tut1_introduction

February 14, 2022

1 Macro 318: Tutorial #1

1.1 Introduction to Programming with Julia

Lecturer: Dawie van Lill (dvanlill@sun.ac.za)

1.2 Introduction

This is the first of four tutorials that are focused on basic computational methods in macroeconomics. These tutorials will provide some basic programming knowledge with application in macroeconomics. The tutorials are quite long and will take some time to work through. However, they are in notebook format, which means that they are interactive.

My advice is to work on the notebook for 30 minutes at a time. Try not to let the amount of information overwhelm you. This is a long tutorial that contains a lots of bit and pieces on programming and the Julia language.

As I mentioned, the first tutorial is supposed to show you the basics of the Julia programming language.

We chose the Julia programming language because it is easy to learn and also very fast!

There are other excellent languages, such as **Python**, R and Matlab, which are useful for economists. However, Julia will work for our purposes. Many of the lessons you learn here can be applied to other languages as well, especially Python.

For those of you that are comfortable with Python, here is a resource that compares much of the functionality between Python and Julia.

This notebook is meant to be used as a **reference**. In other words, I am not expecting you to remember everything in the notebook. The notebook introduces you to several concepts that you might need in the future. While this is a reference, it only covers the very basic components of the language. Some other useful references that I used in constructing these notes are

1. QuantEcon notes
2. Paul Soderlind notes
3. Cameron Pfiffer notes
4. Czech Technical University notes

You can ask me for more resources if you need it.

A good textbook that teaches both Julia and programming is Think Julia, which can be found here

---

1.3 Planned teaching outcomes

In this course we hope to look at different themes in computational macroeconomics. Most of modern macroeconomic research involves utilising computational methods, so it is wortwhile to get comfortable with some basic programming skills. The return on this investment is quite high. If you are interested in this type of work, feel free to contact me to talk about computational economics in Honours and Masters.

Note: We currently don't offer modules at postgraduate level that explicitly teach computational methods. However, I am offering weekly sessions that cover the basics of programming and computation. I will create a class list where people can sign up for the sessions. These sessions are aimed at Honours and Masters students, but you are welcome to attend to see what all the hype is about!

Here are some of the focus areas

1. Fundamentals of programming
2. Optimisation and the consumer problem
3. Solow model
4. Simulation + data work

1.4 Tutorial workflow

The tutorials will be pre-recorded so that you can go through the material on your own time before the live tutorial session. During the live tutorial session I will be answering questions and also going through some exercises. These will be both normal pen and pencil exercises and computer based exercises. All the exercises in the tutorial will be unseen, which means that will need to solve them in class.

This obviously then means that you need to go through the notes before the tutorial in order to solve the problems in class.

For the best experience with this module it is strongly advised that students go through the tutorial material before the tutorials!

1.5 Running the notebooks

The current notes are put together in a Jupyter notebook.

For the purpose of this course you will run the notebooks online (this doesn't require you to install Python, Julia or Jupyter on your system) using Binder.

You can also run these files locally (on your computer) without the need for the internet, but that requires an installation procedure that will be too difficult to explain to 250+ students.

I will, however, make a brief video on how to install Julia, Python, Jupyter and VS Code for those that are interested. This is installation procedure is optional. If you want a step by step guide on how to install most of these components, you can look here

1.6 Topics for the tutorial

Below are the broad topics for the tutorial

1. Variables
2. Data structures
3. Control flow
4. Functions
5. Visualisation
6. Type system and generic programming

Most of these topics are covered in introductory undergraduate computer science courses, so if you are a CS student then you can quickly skim for the Julia syntax. If you are comfortable with a modern programming language like Python then everything here will feel somewhat familiar.

---

2 Your first code!

There are many great guides to that provide an introduction to Julia and programming. The QuantEcon website is an exceptional resource that can be used to teach yourself about computational economics. We will be borrowing some the material from their website in this section. The main difference with these notes is that they are not nearly as comprehensive and are only meant as a starting point for your journey.

Before we start our discussion, let us try and run our first Julia program. For those that have done programming before, this normally entails writing a piece of code that gives us the output `Hello World!`. In Julia this is super easy to do.

[1]:
```
println("Hello World!")
```

```
Hello World!
```

2.1 Introduction to packages

Julia has many useful packages. If we want to include a specific package then we can do the following,

```
import Pkg

Pkg.add("PackageName")

using PackageName
```

[2]:
```
import Pkg
```

Here are some of the packages that we are going to use in this tutorial

[3]:
```
Pkg.add("DataFrames") # Package for working with data
Pkg.add("GLM") # Required for linear regression
```

```
Pkg.add("LinearAlgebra") # Required for linear algebra applications
Pkg.add("Plots")# This is required for plotting
Pkg.add("TypeTree")
```

```
    Updating registry at `~/.julia/registries/General.toml`
   Resolving package versions…
  No Changes to `~/.julia/environments/v1.7/Project.toml`
  No Changes to `~/.julia/environments/v1.7/Manifest.toml`
   Resolving package versions…
  No Changes to `~/.julia/environments/v1.7/Project.toml`
  No Changes to `~/.julia/environments/v1.7/Manifest.toml`
   Resolving package versions…
  No Changes to `~/.julia/environments/v1.7/Project.toml`
  No Changes to `~/.julia/environments/v1.7/Manifest.toml`
   Resolving package versions…
  No Changes to `~/.julia/environments/v1.7/Project.toml`
  No Changes to `~/.julia/environments/v1.7/Manifest.toml`
   Resolving package versions…
  No Changes to `~/.julia/environments/v1.7/Project.toml`
  No Changes to `~/.julia/environments/v1.7/Manifest.toml`
```

The `using` command lets the computer know that you are going to be using the following installed packages in your session. This is similar to `import` in Python or `library()` in R.

[5]:
```
using Base: show_supertypes
using DataFrames
using GLM
using LinearAlgebra
using Plots
using TypeTree
```

We can now use basic functions from these packages. In the following section you will see how we use specific functions from the packages that we have installed.

---

3 Variables and types

After having successfully written your `Hello World!` code in Julia, a natural place to continue your journey is with variables. A variable in a programming language is going to be some sort of symbol that we assign some value. Let us use a concrete example to illustrate.

[6]:
```
x = 2 # assign the value of 2 to the variable x
```

[6]: 2

The `type` of this variable is inferred automatically and we can check what this type is going to be by using the `typeof()` function.

[7]:
```
typeof(x)
```

[7]:
```
Int64
```

It seems that the variable is of type `Int64`, which is a `Signed Integer`.

How do we know this? Well we need to look at the type tree. A type tree gives us information about the different types and their relation to each other.

If we look at the type tree in the code below, we see that `Int64` falls within the `Signed Integer` class.

`Int64` is a **subtype** of the broader `Integer` type, as can be seen below.

Note If this type information seems strange to you right now, don't worry. We will repeatedly talk about types throughout the tutorial. You can always come back and read this section again later to make sure you understand.

[8]:
```
print(join(tt(Integer), "")) # print the subtypes of the Integer type.
```

```
Integer
   Bool
   GeometryBasics.OffsetInteger
   Signed
       BigInt
       Int128
       Int16
       Int32
       Int64
       Int8
   Unsigned
        UInt128
        UInt16
        UInt32
        UInt64
        UInt8
```

We can also see the super types of the `Int64` type. These are all the abstract types above it in the type hierarchy.

[9]:
```
show_supertypes(Int64)
```

```
Int64 <: Signed <: Integer <: Real <: Number <: Any
```

The value `2` is now bound to `x` in the computer's memory. We can now work with `x` as if it represents the value of `2`.

Since an integer is a number, we can perform basic mathematical operations.

[10]:
```
y = x + 2
```

[10]: 4

When we check the type of the new variable `y`, it should also return an integer. The reason for this is that an integer plus an integer always gives back an integer.

[11]:
```
typeof(y)
```

[11]: Int64

We can reassign the variable `x` to another value, even with another type.

[12]:
```
x = 3.1345
```

[12]: 3.1345

[13]:
```
typeof(x)
```

[13]: Float64

Now `x` is a floating point number. As an exercise, create one or two new variables with different types and then check what type they are. I'll provide another example.

[14]:
```
z = 1//2
```

[14]: 1//2

[15]:
```
typeof(z)
```

[15]: Rational{Int64}

3.0.1 Variable names (quick detour)

Generally when we name variables they will start with letters and in Julia the preference is to use snake case. This means variables are written in lower case, with underscores used if the word becomes unreadable. As an example `new_variable` instead of `NewVariable`. The latter is an example of camel case (which will be used for modules and types later in the notes).

In many Julia editing environments you will be able to use math (and other types) of symbols as variable names.

[16]:
```
 = 1; # this is delta
```

There are also some reserved words within the language that you cannot use to name a variable. Words like `if`, `begin`, and so forth. The list of reserve words is not long, so don't be overly concerned about this. If you do use a reserve word, there will most likely be an error.

If you want to know more about the style that you need to use when coding, you can look at the official style guide or the Invenia style guide. However, I recommend only doing that after you have completed the tutorial.

3.1 Data types

We have mentioned some data types in the previous discussion on variables. However, we didn't really explain things well.

One of the best places to start with a discussion on a programming language is with data types. This might sound incredibly dull, but it is actually quite important to understand the different data types that can be used in a programming language and the functions that can be implemented on them.

3.1.1 Primitive data types

There are several important data types that are at the core of computing. Some of these include,

- **Booleans**: true and false
- **Integers**: -3, -2, -1, 0, 1, 2, 3, etc.
- **Floating point numbers**: 3.14, 2.95, 1.0, etc.
- **Strings**: "abc", "cat", "hello there"
- **Characters**: 'f', 'c', 'u'

We start our discussion with Booleans, these are `true` / `false` values.

[17]:
```
a = true
```

[17]:
```
true
```

[18]:
```
typeof(a)
```

[18]:
```
Bool
```

Once againg, the `typeof()` operator tells us what the data type a variable takes. In the case above we have a `Boolean` value. Now let's look at some other well known data types.

Before you run the cell, make sure that you have an idea of what to expect with respect to the input type.

[19]:
```
typeof(122)
```

[19]:
```
Int64
```

[20]:
```
typeof(122.0)
```

[20]:
```
Float64
```

Numbers are represented as floats and integers. Floating point numbers (floats) represent the way in which computers manifest real numbers.

With numbers in mind, we can treat the computer like a calculator. We can perform basic arithmetic operations. Operators perform operations. These common operators are called the **arithmetic operators**.

Expression	Name	Description
`x + y`	binary plus	performs addition
`x - y`	binary minus	performs subtraction
`x * y`	times	performs multiplication
`x / y`	divide	performs division
`x ÷ y`	integer divide	`x / y`, truncated to an integer
`x \ y`	inverse divide	equivalent to `y / x`
`x ^ y`	power	raises `x` to the `y`th power
`x % y`	remainder	equivalent to `rem(x,y)`

Here are some simple examples that utilise these arithmetic operators.

[21]:
```
x = 2; y = 10;
```

The semicolon `;` is used at the end of the expression to suppress the output (similar to Matlab). This mean that it won't show the result from the code that you run.

[22]:
```
x * y
```

[22]: 20

[23]:
```
x ^ y
```

[23]: 1024

[24]:
```
y / x # note: division converts integers to floats
```

[24]: 5.0

[25]:
```
2x - 3y
```

[25]: -26

[26]:
```
x // y
```

[26]: 1//5

If we wanted to display both the text and result from some of the operations above, we could use the `@show` macro. Try it out by putting `@show` in front of your piece of code.

[27]:
```
@show 2x + 3y;
@show 22x * y;
```

```
2x + 3y = 34
(22x) * y = 440
```

Exercise #1 Take a few seconds to try and code up an answer to the following question. This will help you come to grips with the way in which arithmetic operations are defined in Julia.

Determine the value and type of `y` given by the following expression

$$y = \frac{(x+2)^2 - 4}{(x-2)^{p-2}},$$

where `x = 4` and `p = 5`.

Another important data type is the String (and Character)

[28]:
```
typeof("Hello Class!")
```

[28]:
```
String
```

[29]:
```
typeof('h')
```

[29]:
```
Char
```

Some of the operators used on floats and integers above can also be used on strings and characters.

[30]:
```
x = "abc"; y = "def"
```

[30]:
```
"def"
```

[31]:
```
x * y # concatenation
```

[31]:
```
"abcdef"
```

[32]:
```
x ^ 2 # multiples
```

[32]:
```
"abcabc"
```

Another important class of operators that we will often encounter are the **augmentation operators**. This will be especially important in the section on control flow.

[33]:
```
x = 3
```

[33]:
```
3
```

[34]:
```
x += 1 # same as x = x + 1
```

[34]:
```
4
```

[35]:
```
x *= 2 # same as x = x * 2
```

[35]:
```
8
```

[36]:
```
x /= 2 # same as x = x / 2
```

[36]:
```
4.0
```

Now that we have talked about the basic data types, it is important to talk about the different operators that are supported in Julia. We have already used some of the mathematical operators, such as multiplication and division when it came to floating point numbers. However, Julia has another set of common operators that work on Booleans.

[37]:
```
!true # negation operation
```

[37]:
```
false
```

[38]:
```
x = true; y = true
```

[38]:
```
true
```

[39]:
```
x && y # and operator. Returns true if x and y are both true, otherwise false
```

[39]:
```
true
```

[40]:
```
x || y # or operator. Returns true if x aor y is true, otherwise false
```

[40]:
```
true
```

Another important class of operators are the comparison operators. These help to generate true and false values for our conditional statements that we see later in the tutorial.

Operator	Name
`==`	equality
`!=`,	inequality
`<`	less than
`<=`,	less than or equal to
`>`	greater than
`>=`,	greater than or equal to

Below are some examples using comparison operators. The operators always return `true` or `false`.

[41]:
```
x = 3; y = 2;
```

[42]:
```
x < y
```

[42]:
```
false
```

[43]:
```
x <= y
```

[43]:
```
false
```

[44]:
```
x != y
```

[44]: true

[45]:
```
x == y
```

[45]: false

[46]:
```
z = x < y
```

[46]: false

3.1.2 Type conversion

It is possible to convert the type of one variable to another type. One could, as an example convert a float to string, or float to integer.

[47]:
```
x = 1
```

[47]: 1

[48]:
```
y = string(x)
```

[48]: "1"

[49]:
```
z = float(x)
```

[49]: 1.0

[50]:
```
y = string(z)
```

[50]: "1.0"

It is not possible to convert a string to an integer. So this means there are some limitations to conversion. The following bit of code uses exception handling, don't worry too much about it for now. We will talk about this some more in the section on control flow. Here is a link that goes into some more detail on the topic.

[51]:
```
try
    x = Int("22")
    println("This conversion is possible")
    println("The resulting value is $x")
catch
    println("This conversion is not possible")
end
```

```
This conversion is not possible
```

This raises the question as to whether a boolean variables can be converted to an integer? Write a bit of code, given what is provided above to try and check this. Here is some code to get started.

[52]:
```
try
    x = Int() # fill in the missing conversion here
    println("This conversion is possible")
    println("The resulting value is $x")
catch
    println("This conversion is not possible")
end
```

```
This conversion is not possible
```

---

4 Data Structures

There are several types of containers, such as arrays and tuples, in Julia. We explore some of the most commonly used containers here.

Note that we have both mutable and immutable types of containers in Julia. Mutable means that the value within that container can be changed, whereas immutable refers to the fact that values cannot be altered.

4.0.1 Tuples

Let us start with one of the basic types of containers, which are referred to as tuples. These containers are immutable (can't change the values in the container), ordered and of a fixed length. Consider the following example of a tuple,

[53]:
```
x = (10, 20, 30)
```

[53]:
```
(10, 20, 30)
```

We can access the first component of a tuple in the following way,

[54]:
```
x[1]
```

[54]:
```
10
```

We can generally unpack a tuple in the following fashion,

[55]:
```
a, b, c = x
```

[55]:
```
(10, 20, 30)
```

[56]:
```
a
```

[56]:
```
10
```

Named tuples (optional) It is also possible to provide names to each position of a named tuple. Here is an example of a named tuple.

[57]:
```
x = (first = 'a', second = 100, third = (1, 2, 3))
```

[57]:
```
(first = 'a', second = 100, third = (1, 2, 3))
```

[58]:
```
x.third # provides access to the third element in the tuple (which in this case␣
↪is another tuple)
```

[58]:
```
(1, 2, 3)
```

[59]:
```
x[:third] # equivalent to x.third
```

[59]:
```
(1, 2, 3)
```

You can merge named tuples to add new elements to the tuple. Let us say that you want to add the sentence "Programming is awesome!" to the named tuple, you can do it is as follows,

[60]:
```
y = (fourth = "Programming is awesome!", ) # note the comma at the end, if you␣
↪don't have it, then this won't work.
```

[60]:
```
(fourth = "Programming is awesome!",)
```

[61]:
```
merge(x, y)
```

[61]:
```
(first = 'a', second = 100, third = (1, 2, 3), fourth = "Programming is
awesome!")
```

4.0.2 Arrays

One of the most important containers in Julia are arrays.

You will use tuples and arrays quite frequently in your code.

An array is a multi-dimensional grid of values.

Vectors and matrices, such as those from mathematics, are types of arrays in Julia. Let us start with the idea of a vector. A vector is a one-dimensional array.

In Julia, we can create a vector by surrounding a value by brackets and separating the values with commas.

[62]:
```
vector_x = [1, "abc"] # example of a vector (one dimensional array)
```

[62]:
```
2-element Vector{Any}:
 1
  "abc"
```

[63]:
```
typeof(vector_x)
```

[63]:
```
Vector{Any} (alias for Array{Any, 1})
```

As with tuples, we can access the elements of the vector in the following way,

[64]:
```
vector_x[1]
```

[64]:
```
1
```

The big difference between a tuple and array is that we can change the values of the array. Below is an example where we change the first component of the array. This means that arrays are mutable.

[65]:
```
vector_x[1] = "def"
```

[65]:
```
"def"
```

[66]:
```
vector_x
```

[66]:
```
2-element Vector{Any}:
 "def"
 "abc"
```

We can use the `push!()` function to add values to this vector. This grows the size of the vector. You might notice the `!` operator after `push`. This exclamation mark doesn't do anything particularly special in Julia. It is a coding convention to let the user know that the input is going to be altered / changed. In our case it lets us know that the vector is going to be mutated. Let us illustrate.

[67]:
```
push!(vector_x, "hij") # vector_x is mutated here. It changes from 2 element␣
↪vector to 3 element vector.
```

[67]:
```
3-element Vector{Any}:
 "def"
 "abc"
 "hij"
```

We can also create matrices (two dimensional arrays) in the following way,

[68]:
```
matrix_x = [1 2 3; 4 5 6; 7 8 9] # rows separated by spaces, columns separated␣
↪by semicolons.
```

[68]:
```
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9
```

[69]:
```
matrix_y = [1 2 3;
            4 5 6;
            7 8 9] # another way to write the matrix above
```

[69]:
```
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9
```

Another important object in Julia is a sequence. We can create a sequence as follows,

[70]:
```
seq_x = 1:10:21 # this is a sequence that starts at one and ends at 21 with an␣
↪increment of 10.
```

[70]:
```
1:10:21
```

In order to collect the values of the sequence into a vector, we can use the `collect` function.

[71]:
```
collect(seq_x)
```

[71]:
```
3-element Vector{Int64}:
  1
 11
 21
```

4.0.3 Convenience functions

There are several convenience functions that can be applied to arrays. We can look at the length of an array by using the `length` function.

[72]:
```
length(vector_x) # provides information on how long the vector is
```

[72]:
```
3
```

[73]:
```
length(matrix_x)
```

[73]:
```
9
```

[74]:
```
size(vector_x) # provides information in the format of a tuple. This is more␣
↪useful for multidimensional arrays
```

[74]:
```
(3,)
```

[75]:
```
size(matrix_x) # we can now see the size of the matrix as 3 x 3
```

[75]:
```
(3, 3)
```

[76]:
```
zeros(3) # provides a vector of zeros with length 3.
```

[76]:
```
3-element Vector{Float64}:
 0.0
 0.0
```

```
0.0
```

Note that the default type for the zeros in the vector above is `Float64`. If we wanted we could specify these values to be integers as follows,

[77]:
```
zeros(Int, 3)
```

[77]:
```
3-element Vector{Int64}:
 0
 0
 0
```

[78]:
```
ones(3) # same thing as above, but fills with ones
```

[78]:
```
3-element Vector{Float64}:
 1.0
 1.0
 1.0
```

Sometimes we want to create vectors that are filled with random values. The simplest function is `rand()`, which selects random values from the uniform distribution on the interval between zero and one. This means that the each value is picked with equal probability in the interval between zero and one.

[79]:
```
rand(3) # values chosen will lie between zero and one. chosen with equal␣
↪probability.
```

[79]:
```
3-element Vector{Float64}:
 0.6652048620715921
 0.5954573712160555
 0.5449000600501381
```

An alternative is to pick values according to the standard Normal distribution, instead of the uniform distribution as above. The Normal distribution is a distribution which is symmetric around zero, with majority of values lying within three standard deviations of the mean (which is zero). This means that this random value selection tool will select values to the right and left of zero on the real line. In other words, it will select both positive and negative values with about 99% of values falling within three standard deviations from the mean.

[80]:
```
randn(3)
```

[80]:
```
3-element Vector{Float64}:
  0.7503223136476806
  0.12428590665111607
 -1.4359245471314834
```

4.0.4 Indexing

Remember from before that we can extract value from containers.

```
[81]: vector_x[1] # extract the first value
```

```
[81]: "def"
```

```
[82]: matrix_x[2, 2] # retrieve the value in the second row and second column of the␣
      ↪matrix
```

```
[82]: 5
```

```
[83]: vector_x[1:2] # gets the first two values of the vector
```

```
[83]: 2-element Vector{Any}:
       "def"
       "abc"
```

```
[84]: vector_x[2:end] # extracts all the values from the second to the end of the␣
      ↪vector
```

```
[84]: 2-element Vector{Any}:
       "abc"
       "hij"
```

```
[85]: vector_x[:, 1] # provides all the values of the first column
```

```
[85]: 3-element Vector{Any}:
       "def"
       "abc"
       "hij"
```

```
[86]: vector_x[1, :] # provides all the values from the first row
```

```
[86]: 1-element Vector{Any}:
       "def"
```

Next let's move on to a brief discussion on linear algebra.

Exercise #2 Consider a two dimensional array, which we can call `x_2d`.

1. Fill this 4×4 array with values
2. Extract all the column values of this array
3. Extract the value along the second row and third column
4. Determine if the matrix is **singular** (optional) – if you don't know what this means don't worry. You can always look it up on the internet.

---

5 Linear algebra operations

Linear algebra is an incredibly useful branch of mathematics that I would highly recommend students invest time in. If you are interested in the more computational or statistical side of economics then you need to learn some linear algebra. For a more detailed look at linear algebra, the best place to go is the QuantEcon website. The material there is a bit more advanced than we are covering for this course, but it is worthwhile working through if you are interested in doing postgraduate work in economics.

Linear algebra entails operations with matrices and vectors at its core, so let us show some basic commands.

[87]:
```
Q = matrix_x
```

[87]:
```
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9
```

We start with multiplication and addition operations on arrays.

One can multiply a matrix with a scalar and also add a scalar to each component of a matrix.

One can also multiply matrices together when they have the same dimension.

It is also possible to multiply the transpose of a matrix with itself (or another matrix).

All of these operations will have specific applicability in certain situations. I am not speaking at this stage as to why we are doing this, but simply stating the conditions under which we can perform these operations.

[88]:
```
2 * Q # multiply matrix with scalar. All elements of matrix are multiplied by 2.
```

[88]:
```
3×3 Matrix{Int64}:
  2   4   6
  8  10  12
 14  16  18
```

[89]:
```
2 + Q  # Look at the first two line of the error message below
```

```
MethodError: no method matching +(::Int64, ::Matrix{Int64})
For element-wise addition, use broadcasting with dot syntax: scalar .+ array
Closest candidates are:
  +(::Any, ::Any, ::Any, ::Any…) at /usr/share/julia/base/operators.jl:655
  +(::T, ::T) where T<:Union{Int128, Int16, Int32, Int64, Int8, UInt128,␣
↪UInt16, UInt32, UInt64, UInt8} at /usr/share/julia/base/int.jl:87
  +(::Union{Int16, Int32, Int64, Int8}, ::BigInt) at /usr/share/julia/base/
↪gmp.jl:535
  …
```

```
Stacktrace:

 [1] top-level scope

   @ In[89]:1

 [2] eval

   @ ./boot.jl:373 [inlined]

 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::
↪String, filename::String)

   @ Base ./loading.jl:1196
```

The error message above (`no method matching +(::Int64, ::Matrix{Int64})`) is telling us in the first line that there is no way to add an integer to a matrix. It is basically saying that the plus operator does not know what to do when the types of the inputs are an integer and matrix. This is why types play such an important role in Julia. The plus operator looks at the types to decide what action to perform. If there is no action corresponding to the input types provided then we get a `MethodError`.

In order to add the value of `2` to every component of the matrix, we need to `broadcast` the value of `2` across all the matrix components. We can think about this as adding the value of `2` elementwise.

The error message here even tells us that this is the way to solve the problem,

On the second line of the error message we have: For element-wise addition, use broadcasting with dot syntax: scalar .+ array

Broadcasting can be done in Julia using the `.` operator.

[90]:
```
2 .+ Q # notice the `.` in front of the plus sign.
```

[90]:
```
3×3 Matrix{Int64}:
 3   4   5
 6   7   8
 9  10  11
```

[91]:
```
Q*Q # matrix multiplication
```

[91]:
```
3×3 Matrix{Int64}:
  30   36   42
  66   81   96
 102  126  150
```

We can take the transpose of a matrix in two ways with Julia

```
[92]: Q' # technically the adjoint
```

```
[92]: 3×3 adjoint(::Matrix{Int64}) with eltype Int64:
 1  4  7
 2  5  8
 3  6  9
```

```
[93]: transpose(Q)
```

```
[93]: 3×3 transpose(::Matrix{Int64}) with eltype Int64:
 1  4  7
 2  5  8
 3  6  9
```

```
[94]: Q'Q # multiplication of the transpose with the original matrix
```

```
[94]: 3×3 Matrix{Int64}:
  66   78   90
  78   93  108
  90  108  126
```

We will get back to linear algebra in some of the other tutorials, so keep it in the back of your mind. Linear algebra is easily one of my favourite math topics, so we will see it throughout the tutorials and lectures.

If you want to know more about linear algebra in general, I recommend the Essence of Linear Algebra series by 3Blue1Brown on Youtube. It is aimed at anyone with a passing interest in linear algebra and does not suppose any type of math background to enjoy.

5.1 Linear algebra application: OLS

One of the things that you covered in the econometrics section of the course was ordinary least squares. You were told that you can easily run a regression between a dependent variable Y and and independent variable (or multiple variables) X. In Stata the command is pretty easy to do, it should be something like

```
regress y x
```

The mathematical specification for this regression is,

$$Y = X\beta + \epsilon$$

where β represents the OLS coefficient and ϵ is the error term. We cannot obtain the true value for β from our data since we don't have access to the full population dataset. However, we can get an estimate of β.

One of the possible estimators is the least squares estimator. This is where the idea of Ordinary Least Squares enters. I am going to skip the math (you will do this in Honours), but after doing the appropriate derivation you will see that the least squares estimate of β turns out to be

$$\hat{\beta} = (X'X)^{-1}X'Y$$

The prime (') means transpose and $(X'X)^{-1}$ indicates the inverse of $(X'X)$. You do not need to worry about how to take the inverse of a matrix, or even transposing, for now. You simply need to acknowledge that we are looking for a combination of matrix operations on the right hand side of the equation above.

Let us try to code our own version of an OLS estimator using the equation above.

The first thing to do is to generate some data for X. We start with a `100x3` matrix, which means that there are three explanatory variables in our model.

[95]:
```
X = randn(100, 3) # 100x3 matrix of independent variables
```

[95]:
```
100×3 Matrix{Float64}:
  0.275528    -1.10578     1.64411
 -1.44363      0.189027   -0.0209143
  0.871004     0.549976   -0.6829
 -0.470267    -0.43797    -0.341199
 -0.387124    -0.187535   -1.21592
  0.00404923   0.916763    0.471919
 -2.25025      0.579968    0.169318
  0.772607     0.293029    0.163801
 -0.0659606    0.248039   -0.594245
  1.32751     -0.249878   -0.0145082
 -0.709097     1.19777     0.5126
 -0.937251    -1.33809     0.0428325
 -0.339568    -1.0283     -0.217669

 -0.294019    -1.37957    -0.551934
 -0.275287    -0.920476    0.375872
 -1.13841     -1.0912     -0.763173
 -0.271108    -0.463501   -0.509342
  0.289243     1.45808     0.855998
 -0.0469698   -2.82371     1.49859
 -1.61426     -0.978754   -0.198383
 -0.594328     1.11824    -1.20015
  0.0954354   -0.853171    0.661278
  0.164566    -1.64269     0.0290745
  0.385077     0.0528676  -0.137145
  1.27261     -0.534318   -0.147221
```

Next we will change the first column to a constant column containing only ones. This is done so that there is a constant term in the regression. You have spoken about the need for a constant term in the regression with Marisa.

[96]:
```
X[:, 1] = ones(size(X, 1))
```

[96]:
```
100-element Vector{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
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
```

[97]:
```
X # let us see what X looks like now
```

[97]:
```
100×3 Matrix{Float64}:
 1.0  -1.10578     1.64411
 1.0   0.189027   -0.0209143
 1.0   0.549976   -0.6829
 1.0  -0.43797    -0.341199
 1.0  -0.187535   -1.21592
 1.0   0.916763    0.471919
 1.0   0.579968    0.169318
 1.0   0.293029    0.163801
 1.0   0.248039   -0.594245
 1.0  -0.249878   -0.0145082
 1.0   1.19777     0.5126
 1.0  -1.33809     0.0428325
 1.0  -1.0283     -0.217669

 1.0  -1.37957    -0.551934
 1.0  -0.920476    0.375872
```

```
1.0  -1.0912     -0.763173
1.0  -0.463501   -0.509342
1.0   1.45808     0.855998
1.0  -2.82371     1.49859
1.0  -0.978754   -0.198383
1.0   1.11824    -1.20015
1.0  -0.853171    0.661278
1.0  -1.64269     0.0290745
1.0   0.0528676  -0.137145
1.0  -0.534318   -0.147221
```

Next we want to define the true coefficient values and combine them in a vector.

[98]:
```
 = [6.0, 3.0, -1.0]
```

[98]:
```
3-element Vector{Float64}:
  6.0
  3.0
 -1.0
```

This then enables us to generate our Y variable using the first equation from this section.

[99]:
```
Y = X*  + randn(size(X, 1));
```

Now we need to try and get an estimate for β using the OLS equations provided above.

[100]:
```
 _hat = inv(X'X) * X'Y
```

[100]:
```
3-element Vector{Float64}:
  6.181904000205606
  3.0640272563832904
 -0.9653352768848974
```

We see that the estimate for β is relatively close to the true value. Let us now determine how close the $\hat{Y}$ is to the data generating process Y. We can do this by calculating the error and then squaring the errors, which gives us the sum of squared errors.

[101]:
```
Y_hat = X *  _hat # predictions of Y using  _hat
e = Y_hat - Y # difference between predicted and actual values
sse = e'e # sum of squared errors
```

[101]:
```
92.0968476986781
```

We don't have to make things this complicated though. Julia has it's own regression packages, such as `GLM`. IN order to use this package we have to put the data in a `DataFrame`. We will discuss the `DataFrame` concept more in another tutorial.

[102]:
```
data = DataFrame(Y = Y, X1 = X[:, 1], X2 = X[:, 2], X3 = X[:, 3])
```

[102]:

	Y	X1	X2	X3
	Float64	Float64	Float64	Float64
1	1.12886	1.0	-1.10578	1.64411
2	8.48563	1.0	0.189027	-0.0209143
3	8.34962	1.0	0.549976	-0.6829
4	4.18752	1.0	-0.43797	-0.341199
5	5.46993	1.0	-0.187535	-1.21592
6	10.1255	1.0	0.916763	0.471919
7	6.49587	1.0	0.579968	0.169318
8	6.70514	1.0	0.293029	0.163801
9	7.63655	1.0	0.248039	-0.594245
10	4.9111	1.0	-0.249878	-0.0145082
11	9.0848	1.0	1.19777	0.5126
12	2.71971	1.0	-1.33809	0.0428325
13	2.44553	1.0	-1.0283	-0.217669
14	5.2669	1.0	-0.667914	-0.464194
15	8.93432	1.0	0.481102	-1.15658
16	8.68214	1.0	0.478849	-0.591606
17	3.07901	1.0	-1.48058	-0.665691
18	0.294167	1.0	-1.36199	1.58825
19	1.52588	1.0	-1.30011	0.507774
20	4.8279	1.0	-0.0705876	0.752122
21	5.6718	1.0	-0.287182	-0.217418
22	5.18943	1.0	-0.636351	0.0149283
23	10.3614	1.0	0.611613	-1.41076
24	4.85441	1.0	-0.342739	0.252446
25	8.19357	1.0	0.10864	-0.138563
26	4.51378	1.0	-0.852136	1.36869
27	10.0376	1.0	1.16916	-0.726377
28	-2.31734	1.0	-1.76523	0.225482
29	8.08329	1.0	1.44129	2.12359
30	2.44548	1.0	-1.44495	0.880478
...	...	...	...	...

[103]:
```
lm(@formula(Y ~ X1 + X2 + X3), data) # This is similar to the command you␣
↪issued in Stata. lm stands for linear model.
```

[103]:
```
StatsModels.TableRegressionModel{LinearModel{GLM.LmResp{Vector{Float64}},
GLM.DensePredChol{Float64, CholeskyPivoted{Float64, Matrix{Float64}}}},
Matrix{Float64}}

Y ~ 1 + X1 + X2 + X3

Coefficients:

                Coef.  Std. Error      t  Pr(>|t|)  Lower 95%  Upper 95%
```

(Intercept)	0.0	NaN	NaN	NaN	NaN	NaN
X1	6.1819	0.100335	61.61	<1e-78	5.98277	6.38104
X2	3.06403	0.10623	28.84	<1e-48	2.85319	3.27486
X3	-0.965335	0.0934273	-10.33	<1e-16	-1.15076	-0.779908

You see the result for coefficient estimates are the same as those we achieved with our linear algebra inspired version of OLS.

5.1.1 Broadcasting

Before we continue to the next section, let us just take a second to think about the idea of broadcasting.

If you have a particular vector, such as [1, 2, 3], then you might want to apply a certain operation to each of the elements in that vector. Perhaps you want to find out what the sin of each of those values are independently? In Julia we can simply use our dot syntax to enable this.

[104]:
```
x_vec = [1.0, 2.0, 3.0];
```

There is one way in which you could do this. You could write the following for loop that performs the task (we will speak about loops soon).

[105]:
```
y_vec = similar(x_vec)
for (i, x) in enumerate(x_vec)
    y_vec[i] = sin(x)
end

y_vec # This now gives you sin of the x vector
```

[105]:
```
3-element Vector{Float64}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
```

Writing a loop like this for a simple operation seems wasteful, so Julia offers a better alternative. The dot operator.

[106]:
```
sin.(x) # notice the dot operator. What happens without the dot operator?
```

```
ArgumentError: broadcasting over dictionaries and `NamedTuple`s is␣
↪reserved
```

```
Stacktrace:
```

```
        [1] broadcastable(#unused#::NamedTuple{(:first, :second, :third),␣
↪Tuple{Char, Int64, Tuple{Int64, Int64, Int64}}})

          @ Base.Broadcast ./broadcast.jl:705

        [2] broadcasted(::Function, ::NamedTuple{(:first, :second, :third),␣
↪Tuple{Char, Int64, Tuple{Int64, Int64, Int64}}})

          @ Base.Broadcast ./broadcast.jl:1295

        [3] top-level scope

          @ In[106]:1

        [4] eval

          @ ./boot.jl:373 [inlined]

        [5] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::
↪String, filename::String)

          @ Base ./loading.jl:1196
```

You can see with this example that the sin() function has been applied to elementwise to the components of the vector. This is true for any function that you want to broadcast. This even applies to user defined functions, which we will discuss toward the end of the tutorial.

---

6 Control flow

In this section we will be looking at conditional statements and loops.

Conditional statements provide branches to a program depending on a certain condition. The most recognisable conditional statement is the if-else statement. Consider the example below.

[107]:
```
x = 1

if x < 2
    print("first")
elseif x > 4
    print("second")
elseif x < 0
    print("third")
else
```

```
    print("fourth")
end
```

```
first
```

One can also represent conditional statements via a ternary operator, this is often more compact.

[108]:
```
x, y, z = 1, 3, true
```

[108]:
```
(1, 3, true)
```

Can you make sense of what the expression below is saying? If you don't know then that is completely understandable.

[109]:
```
z ? x : y
```

[109]:
```
1
```

The above can also be written as follows, in a more lengthy if-else fashion

[110]:
```
if z
    x
else
    y
end
```

[110]:
```
1
```

A shorter, and perhaps more understandable way would be the following.

[111]:
```
ifelse(z, x, y) # This means, return x if z is true, otherwise return y.
```

[111]:
```
1
```

6.1 Loops

Avoid repeating code at ALL COSTS. Do not repeat yourself. This is the DRY principle in coding.

[112]:
```
x = [0,1,2,3,4]
```

[112]:
```
5-element Vector{Int64}:
 0
 1
 2
 3
 4
```

[113]:
```
y_1 = [] # empty array (list in python)
```

[113]: Any[]

[114]:
```
append!(y_1, x[1] ^ 2)
append!(y_1, x[2] ^ 2)
append!(y_1, x[3] ^ 2)
append!(y_1, x[4] ^ 2)
append!(y_1, x[5] ^ 2)
```

[114]:
```
5-element Vector{Any}:
  0
  1
  4
  9
 16
```

Generally we will encounter two types of loops. We get for loops and while loops. We will mostly focus on for loops here. They are best in cases when you know exactly how many times you want to loop. On the other hand, while loops are used when you want to keep looping till something in particular happens (some condition is met).

Let us try and fill our array using a for loop.

[115]:
```
y_2 = []
```

[115]: Any[]

In the following loop we will have 5 iterations. The array y_2 will be altered. We refer to y_2 as a **global** variable. We will talk about variable scope, local and global variables, later when we get to functions.

[116]:
```
for i in x
    append!(y_2, i ^ 2)
    println(y_2)
end
```

```
Any[0]
Any[0, 1]
Any[0, 1, 4]
Any[0, 1, 4, 9]
Any[0, 1, 4, 9, 16]
```

[117]:
```
y_3 = []
```

[117]: Any[]

[118]:
```
for i in 0:4
    append!(y_3, i ^ 2)
    println(y_3)
end
```

```
Any[0]
Any[0, 1]
Any[0, 1, 4]
Any[0, 1, 4, 9]
Any[0, 1, 4, 9, 16]
```

[119]:
```
y_4 = [i ^ 2 for i in x] # array comprehension -- we will mention this again␣
↪later on
```

[119]:
```
5-element Vector{Int64}:
  0
  1
  4
  9
 16
```

We could also use double for loops! This means that we nest a loop within another loop. This is used a lot in practice. Let us look at the following code to fill the entries of a matrix.

[120]:
```
(a, b) = (4, 3)
```

[120]:
```
(4, 3)
```

[121]:
```
matrix_x = zeros(Int, (a, b))
```

[121]:
```
4×3 Matrix{Int64}:
 0  0  0
 0  0  0
 0  0  0
 0  0  0
```

You should be able to evaluate this piece of code if you go through it slowly. You have been provided all the tools to do this.

[122]:
```
for i in 1:a
    for j in 1:b
    matrix_x[i, j] = 10 * i + j
    end
    return matrix_x
end
```

[122]:
```
4×3 Matrix{Int64}:
 11  12  13
  0   0   0
  0   0   0
  0   0   0
```

6.2 Iterables

[123]:
```
actions = ["watch Netflix", "like 318 homework"]
```

[123]:
```
2-element Vector{String}:
 "watch Netflix"
 "like 318 homework"
```

[124]:
```
for action in actions
    println("Peter doesn't $action")
end
```

```
Peter doesn't watch Netflix
Peter doesn't like 318 homework
```

[125]:
```
for i in 1:3
    println(i)
end
```

```
1
2
3
```

[126]:
```
x_values = 1:5
```

[126]:
```
1:5
```

[127]:
```
for x in x_values
    println(x * x)
end
```

```
1
4
9
16
25
```

[128]:
```
for i in eachindex(x_values)
    println(x_values[i] * x_values[i])
end
```

```
1
4
9
16
25
```

[129]:
```
countries = ("Japan", "Korea", "China")
```

[129]:
```
("Japan", "Korea", "China")
```

[130]:
```
cities = ("Tokyo", "Seoul", "Beijing")
```

[130]:
```
("Tokyo", "Seoul", "Beijing")
```

[131]:
```
for (country, city) in zip(countries, cities)
    println("The capital of $country is $city")
end
```

```
The capital of Japan is Tokyo
The capital of Korea is Seoul
The capital of China is Beijing
```

[132]:
```
for (i, country) in enumerate(countries)
    println(i, " ", country)

    #city = cities[i]
    #println("The capital of $country is $city")

end
```

```
1 Japan
2 Korea
3 China
```

6.3 Comprehensions

[133]:
```
doubles = [ 2i for i in 1:4 ]
```

[133]:
```
4-element Vector{Int64}:
 2
 4
 6
 8
```

[134]:
```
[ i + j for i in 1:3, j in 4:6 ]
```

[134]:
```
3×3 Matrix{Int64}:
 5  6  7
```

```
 6  7  8
 7  8  9
```

6.4 Break and continue (optional)

I don't think that break and continue statements are easy to read and don't often use them. I think there are easier ways to write loops. However, here is an example that uses these control flow commands.

[135]:
```
y_array = []
x_array = collect(0:9)
```

[135]:
```
10-element Vector{Int64}:
 0
 1
 2
 3
 4
 5
 6
 7
 8
 9
```

You don't have to spend too much time with this code. It is simply here to illustrate a basic point. Sometimes we will include break and continue statements within a loop that allow skipping and termination of loops at particular points.

In the example below the continue statement states that if i == 1 we continue the loop and then go to the next iteration. If we reach the point where i == 4 then the break command tells the computer to exit the for loop.

[136]:
```
for (i, x) in enumerate(x_array)

    if i == 1
        continue
    elseif i == 4
        break
    end

    append!(y_array, x^2)
end

println(y_array)
```

```
Any[1, 4]
```

---

7 Functions and methods

Functions are key to writing more complex code.

You should always write in terms of functions if possible.

From mathematics we know that a function is an abstraction that takes in some input and returns an output. In the world of computing the idea is similar. Let us take a look at some functions below.

The most common way to write a function is the following.

[137]:
```
function f(x) # function header
    return x ^ 2 # body of the function
end
```

[137]: f (generic function with 1 method)

This function accepts one input and returns one output. Namely, it takes the value of x and then squares it.

[138]:
```
f(2)
```

[138]: 4

A shorter way to write the same function is,

[139]:
```
g(x) = x ^ 2
```

[139]: g (generic function with 1 method)

[140]:
```
g(2)
```

[140]: 4

We could also use an anonymous function, as follows.

[141]:
```
h = (x) -> x ^ 2 # Lambda or anonymous function
```

[141]: #7 (generic function with 1 method)

[142]:
```
h(2)
```

[142]: 4

With the following function we have two inputs, x and y, that return one output.

[143]:
```
function k(x, y)
    """ This is a function that squares things

    k(x, y; a)
```

```
    a is this keyword argument
    """
    return x ^ 2 + y ^ 2
end
```

[143]: k (generic function with 1 method)

[144]:
```
k(2, 2)
```

[144]: 8

[145]:
```
function m(x, y)
    z = x ^ 2
    q = y ^ 2

    return z, q
end
```

[145]: m (generic function with 1 method)

[146]:
```
z, q = m(2, 2)
```

[146]: (4, 4)

[147]:
```
z
```

[147]: 4

[148]:
```
q
```

[148]: 4

7.1 Type annotations for functions (optional)

I just want to squeeze in a quick word about types here before we continue. This is an optional section and has more to do with the Julia type system. It is a bit more technical, so you can skip on the first reading.

Within the Julia type system, we can restrict our function to only accept a certain type of input. We covered the different primitive types such as integers and floats. In order to do this all that you need to do is add ::TypeName after the input argument for the function. Let us showcase this with an example.

[149]:
```
fun_1(x::Int64) = x + 2
fun_1(x::Float64) = x * 2
```

[149]: fun_1 (generic function with 2 methods)

What you see above is that this function, fun_1, is a generic function with two **methods**. In the case above, the output that you receive depends on the **type** of input that you provide. This is an example of *multiple dispatch*. This is one of the unique selling points of Julia, that not many other languages have.

Let us show how the return value differs based on different inputs.

[150]:
```
fun_1(1)
```

[150]: 3

[151]:
```
fun_1(1.0)
```

[151]: 2.0

7.2 What is return? (optional)

We mentioned the return value in the previous section so let us talk about the return part of the function body. What does it actually represent?

return basically instructs the program to exit the function and return the specified value.

Let us look at an example of where the return part will prevent execution of other parts of the function.

[152]:
```
function return_example_1(x)

    if x > 5
        return x + 5 # this will exit the function immediately

        println(x - 9) # this will never execute, it is after the return
    end

    return x / 5 # if x <= then this will be returned
end
```

[152]: return_example_1 (generic function with 1 method)

[153]:
```
return_example_1(1)
```

[153]: 0.2

[154]:
```
return_example_1(10) # notice that the `println(x - 9)` doesn't execute
```

[154]: 15

You don't have to write return in the body of the function. In Julia the function will return the last line of the function. However, it is good coding practice to write return so that you can see within the function what is being returned.

[155]:
```
function return_example_2a(x)
    y = x / 2 # this is the part that will be returned
end
```

[155]: return_example_2a (generic function with 1 method)

[156]:
```
return_example_2a(4)
```

[156]: 2.0

[157]:
```
function return_example_2b(x)
    y = x % 2
    x # this is now the part that will be returned
end
```

[157]: return_example_2b (generic function with 1 method)

[158]:
```
return_example_2b(4)
```

[158]: 4

You can also get multiple values returned,

[159]:
```
function return_example_2c(x)
    y = x % 2
    y + x, y - x, y * x, y / x # this will generate multiple return values
end
```

[159]: return_example_2c (generic function with 1 method)

[160]:
```
return_example_2c(4)
```

[160]: (4, -4, 0, 0.0)

You will notice that the return above has parentheses around it, which means that the values are contained in a tuple. Remember our discussion on tuples (an immutable type) from earlier.

7.3 Scope

Global variables are generally considered to be a bad idea.

If you want to be a good programmer, you need to understand the basics of local scoping.

Most of the code in the notebook has been contained in the global scope, but if you are writing scripts and source code you will most likely want to wrap code in functions to avoid variables entering the global scope.

When you copy variables inside functions, they become local and the function is said to become a **closure**.

We will show what we mean through some examples.

[161]:
```
a = 2 # global variable
```

[161]: 2

[162]:
```
function f(x)
    return x ^ a
end
```

[162]: f (generic function with 1 method)

[163]:
```
function g(x; a = 2)
    return x ^ a
end
```

[163]: g (generic function with 1 method)

[164]:
```
function h(x)
    a = 2 # a is local
    return x ^ a
end
```

```
cannot define function h; it already has a value


Stacktrace:
 [1] top-level scope
   @ none:0
 [2] top-level scope
   @ In[164]:1
 [3] eval
   @ ./boot.jl:373 [inlined]
 [4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::
↪String, filename::String)
```

```
@ Base ./loading.jl:1196
```

[165]:
```
f(2), g(2), h(2)
```

[165]: (4, 4, 4)

[166]:
```
a += 1
```

[166]: 3

[167]:
```
f(2), g(2), h(2)
```

[167]: (8, 4, 4)

Never rely on global variables!

7.4 Keyword arguments

We can give function arguments default values. If an argument is not provided then the default value is used. Alternatively, we can use keyword arguments.

The difference between keyword and standard (positional) arguments is that they are parsed and bounded by name rather than the order in the function call.

[168]:
```
function p(x, y; a = 2, b = 2)
    return x ^ a + y ^ b
end
```

[168]: p (generic function with 1 method)

Note the ; in our function call.

Using keyword arguments is generally discouraged, but there are times when it is useful.

[169]:
```
p(2, 4)
```

[169]: 20

[170]:
```
p(2, 4, b = 4)
```

[170]: 260

[171]:
```
p(2, 4, a = 4)
```

[171]: 32

8 Visualisation

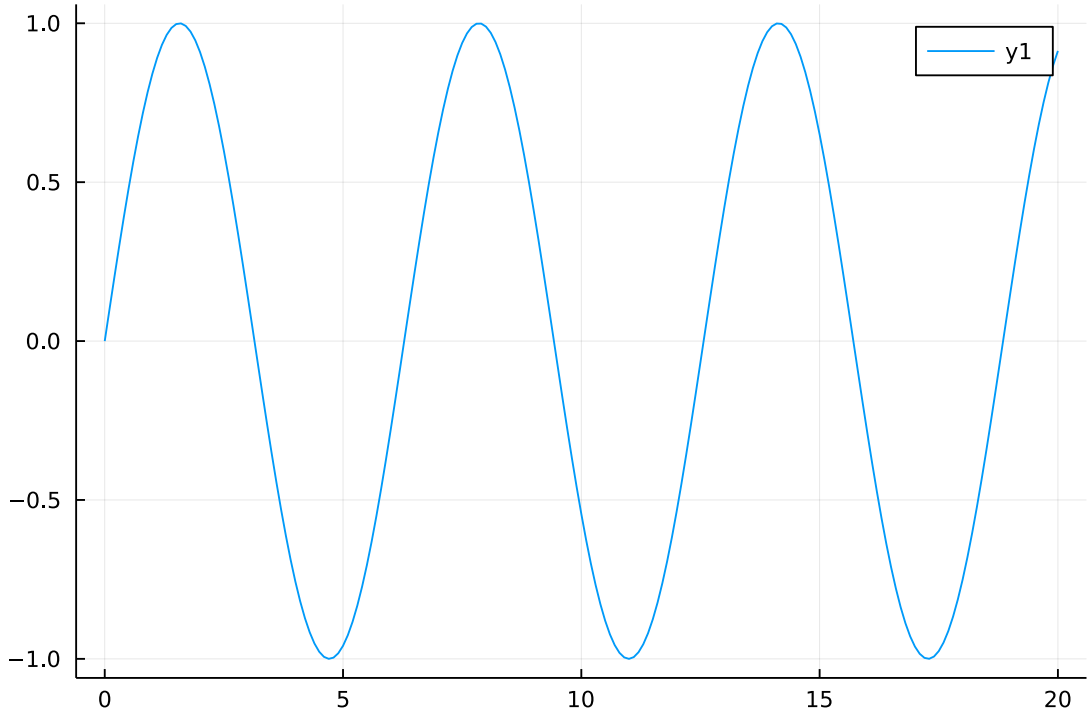
Let us show the basic principles of plotting Julia. There is a lot to learn about visualisation in Julia, we only showcase a small portion here.

[172]:
```
xs = 0:0.1:20;
```

[173]:
```
ys = sin.(xs); # we are using dot syntax - indicates broadcasting over the entire sequence
```

[174]:
```
plot(xs, ys)
```

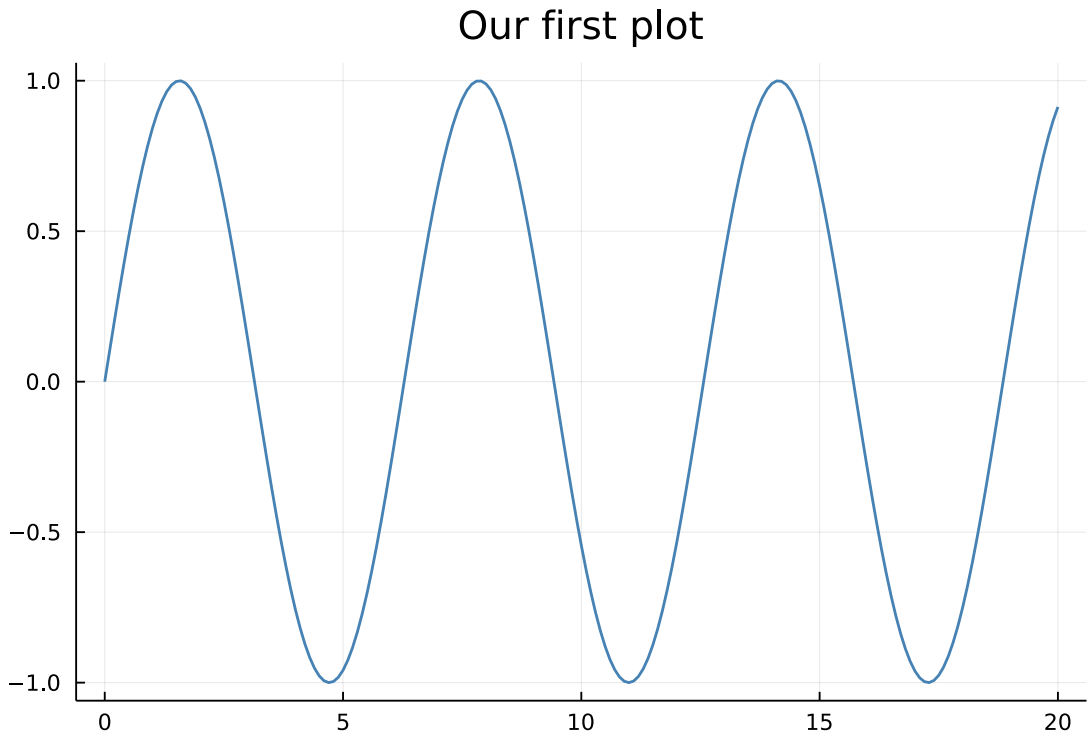
[174]:



We can add titles, legends, colors and other components with keywords. You can check the Plots.jl package documentation for the different keywords for plots.

[175]:
```
plot(xs, ys, color = :steelblue, title = "Our first plot", legend = false, lw = 1.5) # lw stands for line width
```

[175]:



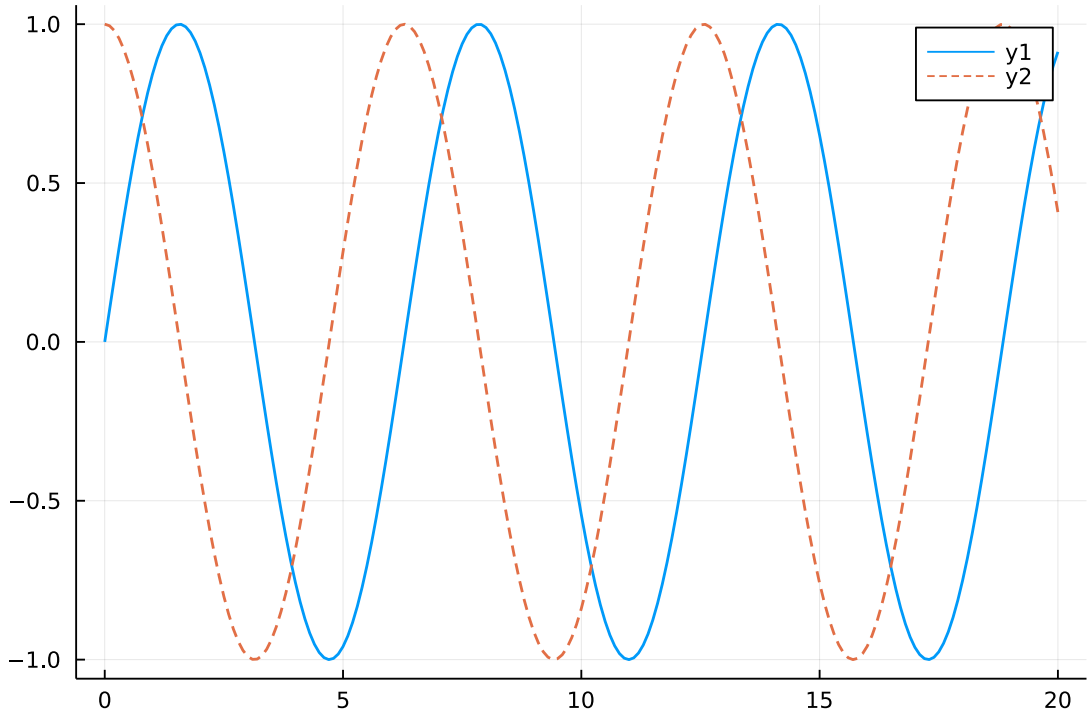
Another nice feature of plotting is the ability to overlay plots on top of each other. Let us define two different plots and then overlay.

[176]:
```
ys_1 = sin.(xs);
ys_2 = cos.(xs);
```

If you look carefully at the second line of the code below you will see that there is a ! operator. This exclamation mark means that we are mutating / altering the current plot. It is a nice coding convention to indicate to us that the original plot will now be altered.

[177]:
```
plot(xs, ys_1, lw = 1.5, ls = :solid)
plot!(xs, ys_2, lw = 1.5, ls = :dash)
```

[177]:



Lets look at a more complex plotting example

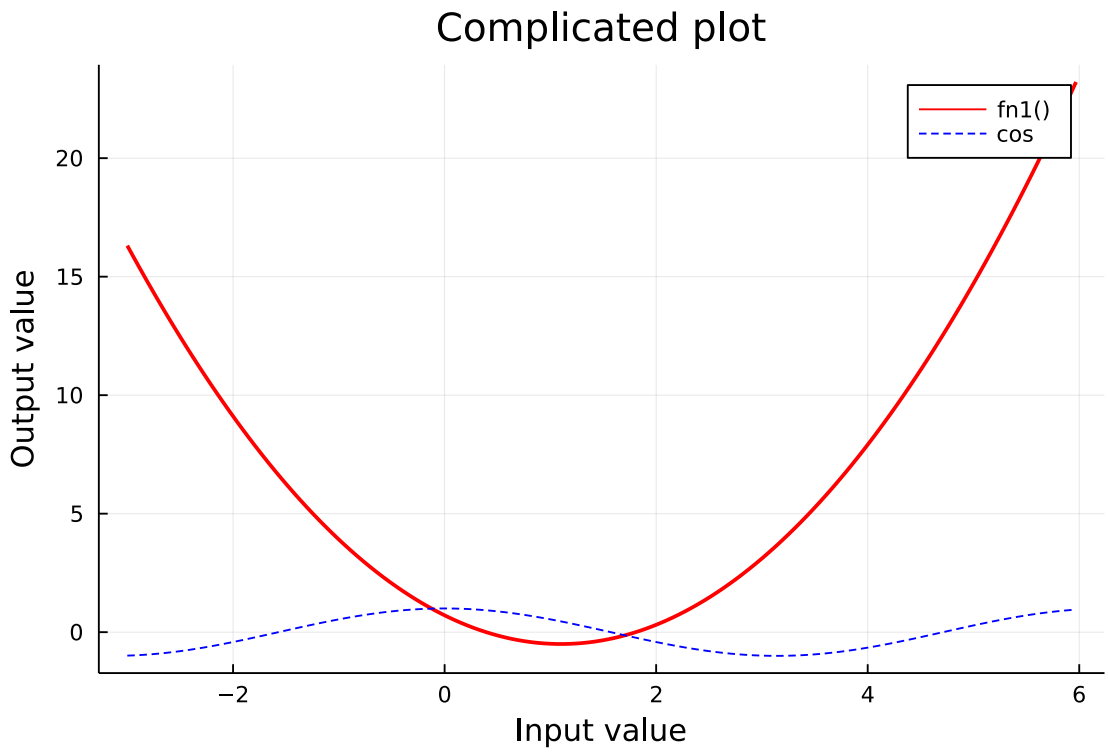
[178]:
```
x = -3:6/99:6
```

[178]:
```
-3.0:0.06060606060606061:5.96969696969697
```

[179]:
```
function fn1(x) #define a new function to plot
    y = (x-1.1)^2 - 0.5
    return y
  end;
```

[180]:
```
plot( [x x],[fn1.(x) cos.(x)],
            linecolor = [:red :blue],
            linestyle = [:solid :dash],
            linewidth = [2 1],
            label = ["fn1()" "cos"],
            title = "Complicated plot",
            xlabel = "Input value",
            ylabel = "Output value" )
```

[180]:



We will use plotting in all of the future tutorials, so this is worthwhile investing some time trying to learn the basics. For those of you who are interested in visualisation in programming, for Python you will most likely work with matplotlib and in R the ggplot package is quite popular.

---

9 Type system (advanced + optional)

This final section is optional, but I would recommend simply reading through it. There are some really valuable things about the way that Julia works as a language. However, if you have no interest in this type of thing you can safely skip the section.

Julia is different from languages like Python and R, in that it does not really work with objects. In Julia the important abstraction is the type.

Everything in Julia has some type, which in turn may be a subtype of something else. In this section we will explore the basics of the type system. Once again, this is a bit more complicated / technical, but worth exploring if you really like the Julia language.

Let us consider a type that we have dealt with, for example the Int64 type.

Int64 is a concrete type in Julia, as opposed to an abstract type. Concretes types are the last in the type hierarchy. In other words, there are no subtypes

below Int64 in Julia.

However, there are types that lie ``above'' Int64 in the type hierarchy. If you consider the subgraph of Int64 you will see the following.

[181]:
```
Int64 <: Signed <: Integer <: Real <: Number <: Any
```

[181]:
```
true
```

What this line of code says is that Int64 is a subtype of all the components to the right of it. The <: operator basically says that the left component is a subtype of the right.

In other words, Int64 is a subtype of the Integer type.

The Integer type in this case is a abstract type (which we cover in the section below), since there are subtypes below it (such as Signed and Int64).

You can take a look at the type tree below, which should give some indication on the hierarchy between types. You will see that concrete types are last in the hierarchy (the terminal nodes). Examples include Int128, Int64 and so forth.

[182]:
```
print(join(tt(Integer), ""))
```

```
Integer
   Bool
   GeometryBasics.OffsetInteger
   Signed
       BigInt
       Int128
       Int16
       Int32
       Int64
       Int8
   Unsigned
       UInt128
       UInt16
       UInt32
       UInt64
       UInt8
```

9.0.1 Type annotations

We have seen type annotations in our section on functions. In other words, we have something along the lines of x::Int64. This indicates that x is of the type Int64.

This is basically how you tell Julia's compiler that x is of the type Int64.

However, what if we are not sure that x is going to be an Int64, but we know that it will be a number. Then we it is possible to say x::Number.

If we state that something is of a certain type but this is not true, then we will receive a TypeError.

[183]:
```
10::Float64
```

```
TypeError: in typeassert, expected Float64, got a value of type Int64



Stacktrace:

 [1] top-level scope

   @ In[183]:1

 [2] eval

   @ ./boot.jl:373 [inlined]

 [3] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::
↪String, filename::String)

   @ Base ./loading.jl:1196
```

In this error message it tells you: ``expected Float64, got a value of type Int64''. This means you put down an integer and the compiler was expecting a floating point number. However, if you were uncertain of the type of the input you could have had a more general type, such as Number instead.

[184]:
```
10::Number
```

[184]: 10

What can we take from this, in defining functions we can often use type annotations to give an idea fo the type that we expect. There is a good reason for doing this with functions, which is related to the concept of **multiple dispatch** (a key feature that makes Julia a really special language).

9.0.2 Abstract types

At the core of the type system is the abstract type. Abstract types cannot be **instantiated**. This means that we cannot create them explicitly. They are just a way to **organise** the types that we have.

The things that we can instantiate are called **structs**.

A struct is a wrapper around some field. These structs are referred to as *composite types*.

This is not easy to understand at first, so let us use some example to make this clear.

In the following example we will be constructing a struct. This struct has no supertype defined here.

[185]:
```
struct FirstStruct
    field::Float64
end
```

This struct contains one field, which is of the type Float64. The input for this struct need to be of type Float64, otherwise the call will not work.

We create an instance of this struct as follows,

[186]:
```
x = FirstStruct(1.2)
```

[186]:
```
FirstStruct(1.2)
```

We can access the field value for this composite type as follows,

[187]:
```
x.field
```

[187]:
```
1.2
```

Next we will combine **structs** and **abstract types**.

Remember that abstract types are there to organise the structs.

We start with the highest level type first.

In our example we are going to be creating a Happiness type, which reflects the overall happiness in the economy.

[188]:
```
abstract type Happiness end
```

Next we think about specific **subtypes** that might be related in some way to happiness within the economy.

In this case I thought about GDP, which contains components such as investment and consumption. These factors might contribute to happiness. GDP is a measure of income, not wealth. So perhaps not the best way to determine happiness, but let us use it.

In addition, health considerations might also be important for happiness within the economy.

You can imagine your own particular type system here.

[189]:
```
abstract type GDP <: Happiness end
abstract type Health <: Happiness end
```

In our example, we can decompose GDP into its to main components. First, we have consumption expenditure on either goods and services. We can think about an increase in consumption indicating higher GDP and therefore greater happiness. This is very superficial, but play along for now.

We can also look at different types of investment. Most important for happiness might be residential investment. However, an increase in investment normally leads to higher job creation and in return higher levels of happiness.

[190]:
```
struct Consumption <: GDP
    goods::Float64
    services::Float64
end

struct Investment <: GDP
    inventory::Float64
    residential::Float64
    non_residential::Float64
end
```

In terms of the health component, let us say that Covid is currently the only disease that people care about. If you have Covid this tends to make you unhappy, and if you don't you are lucky and generally happy about it.

[191]:
```
abstract type Sick <: Health end

struct Covid <: Sick
    positive::Bool
end
```

Now we need to instantiate some of these structs that we have made,

[192]:
```
consumption_2021 = Consumption(1205.45, 980.57)
investment_2021 = Investment(130.86, 201.54, 335.12)
```

[192]: Investment(130.86, 201.54, 335.12)

[193]:
```
covid_pos_2021 = Covid(true)
```

[193]: Covid(true)

Now we can create a function that determines if we are happy based on the type of the input. This is where the idea of multiple dispatch comes in.

[194]:
```
happy(x::Investment) = true
```

[194]: happy (generic function with 1 method)

So far this function, called happy(), gives us the result true if the input type is Investment. Let us extend the function even more.

[195]:
```
happy(x::Consumption) = if consumption_2021.goods + consumption_2021.services >␣
↪1000 return true end
```

[195]: happy (generic function with 2 methods)

We now see that the function happy() is a function with two methods. That means that for different inputs it will perform different actions. In this case if the type is Investment the returned value will be true. If the input is Consumption then combined the values of the goods and services fields need to be greater than 1000 for the value to be true.

[196]:
```
happy(x::Covid) = false
```

[196]: happy (generic function with 3 methods)

This last bit of code now tells us that whenever the input type is reflects us being Covid positive, then we are not happy.

[197]:
```
map(happy, [consumption_2021, investment_2021, covid_pos_2021])
```

[197]:
```
3-element Vector{Bool}:
 1
 1
 0
```

We can see from the output above that instances of Consumption and Investment return true values. Boolean output of 1 is the same as true. In the case of being Covid positive we are not happy.