```julia
# Install some packages
begin
    using LinearAlgebra
    using PlutoUI
    using PlutoTeachingTools
    using Plots
end
```

Table of Contents

Exercice Session 0: Getting Started with Julia

This first session is meant for you to get acquainted with the Julia programming language.

The following tutorial will give you an interactive tour of the main Julia functionalities we will need for the rest of the semester. By reading and running this notebook, you should be able to get a decent overview of Julia.

References:

- When in doubt about syntax, have a look at the Julia Cheatsheet
- This introduction is inspired by the MIT lecture Introduction to Computational Thinking where you can find more examples of Julia code and Pluto notebooks.
- Perhaps you also find the Comparative cheatsheet Python <-> Julia <-> Matlab useful.
- Some more tips are also available in the lecture notes

Pluto

Pluto is a browser-based notebook framework for Julia. All programming for this lecture will be done within Pluto notebooks.

Pluto allows for running code in an interactive fashion while also presenting computation results in a readable way, also integrating formatted text.

Cells

All Pluto inputs and outputs appear in **cells**. Cells can contain **text** or **code** and can be **visible** or **invisible**.

A cell can be run by **clicking the play button** below it or by hitting `Shift + Enter`. Its **visibility** can be toggled by clicking the *eye* button on its left (note that this only hides the cell's input, not its output).

Text cells

Text can be formatted by using Markdown. Markdown cells begin with `"md` and end with `"`.

Exercise

Modify the cell below, **run** it and look at the output. Then **toggle its visibility**.

Title

Subtitle

Text can be put in **bold** or *italic*.

- List element 1
- List element 2

`inline quote`

```julia
# code block
using Plots
plot(sin)
```

[Test link](https://epfl.ch)

````
md" # Title
## Subtitle

Text can be put in **bold** or *italic*.

- List element 1
- List element 2

`inline quote`

```julia
# code block
using Plots
plot(sin)
```

[Test link](https://epfl.ch)
"
````

Exercises

In the notebooks, there will be many interactive exercises, denoted by a green box such as this:

Exercise

This is what an exercise looks like!

Many exercises are interactive:

- The statement is in a green box.
- There is a cell below where you have to complete the code.
- There is an immediate feedback box below.

Here is a first exercise to get you started:

Exercise

Change the following line to `i_am_ready = true` then run the cell.

```
i_am_ready = false
```

```
1  i_am_ready = false
```

Almost there!

This is an interactive feedback cell.

You have not completed this exercise yet. Change the line above to `i_am_ready = true` then run the cell.

Julia Tutorial

Now on to the Julia language itself. The rest of this notebook demonstrates all the basic Julia functionalities we will need for the lecture. It is written in an interactive and conversational way, the goal being that by the time you have read and run the provided commands you will have a basic working understanding of how to use Julia.

Please read and run all the cells below. Feel free to experiment by modifying the content of the cells.

Expressions

Julia supports common math operations:

```
5
```

```julia
2 + 3
```

```
-1
```

```julia
2 - 3
```

```
6
```

```julia
2 * 3
```

```
0.6666666666666666
```

```julia
2 / 3
```

```
8
```

```julia
2 ^ 3
```

By default Julia displays the output of the last operation. (You can suppress the output by adding `;` (a semicolon) at the end.)

Some more complex examples:

```
14
```

```julia
2 + 3 * 4
```

```
20
```

```julia
(2 + 3) * 4
```

```
302
```

```
2 + 3 * 10 ^ 2
```

```
302
```

```
2 + 3 * 10^2
```

Variables

We can define a variable using `=` (assignment). Then we can use its value in other expressions:

```
x = 3
```

```
x = 3
```

```
y = 6
```

```
y = 2 * x
```

Pluto is **reactive**, meaning that when a variable changes, all other variables depending on it are automatically updated.

Exercise

Try modifying `z` in the below cell and run it. See how the value of `w` in the cell below is automagically updated.

```
z = 10
```

```
z = 10
```

```
w = 20
```

```
w = 2 * z
```

A note on multiple expressions per cell

In Pluto, multiple expressions per cell are not allowed, as show by the error below:

Error message

```
Multiple expressions in one cell.
How would you like to fix it?

  • Split this cell into 2 cells, or
  • Wrap all code in a begin ... end block.
```

```
var1 = 2
var2 = 3
```

One solution is to split the code across multiple cells:

```
var1 = 2
```

```
var2 = 3
```

Another solution is to put the code inside a `begin` ... `end` block:

```
begin
    var1 = 2
    var2 = 3
end
# var1 and var2 are available in other cells
```

Either way, the variables `a` and `b` will be available in future cells.

You might also encounter `let` ... `end` blocks, which do not make the variables available to future cells:

```
let
    var1 = 2
    var2 = 3
end
# var1 and var2 are NOT available outside of let...end
```

Exercise

In the following cell, assign `2` to the `var1` variable and `3` to the `var2` variable

```
# Set var1 to 2 and var2 to 3
```

Oopsie!

Make sure that you define a variable called `var1`

Types

In Julia, every value has a type that determines what can be done with it. We can find the type of values using `typeof`:

```
Int64
```

```
typeof(10)
```

```
Float64
```

```
typeof(10.0)
```

```
Float64
```

```
typeof(2.5)
```

We can also ask for the type of a variable:

```
Int64
```

```
typeof(y)
```

`Int64` means that this variable contains a signed 64-bit integer. In practice, this means that it is number without any decimal part.

`Float64` means that this variable contains a 64-bit floating point number. In practice, this means a number that can have a decimal part.

As you learn Julia, you will discover more and more types. Sometimes, it can get quite complicated:

```
@NamedTuple{x::Vector{Vector{Vector{Vector{Vector{Int64}}}}}}
```

```
typeof((; x=[[[[[1]]]]]))
```

Strings

Another common type is `String`, which is used for text:

```
String
```

```julia
typeof("Some text")
```

Variables can be inserted in a string using `$`:

```
"variable is 10"
```

```julia
let
    variable = 10
    "variable is $variable"
end
```

To add a `$` to a string, escape it with `\\` as follows:

```julia
println("0.02 \$")
```

```
0.02 $
```

To combine strings, use the `*` operator:

```
"part 1 part 2 part 3"
```

```julia
"part 1 " * "part 2 " * "part 3"
```

Symbols

You may occasionally encounter symbols. They are typically used to represent identifiers and start with `:`.

```
Symbol
```

```julia
typeof(:name)
```

Keep in mind they are different from strings:

```
false
```

```julia
:name == "name"
```

Functions

Julia comes with many built-in functions, for example `exp(x)` to compute e^x, `cos(x)` to compute $\cos(x)$, and many others...

Typing the function's name gives some basic information about the function:

```
exp (generic function with 14 methods)
```

```julia
exp
```

```
cos (generic function with 19 methods)
```

```julia
cos
```

To call a function we must use parentheses:

```
1.0
```

```julia
exp(0)
```

```
2.718281828459045
```

```julia
exp(1)
```

```
1.0
```

```julia
cos(0)
```

```
-1.0
```

```julia
cos(π)
```

Creating new functions

Of course, we can also write our own functions. For simple functions, we can use a short-form, one-line function definition:

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

```julia
f(x) = 2 + x
```

As before, typing the function's name gives some basic information about the function.

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

```julia
f
```

To call it we must use parentheses:

12

```julia
f(10)
```

For longer functions we use the following syntax with the `function` keyword and `end`:

g (generic function with 1 method)

```julia
function g(x, y)
    z = x + y
    return z^2
end
```

9

```julia
g(1, 2)
```

Note that the final `return` is not necessary. In functions, the last expression is automatically returned:

h (generic function with 1 method)

```julia
function h(x, y)
    z = x + y
    z ^ 2
end
```

9

```julia
h(1, 2)
```

Exercise

Complete the function `line(a, b, x)` below, which should return $ax + b$.

line (generic function with 1 method)

```julia
function line(a, b, x)
    nothing
end
```

Here we go!

Replace `nothing` with your answer.

Conditionals: `if`, `elseif`, `else`

We can evaluate whether a condition is true or not by using comparison operators:

- `<`: smaller than
- `<=`: smaller or equal
- `>`: greater than
- `>=`: greater or equal
- `==`: equal
- `!=`: not equal

```
a = 3
```

```julia
a = 3
```

```
true
```

```julia
a < 5
```

```
true
```

```julia
a != 5
```

```
false
```

```julia
a >= 10
```

```
Bool
```

```julia
typeof(a >= 10)
```

We see that conditions have a boolean (`true` or `false`) value. The corresponding Julia type is `Bool`.

We can then use `if` to control what we do based on that value:

```
"small"
```

```julia
if a < 5
    "small"
else
    "big"
end
```

Note that the `if` also returns the last value that was evaluated, in this case the string `"small"` or `"big"`. Since Pluto is reactive, changing the definition of `a` above will automatically cause this to be reevaluated!

Intermediate checks can be added using `elseif`:

```
"1 to 9"
```

```
if a < 0
    "negative"
elseif a == 0
    "zero"
elseif a < 10
    "1 to 9"
else
    "10 or larger"
end
```

Conditionals: logical operators

Comparisons can be combined using logical operators:

- `a && b`: checks that `a` **and** `b` are `true`.
- `a || b`: checks that at least `a` **or** `b` is `true`.
- `!a`: checks that `a` is **not** `true`.

For example:

```
true
```

```
1 < 2 && 2 < 3
```

```
true
```

```
1 < 2 || 2 < 1
```

```
false
```

```
1 < 2 && 2 < 1
```

```
true
```

```
!(1 == 2)
```

Exercise

Complete the function `is_leap_year(year)` below. The function should return `true` if `year` is a leap year, and `false` if it is not. Here is a reminder of the algorithm:

- Every year divisible by 4 is a leap year, except that:
- Every year divisible by 100 **is not** a leap year, except that:
- Every year divisible by 400 **is** a leap year.

To check if the year is divisible by some number `n`, check that `year % n == 0`. (`%` is called the modulo operator).

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

```julia
function is_leap_year(year)
    nothing
end
```

Here we go!

Replace `nothing` with your answer.

For loops

Use `for` to loop through a pre-determined set of values:

```
55
```

```julia
let
    s = 0

    for i in 1:10
        s += i    # Equivalent to s = s + i
    end

    s
end
```

Here, `1:10` is a **range** representing the numbers from 1 to 10:

```
UnitRange{Int64}
```

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

Above we used a `let` block to define a new local variable `s`. But blocks of code like this are usually better inside functions, so that they can be reused. For example, we could rewrite the above as follows:

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

```julia
function mysum(n)
    s = 0

    for i in 1:n
        s += i
    end

    return s
end
```

```
5050
```

```julia
mysum(100)
```

`for` loops work over many other things than ranges. Here is an example of looping over a vector (see below). `$element` includes the value of the `element` variable in a string.

```julia
for element in [1, 10, 100, 1000]
    println("Looping over $element")
end
```

```
Looping over 1
Looping over 10
Looping over 100
Looping over 1000
```

Exercise: Fibonacci

Before we move on, let us implement a classic exercise.

Exercise

Complete the below cell by writing a function `fibonacci` that accepts an integer n and returns the n-th term of the Fibonacci sequence as an integer, that is the sequence F_n with

$$
\begin{aligned}
F_0 &= 0 \\
F_1 &= 1 \\
F_2 &= 1 \\
F_3 &= 2 \\
&\ldots \\
F_n &= F_{n-1} + F_{n-2}
\end{aligned}
$$

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

```julia
function fibonacci(n)
    nothing
end
```

Here we go!

Replace `nothing` with your answer.

Arrays

An array is a collection of elements. They can be one-dimensional, corresponding to a list or vector. They can be two-dimensional, corresponding to a grid of numbers or matrix. They can have more dimensions.

Arrays have the type `Array{T, N}` where:

- `T` is the type of element inside the array.
- `N` is the number of dimensions in the array: `1` for a vector, `2` for a matrix, etc...

`Vector{T}` is an alias for `Array{T, 1}` and `Matrix{T}` is an alias for `Array{T, 1}`.

Tip

If you are already familiar with Python or Matlab, you should check out this comparative cheatsheet already linked in the introduction, which covers a lot of Julia's array syntax: Comparative cheatsheet Python <-> Julia <-> Matlab.

1D arrays (`Vector` s)

We can make a `Vector` (1-dimensional, or 1D array) using square brackets:

```
v = ▸[1, 2, 3]
```

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

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

```julia
typeof(v)
```

The type tells us that this is a 1D array of integers. **Don't forget the commas (,) between the** elements, or you will create a matrix instead:

```
1×3 Matrix{Int64}:
 1  2  3
```

```julia
[1 2 3]
```

We access elements using square brackets:

```
1
```

```julia
v[1]
```

```
2
```

```julia
v[2]
```

The special syntax **`end`** can be used to refer to the end of the array. For example, `v[end]` will access the last element, and `v[end-1]` the one before.

```
2
```

```julia
v[end-1]
```

In Julia, arrays start at `1`. Accessing an array at index `0` will cause an error:

Error message

```
BoundsError: attempt to access 3-element Vector{Int64} at index [0]
```

Show stack trace...

```
v[0]
```

Arrays can be modified:

```
10
```

```
v[2] = 10
```

However types matter! We cannot store a decimal number in an array of `Int64`:

Error message

```
InexactError: Int64(2.5)
```

Show stack trace...

```
v[2] = 2.5
```

Note that Pluto does not automatically update cells when you modify elements of an array, but the value does change.

A nice way to create `Vector`s following a certain pattern is to use an **array comprehension**:

```
v2 = ▶[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
v2 = [i^2 for i in 1:10]
```

Element-wise array operations

Arrays can be added together, or subtracted, using the regular `+` and `-` operators.

```
vec1 = ▶[1, 2, 3]
```

```
1 vec1 = [1, 2, 3]
```

```
vec2 = ▶[1, 4, 9]
```

```
1 vec2 = [1, 4, 9]
```

```
▶[2, 6, 12]
```

```
1 vec1 + vec2
```

```
▶[0, -2, -6]
```

```
1 vec1 - vec2
```

This adds and substracts corresponding elements of each array. We call this an **element-wise** operation.

In general, other mathematical operator or functions will not work on arrays. To apply operators or functions to an array **element-wise**, we use special syntax:

- Adding `.` before an operator will apply the operator element-wise. For example, `.*` and `./` will perform element-wise multiplication.
- Adding `.` after a function will apply the function element-wise. For example, `cos.` will compute the cosine element-wise.

```
▶[1, 8, 27]
```

```
1 vec1 .* vec2
```

```
▶[1.0, 0.5, 0.333333]
```

```
1 vec1 ./ vec2
```

```
▶[1, 4, 9]
```

```
1 vec1 .^ 2
```

```
true
```

```
1 vec1.^2 == vec2
```

```
[2.71828, 7.38906, 20.0855]
```

```
exp.(vec1)
```

```
[1.0, -1.0, 1.0]
```

```
cos.([0, π, 2π])
```

Let's put this into practice with the Fibonacci function you wrote above:

Exercise

Complete the following function that will compute multiple Fibonacci numbers at once. It will receive a vector of integers, and should return a vector with the corresponding Fibonacci numbers. For example, `many_fibonacci([1, 2, 4])` should return `[1, 1, 3]`.

Your answer should be very short and use the `fibonacci` function that you wrote above.

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

```
many_fibonacci(ns) = fibonacci(0)
```

Here we go!

First, implement the `fibonacci` function in the exercise a few sections above.

2D arrays (matrices)

We can make small matrices (2D arrays) with square brackets too:

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

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

```
Matrix{Int64} (alias for Array{Int64, 2})
```

```
typeof(M)
```

The `2` in the type confirms that this is a 2D array.

This won't work so easily for larger matrices, though. For that we can use e.g.

```
5×5 Matrix{Float64}:
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0
```

```
zeros(5, 5)
```

Note that `zeros` gives `Float64`s by default. We can also specify a type for the elements:

```
4×5 Matrix{Int64}:
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
```

```
zeros(Int, 4, 5)
```

Same as the arrays themselves. E.g. contrast

```
▸[1.0, 2.0, 3.0]
```

```
Float64[1, 2, 3]
```

which creates a `Float64` array versus

```
▸[1, 2, 3]
```

```
Int[1, 2, 3]  # or just [1, 2, 3]
```

which creates an integer array. We can then fill in the values we want by manipulating the elements, e.g. with a `for` loop.

A nice alternative syntax to create matrices following a certain pattern is an array comprehension with a *double* `for` loop:

```
5×6 Matrix{Int64}:
 2  3  4  5   6   7
 3  4  5  6   7   8
 4  5  6  7   8   9
 5  6  7  8   9  10
 6  7  8  9  10  11
```

```
[i + j for i in 1:5, j in 1:6]
```

To access matrix elements directly, we need to use two indices:

```
2
```

```julia
M[1, 2]
```

```
4
```

```julia
M[2, 2]
```

Element-wise vs matrix operations

Element-wise operations also work on matrices, using the `.` syntax explained above:

```
2×2 Matrix{Float64}:
  2.71828   7.38906
 20.0855   54.5982
```

```julia
exp.(M)
```

Many functions can be applied to the whole square matrix, giving a different result than applying the function element-wise.

```
2×2 Matrix{Float64}:
  51.969   74.7366
 112.105  164.074
```

```julia
exp(M)
```

Oops, we computed the matrix exponential instead of the element-wise exponential.

As another example, **matrix multiplication** is performed with `*` whereas element-wise multiplication is performed with `.*`:

```
2×2 Matrix{Int64}:
  7  10
 15  22
```

```julia
M * M
```

```
2×2 Matrix{Int64}:
 1   4
 9  16
```

```julia
M .* M
```

As a final example, let's revisit two variations of the famous $\cos(x)^2 + \sin(x)^2 = 1$ equation:

```
2×2 Matrix{Float64}:
  1.0          -1.94289e-16
 -2.77556e-16   1.0
```

```julia
cos(M)^2 + sin(M)^2
```

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

```
cos.(M).^2 + sin.(M).^2
```

Element-wise, we indeed get `1` for every entry. For the whole matrix, we get an identity matrix which is indeed the `1` of 2x2 matrices.

Note: `-1.66533e-16` means -1.66533×10^{-16} which is a very small number. It happens due to the imprecision of floating-point arithmetic. It would be `0` if computer math were exact.

The End

This concludes the tutorial section of this first notebook.

Feel free to keep reading for a look at a more advanced example, or skip the following section and come back to it later. In any case, a second notebook awaits you on Moodle with an introduction to plotting in Julia.

Optional: Step-by-step replication of the Arrhenius fit from the lecture

(This part is intented as a deeper dive into Julia's capabilities.)

We here reproduce in full detail the Arrhenius fit example from the lecture.

- We were given the data:

```
data = """
# Temperature(K)  Rate(1/s)
  250.0           1.65657
  260.0           1.70327
  270.0           1.74472
  280.0           1.78110
  290.0           1.81259
  300.0           1.83940
  310.0           1.86171
  320.0           1.87971
  330.0           1.89358
  340.0           1.90352
  350.0           1.90968
""";

```

This data is in plain text form, so we need to preprocess it a little in order to be able to plot it graphically.

First we split the data into lines:

```
lines =
▸["# Temperature(K)  Rate(1/s)", "  250.0           1.65657", "  260.0           1.70327"
```

```
lines = split(data, "\n")
```

The head line

```
"# Temperature(K)  Rate(1/s)"
```

```
lines[1]
```

is not needed, and similarly the last line

```
""
```

```
lines[end]
```

is empty, so we strip them using Julia's array masks:

```
lines_relevant =
▸["  250.0            1.65657", "  260.0            1.70327", "  270.0            1.74472",
```

```
lines_relevant = lines[2:end-1]
```

Repeating the spliting on the lines, we can obtain the temperature and rate data in separate arrays:

```
temperature_string =
▸["250.0", "260.0", "270.0", "280.0", "290.0", "300.0", "310.0", "320.0", "330.0", "340.0",
```

```
temperature_string = [split(line)[1] for line in lines_relevant]
```

```
rate_string =
▸["1.65657", "1.70327", "1.74472", "1.78110", "1.81259", "1.83940", "1.86171", "1.87971", '
```

```
rate_string = [split(line)[2] for line in lines_relevant]
```

To get floating-point numbers out of these strings, we `parse` them:

```
▸[250.0, 260.0, 270.0, 280.0, 290.0, 300.0, 310.0, 320.0, 330.0, 340.0, 350.0]
```

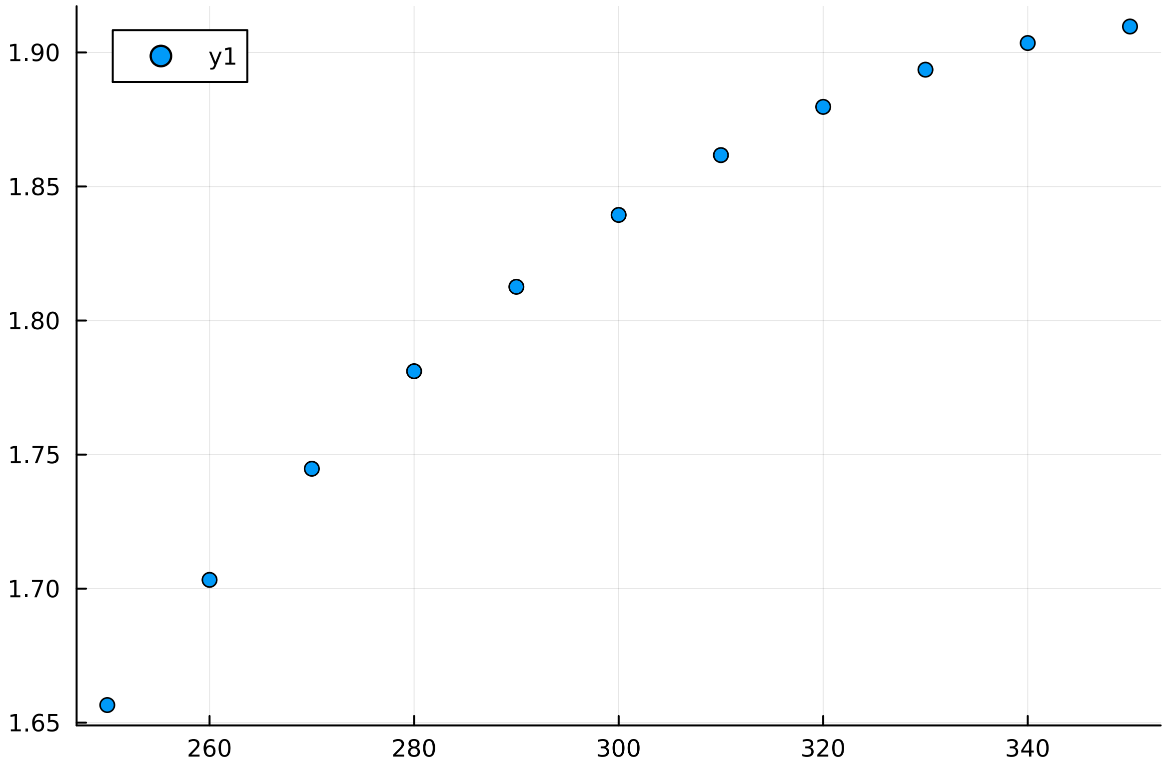
```
[parse(Float64, string) for string in temperature_string]
```

More compactly we could have written this as

```
begin
    temperature = [parse(Float64, split(line)[1]) for line in lines[2:end-1]]
    rate        = [parse(Float64, split(line)[2]) for line in lines[2:end-1]]
end;
```

where `begin ... end` allows to combine multiple statements in one cell.

Finally we plot:



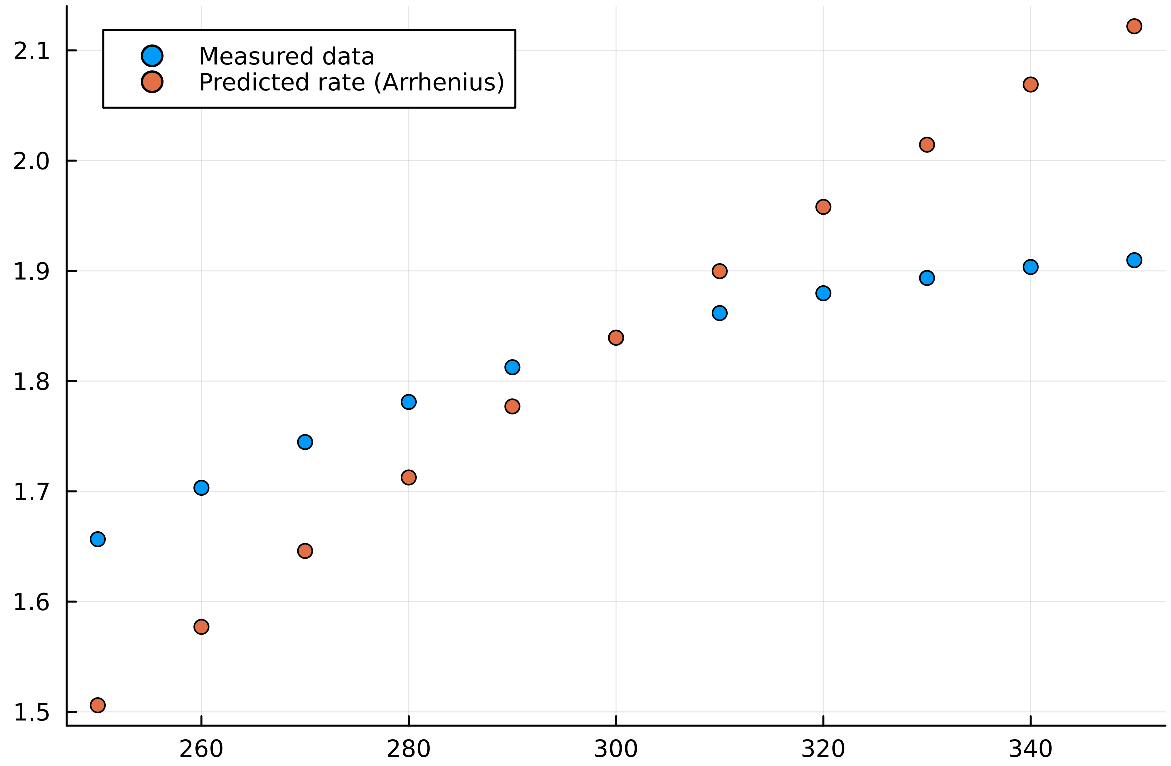

```
scatter(temperature, rate)
```

The best fit Arrhenius equation is given by the function

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

```
function arrhenius(T)
    5exp(-300 / T)
end
```

Let's first investigate which values this function would take at these points and plot it in the same graph:

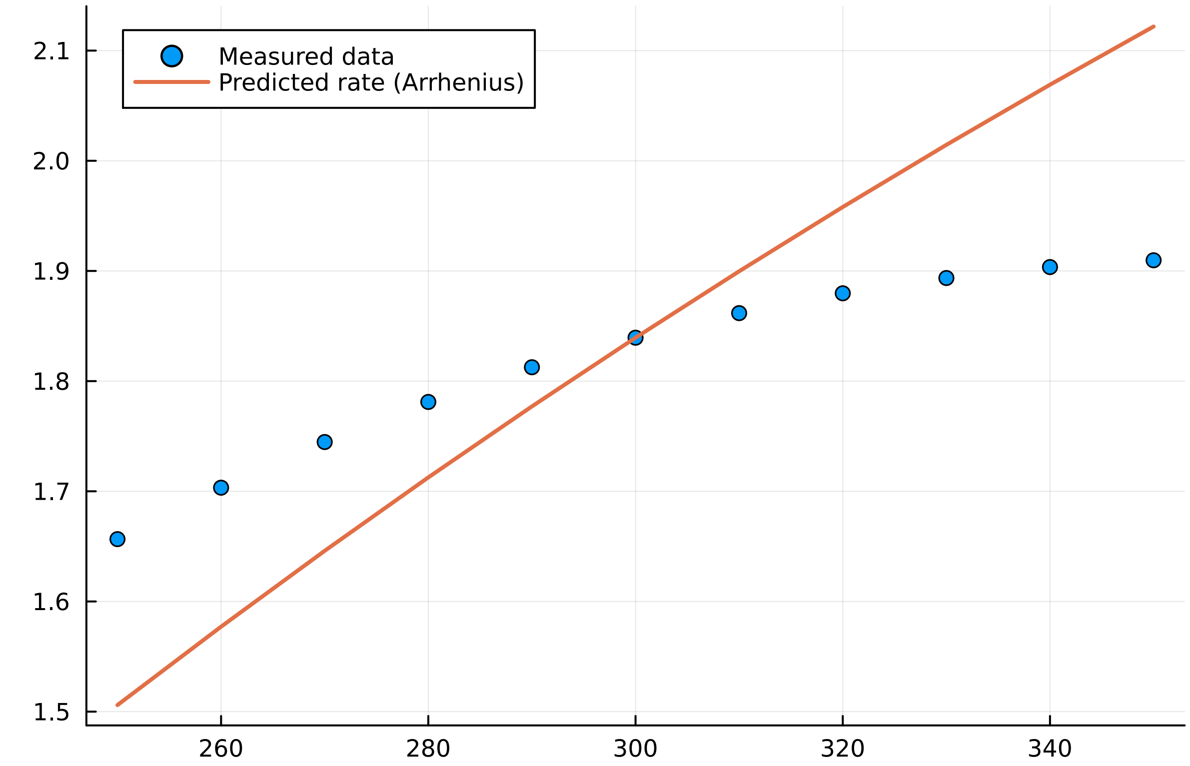



```
begin
    predicted_rates = [arrhenius(t) for t in temperature]

    plt1 = scatter(temperature, rate, label="Measured data")
    scatter!(plt1, temperature, predicted_rates, label="Predicted rate (Arrhenius)")
    plt1
end
```

Note that here the first `scatter` call returns an object, which represents the plotting canvas. Using a second `scatter!` call we can add other entities for plotting to it.

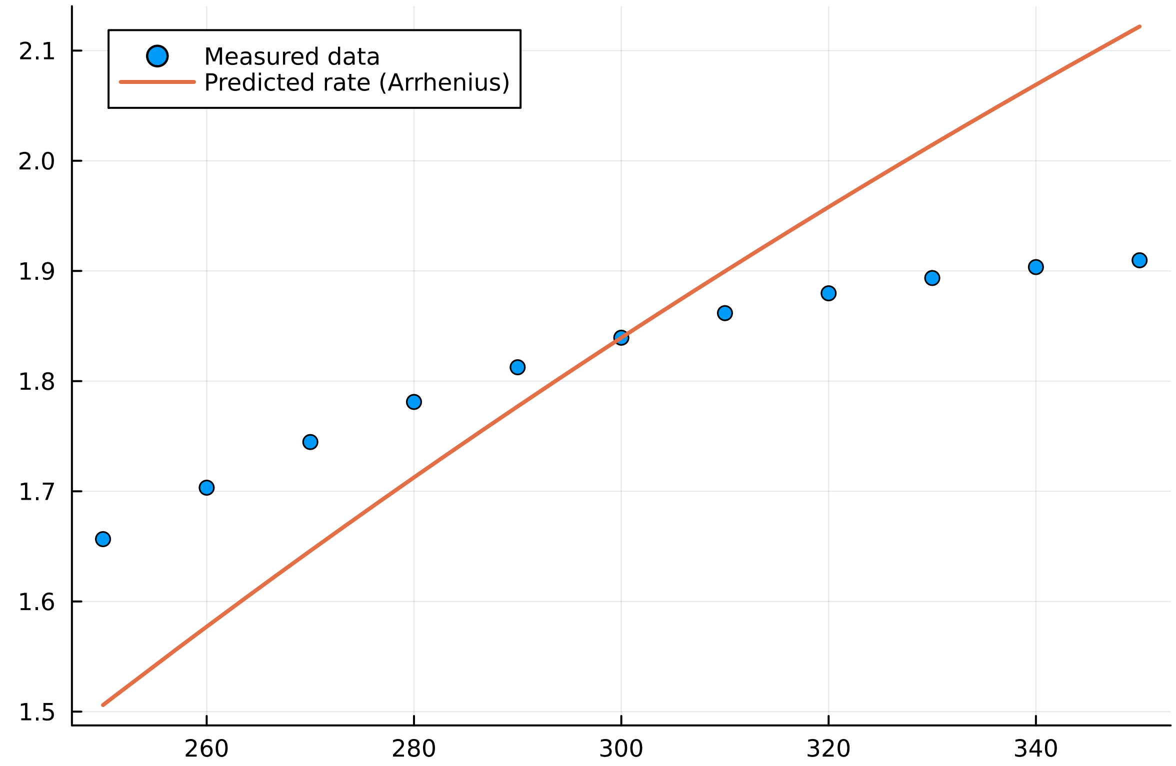
If we want to plot a continuous graph instead of data points, we can use `plot` or `plot!`, e.g.:




```
begin
    # Note: predicted_rates already defined above

    plt2 = scatter(temperature, rate, label="Measured data")
    plot!(plt2, temperature, predicted_rates, label="Predicted rate (Arrhenius)",
          linewidth=2)
    plt2
end
```

For convenience, functions like `arrhenius` can also be plotted directly, without evaluating them explicitly:




```
begin
    plt3 = scatter(temperature, rate, label="Measured data")
    plot!(plt3, arrhenius, label="Predicted rate (Arrhenius)",
          linewidth=2)
    plt3
end
```

Note, that also a similar `let ... end` block exists to combine multiple statements. The point of `let` is to *hide* variable names and content from the outside the cell. This is needed because in Pluto notebooks each variable name may only be used a single time in the public context, e.g. defining `a` twice leads to a warning and the disabling of one cell.

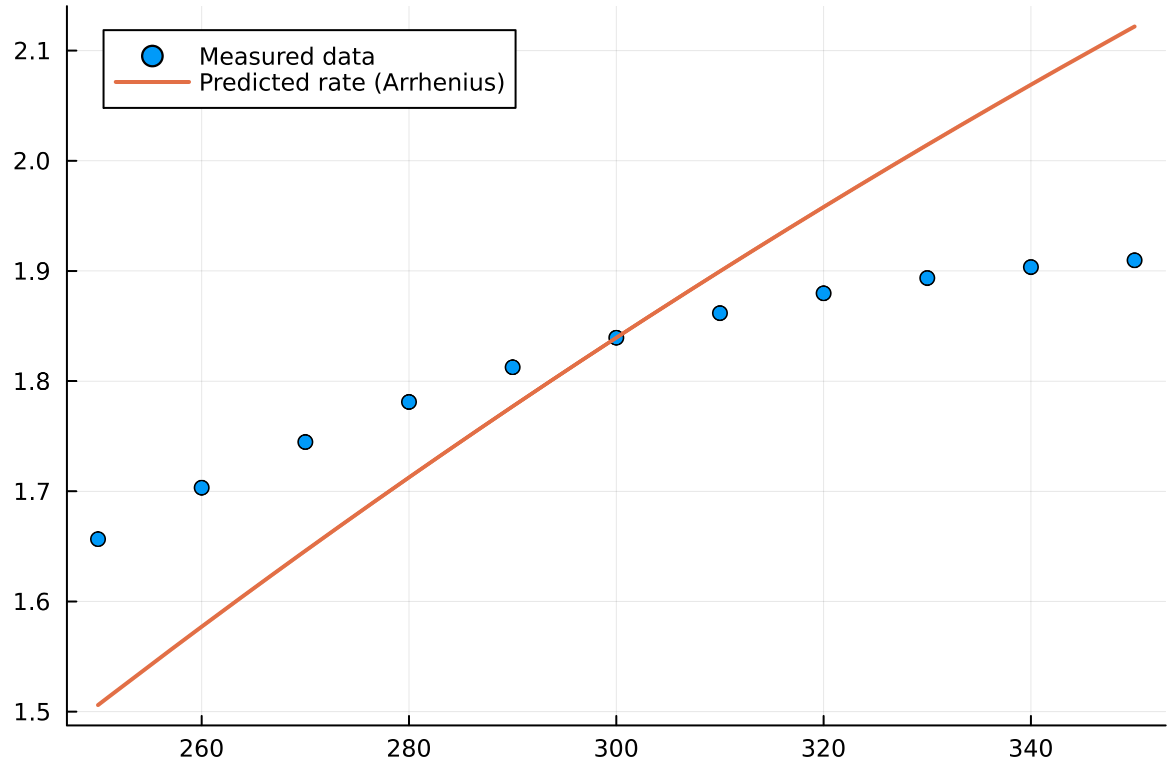
```
variable = 4
```

```
variable = 5
```

```
variable = 5
```

This is a safety feature to avoid overwriting computational results.

Sometimes (especially for setting up plots), we have no interest in using the data outside of the cell anyway. In this case using `let` ... `end` is usually better:




```julia
let
    p = scatter(temperature, rate, label="Measured data")
    plot!(p, arrhenius, label="Predicted rate (Arrhenius)",
          linewidth=2)
    p
end
```