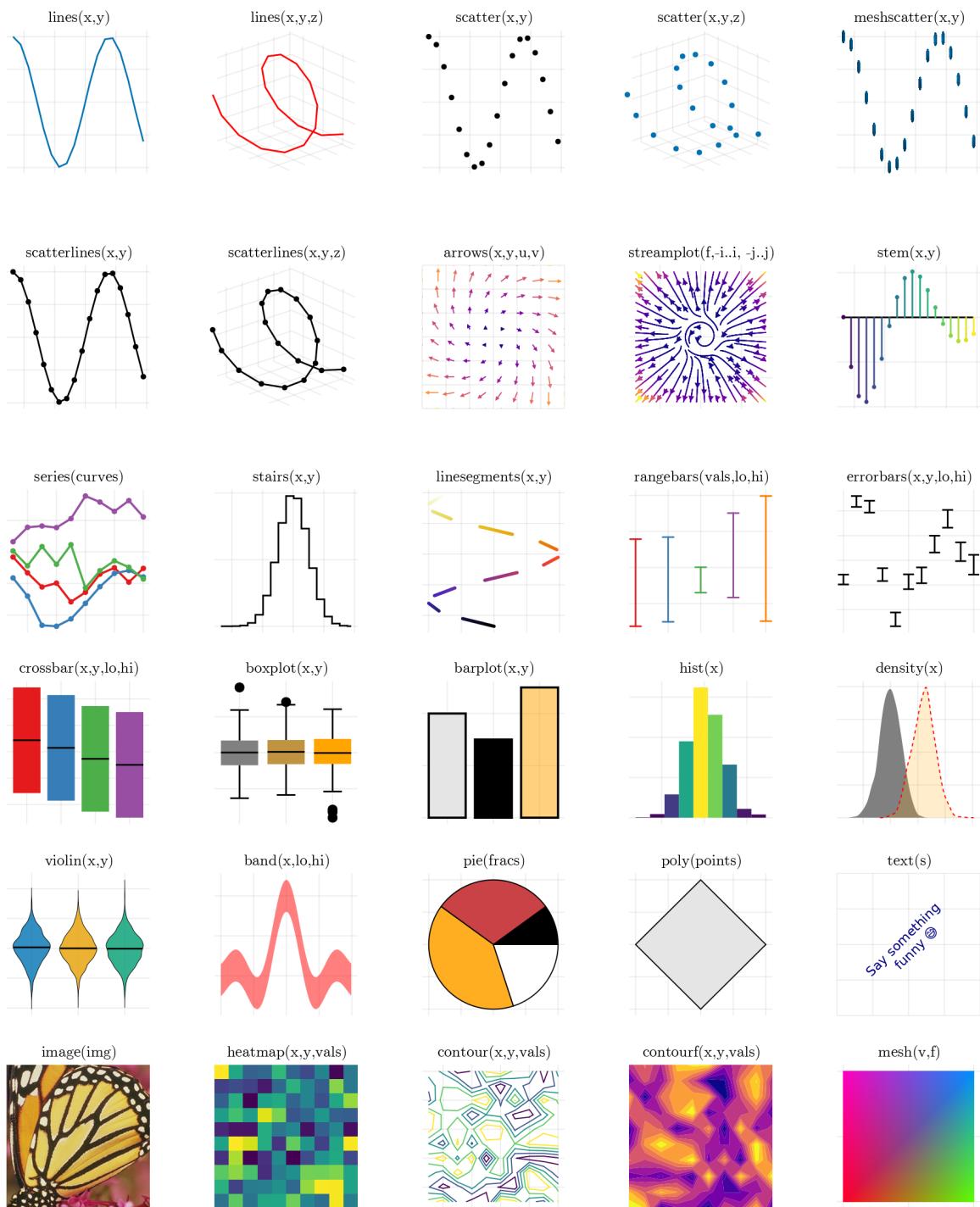
Figure 7.5: Bubble plot

where we have decomposed the tuple `FigureAxisPlot` into `fig`, `ax`, `pltobj`, in order to be able to add a `Legend` and `Colorbar` outside of the plot object. We will discuss layout options in more detail in Section 7.6.

We have done some basic but still interesting examples to show how to use `Makie.jl` and by now you might be wondering, what else can we do? What are all the possible plotting functions available in `Makie.jl`? To answer this question, a *cheat sheet* is shown in Figure 7.6. These work especially well with `CairoMakie.jl` backend.



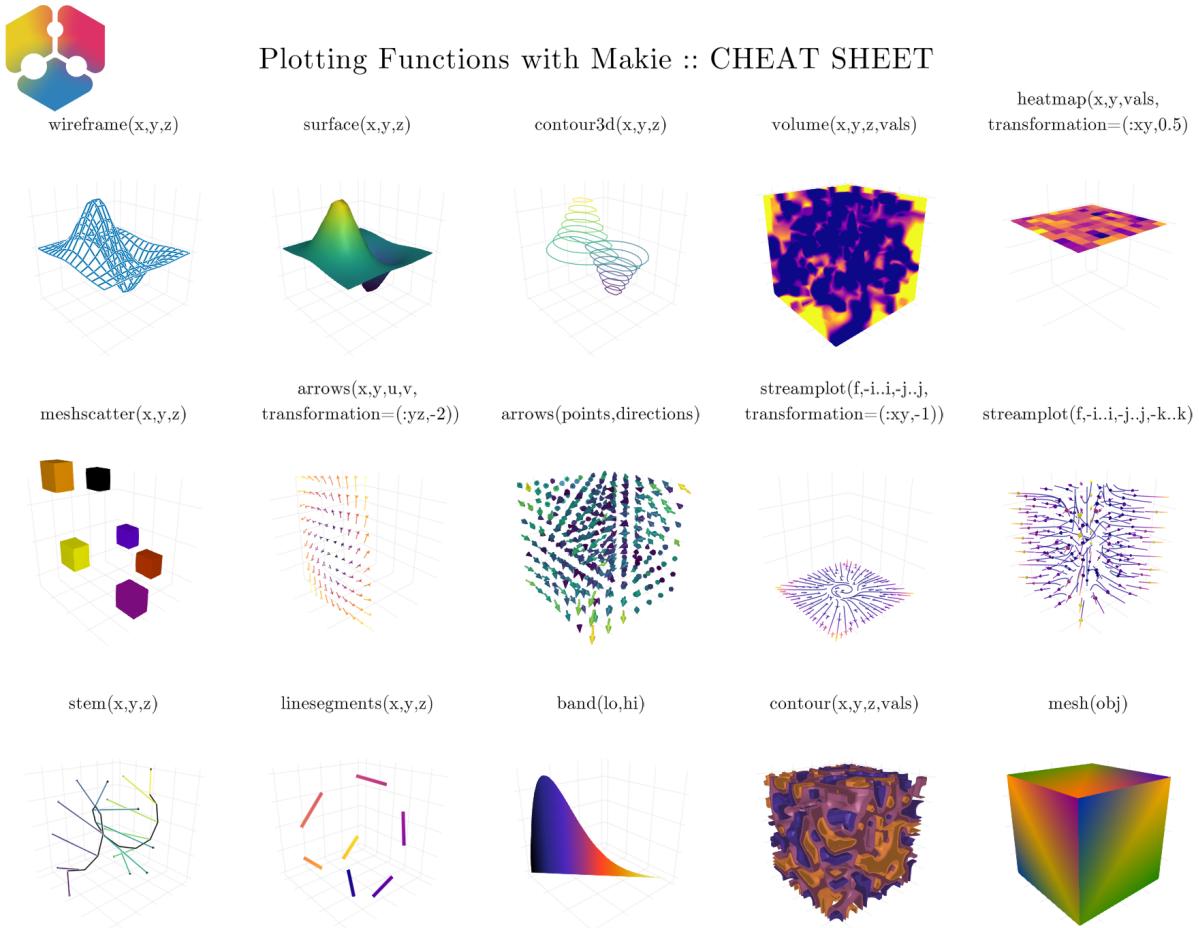
Plotting Functions with Makie :: CHEAT SHEET



Learn more in Julia Data Science. <https://juliadatascience.io> · <http://makie.juliaplots.org> · Makie v0.15.0 · CairoMakie v0.6.3 · Updated: 2021-08-03

Figure 7.6: Plotting functions: Cheat Sheet. Output given by Cairomakie.

For completeness, in Figure 7.7 we show the corresponding functions *cheat sheet* for `GLMakie.jl`, which as a backend supports mostly 3D plots. Those will be explained in detail in Section 7.7.



Learn more in Julia Data Science. <https://juliadatascience.io> · <http://makie.julialplots.org> · Makie v0.15.0 · GLMakie v0.4.4 · Updated: 2021-08-03

Figure 7.7: Plotting functions: Cheat Sheet. Output given by GLMakie.

Now, that we have an idea of all the things we can do, let's go back and continue with the basics. It's time to learn how to change the general appearance of our plots.

7.3 Themes

There are several ways to affect the general appearance of your plots. Either, you could use a predefined theme or your own custom theme. For example, to use the predefined dark theme via `with_theme(your_plot_function, theme_dark())`. Or, build your own with `Theme(kwargs)` or even update the one that is active with `update_theme!(kwargs)`.

You can also do `set_theme!(theme; kwargs...)` to change the current default theme to `theme` and override or add attributes given by `kwargs`. If you do this and want to reset all previous setting just do `set_theme!()` with no arguments. See it in action in the following examples:

```
function demo_themes()
    Random.seed!(123)
    n = 6
    y = cumsum(randn(n, 10), dims = 2)
```

```

labels = ["\$i" for i in 1:n]

fig, _ = series(y; labels = labels, markersize = 10, color=:Set1,
    axis = (; xlabel = "time (s)", ylabel = "Amplitude",
    title = "Measurements"), figure = (;resolution = (600,300)))
xh = LinRange(-3,0.5,20)
yh = LinRange(-3.5,3.5, 20)
hmap = heatmap!(xh, yh, randn(20,20); colormap = :plasma)
limits!(-3.1,13,-6,5.1)
axislegend("legend"; merge = true)
Colorbar(fig[1,2], hmap)
fig
end

```

```

with_theme(demo_themes, theme_dark())
with_theme(demo_themes, theme_black())
with_theme(demo_themes, theme_ggplot2())
with_theme(demo_themes, theme_minimal())
with_theme(demo_themes, theme_light())

```

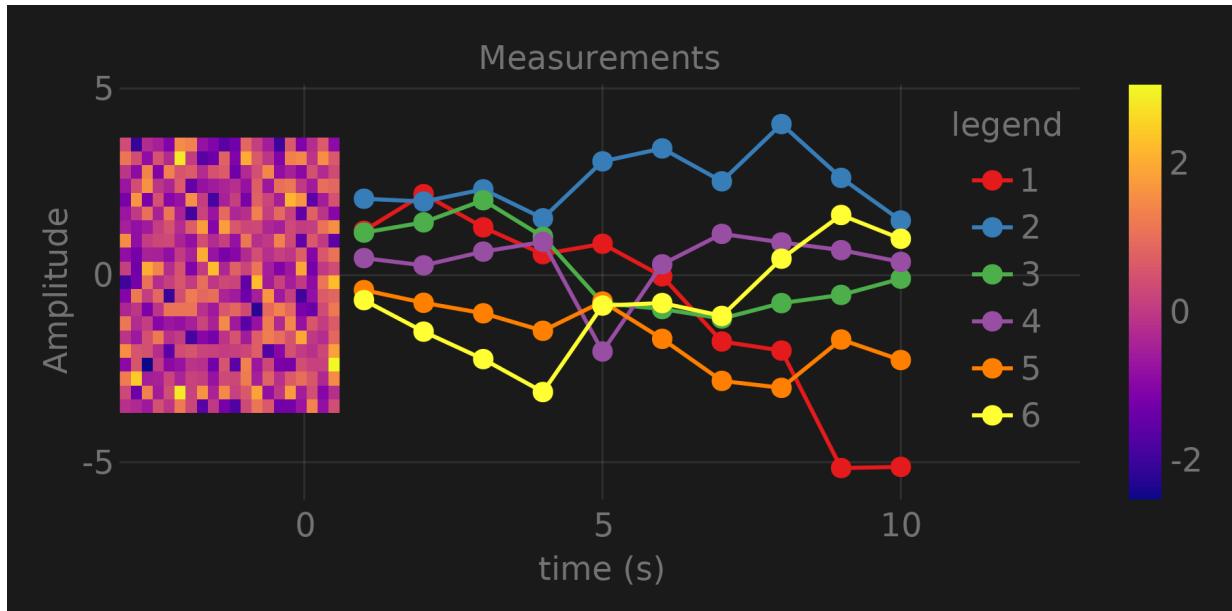


Figure 7.8: Theme dark.

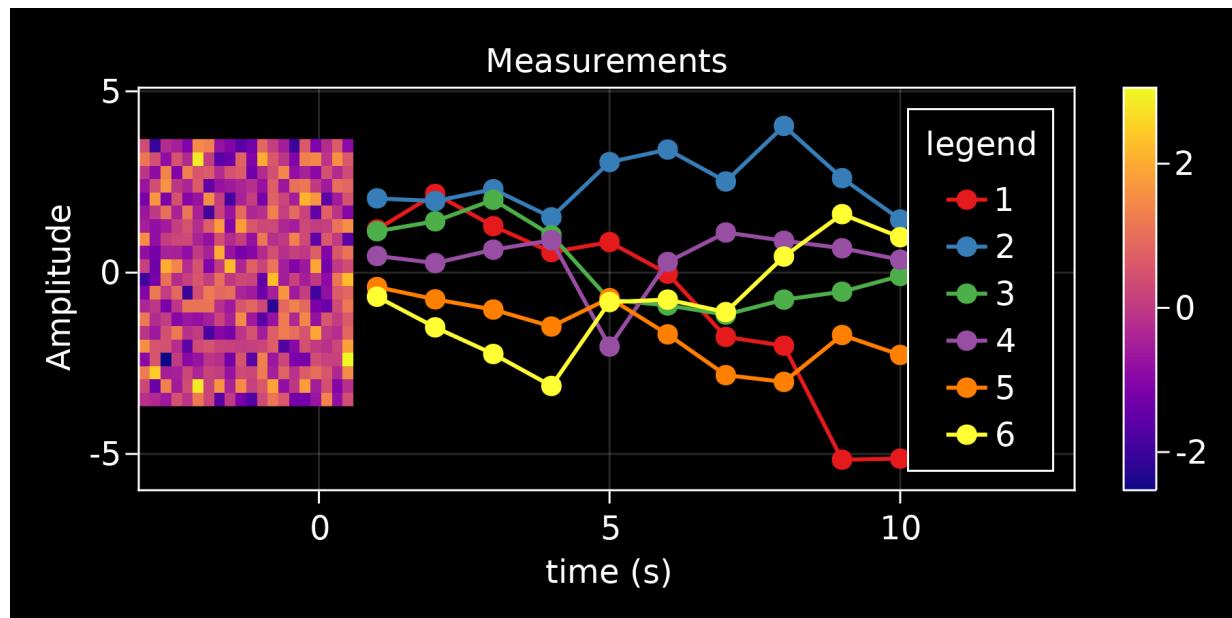


Figure 7.9: Theme black.

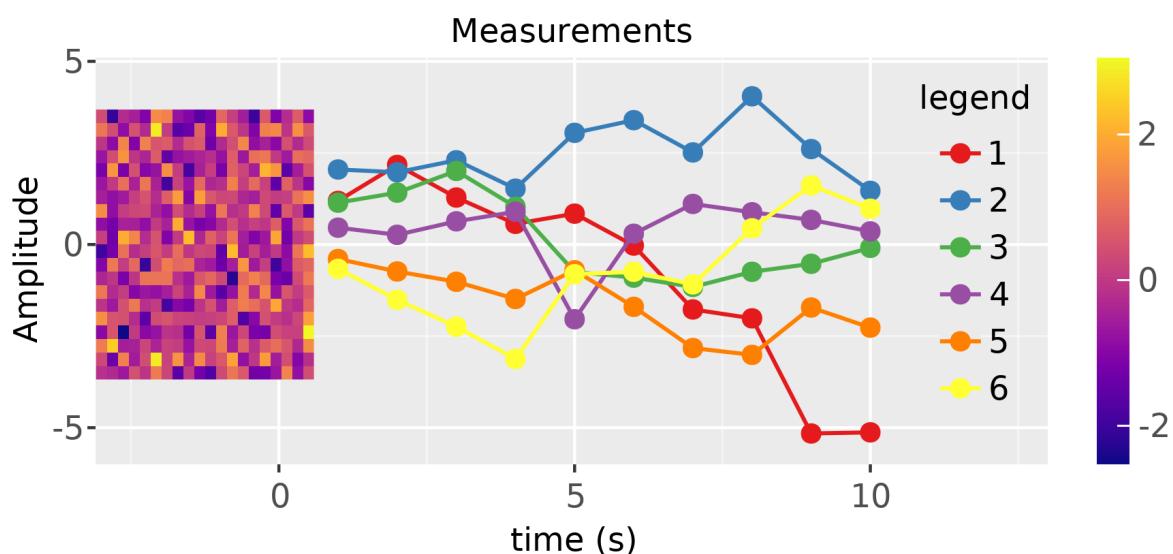


Figure 7.10: Theme ggplot2.

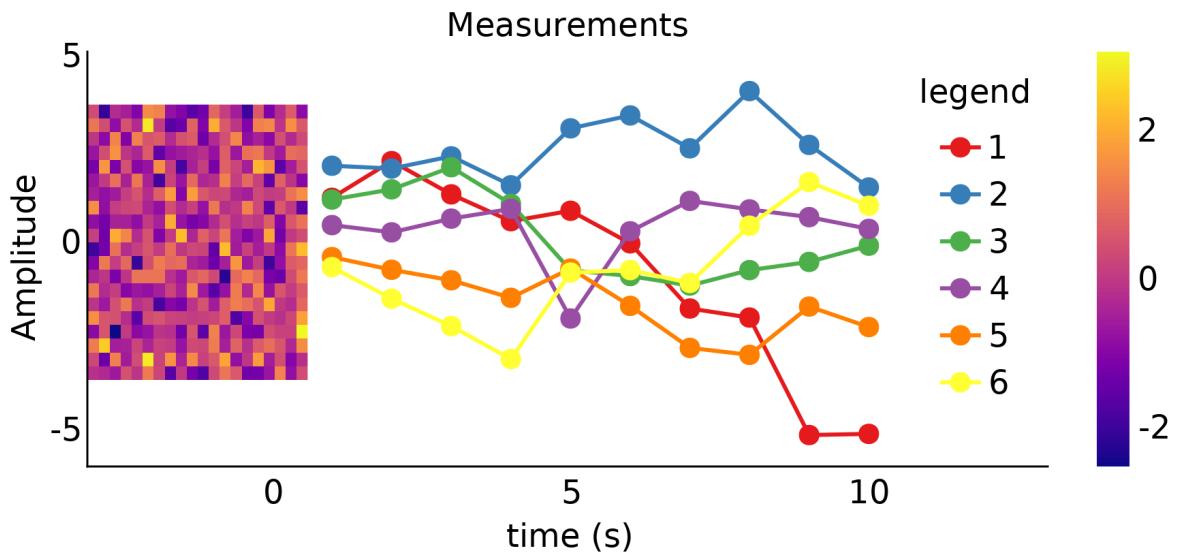


Figure 7.11: Theme minimal.

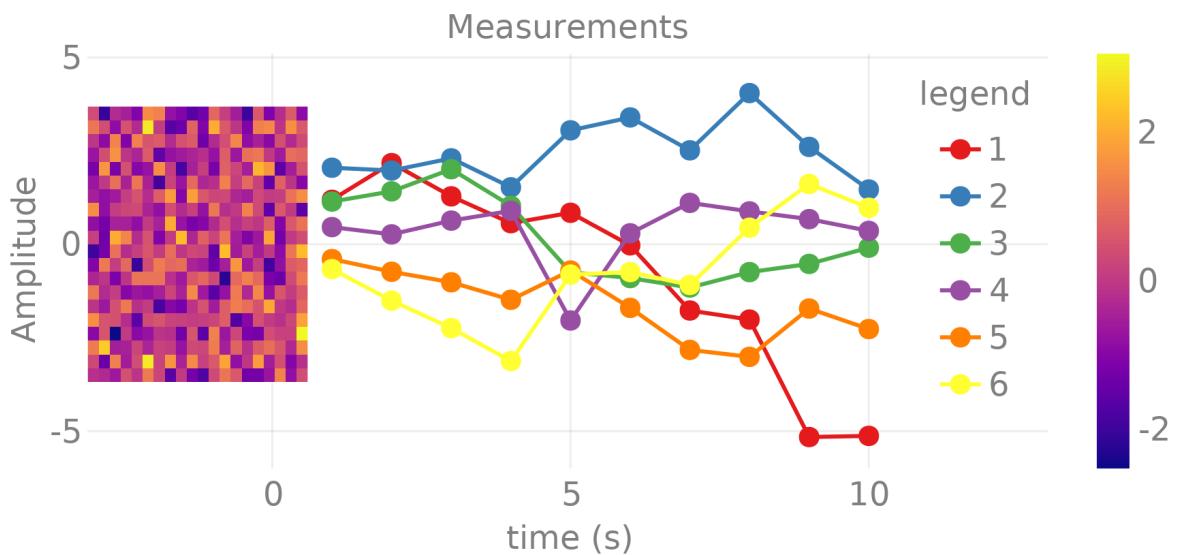


Figure 7.12: Theme light.

Or, defining a custom `Theme` by doing `with_theme(your_plot_function, your_theme())`. For instance, the following theme could be a simple version for a publication quality template:

```
publication_theme() = Theme(
    fontsize = 16, font="CMU Serif",
    Axis = (xlabelsize= 20, xgridstyle=:dash, ygridstyle=:dash,
            xtickalign = 1, ytickalign=1, yticksize=10, xticksize=10,
            xlabelpadding = -5, xlabel = "x", ylabel = "y"),
    Legend = (framecolor = (:black, 0.5), bgcolor = (:white, 0.5)),
    Colorbar = (tickszie=16, tickalign = 1, spinewidth = 0.5)
)
```

```
function plot_with_legend_and_colorbar()
    n = 15
    x = LinRange(6,9,n)
    y = LinRange(2,5,n)
    m = randn(n,n)

    fig, ax, _ = lines(1:10; label = "line")
    scatter!(1:10; label = "line")
    hm = heatmap!(ax, x, y, m; colormap = :Spectral_11)
    axislegend("legend"; position = :lt, merge = true)
    Colorbar(fig[1,2], hm, label = "values")
    ax.title = "my custom theme"
    fig
end
```

```
with_theme(plot_with_legend_and_colorbar, publication_theme())
```

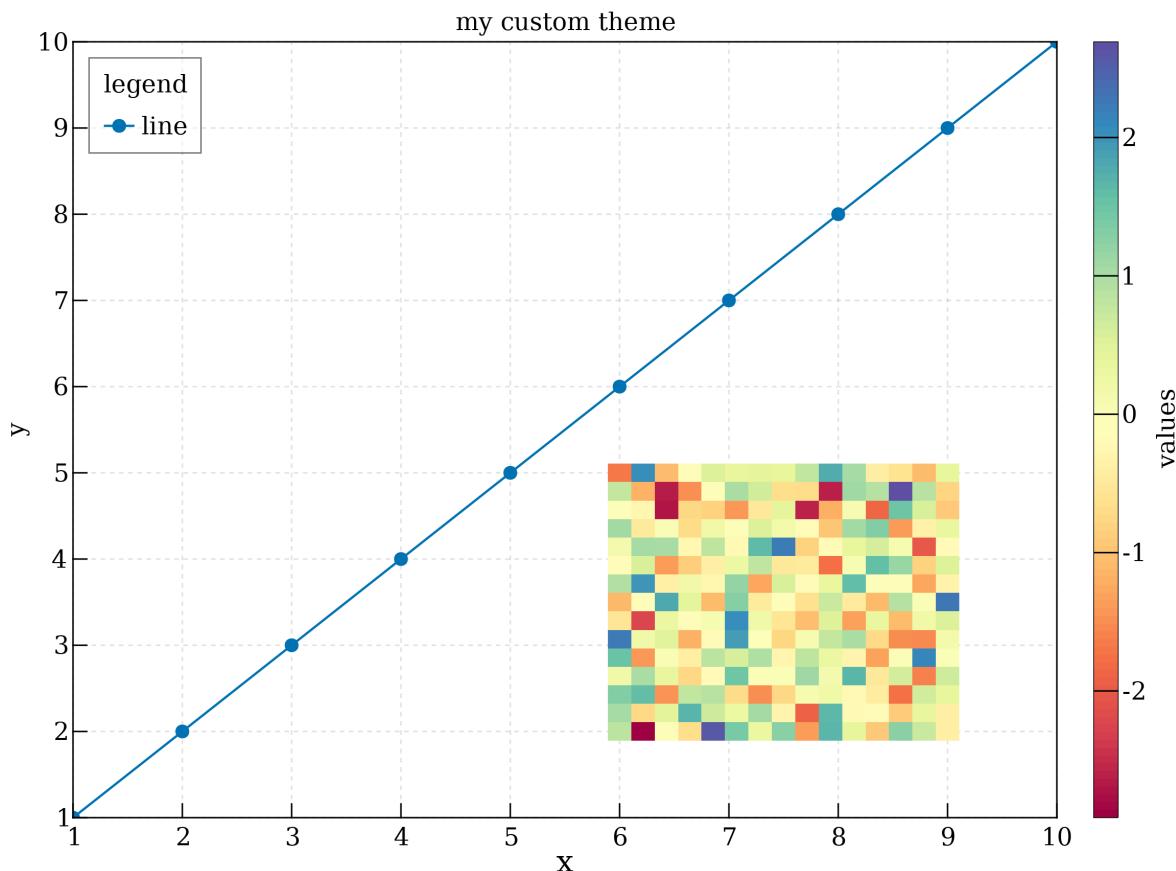


Figure 7.13: Themed plot with Legend and Colorbar.

Now, if something needs to be changed after `set_theme!(your_theme)`, we can do it with `update_theme!(resolution = (500,400), fontsize = 18)`. Another approach will be to pass additional arguments to the `with_theme` function:

```
fig = (resolution = (410,400), figure_padding = 1, backgroundcolor= :grey90)
```

```

ax = (; aspect = DataAspect())
cbar = (; height = Relative(4/5))

with_theme(publication_theme(); fig..., Axis = ax, Colorbar = cbar) do
    plot_with_legend_and_colorbar()
end

```

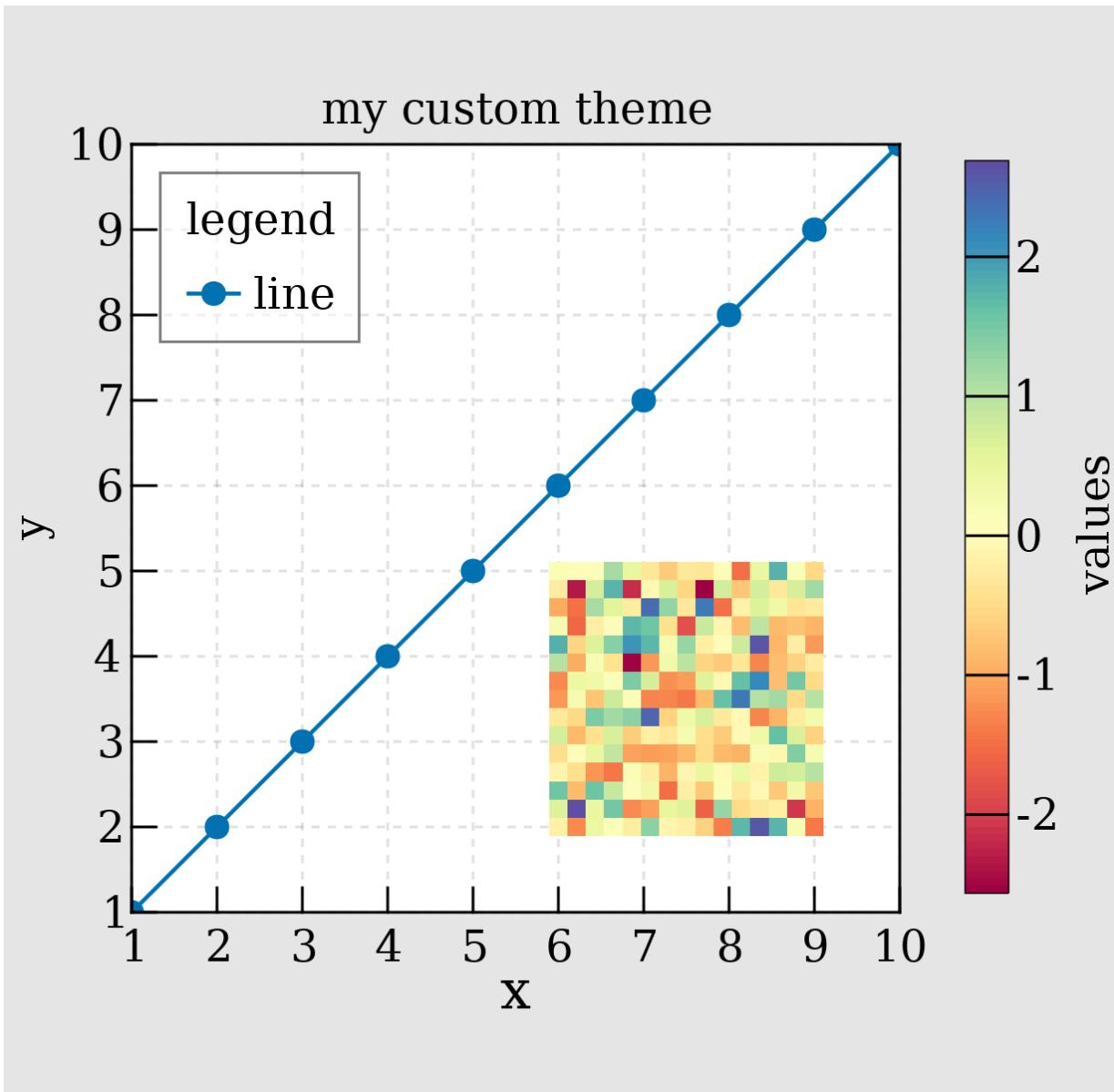


Figure 7.14: Theme with extra args.

Now, let's move on and do a plot with LaTeX strings and a custom theme.

7.4 using LaTeXStrings.jl

LaTeX support in `Makie.jl` is also available. Simple use cases are shown below (Figure 7.15). A basic example includes LaTeX strings for x-y labels and legends:

```
function LaTeX_Strings()
    x = π0:0.05:4
    lines!(x, x -> sin(3x)/(cos(x)+2)/x; label=L"\frac{\sin(3x)}{x(\cos(x)+2)}",
        figure=(;resolution=(600,400), ), axis = (; xlabel = L"x"))
    lines!(x, x-> cos(x)/x; label = L"\cos(x)/x")
    lines!(x, x-> exp(-x); label = L"e^{-x}")
    limits!(-0.5,13,-0.6,1.05)
    axislegend(L"f(x)")
    current_figure()
end
```

```
with_theme(LaTeX_Strings, publication_theme())
```

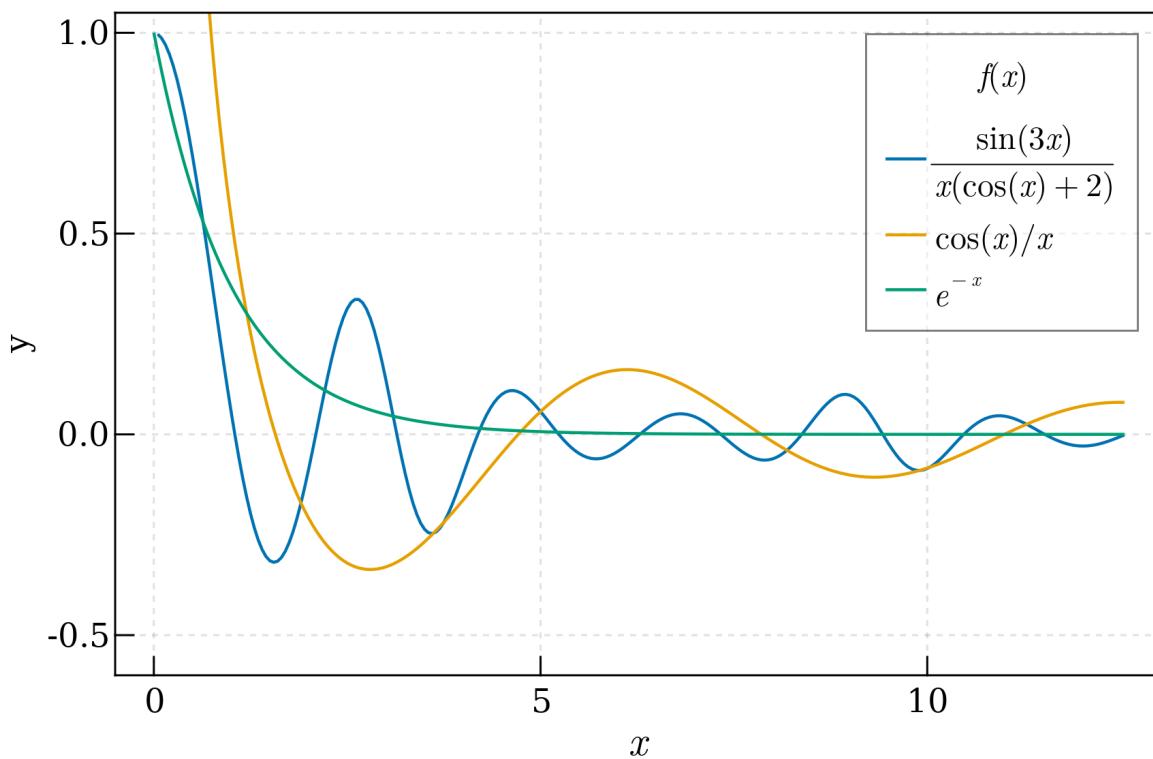


Figure 7.15: Plot with LaTeX strings.

A more involved example will be one with some equation as `text` and increasing legend numbering for curves in a plot.

```
using LaTeXStrings
```

```
function multiple_lines()
    x = collect(0:10)

    fig = Figure(resolution = (600,400), font="CMU Serif")
    ax = Axis(fig[1,1], xlabel = L"x", ylabel = L"f(x,a)")
    for i in 0:10
        lines!(ax, x, i .* x; label = latexstring("$(i) x"))
    end
end
```

```

end
axislegend(L"f(x)"; position = :lt, nbanks = 2, labelsize = 14)
text!(L"f(x,a) = ax", position = (4,80))
fig
end
multiple_lines()

```

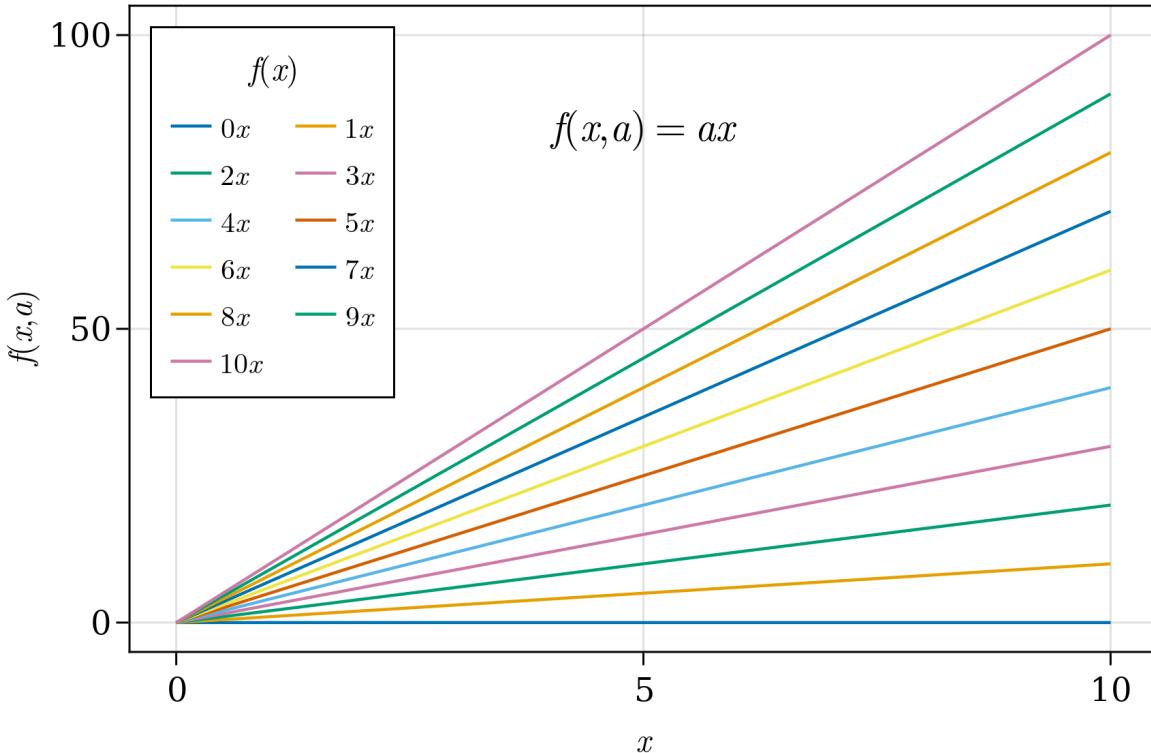


Figure 7.16: Multiple lines.

But, some lines have repeated colors, so that's no good. Adding some markers and line styles usually helps. So, let's do that using `Cycles` for these types:

```

function multiple_scatters_and_lines()
    x = collect(0:10)

    cycle = Cycle([:color, :linestyle, :marker], covary = true)
    set_theme!(Lines = (cycle = cycle,), Scatter = (cycle = cycle,))

    fig = Figure(resolution = (600,400), font="CMU Serif")
    ax = Axis(fig[1,1], xlabel = L"x", ylabel = L"f(x,a)")
    for i in x
        lines!(ax, x, i .* x; label = latexstring("$(i) x"))
        scatter!(ax, x, i .* x; markersize = 13,
                 strokewidth = 0.25, label = latexstring("$(i) x"))
    end
    axislegend(L"f(x)"; merge = true, position = :lt, nbanks=2, labelsize=14)
    text!(L"f(x,a) = ax", position = (4,80))
    set_theme!() # reset to default theme

```

```
fig
end
multiple_scatters_and_lines()
```

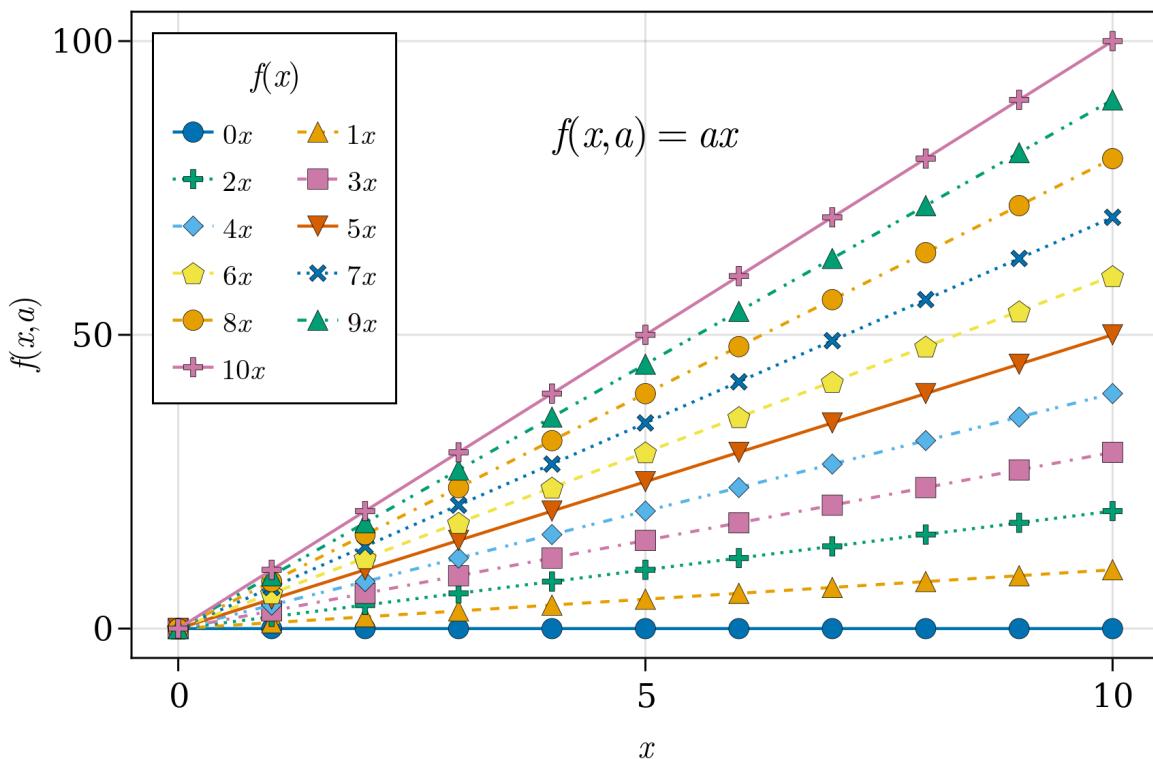


Figure 7.17: Multiple scatters and lines.

And voilà. A publication quality plot is here. What more can we ask for? Well, what about different default colors or palettes. In our next section, we will see how to use again `cycles` and know a little bit more about them, plus some additional keywords in order to achieve this.

7.5 Colors and Colormaps

Choosing an appropriate set of colors or colorbar for your plot is an essential part when presenting results. Using `Colors.jl` is supported in `Makie.jl` so that you can use named colors or pass RGB or RGBA values. Regarding colormaps, all those that work with `Plots.jl` also do here. Additionally, colormaps from `ColorSchemes.jl` and `PerceptualColourMaps.jl` can also be used. It is worth knowing that you can reverse a colormap by doing `Reverse(:colormap_name)` and obtain a transparent color or colormap with `color= (:red,0.5)` and `colormap = (:viridis, 0.5)`.

Different use cases will be shown next. Then we will define a custom theme with new colors and a colorbar palette.

By default `Makie.jl` has a predefined set of colors in order to cycle through them automatically. As shown in the previous figures, where no specific color was set. Overwriting these defaults is done by calling

the keyword `color` in the plotting function and specifying a new color via a `Symbol` or `String`. See this in action in the following example:

```
function set_colors_and_cycle()
    # Epicycloid lines
    x(r,kθ,) = r*(k .+ 1.0).*cosθ.() .- r*cos.((k .+ 1.0) .* θ)
    y(r,kθ,) = r*(k .+ 1.0).*sinθ.() .- r*sin.((k .+ 1.0) .* θ)θ
    = LinRangeπ(0,6.2,1000)

    axis = (; xlabel = L"x(\theta)", ylabel = L"y(\theta)",
              title = "Epicycloid", aspect= DataAspect())
    figure = (;resolution = (500,400), font= "CMU Serif")

    fig, ax, _ = lines(xθ(1,1,), yθ(1,1,); color = "firebrick1", # string
                        label = L"1.0", axis = axis, figure = figure)
    lines!(ax, xθ(4,2,), yθ(4,2,); color = :royalblue1, #symbol
          label = L"2.0")
    for k in 2.5:0.5:5.5
        lines!(ax, x(2k,kθ,), y(2k,kθ,); label = latexstring("$(k)") #cycle
    end
    Legend(fig[1, 2], ax, latexstring("k, r = 2k"), merge = true)
    fig
end
set_colors_and_cycle()
```

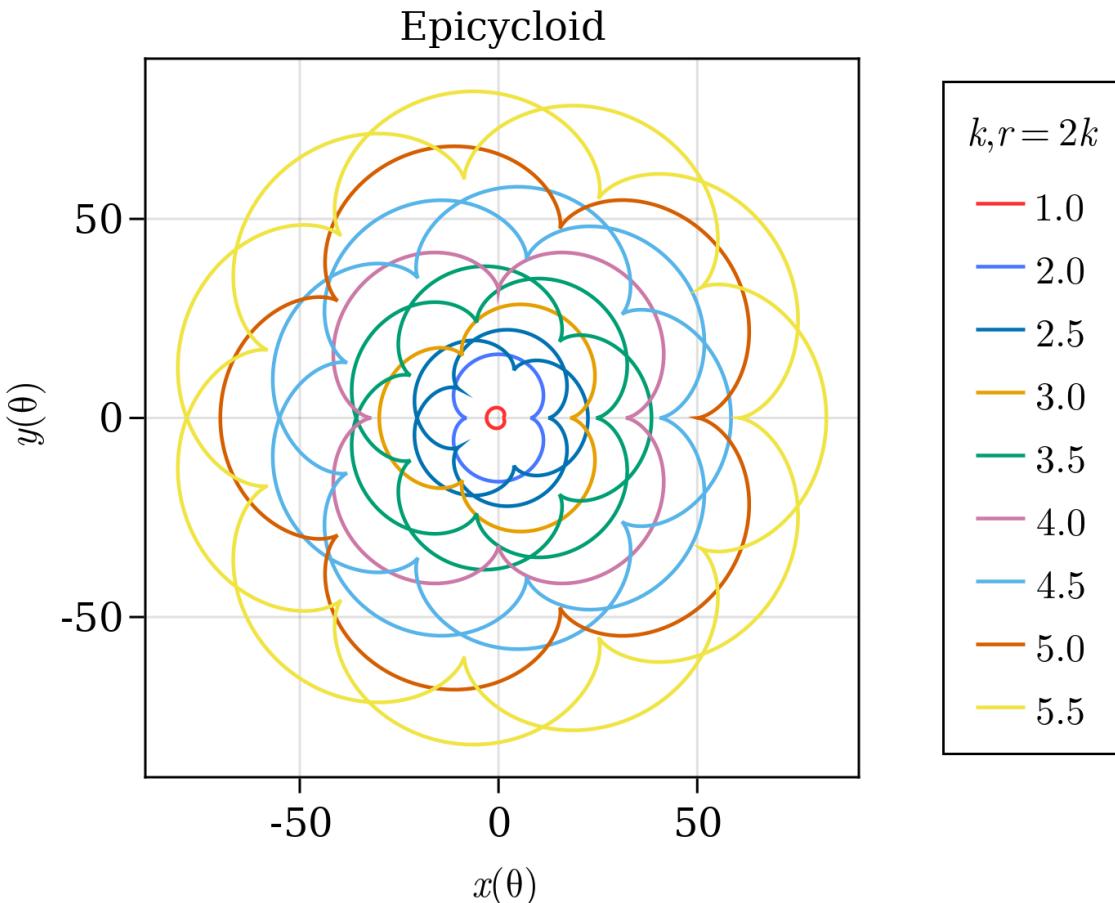


Figure 7.18: Set colors and cycle.

Where, in the first two lines we have used the keyword `color` to specify our color. The rest is using the default cycle set of colors. Later, we will learn how to do a custom cycle.

Regarding colormaps, we are already familiar with the keyword `colormap` for heatmaps and scatters. Here, we show that a colormap can also be specified via a `symbol` or a `String`, similar to `colors`. Or, even a vector of RGB colors. Let's do our first an example by calling `colormaps` as a `Symbol`, `String` and `cgrad` for categorical values. See `?cgrad` for more information.

```
figure = (;resolution = (400,300), font= "CMU Serif")
axis = (; xlabel = L"x", ylabel = L"y", aspect= DataAspect())

fig, ax, pltobj = heatmap(rand(20,20); colorrange = (0,1),
    colormap = Reverse(:viridis), axis = axis, figure = figure)
Colorbar(fig[1,2], pltobj, label = "Reverse colormap Sequential")
fig
```

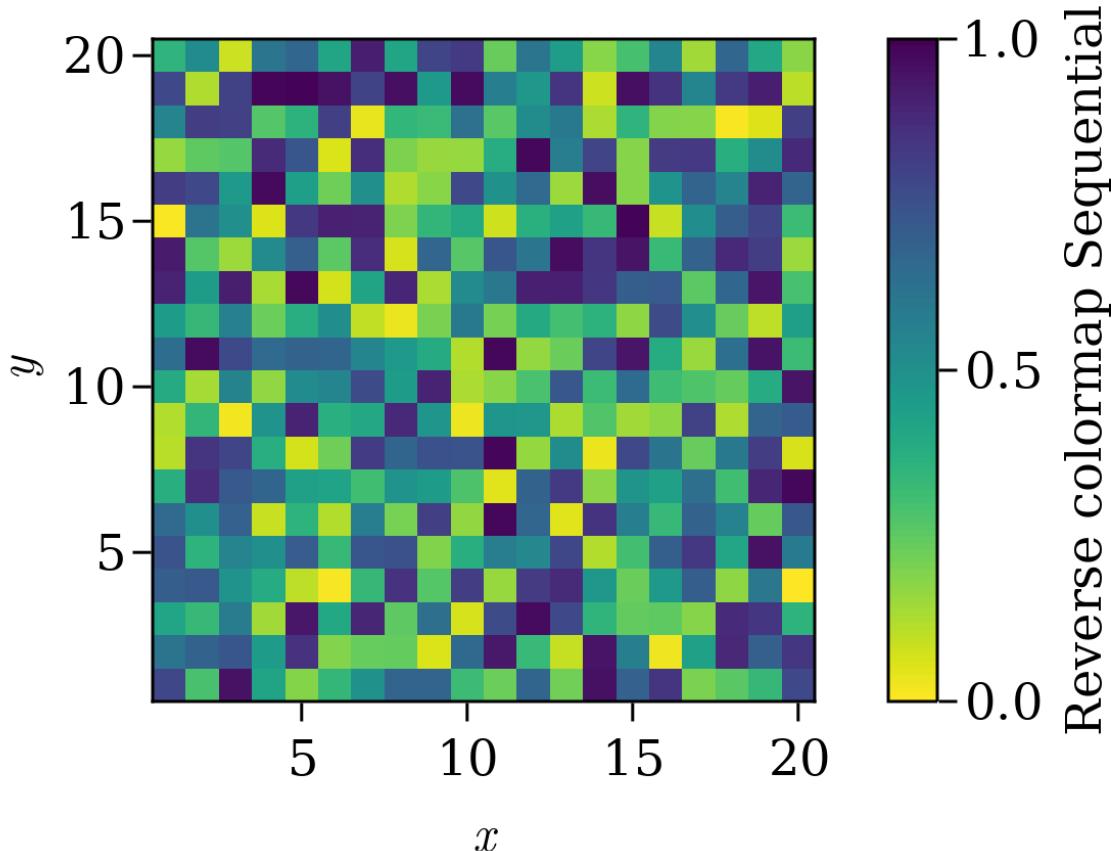


Figure 7.19: Reverse colormap sequential and colorrange.

When setting a `colorrange` usually the values outside this range are colored with the first and last color from the colormap. However, sometimes it is better to specify the color you want at both ends. We do that with `highclip` and `lowclip`:

```
using ColorSchemes

figure = (;resolution = (400,300), font= "CMU Serif")
axis = (; xlabel = L"x", ylabel = L"y", aspect= DataAspect())

fig, ax, pltobj = heatmap(randn(20,20); colorrange = (-2,2),
    colormap = "diverging_rainbow_bgymr_45_85_c67_n256",
    highclip = :black, lowclip = :white, axis = axis, figure = figure)
Colorbar(fig[1,2], pltobj, label = "Diverging colormap")
fig
```

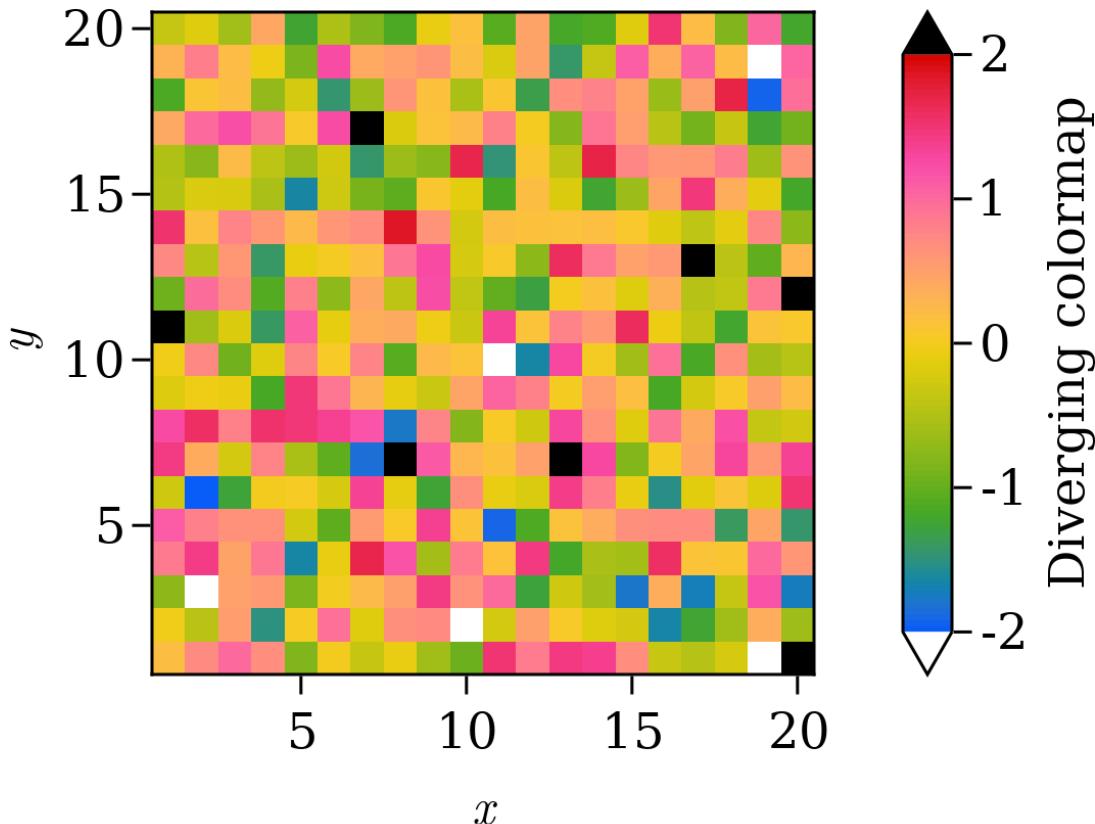


Figure 7.20: Diverging Colormap with low and high clip.

But we mentioned that also RGB vectors are valid options. For our next example you could pass the custom colormap `perse` or use `cgrad` to force a categorical colorbar.

```
using Colors, ColorSchemes

figure = (;resolution = (400,300), font= "CMU Serif")
axis = (; xlabel = L"x", ylabel = L"y", aspect= DataAspect())
cmap = ColorScheme(range(colorant"red", colorant"green", length=3))
mygrays = ColorScheme([RGB{Float64}(i, i, i) for i in [0.0,0.5,1.0]])

fig, ax, pltobj = heatmap(rand(-1:1,20,20);
    colormap = cgrad(mygrays, 3, categorical = true, rev = true), # cgrad and Symbol, mygrays,
    axis = axis, figure = figure)
cbar = Colorbar(fig[1,2], pltobj, label = "Categories")
cbar.ticks = ([-0.66,0,0.66], ["-1","0","1"])
fig
```

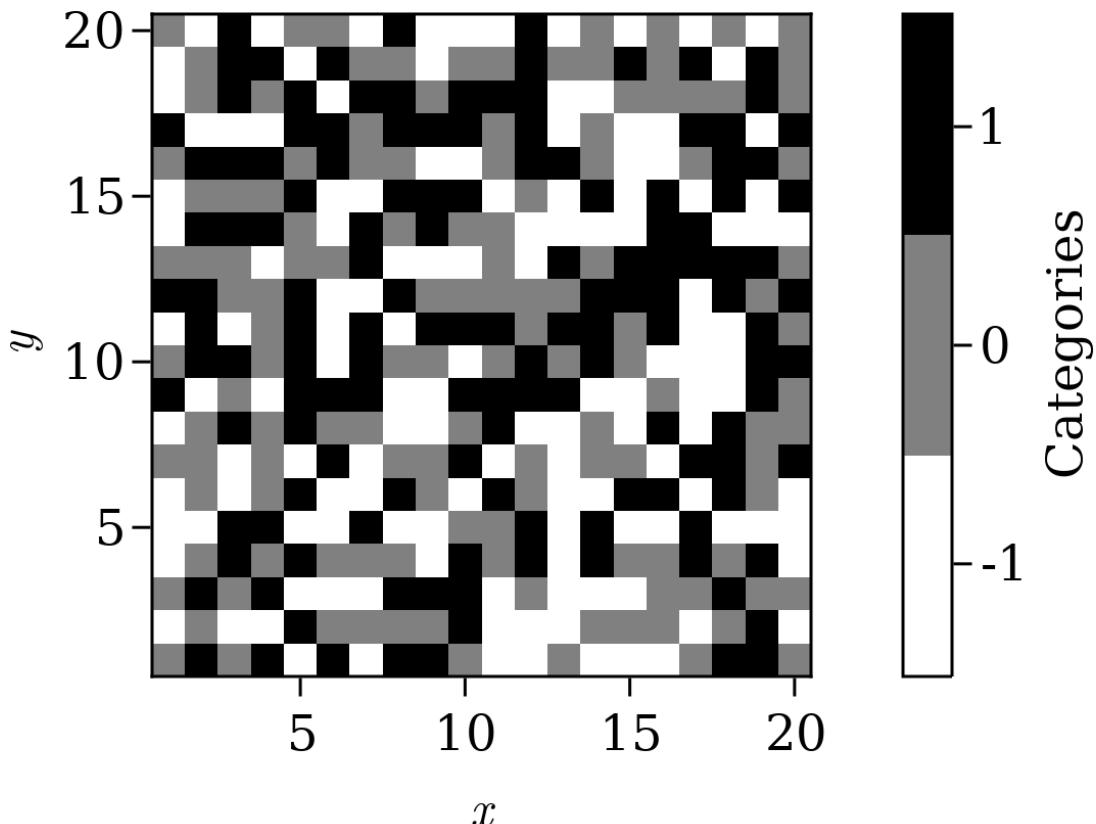


Figure 7.21: Categorical Colormap.

Lastly, the ticks in the colorbar for the categorial case are not centered by default in each color. This is fixed by passing custom ticks, as in `cbar.ticks = (positions, ticks)`. The last situation is when passing a tuple of two colors to `colormap` as symbols, strings or a mix. You will get an interpolated colormap between these two colors.

Also, hex coded colors are also accepted. So, on top of our heatmap let's put one semi-transparent point using this.

```
figure = (;resolution = (400,300), font= "CMU Serif")
axis = (; xlabel = L"x", ylabel = L"y", aspect= DataAspect())

fig, ax, pltobj = heatmap(rand(20,20); colorange = (0,1),
    colormap = (:red, "black"), axis = axis, figure = figure)
scatter!(ax,[11],[11],color="#C0C0C0", 0.5,markersize=150)
Colorbar(fig[1,2], pltobj, label = "2 colors")
fig
```

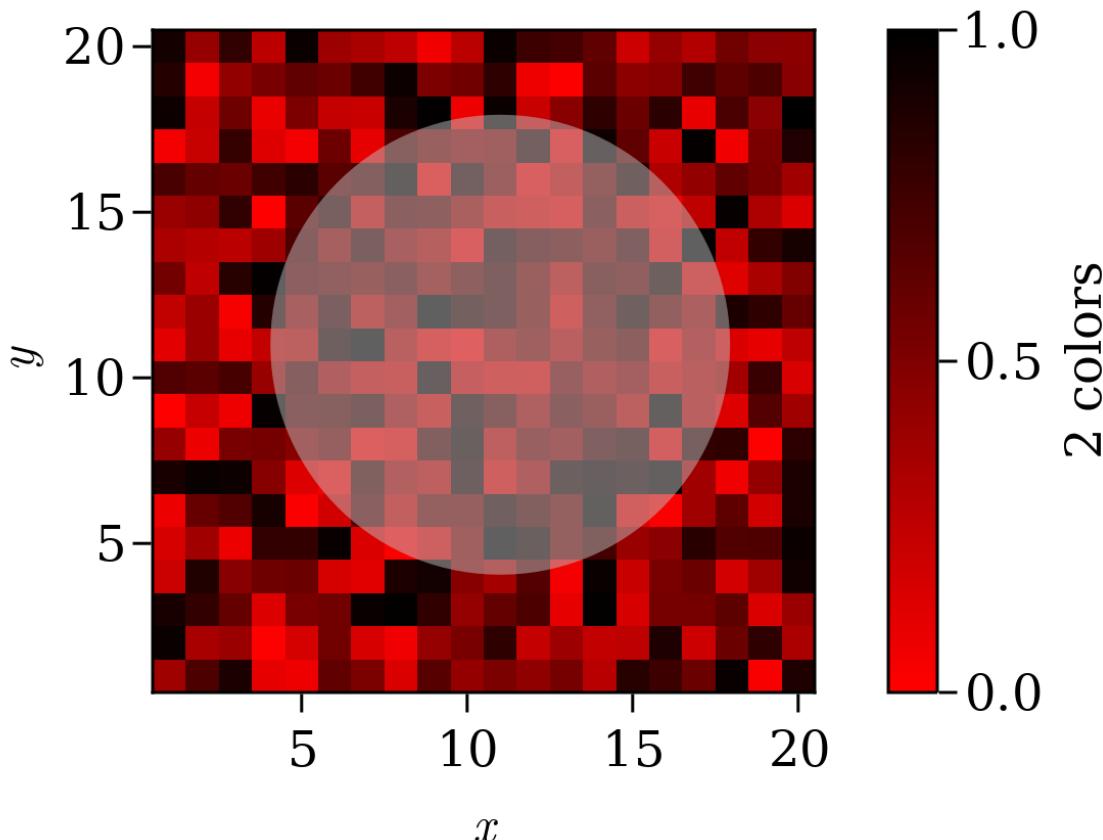


Figure 7.22: Colormap from two colors.

7.5.1 Custom cycle

Here, we could define a global `Theme` with a new cycle for colors, however that is **not the recommend way** to do it. It's better to define a new theme and use as shown before. Lets define a new one with a `cycle` for `:color`, `:linestyle`, `:marker` and a new `colormap` default. Lets add this new attributes to our previous `publication_theme`.

```
function new_cycle_theme()
    # https://nanx.me/ggsci/reference/pal_locuszoom.html
    my_colors = ["#D43F3AFF", "#EEA236FF", "#5CB85CFF", "#46B8DAFF",
                 "#357EBDFF", "#9632B8FF", "#B8B8B8FF"]
    cycle = Cycle([:color, :linestyle, :marker], covary = true) # altogether
    my_markers = [:circle, :rect, :utriangle, :dtriangle, :diamond,
                  :pentagon, :cross, :xcross]
    my_linestyle = [nothing, :dash, :dot, :dashdot, :dashdotdot]

    Theme(
        fontsize = 16, font="CMU Serif",
        colormap = :linear_bmy_10_95_c78_n256,
        palette=(color=my_colors,marker=my_markers,linestyle=my_linestyle),
        Lines = (cycle = cycle,), Scatter = (cycle = cycle,),
```

```

Axis = (xlabelsize= 20,xgridstyle=:dash,ygridstyle=:dash,
        xtickalign = 1, ytickalign=1,yticksize=10, xtickszie=10,
        xlabelpadding = -5, xlabel = "x", ylabel = "y"),
Legend = (framecolor = (:black, 0.5), bgcolor = (:white, 0.5)),
Colorbar = (tickszie=16, tickalign = 1, spinewidth = 0.5)
)
end

```

And apply it to a plotting function like the following:

```

function scatters_and_lines()
    x = collect(0:10)
    xh = LinRange(4,6,25)
    yh = LinRange(70,95,25)
    h = randn(25,25)

    fig = Figure(resolution = (650,400), font="CMU Serif")
    ax = Axis(fig[1,1], xlabel = L"x", ylabel = L"f(x,a)")
    for i in x
        lines!(ax, x, i .* x; label = latexstring("$(i) x"))
        scatter!(ax, x, i .* x; markersize = 13, strokewidth = 0.25,
                 label = latexstring("$(i) x"))
    end
    hm = heatmap!(xh, yh, h)
    axislegend(L"f(x)"; merge = true, position = :lt, nbanks=2, labelsize=14)
    Colorbar(fig[1,2], hm, label = "new default colormap")
    limits!(ax,-0.5,10.5,-5,105)
    colgap!(fig.layout, 5)
    fig
end

```

```
with_theme(scatters_and_lines, new_cycle_theme())
```

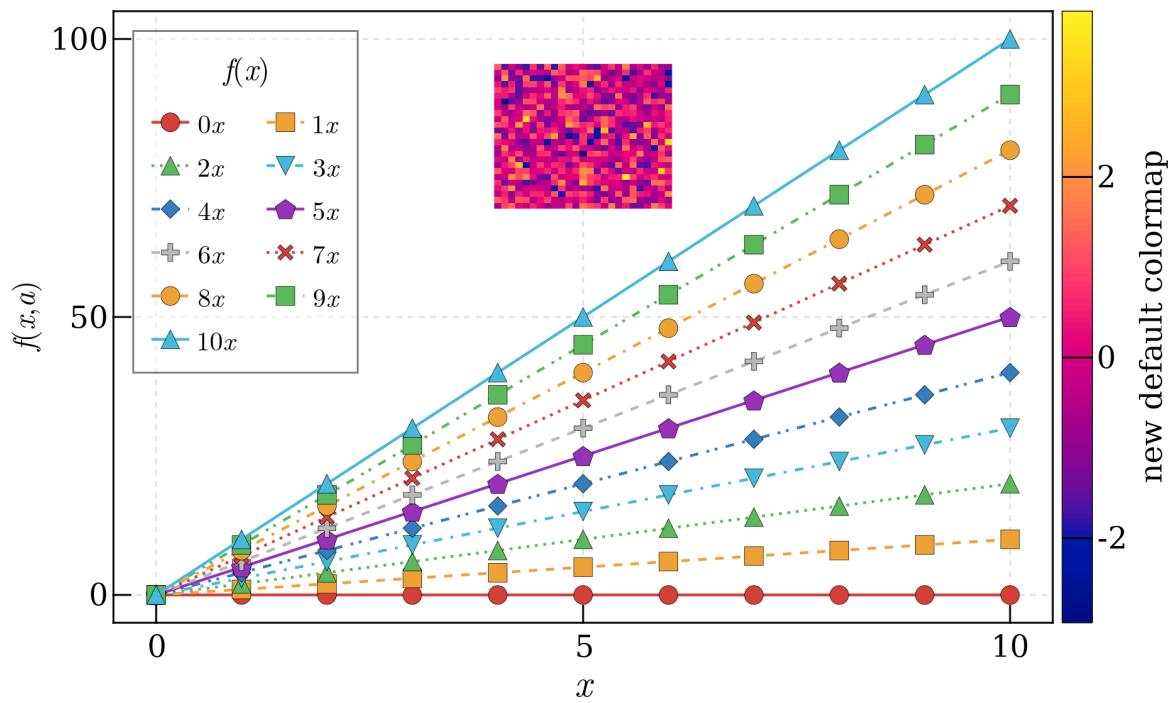


Figure 7.23: Custom theme with new cycle and colormap.

At this point you should be able to have **complete control** over your colors, line styles, markers and colormaps for your plots. Next, we will dive into how to manage and control **layouts**.

7.6 Layouts

A complete *canvas/layout* is defined by `Figure`, which can be filled with content after creation. We will start with a simple arrangement of one `Axis`, one `Legend` and one `Colorbar`. For this task we can think of the canvas as an arrangement of `rows` and `columns` in indexing a `Figure` much like a regular `Array/Matrix`. The `Axis` content will be in `row 1, column 1`, e.g. `fig[1, 1]`, the `Colorbar` in `row 1, column 2`, namely `fig[1, 2]`. And the `Legend` in `row 2` and across `column 1 and 2`, namely `fig[2, 1:2]`.

```
function first_layout()
    Random.seed!(123)
    x, y, z = randn(6), randn(6), randn(6)

    fig = Figure(resolution = (600,400), backgroundcolor = :grey90)
    ax = Axis(fig[1,1], backgroundcolor = :white)
    pltobj = scatter!(ax, x, y; color = z, label = "scatters")
    lines!(ax, x, 1.1y; label = "line")
    Legend(fig[2,1:2], ax, "labels", orientation = :horizontal)
    Colorbar(fig[1, 2], pltobj, label = "colorbar")
    fig
end
first_layout()
```

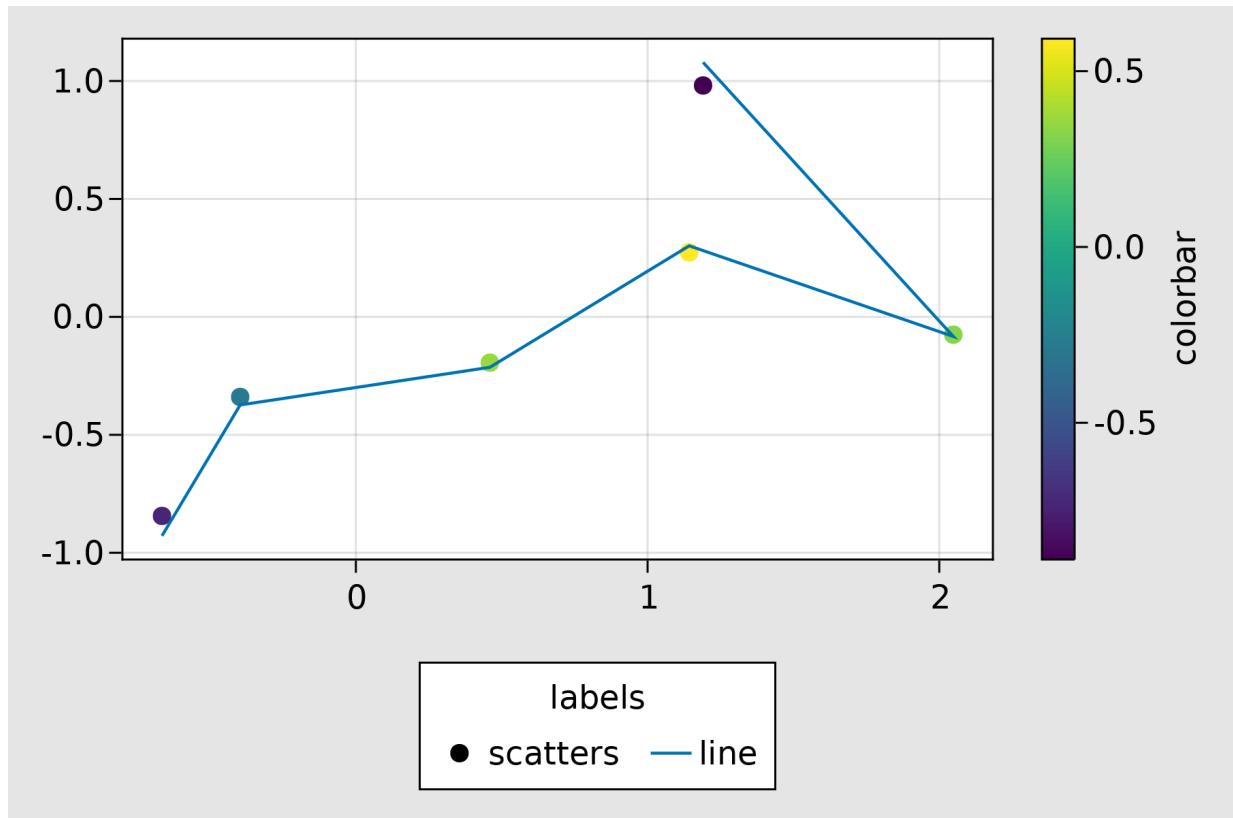


Figure 7.24: First layout.

This does not look good. Next, we fix the spacing problems using the following keywords and methods:

- `figure_padding = (left, right, bottom, top)`
- `padding = (left, right, bottom, top)`

Taking into account the actual size for a `Legend` or `Colorbar` is done by

- `tellheight = true` or `false`
- `tellwidth = true` or `false`

Setting these to `true` will take into account the actual size (height or width) for a `Legend` or `Colorbar`.

Consequently, things will be resized accordingly.

The space between columns and rows is specified as

- `colgap!(fig.layout, col, separation)`
- `rowgap!(fig.layout, row, separation)`

`Column gap` (`colgap!`), if `col` is given then the gap will be applied to that specific column. `Row gap` (`rowgap!`), if `row` is given then the gap will be applied to that specific row.

Also, we will see how to put content into the **protrusions**, i.e. the space reserved for `title`: `x` and `y`; either `ticks` or `label`. We do this by plotting into `fig[i, j, protrusion]` where `protrusion` can be `Left()`, `Right()`, `Bottom()` and `Top()`, or for each corner `TopLeft()`, `TopRight()`, `BottomRight()`, `BottomLeft()`. See below how these options are being used:

```

function first_layout_fixed()
    Random.seed!(123)
    x, y, z = randn(6), randn(6), randn(6)

    fig = Figure(figsize_padding = (0,3,5,2), resolution = (600,400),
                backgroundcolor = :grey90, font="CMU Serif")
    ax = Axis(fig[1,1], xlabel = L"x", ylabel = L"y",
              title = "Layout example", backgroundcolor = :white)

    pltobj = scatter!(ax, x, y; color = z, label = "scatters")
    lines!(ax, x, 1.1y, label = "line")
    Legend(fig[2, 1:2], ax, "Labels", orientation = :horizontal,
           tellheight = true, titleposition = :left)
    Colorbar(fig[1, 2], pltobj, label = "colorbar")

    # additional aesthetics
    Box(fig[1, 1, Right()], color = (:slateblue1,0.35))
    Label(fig[1, 1, Right()], "protrusion", textsize = 18,
          rotation = pi/2, padding = (3,3,3,3))
    Label(fig[1, 1, TopLeft()], "(a)", textsize = 18, padding = (0,3,8,0))
    colgap!(fig.layout, 5)
    rowgap!(fig.layout, 5)
    fig
end
first_layout_fixed()

```

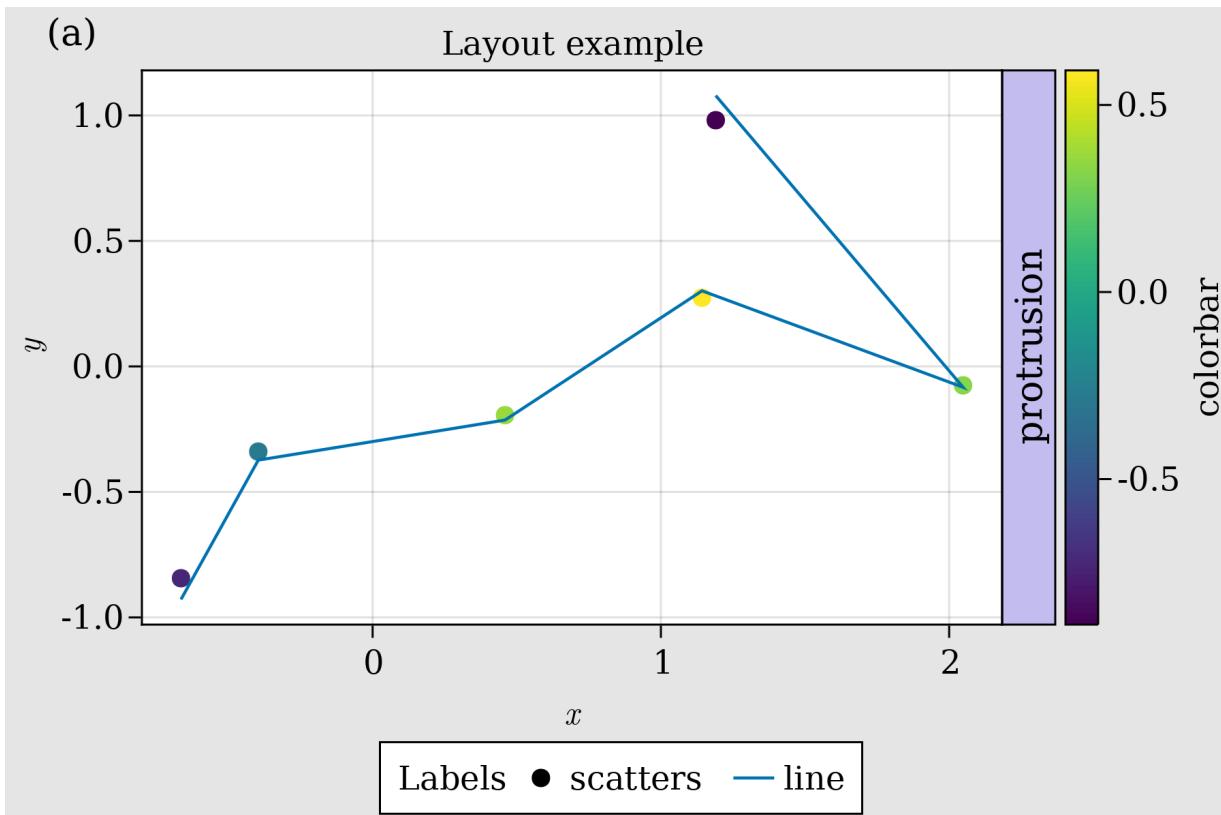


Figure 7.25: First layout fixed.

Here, having the label `(a)` in the `TopLeft()` is probably not necessary, this will only make sense for more than two plots. For our next example let's keep using the previous tools and some more to create a richer and complex figure.

You can hide decorations and axis' spines with:

- `hidedecorations!(ax; kwargs...)`
- `hidexdecorations!(ax; kwargs...)`
- `hideydecorations!(ax; kwargs...)`
- `hidespines!(ax; kwargs...)`

Remember, we can always ask for help to see what kind of arguments we can use, e.g.,

```
help(hidespines!)
```

```
hidespines!(la::Axis, spines::Symbol... = (:l, :r, :b, :t)...)
```

Hide all specified axis spines. Hides all spines by default, otherwise choose with the symbols `:l`, `:r`, `:b` and `:t`.

`hidespines!` has the following function signatures:

```
(Vector, Vector)
(Vector, Vector, Vector)
(Matrix)
```

Available attributes for `Combined{Makie.MakieLayout.hidespines!, T}` where `T` are:

Alternatively, for decorations

```
help(hidedecorations!)
```

```
hidedecorations!(la::Axis)
```

Hide decorations of both x and y-axis: label, ticklabels, ticks and grid.

`hidedecorations!` has the following function signatures:

```
(Vector, Vector)
(Vector, Vector, Vector)
(Matrix)
```

Available attributes for `Combined{Makie.MakieLayout.hidedecorations!, T}` where `T` are:

For elements that **you don't want to hide**, just pass them with `false`, i.e. `hideydecorations!(ax; ticks=false, grid=false)`.

Synchronizing your Axis is done via:

- `linkyaxes!` `linkyaxes!` `linkxaxes!`

This could be useful when shared axis are desired. Another way of getting shared axis will be by setting `limits!`.

Setting `limits` at once or independently for each axis is done by calling

- `limits!(ax; l,r,b,t)` You can also do `ylims!(low, high)` or `xlims!(low, high)`, and even open ones by doing `ylims!(low=0)` or `xlims!(high=1)`.

Now, the example:

```
function complex_layout_double_axis()
    Random.seed!(123)
    x = LinRange(0,1,10)
    y = LinRange(0,1,10)
    z = rand(10,10)

    fig = Figure(resolution = (600,400), font="CMU Serif",
                 backgroundcolor = :grey90)
    ax1 = Axis(fig, xlabel = L"x", ylabel = L"y")
    ax2 = Axis(fig, xlabel = L"x")
    heatmap!(ax1, x, y, z; colormode = (0,1))
    series!(ax2, abs.(z[1:4,:]); labels=["lab $i" for i in 1:4],
             color = :Set1_4)
    hm = scatter!(10x, y; color = z[1,:],label = "dots",colormode= (0,1))

    hideydecorations!(ax2, ticks = false, grid = false)
    linkyaxes!(ax1, ax2)
    #layout
    fig[1,1] = ax1
    fig[1,2] = ax2
    Label(fig[1, 1,TopLeft()], "(a)",textsize = 18,padding = (0,6,8,0))
    Label(fig[1, 2,TopLeft()], "(b)",textsize = 18,padding = (0,6,8,0))
    Colorbar(fig[2,1:2], hm, label = "colorbar", vertical = false,
              flipaxis = false)
    Legend(fig[1,3], ax2, "Legend")
    colgap!(fig.layout, 5)
    rowgap!(fig.layout, 5)
    fig

end
complex_layout_double_axis()
```

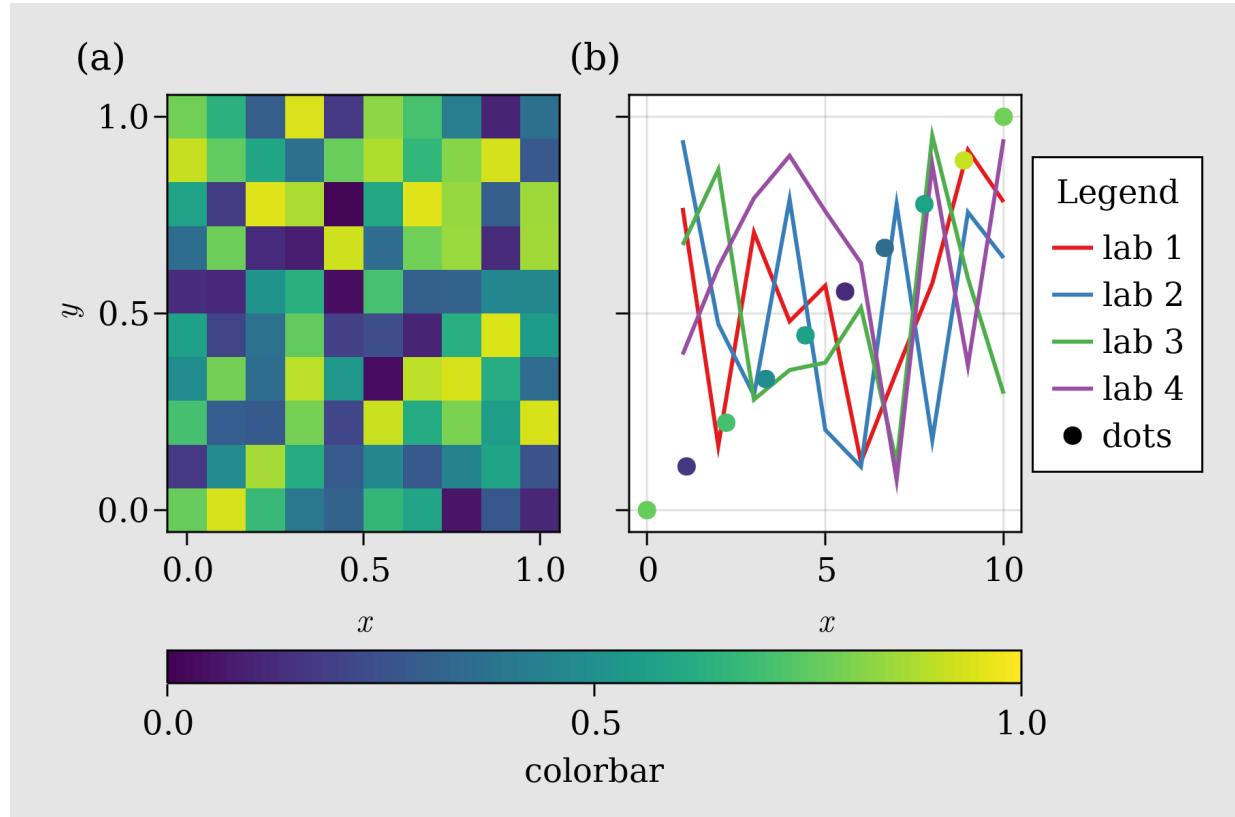


Figure 7.26: Complex layout double axis.

So, now our `Colorbar` needs to be horizontal and the bar ticks need to be in the lower part. This is done by setting `vertical = false` and `flipaxis = false`. Additionally, note that we can call many `Axis` into `fig`, or even `Colorbar`'s and `Legend`'s, and then afterwards build the layout.

Another common layout is a grid of squares for heatmaps:

```
function squares_layout()
    Random.seed!(123)
    letters = reshape(collect('a':'d'), (2,2))
    fig = Figure(resolution =(500,400), fontsize = 14, font="CMU Serif",
                 backgroundcolor = :grey90)
    axs = [Axis(fig[i, j], aspect = DataAspect()) for i in 1:2, j in 1:2]
    hms = [heatmap!(axs[i,j], randn(10,10), colorrange = (-2,2))
           for i in 1:2, j in 1:2]

    Colorbar(fig[1:2, 3], hms[1], label = "colorbar")
    [Label(fig[i, j], TopLeft(), "($letters[i,j])", textsize = 16,
          padding = (-24,0,-16,0)) for i in 1:2, j in 1:2]

    colgap!(fig.layout, 5)
    rowgap!(fig.layout, 5)
    fig
end
squares_layout()
```

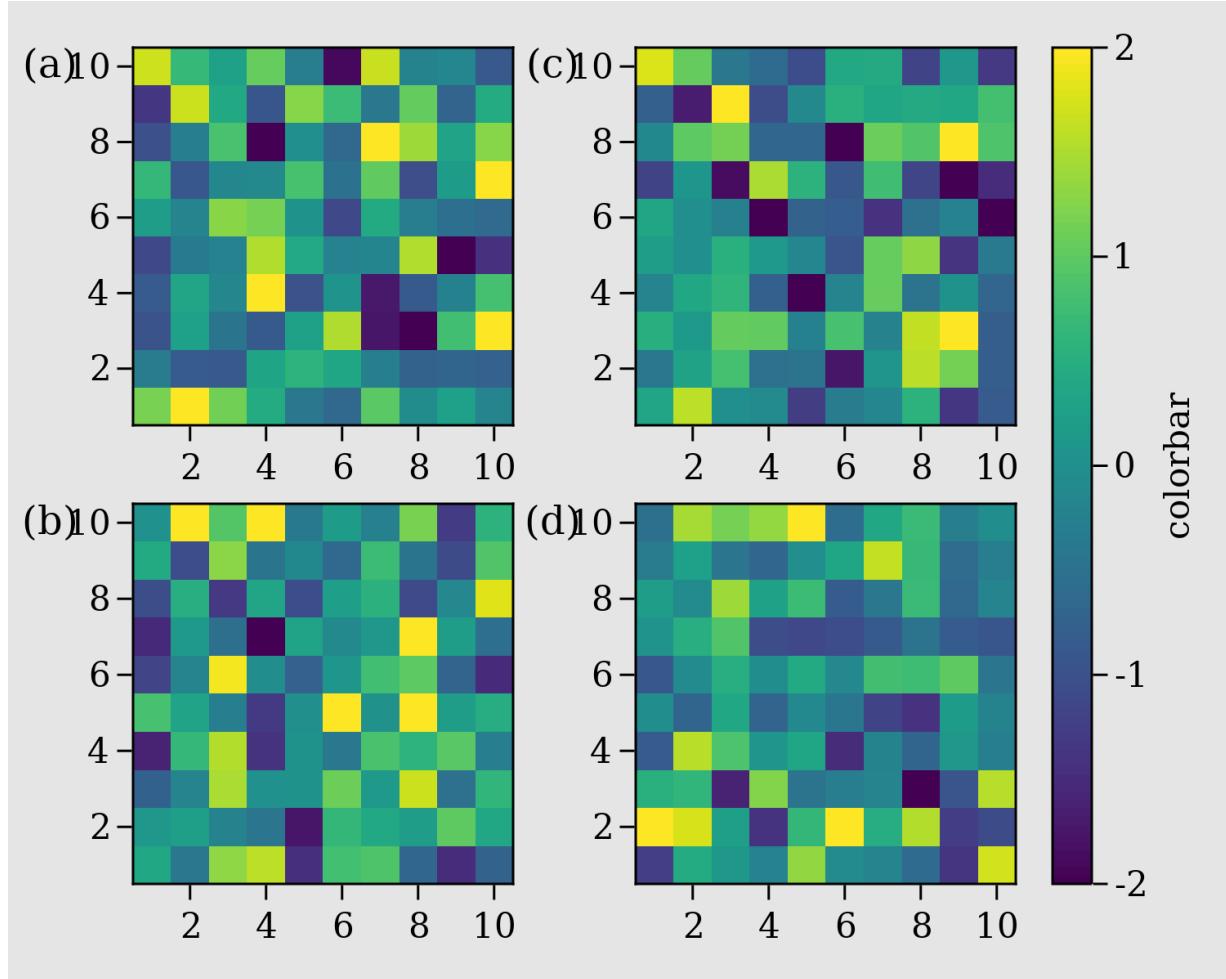


Figure 7.27: Squares layout.

where all labels are in the **protrusions** and each `Axis` has an `AspectData()` ratio. The `Colorbar` is located in the third column and expands from row 1 up to row 2.

The next case uses the so called `Mixed()` **alignmode**, which is especially useful when dealing with large empty spaces between `Axis` due to long ticks.

```
function mixed_mode_layout()
    Random.seed!(123)
    longlabels = ["$(today() - Day(1))", "$(today())", "$(today() + Day(1))"]

    fig = Figure(resolution = (600,400), fontsize = 12,
                 backgroundcolor = :grey90, font="CMU Serif")
    ax1 = Axis(fig[1,1])
    ax2 = Axis(fig[1,2],xticklabelrotation = pi/2,alignmode = Mixed(bottom=0),
               xticks = ([1,5,10], longlabels))
    ax3 = Axis(fig[2,1:2])
    ax4 = Axis(fig[3,1:2])

    lines!(ax1, 1:10, rand(10))
    lines!(ax2, 1:10, rand(10))
    lines!(ax3, 1:10, rand(10))
    lines!(ax4, 1:10, rand(10))
```

```

hidexdecorations!(ax3; ticks = false, grid = false)

Box(fig[2:3,1:2, Right()], color = (:slateblue1, 0.35))
Label(fig[2:3,1:2, Right()], "protrusion", rotation = pi/2, textsize = 14,
      padding = (3,3,3,3))
Label(fig[1,1:2, Top()], "Mixed alignmode", textsize = 16,
      padding = (0,0,15,0))

colsizer!(fig.layout, 1, Auto(2))
rowsizer!(fig.layout, 2, Auto(0.5))
rowsizer!(fig.layout, 3, Auto(0.5))
rowgap!(fig.layout, 1, 15)
rowgap!(fig.layout, 2, 0)
colgap!(fig.layout, 5)
fig
end
mixed_mode_layout()

```

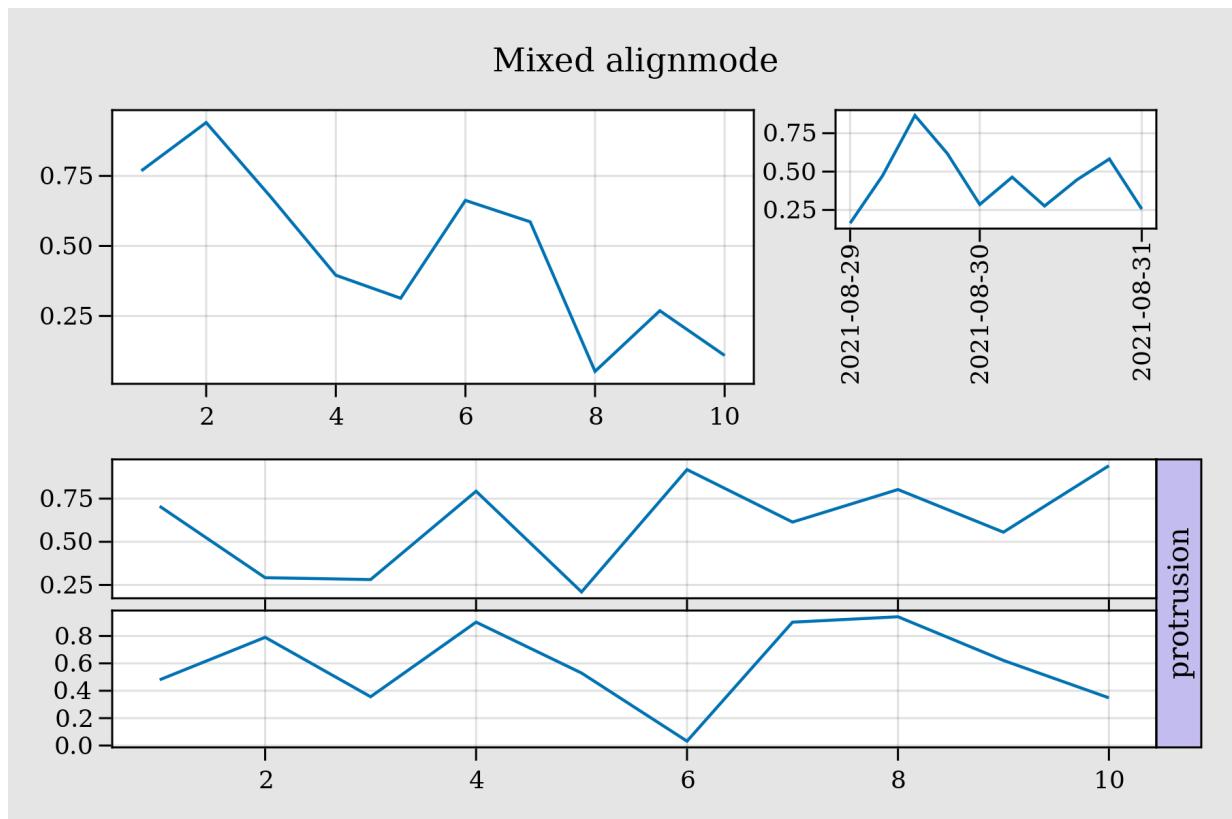


Figure 7.28: Mixed mode layout.

Here, the argument `alignmode = Mixed(bottom=0)` is shifting the bounding box to the bottom, so that this will align with the panel on the left filling the space.

Also, see how `colsizer!` and `rowsizer!` are being used for different columns and rows. You could also put a number instead of `Auto()` but then everything will be fixed. And, additionally, one could also give a `height` or `width` when defining the `Axis`, as in `Axis(fig, height = 50)` which will be fixed as well.

7.6.1 Nested Axis (*subplots*)

It is also possible to define a set of `Axis` (*subplots*) explicitly, and use it to build a main figure with several rows and columns. For instance, the following its a “complicated” arrangement of `Axis`:

```
function nested_sub_plot!(fig)
    color = rand(RGBf0)
    ax1 = Axis(fig[1,1], backgroundcolor = (color,0.25))
    ax2 = Axis(fig[1,2], backgroundcolor = (color,0.25))
    ax3 = Axis(fig[2,1:2], backgroundcolor = (color,0.25))
    ax4 = Axis(fig[1:2,3], backgroundcolor = (color,0.25))
    return (ax1,ax2,ax3,ax4)
end
```

which, when used to build a more complex figure by doing several calls, we obtain:

```
function main_figure()
    fig = Figure()
    Axis(fig[1,1])
    nested_sub_plot!(fig[1,2])
    nested_sub_plot!(fig[1,3])
    nested_sub_plot!(fig[2,1:3])
    fig
end
main_figure()
```

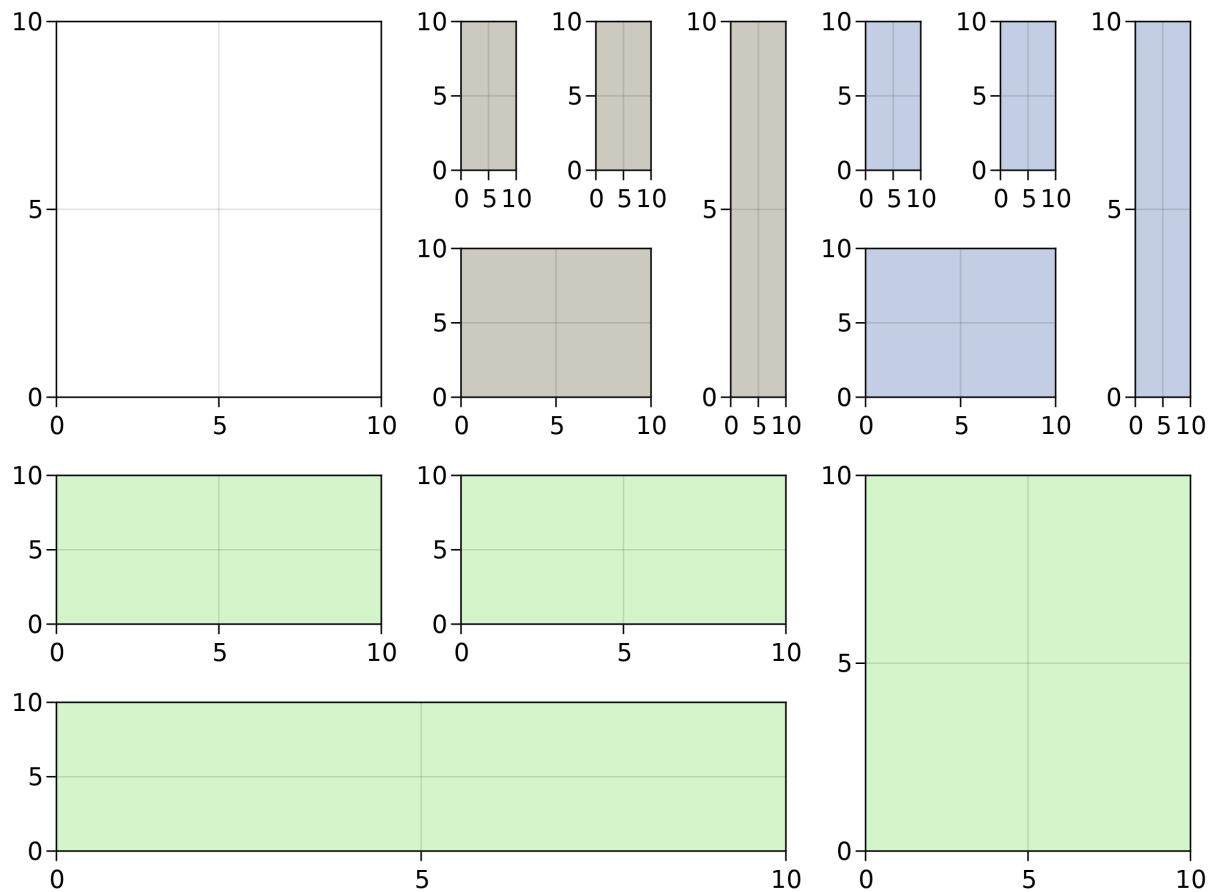


Figure 7.29: Main figure.

Note that different subplot functions can be called here. Also, each `Axis` here is an independent part of `Figure`. So that, if you need to do some `rowgap!`'s or `colsizes!`'s operations, you will need to do it in each one of them independently or to all of them together.

For grouped `Axis` (`subplots`) we can use `GridLayout()` which, then, could be used to composed a more complicated `Figure`.

7.6.2 Nested GridLayout

By using `GridLayout()` we can group subplots, allowing more freedom to build complex figures. Here, using our previous `nested_sub_plot!` we define three sub-groups and one normal `Axis`:

```
function nested_Grid_Layouts()
    fig = Figure(backgroundcolor = RGBf0(0.96, 0.96, 0.96),)
    ga = fig[1, 1] = GridLayout()
    gb = fig[1, 2] = GridLayout()
    gc = fig[1, 3] = GridLayout()
    gd = fig[2, 1:3] = GridLayout()
    gA = Axis(ga[1,1])
    nested_sub_plot!(gb)
    axsc = nested_sub_plot!(gc)
    nested_sub_plot!(gd)
```

```
[hidedecorations!(axsc[i],grid=false,ticks=false) for i in 1:length(axsc)]  
  
colgap!(gc, 5)  
rowgap!(gc, 5)  
rowsize!(fig.layout, 2, Auto(0.5))  
colsizes!(fig.layout, 1, Auto(0.5))  
fig  
end  
nested_Grid_Layouts()
```

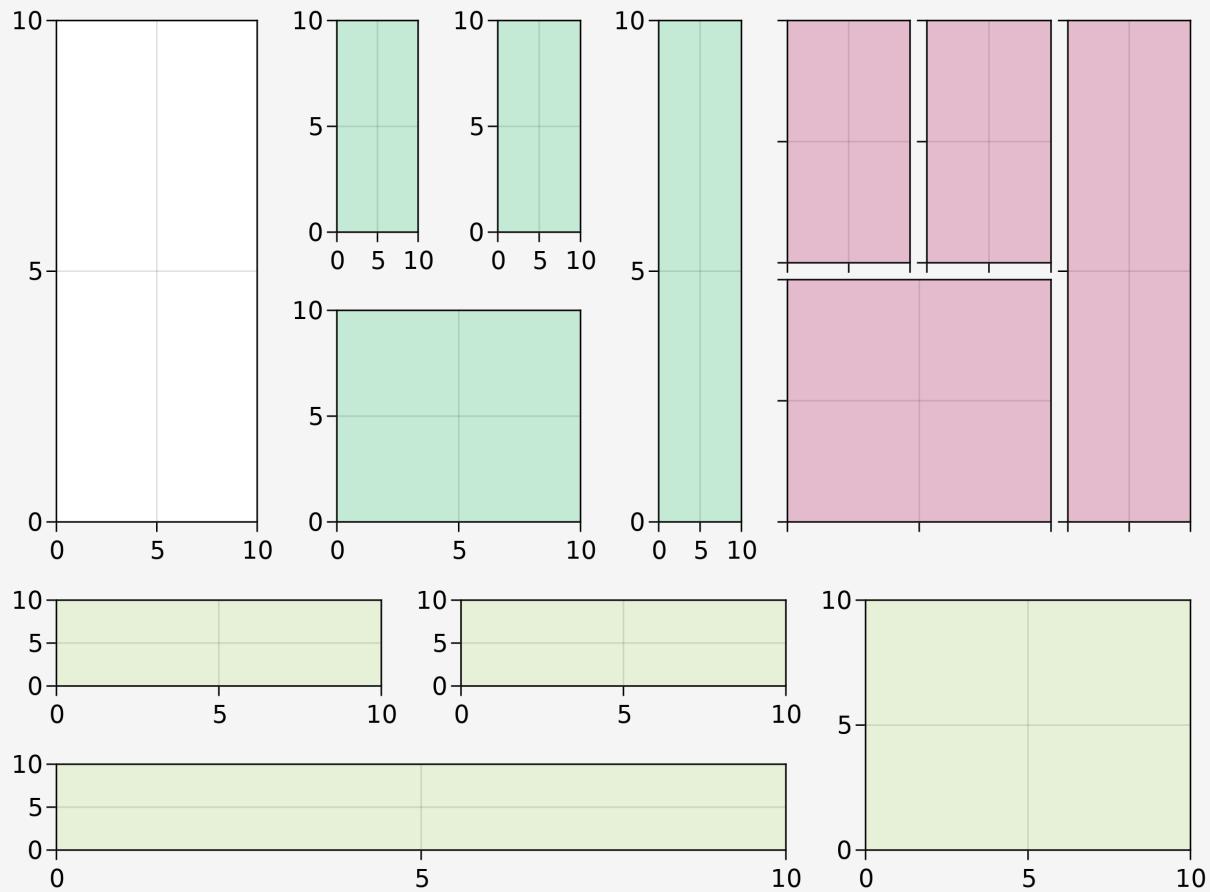


Figure 7.30: Nested Grid Layouts.

Now, using `rowgap!` or `colsizes!` over each group is possible and `rowsize!`, `colsizes!` can also be applied to the set of `GridLayout()`s.

7.6.3 Inset plots

Currently, doing `inset` plots is a little bit tricky. Here, we show two possible ways of doing it by initially defining auxiliary functions. The first one is by doing a `BBox`, which lives in the whole `Figure` space:

```
function add_box_inset(fig; left=100,right=250,bottom = 200,top= 300,  
    bgcolor=:grey90)  
    inset_box = Axis(fig, bbox = BBox(left, right, bottom, top),
```

```

    xticklabelsize = 12, yticklabelsize=12, backgroundcolor = bgcolor)

# bring content upfront
# bring content upfront
translate!(inset_box.scene, 0, 0, 10)
elements = keys(inset_box.elements)
filtered = filter(ele -> ele != :xaxis && ele != :yaxis , elements)
foreach(ele -> translate!(inset_box.elements[ele], 0, 0, 9), filtered)
return inset_box
end

```

Then, the `inset` is easily done, as in:

```

function figure_box_inset()
    fig = Figure(resolution=(600,400))
    ax = Axis(fig[1,1], backgroundcolor = :white)
    inset_ax1 = add_box_inset(fig; left=100,right=250,bottom = 200,top= 300,
        bgcolor = :grey90)
    inset_ax2 = add_box_inset(fig; left=500,right=600,bottom = 100,top= 200,
        bgcolor = (:white,.65))

    lines!(ax, 1:10)
    lines!(inset_ax1, 1:10)
    scatter!(inset_ax2, 1:10, color = :black)
    fig
end
figure_box_inset()

```

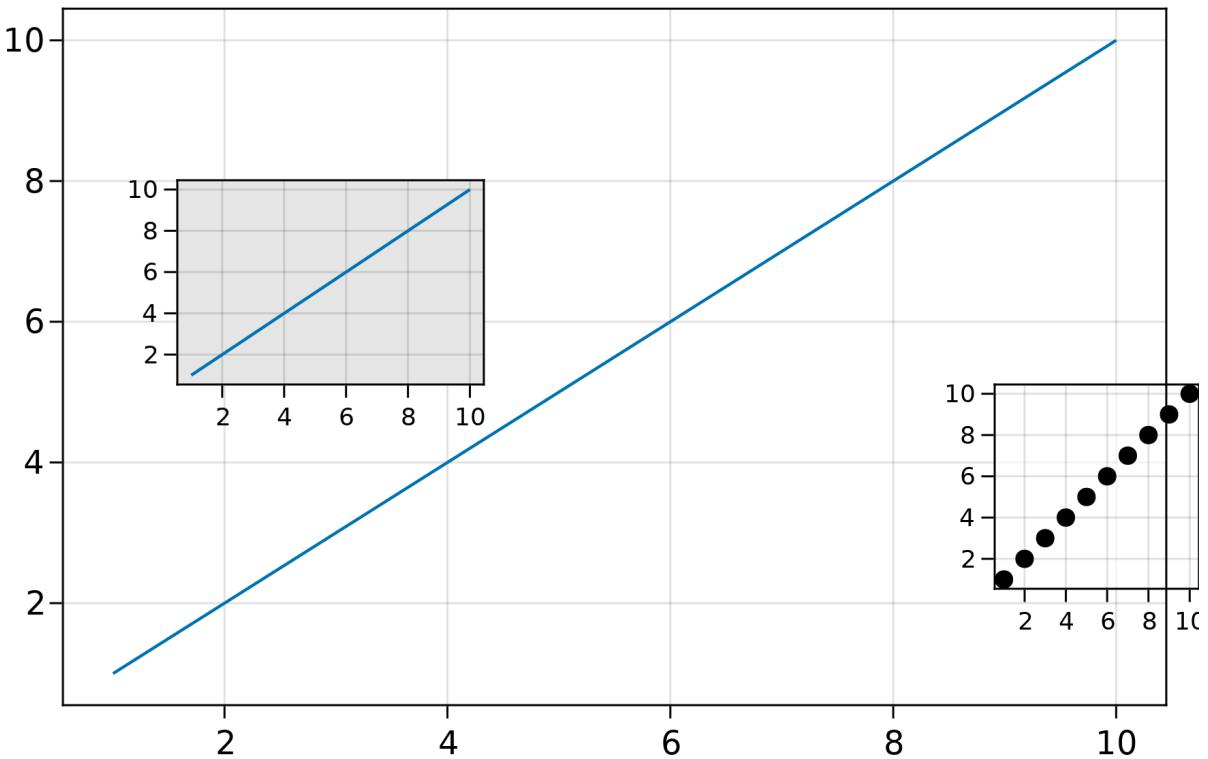


Figure 7.31: Figure box inset.

where the `Box` dimensions are bound by the `Figure`'s resolution. Note, that an inset can be also outside the `Axis`. The other approach, is by defining a new `Axis` into a position `fig[i,j]` specifying his `width`, `height`, `halign` and `valign`. We do that in the following function:

```
function add_axis_inset(; pos = fig[1,1], halign=0.1, valign=0.5,
    width = Relative(0.5), height= Relative(0.35), bgcolor=:lightgray)

    inset_box = Axis(pos, width = width, height = height,
        halign=halign, valign=valign, xticklabelsize = 12, yticklabelsize=12,
        backgroundcolor = bgcolor)

    # bring content upfront
    translate!(inset_box.scene, 0, 0, 10)
    elements = keys(inset_box.elements)
    filtered = filter(ele -> ele != :xaxis && ele != :yaxis , elements)
    foreach(ele -> translate!(inset_box.elements[ele], 0, 0, 9), filtered)
    return inset_box
end
```

See that in the following example the `Axis` with gray background will be rescaled if the total figure size changes. The `insets` are bound by the `Axis` positioning.

```
function figure_axis_inset()
    fig = Figure(resolution=(600,400))
    ax = Axis(fig[1,1], backgroundcolor = :white)
    inset_ax1 = add_axis_inset(; pos = fig[1,1], halign=0.1, valign=0.65,
```

```

    width = Relative(0.3), height= Relative(0.3), bgcolor= :grey90)
inset_ax2 = add_axis_inset(; pos = fig[1,1], halign=1, valign=0.25,
    width = Relative(0.25), height= Relative(0.3), bgcolor= (:white,0.65))

lines!(ax, 1:10)
lines!(inset_ax1, 1:10)
scatter!(inset_ax2, 1:10, color = :black)
fig
end
figure_axis_inset()

```

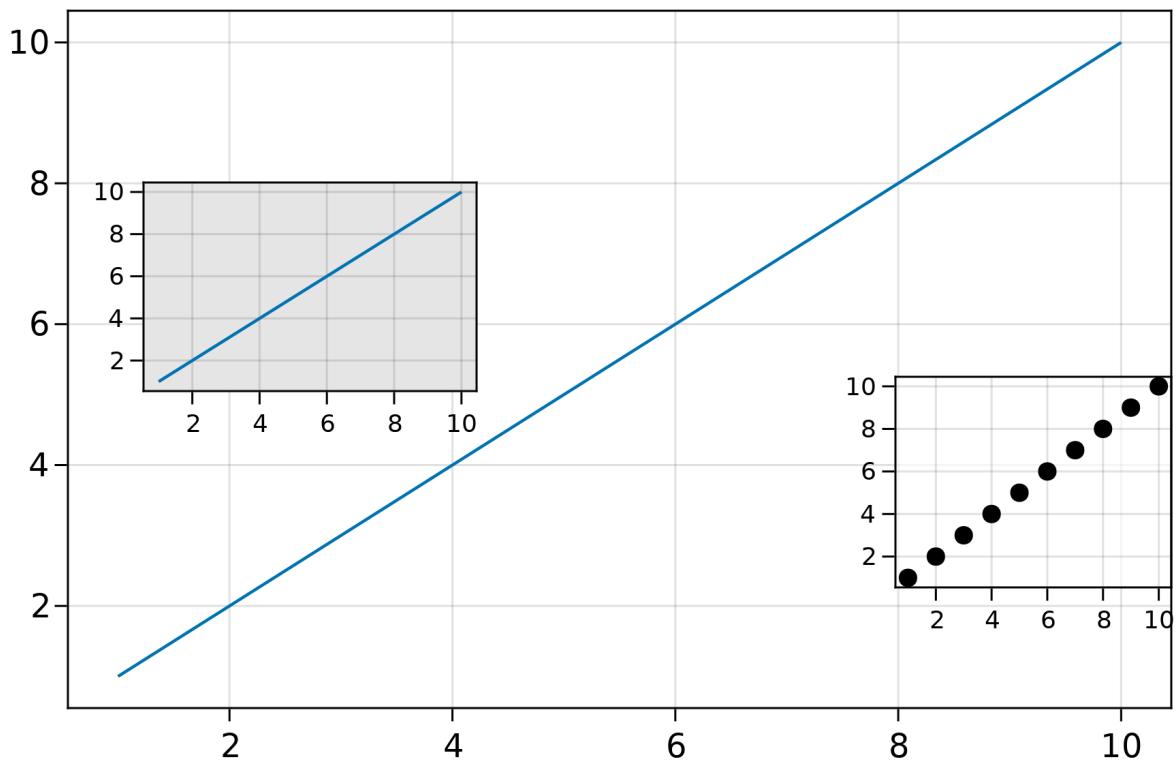


Figure 7.32: Figure axis inset.

And this should cover most used cases for layouting with Makie. Now, let's do some nice 3D examples with `GLMakie.jl`.

7.7 GLMakie.jl

`CairoMakie.jl` supplies all our needs for static 2D images. But sometimes we want interactivity, especially when we are dealing with 3D images. Visualizing data in 3D is also a common practice to gain insight from your data. This is where `GLMakie.jl` might be helpful, since it uses OpenGL as a backend that adds interactivity and responsiveness to plots. Like before, a simple plot includes, of course, lines and points. So, we will start with those and since we already know how layouts work, we will put that into practice.

7.7.1 Scatters and Lines

For scatter plots we have two options, the first one is `scatter(x, y, z)` and the second one is `meshscatter(x, y, z)`. In the first one markers don't scale in the axis directions, but in the later they do because they are actual geometries in 3D space. See the next example:

```
using GLMakie
GLMakie.activate!()
```

```
function scatters_in_3D()
    Random.seed!(123)
    xyz = randn(10, 3)
    x, y, z = xyz[:,1], xyz[:,2], xyz[:,3]
    fig = Figure(resolution = (1800, 600), fontsize = 26)
    ax1 = Axis3(fig; aspect = (1, 1, 1), perspectiveness = 0.5)
    ax2 = Axis3(fig; aspect= (1,1,1), perspectiveness = 0.5,)
    ax3 = Axis3(fig; aspect= :data, perspectiveness = 0.5,)

    scatter!(ax1, x, y, z; markersize = 50)
    meshscatter!(ax2, x, y, z; markersize = 0.25)
    hm = meshscatter!(ax3, x, y, z; markersize = 0.25,
        marker = FRect3D(Vec3f0(0), Vec3f0(1)), color = 1:size(xyz)[2],
        colormap = :plasma, transparency = false)
    cbar = Colorbar(fig, hm, label = "values", height = Relative(0.5))
    fig[1,1] = ax1
    fig[1,2] = ax2
    fig[1,3] = ax3
    fig[1,4] = cbar
    fig
end
scatters_in_3D()
```

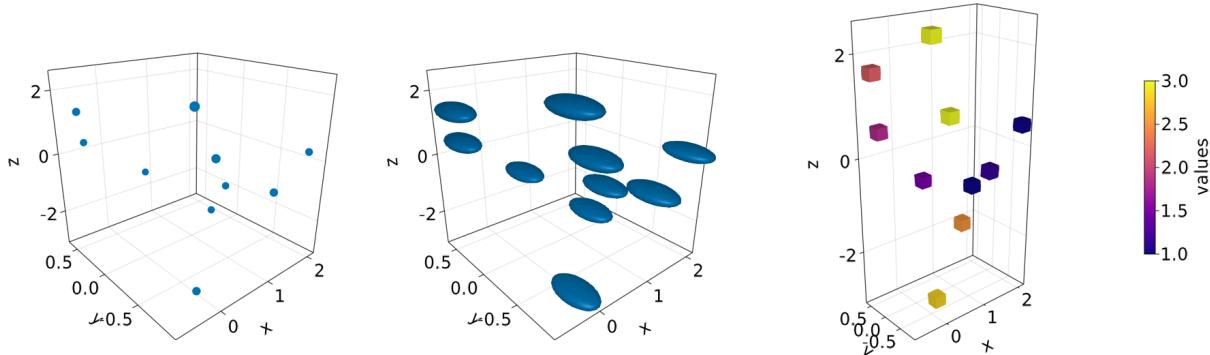


Figure 7.33: Scatters in 3D.

Note also, that a different geometry can be passed as markers, i.e., a square/rectangle and we can assign a `colormap` for them as well. In the middle panel one could get perfect spheres by doing `aspect = :data` as in the right panel.

And doing `lines` or `scatterlines` is also straightforward:

```

function lines_in_3D()
    Random.seed!(123)
    xyz = randn(10, 3)
    x, y, z = xyz[:,1], xyz[:,2], xyz[:,3]
    fig = Figure(resolution = (1800,600), fontsize = 26)
    ax1 = Axis3(fig; aspect= (1,1,1), perspectiveness = 0.5,)
    ax2 = Axis3(fig; aspect= (1,1,1), perspectiveness = 0.5,)
    ax3 = Axis3(fig; aspect= :data, perspectiveness = 0.5,)

    lines!(ax1, x, y, z; color = 1:size(xyz)[2], linewidth=3)
    scatterlines!(ax2, x, y, z; markersize = 50)
    hm = meshscatter!(ax3, x, y, z; markersize = 0.2,
                      color = 1:size(xyz)[2])
    lines!(ax3, x, y, z; color = 1:size(xyz)[2])
    cbar = Colorbar(fig, hm; label = "values", height = 15, vertical = false,
                    flipaxis = false, ticksize=15, tickalign = 1, width = Relative(3.55/4))
    fig[1,1] = ax1
    fig[1,2] = ax2
    fig[1,3] = ax3
    fig[2,1] = cbar
    fig
end
lines_in_3D()

```

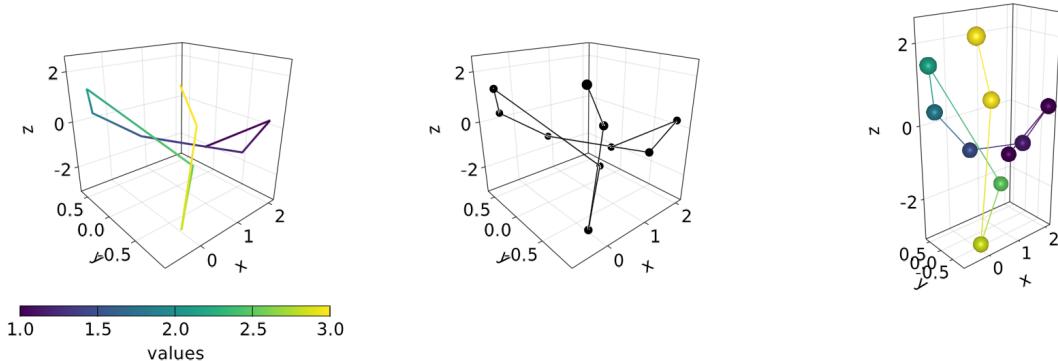


Figure 7.34: Lines in 3D.

Plotting a `surface` is also easy to do as well as a `wireframe` and `contour` lines in 3D.

7.7.2 Surfaces, wireframe, contour, contourf and contour3d

To show these cases we'll use the following `peaks` function:

```

function peaks(; n = 49)
    x = LinRange(-3, 3, n)
    y = LinRange(-3, 3, n)
    a = 3 * (1 .- x').^2 .* exp.(-(x'.^2) .- (y .+ 1).^2)
    b = 10 * (x' / 5 .- x'.^3 .- y.^5) .* exp.(-x'.^2 .- y.^2)
    c = 1 / 3 * exp.(-(x' .+ 1).^2 .- y.^2)

```

```

    return (x, y, a .- b .- c)
end

```

The output for the different plotting functions is

```

function plot_peaks_function()
    x, y, z = peaks()
    x2, y2, z2 = peaks(;n = 15)

    fig = Figure(resolution = (1800,600), fontsize = 26)
    ax1 = Axis3(fig[1,1]; aspect= (1,1,1))
    ax2 = Axis3(fig[1,2]; aspect= (1,1,1))
    ax3 = Axis3(fig[1,3]; aspect= (1,1,1))

    hm = surface!(ax1, x, y, z)
    wireframe!(ax2, x2, y2, z2)
    contour3d!(ax3, x, y, z; levels = 20)
    Colorbar(fig[1,4], hm, height = Relative(0.5))
    fig
end
plot_peaks_function()

```

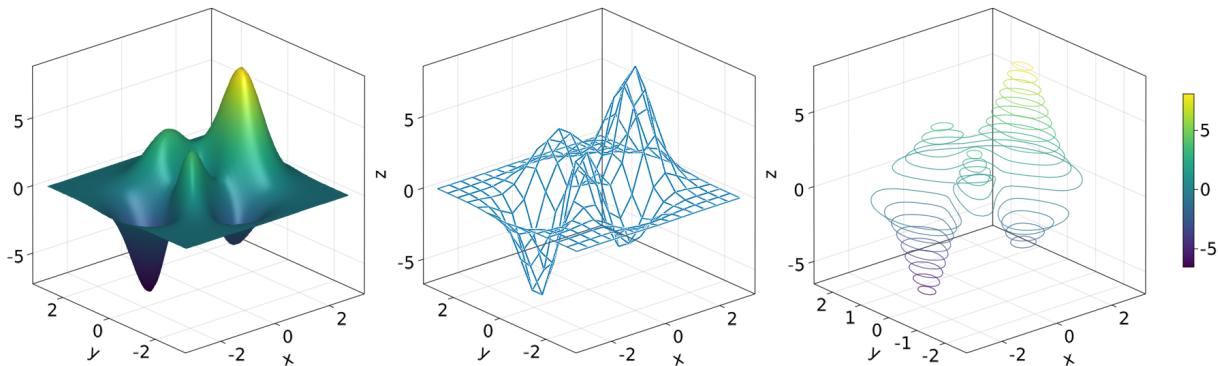


Figure 7.35: Plot peaks function.

But, it can also be plotted with a `heatmap(x, y, z)`, `contour(x, y, z)` or `contourf(x, y, z)`:

```

function heatmap_contour_and_contourf()
    x, y, z = peaks()

    fig = Figure(resolution = (1800,600), fontsize = 26)
    ax1 = Axis(fig[1,1]; aspect = DataAspect())
    ax2 = Axis(fig[1,2]; aspect = DataAspect())
    ax3 = Axis(fig[1,3]; aspect = DataAspect())

    hm = heatmap!(ax1, x, y, z)
    contour!(ax2, x, y, z; levels = 20)
    contourf!(ax3, x, y, z)
    Colorbar(fig[1,4], hm, height = Relative(0.5))
    fig
end

```

```
heatmap_contour_and_contourf()
```

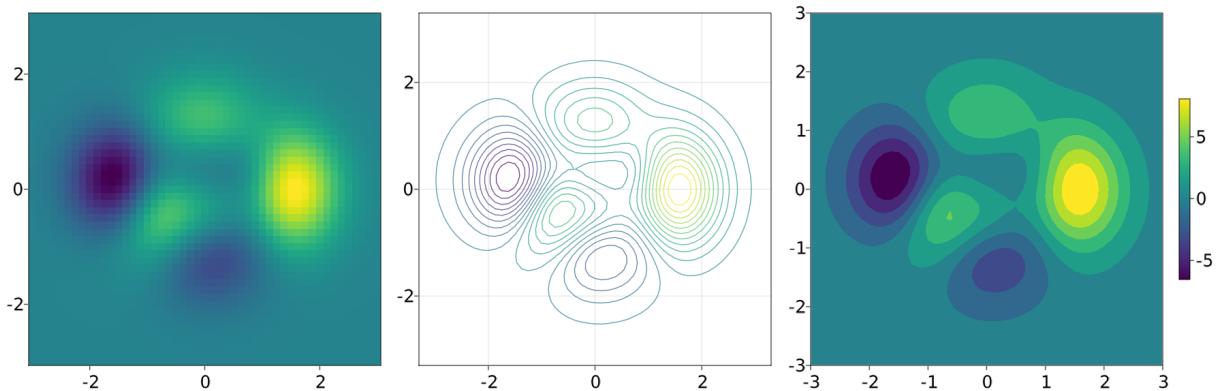


Figure 7.36: Heatmap contour and contourf.

Additionally, by changing Axis to an Axis3, these plots will be automatically be in the x-y plane:

```
function heatmap_contour_and_contourf_in_a_3d_plane()
    x, y, z = peaks()

    fig = Figure(resolution = (1800,600), fontsize = 26)
    ax1 = Axis3(fig[1,1])
    ax2 = Axis3(fig[1,2])
    ax3 = Axis3(fig[1,3])

    hm = heatmap!(ax1, x, y, z)
    contour!(ax2, x, y, z; levels = 20)
    contourf!(ax3, x, y, z)
    Colorbar(fig[1,4], hm, height = Relative(0.5))
    fig
end
heatmap_contour_and_contourf_in_a_3d_plane()
```

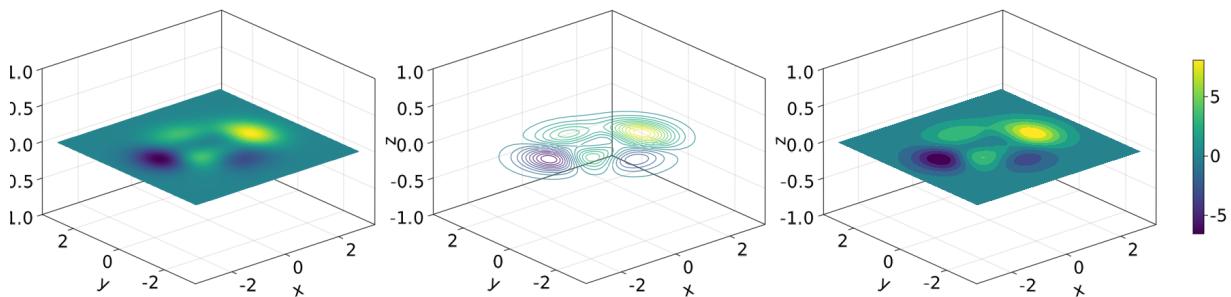


Figure 7.37: Heatmap contour and contourf in a 3d plane.

Something else that is easy to do is to mix all these plotting functions into just one plot, namely:

```
using Downloads, FileIO
```

```
function mixing_surface_contour3d_contour_and_contourf()
    url = "https://raw.githubusercontent.com/JuliaImages/TestImages.jl/images/images/coffee.png"
    img = load(Downloads.download(url))

    x, y, z = peaks()
    cmap = :Spectral_11
    fig = Figure(resolution = (1400,800), fontsize = 26)
    ax1 = Axis3(fig[1,1]; aspect = (1,1,1), elevation = pi/6,
                perspectiveness = 0.5, xzpanelcolor= (:black,0.75),
                yzpanelcolor= :black, zgridcolor = :grey70,
                ygridcolor = :grey70, xgridcolor = :grey70)
    ax2 = Axis3(fig[1,3]; aspect = (1,1,1), elevation = pi/6,
                perspectiveness = 0.5)

    hm = surface!(ax1, x, y, z; colormap = (cmap, 0.95), shading = true)
    contour3d!(ax1, x, y, z .+ 0.02; colormap =cmap,
               levels = 20, linewidth = 2 )
    xmin, ymin, zmin = minimum(ax1.finallimits[])
    xmax, ymax, zmax = maximum(ax1.finallimits[])
    contour!(ax1, x, y, z; colormap = cmap, levels = 20,
              transformation = (:xy, zmax))
    contourf!(ax1, x, y, z; colormap = cmap,
              transformation = (:xy, zmin))
    Colorbar(fig[1,2], hm, width = 15, ticksize=15, tickalign = 1,
              height = Relative(0.35))
    # transformations into planes
    heatmap!(ax2, x, y, z; colormap = :viridis,
              transformation = (:yz, 3.5))
    contourf!(ax2, x, y, z; colormap = :CMRmap,
              transformation = (:xy, -3.5))
    contourf!(ax2, x, y, z; colormap = :bone_1,
              transformation = (:xz, 3.5))
    image!(ax2, -3..3, -3..2, rotr90(img);
           transformation = (:xy, 3.8))
    xlims!(ax2, -3.8,3.8)
    ylims!(ax2, -3.8,3.8)
    zlims!(ax2, -3.8,3.8)
    fig
end
mixing_surface_contour3d_contour_and_contourf()
```

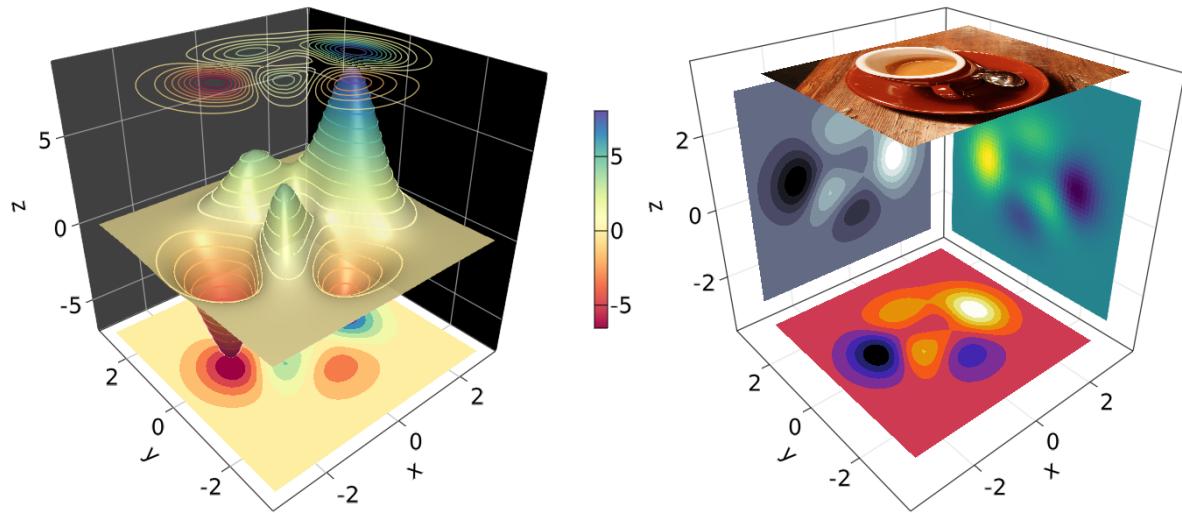


Figure 7.38: Mixing surface contour3d contour and contourf.

Not bad, right? From there is clear that any heatmap's, contour's, contourf's or image can be plotted into any plane.

7.7.3 Arrows and Streamplots

arrows and streamplot are plots that might be useful when we want to know the directions that a given variable will follow. See a demonstration below:

```
using LinearAlgebra

function arrows_and_streamplot_in_3d()
    ps = [Point3f0(x, y, z) for x in -3:1:3 for y in -3:1:3 for z in -3:1:3]
    ns = map(p -> 0.1*rand() * Vec3f0(p[2], p[3], p[1]), ps)
    lengths = norm.(ns)
    flowField(x,y,z) = Point(-y + x*(-1+x^2+y^2)^2, x + y*(-1+x^2+y^2)^2,
                               z + x*(y-z^2))

    fig = Figure(resolution = (1400,800), fontsize = 26)
    ax1 = Axis3(fig[1,1]; aspect = (1,1,1), perspectiveness = 0.5)
    ax2 = Axis3(fig[1,2]; aspect = (1,1,1), perspectiveness = 0.5)
    arrows!(ax1, ps, ns, color=lengths, linewidth = 0.1,
           arrowsize = Vec3f0(0.2, 0.2, 0.3), align = :center)
    streamplot!(ax2, flowField, -4..4, -4..4, -4..4, colormap = :plasma,
               gridsize= (7,7), arrow_size = 0.25, linewidth=1)
    fig
end

arrows_and_streamplot_in_3d()
```

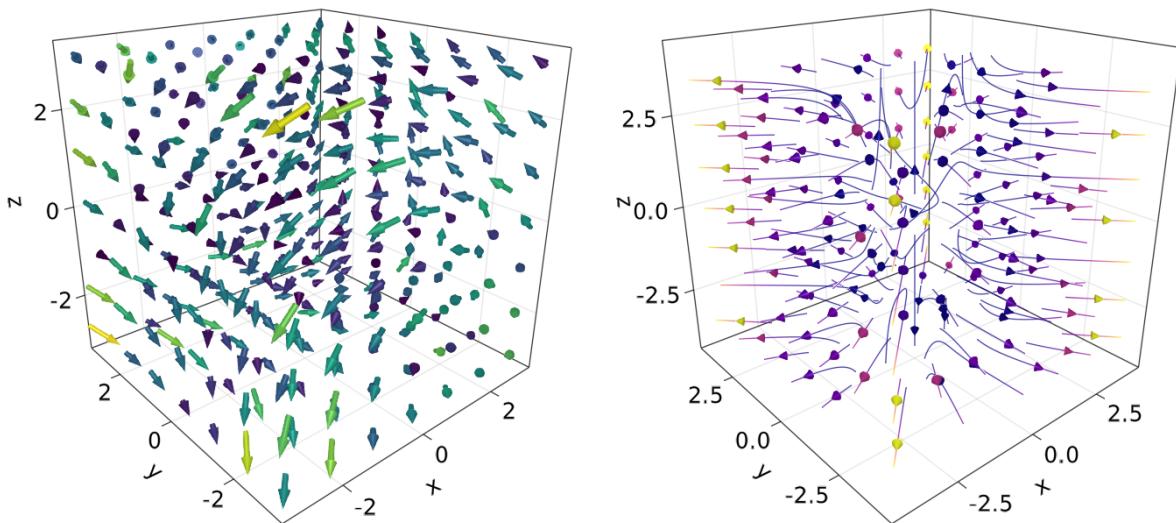


Figure 7.39: Arrows and streamplot in 3d.

Other interesting examples are a `mesh(obj)`, a `volume(x, y, z, vals)`, and a `contour(x, y, z, vals)`.

7.7.4 Meshes and Volumes

Drawing Meshes comes in handy when you want to plot geometries, like a `sphere` or a Rectangle, i. e. `FRect3D`. Another approach to visualize points in 3D space is by calling the functions `volume` and `contour`, which implements ray tracing to simulate a wide variety of optical effects. See the next examples:

```
using GeometryBasics
```

```
function mesh_volume_contour()
    # mesh objects
    rectMesh = FRect3D(Vec3f0(-0.5), Vec3f0(1))
    recmesh = GeometryBasics.mesh(rectMesh)
    sphere = Sphere(Point3f0(0), 1)
    # https://juliageometry.github.io/GeometryBasics.jl/stable/primitives/
    spheremesh = GeometryBasics.mesh(Tesselation(sphere, 64))
    # uses 64 for tesselation, a smoother sphere
    colors = [rand() for v in recmesh.position]
    # cloud points for volume
    x = 1:10
    y = 1:10
    z = 1:10
    vals = randn(10,10,10)
    # figure
    fig = Figure(resolution = (1800,600), fontsize = 26)
    ax1 = Axis3(fig[1,1]; aspect = (1,1,1), perspectiveness = 0.5)
    ax2 = Axis3(fig[1,2]; aspect = (1,1,1), perspectiveness = 0.5)
```

```

ax3 = Axis3(fig[1,3]; aspect = (1,1,1), perspectiveness = 0.5)

mesh!(ax1, recmesh; color= colors, colormap = :rainbow, shading = false)
mesh!(ax1, spheremesh; color = (:white,0.25), transparency = true)
volume!(ax2, x, y, z, vals; colormap = Reverse(:plasma))
contour!(ax3, x, y, z, vals; colormap = Reverse(:plasma))

fig
end
mesh_volume_contour()

```

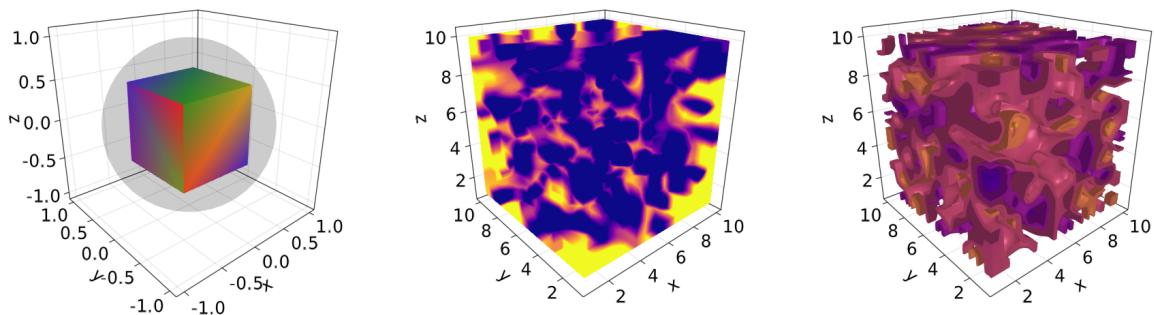


Figure 7.40: Mesh volume contour.

Note that here we are plotting two meshes in the same axis, one transparent sphere and a cube. So far, we have covered most of the 3D use-cases. Another example is `?linesegments`.

Taking as reference the previous example one can do the following custom plot with spheres and rectangles:

```

using GeometryBasics, Colors

function grid_spheres_and_rectangle_as_plate()
    Random.seed!(123)
    rectMesh = FRect3D(Vec3f0(-1,-1,2.1), Vec3f0(22,11,0.5))
    recmesh = GeometryBasics.mesh(rectMesh)
    colors = [RGBA(rand(4)...) for v in recmesh.position]
    fig = with_theme(theme_dark()) do
        fig = Figure(resolution = (1600,800), fontsize = 26)
        ax1 = Axis3(fig[1,1]; aspect = (1,1,1), perspectiveness = 0.5,
                   azimuth= 0.7223275083269882)
        ax2 = Axis3(fig[1,2], aspect=:data, perspectiveness = 0.5,)

        for i in 1:2:10, j in 1:2:10, k in 1:2:10
            sphere = Sphere(Point3f0(i,j,k), 1)
            spheremesh = GeometryBasics.mesh(Tesselation(sphere, 32))
            mesh!(ax1, spheremesh; color = RGBA(i*0.1,j*0.1,k*0.1, 0.75),
                  transparency = false, shading = false)
        end
        cbarPal = :plasma
        cmap = get(colorschemes[cbarPal], LinRange(0,1,50))
        for i in 1:2.5:20, j in 1:2.5:10, k in 1:2.5:4

```

```

        sphere = Sphere(Point3f0(i,j,k), 1)
        spheremesh = GeometryBasics.mesh(Tesselation(sphere, 32))
        mesh!(ax2, spheremesh; color = cmap[rand(1:50)],
        lightposition = Vec3f0(10, 5, 2),
        ambient = Vec3f0(0.95, 0.95, 0.95), backlight = 1f0,
        transparency = false, shading = true)
    end
    mesh!(recmesh; color= colors, colormap = :rainbow, shading = false,
        transparency = false)
    #hidedecorations!(ax2)
    fig
end
fig
grid_spheres_and_rectangle_as_plate()

```

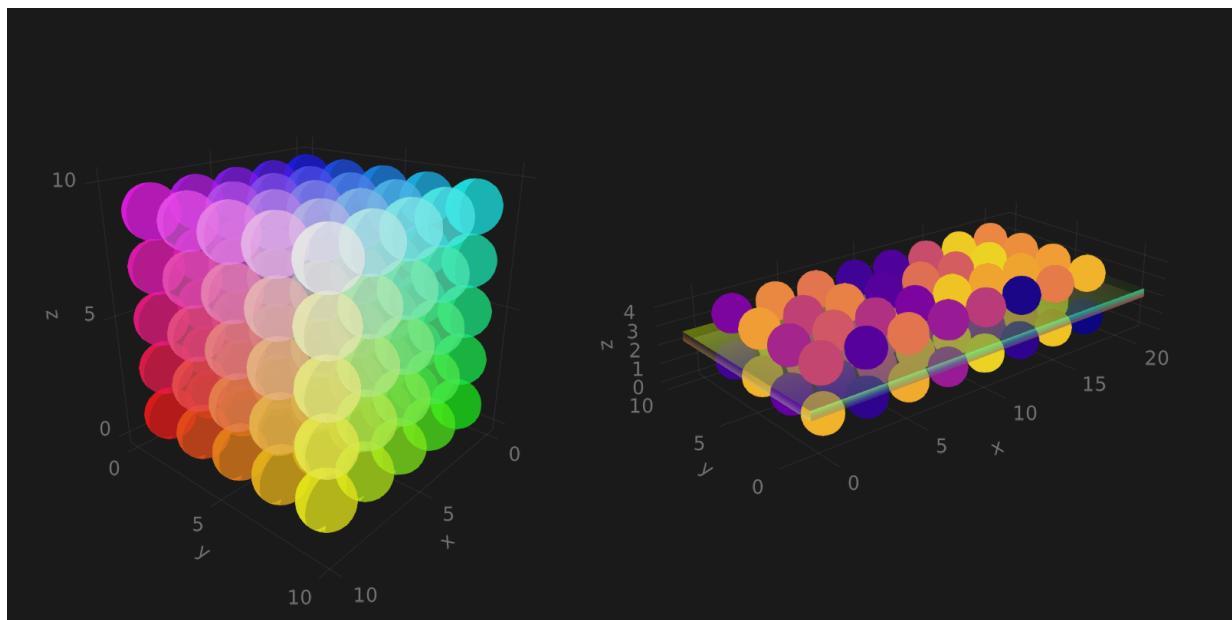


Figure 7.41: Grid spheres and rectangle as plate.

Here, the rectangle is semi-transparent due to the alpha channel added to the RGB color. The rectangle function is quite versatile, for instance 3D boxes are easy do implement which in turn could be used for plotting a 3D histogram. See our next example:

```

function histogram_or_bars_in_3d()
    x, y, z = peaks(;n = 15)δ
    x = (x[2] - x[1])/2δ
    y = (y[2] - y[1])/2

    cbarPal = :Spectral_11
    ztmp = (z .- minimum(z))./(maximum(z .- minimum(z)))
    cmap = get(colorschemes[cbarPal], ztmp)
    cmap2 = reshape(cmap, size(z))
    ztmp2 = abs.(z) ./ maximum(abs.(z)) .+ 0.15

    fig = Figure(resolution = (1400,800), fontsize = 26)

```

```

ax1 = Axis3(fig[1,1]; aspect = (1,1,1), elevation = π/6, perspectiveness = 0.5)
ax2 = Axis3(fig[1,2]; aspect = (1,1,1), perspectiveness = 0.5)

for (idx, i) in enumerate(x), (idy,j) in enumerate(y)
    rectMesh = FRect3D(Vec3f0(i - δx, j - δy, 0), Vec3f0δ(2x, δ2y, z[idx, idy]))
    recmesh = GeometryBasics.mesh(rectMesh)
    mesh!(ax1, recmesh; color= cmap2[idx,idy], shading = false)
end
for (idx, i) in enumerate(x), (idy,j) in enumerate(y)
    rectMesh = FRect3D(Vec3f0(i - δx, j - δy, 0), Vec3f0δ(2x, δ2y, z[idx, idy]))
    recmesh = GeometryBasics.mesh(rectMesh)
    lines!(ax2, recmesh; color= (cmap2[idx,idy], ztmp2[idx, idy]))
    mesh!(ax2, recmesh; color= (cmap2[idx,idy], 0.25),
          shading = false, transparency = true)
end
fig
end
histogram_or_bars_in_3d()

```

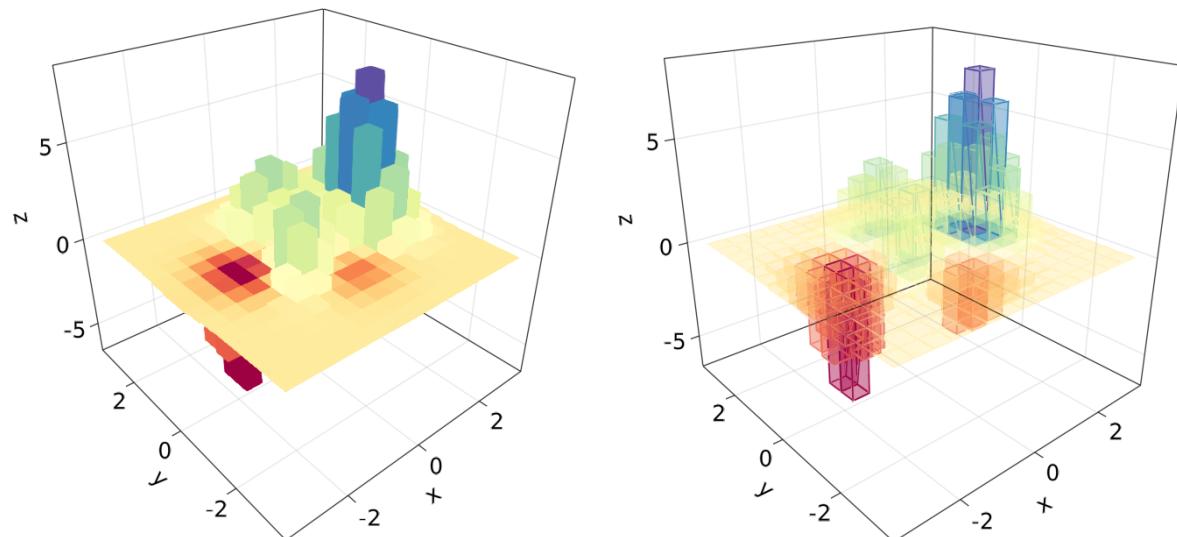


Figure 7.42: Histogram or bars in 3d.

Note, that you can also call `lines` or `wireframe` over a mesh object.

7.7.5 Filled Line and Band

For our last example we will show how to do a filled curve in 3D with `band` and some `linesegments`:

```

function filled_line_and_linesegments_in_3D()
    xs = LinRange(-3, 3, 10)
    lower = [Point3f0(i,-i,0) for i in LinRange(0,3,100)]

```

```

upper = [Point3f0(i,-i, sin(i) * exp(-(i+i))) for i in range(0,3, length=100)]

fig = Figure(resolution = (1400,800), fontsize = 26)
ax1 = Axis3(fig[1,1]; elevation = pi/6, perspectiveness = 0.5)
ax2 = Axis3(fig[1,2]; elevation = pi/6, perspectiveness = 0.5)

band!(ax1, lower, upper, color = repeat(norm.(upper), outer=2), colormap = :CMRmap)
lines!(ax1, upper, color = :black)
linesegments!(ax2, cos.(xs), xs, sin.(xs), linewidth = 5,
    color = 1:length(xs), colormap = :plasma)
fig
end
filled_line_and_linesegments_in_3D()

```

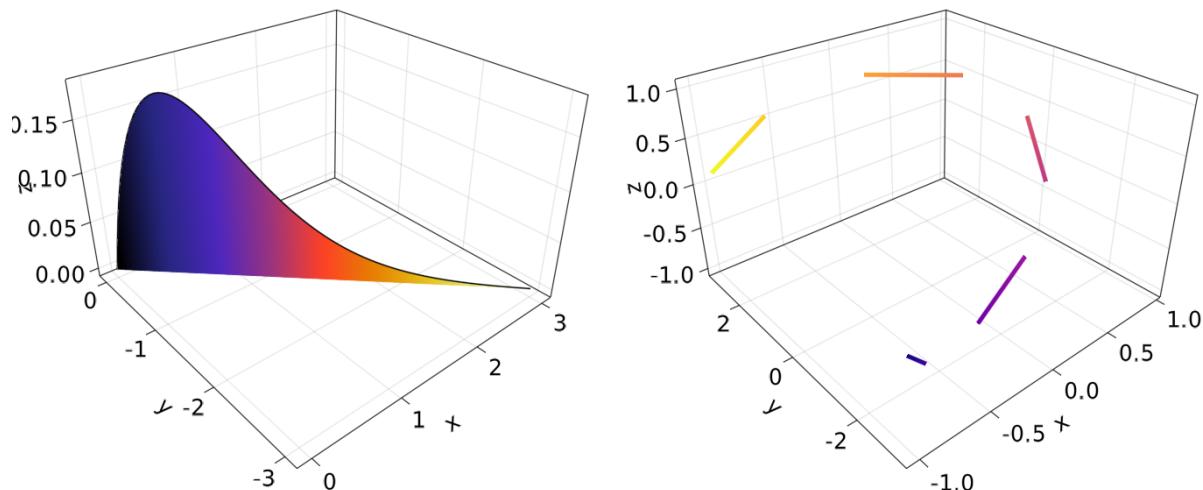


Figure 7.43: Filled line and linesegments in 3D.

Finally, our journey doing 3D plots has come to an end. You can combine everything we exposed here to create amazing 3D images! Now, it's time to dig into the basic rules to create animations.

7.8 Animations

Recording an animation with CairoMakie (GLMakie).

Test animation:

```

function first_animation()
    Random.seed!(123)
    npts = 100
    initms = 8*rand(npts) # initial marker size
    msize = Node(initms) # this is the variable that will change

```

```
# first frame, initial plot
fig, ax = scatter(2*rand(npts), rand(npts); markersize = msizes,
    color = initms, colormap = (:viridis, 0.75), strokewidth = 0.5,
    strokecolor = :white, figure = (; resolution=(600,400)),
    axis = (xlabel = "x", ylabel = "y",))
limits!(ax, 0,2,0,1)
# the animation is done by updating the node values
record(fig, "animScatters.mp4") do io
    for i in 1:0.1:8
        msizes[] = i*initms
        recordframe!(io) # record a new frame
    end
end
end
```

does it work?

8 Appendix

This book is built with Julia `jl string(VERSION)` and the following packages:

```
Books 0.7.11
CSV 0.8.5
CairoMakie 0.6.5
CategoricalArrays 0.10.0
ColorSchemes 3.14.0
Colors 0.12.8
DataFrames 1.2.2
FileIO 1.11.0
GLMakie 0.4.6
GeometryBasics 0.4.1
LaTeXStrings 1.2.1
Makie 0.15.2
Plots 1.21.1
QuartzImageIO 0.7.4
Reexport 1.2.2
StatsPlots 0.14.26
XLSX 0.7.6
```

8.1 Notation

In this book, we try to keep notation as consistent as possible. This makes reading and writing code easier. We can define the notation into three parts.

8.1.1 Julia Style Guide

Firstly, we attempt to stick to the conventions from the Julia Style Guide. Most importantly, we write functions not scripts (see also Section 1.2). Furthermore, we use naming conventions consistent with Julia `base/`, meaning:

- Use camelcase for modules: `module JuliaDataScience, struct MyPoint`. (Note that camelcase is called “camelcase,” because the capitalization of words, such as “iPad” or “CamelCase,” make the word look like a camel.)
- Write function names in lowercase letters and separate the words by underscores. It is also allowed to omit the separator when naming functions. For example, these function names are all in line with the conventions: `my_function`, `myfunction` and `string2int`.

Also, we avoid brackets in conditionals, that is, we write `if a == b` instead of `if (a == b)` and use 4 spaces per indentation level.

8.1.2 BlueStyle

The Blue Style Guide adds multiple conventions on top of the default Julia Style Guide. Some of these rules might sound pedantic, but we found that they make the code more readable.

From the style guide, we attempt to adhere specifically to:

- At most 92 characters per line in code (in Markdown files, longer lines are allowed).
- With `using`, `import` at most one module per line
- No trailing whitespace. Trailing whitespace makes inspecting changes in code more difficult since they do not change the behavior of the code but do show up as changes.
- Avoid extraneous spaces inside brackets. So, write `string(1, 2)` instead of `string(1 , 2)`.
- Global variables should be avoided.
- Try to limit function names to one or two words.
- Use the semicolon to clarify whether an argument is a keyword argument or not. For example, `func(x; y=3)` instead of `func(x, y=3)`.
- Avoid using multiple spaces to align things. So, write

```
a = 1
lorem = 2
```

instead of

```
a      = 1
lorem = 2
```

- Whenever appropriate, surround binary operators with a space, for example, `1 == 2` or `y = x + 1`.
- Indent triple-quotes and triple-backticks:

```
s = """
    my long text:
    [...]
    the end.
"""
```

- Do not omit zeros in floats (even though Julia allows it). Hence, write `1.0` instead of `1.` and write `0.1` instead of `.1`.
- Use `in` in for loops and not `=` or `\` (even though Julia allows it).

8.1.3 Our additions

- In text, we reference the function call `M.foo(3, 4)` as `M.foo` and not `M.foo(...)` or `M.foo()`.

- When talking about packages, like the `DataFrames` package, we explicitly write `DataFrames.jl` each time. This makes it easier to recognize that we are talking about a package.
- For filenames, we stick to “`file.txt`” and not `file.txt` or `file.txt`, because it is consistent with the code.
- For column names in tables, like the column `x`, we stick to `column :x`, because it is consistent with the code.
- Do not use unicode symbols in inline code. This is simply a bug in the PDF generation that we have to workaround for now.
- The line before each code block ends with a colon (`:`) to indicate that the line belongs to the code block.

8.2 Project Management in Julia

References

- Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1), 65–98.
- Chen, M., Mao, S., & Liu, Y. (2014). Big data: A survey. *Mobile Networks and Applications*, 19(2), 171–209.
- Domo. (2018). *Data never sleeps 6.0*. https://www.domo.com/assets/downloads/18_domo_data-never-sleeps-6+verticals.pdf
- Fitzgerald, S., Jimenez, D. Z., S., F., Yorifuji, Y., Kumar, M., Wu, L., Carosella, G., Ng, S., Parker, P., R. Carter, & Whalen, M. (2020). IDC FutureScape: Worldwide digital transformation 2021 predictions. *IDC FutureScape*.
- Gantz, J., & Reinsel, D. (2012). The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*, 2007(2012), 1–16.
- Khan, N., Yaqoob, I., Hashem, I. A. T., Inayat, Z., Mahmoud Ali, W. K., Alam, M., Shiraz, M., & Gani, A. (2014). Big data: Survey, technologies, opportunities, and challenges. *The Scientific World Journal*, 2014.
- Meng, X.-L. (2019). Data science: An artificial ecosystem. *Harvard Data Science Review*, 1(1). <https://doi.org/10.1162/99608f92.ba20f892>
- Perkel, J. M. (2019). Julia: Come for the syntax, stay for the speed. *Nature*, 572(7767), 141–142. <https://doi.org/10.1038/d41586-019-02310-3>
- Storopoli, J. (2021). *Bayesian statistics with julia and turing*. <https://storopoli.io/Bayesian-Julia>
- tanmay bakshi. (2021). *Baking Knowledge into Machine Learning ModelsChris Rackauckas on Tech-LifeSkills w/Tanmay Ep.55*. <https://youtu.be/moyPlhvW4Nk>
- TEDx Talks. (2020). *A programming language to heal the planet together: Julia | Alan Edelman | TEDxMIT*. <https://youtu.be/qGW0GT1rCvs>
- The Julia Programming Language. (2019). *JuliaCon 2019 | The Unreasonable Effectiveness of Multiple Dispatch | Stefan Karpinski*. <https://youtu.be/kc9HwsxE1OY>
- Wickham, H. (2011). The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1), 1–29.