AGEC 652 - Lecture 1.3

An introduction to Julia

Diego S. Cardoso

Spring 2022

Software requirements

By now you hopefully have installed

- Julia
- Visual Studio Code with Julia extension
- Jupyter
  - We will take a quick look on how to use VS Code and Jupyter

Programming with Julia

*These slides are based on Software Carpentry, notes by Ivan Rudik and Grant Mcdermott, QuantEcon, and Julia documentation.

Why learn Julia?

Reason 1: It is easy to learn and use

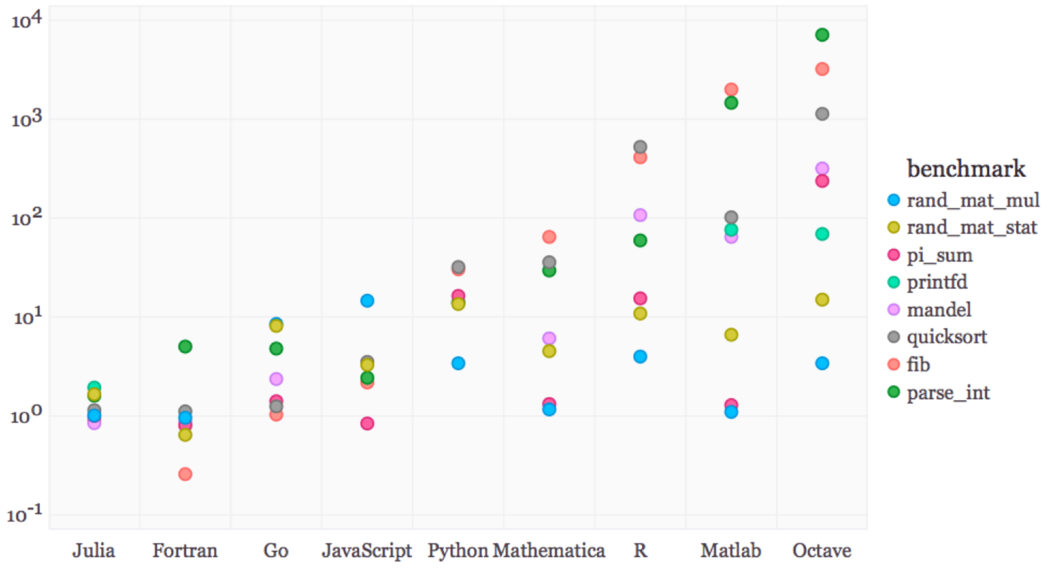
Julia is a *high-level* language

- Low-level = you write instructions are closer to what the hardware understands (Assembly, C++, Fortran)
  - E.g.:
  - These are usually the fastest because there is little to translate (what a compiler does) and you can optimize your code depending on your hardware
- High-level means you write in closer to human language (Julia, R, Python)
  - The compiler has to do a lot more work to translate your instructions

Why learn Julia?

Reason 2: Julia delivers C++ and Fortran speed

Sounds like magic, but it's just a clever combination of design choices targeting numerical methods



*In this graph, time to execute in C++ is 1

Why learn Julia?

Reason 3: Julia is free, open-source, and popular

- You don't need expensive licenses to use (unlike Matlab)
- The people who want to use or verify what you did also don't have to pay
- There is a large and active community of users and developers
    - So it's easy to get help and new packages

Time for an IDE showcase

We'll stop the slide show for a while to see two recommended *Integrated Development Environments*, or *IDEs*

- Visual Studio (VS) code
- Jupyter Lab notebooks

Intro to programming

Programming $\equiv$ writing a set of instructions

- There are hard rules you can't break if you want your code to work
- There are elements of style (e.g. Strunk and White) that make your code easier to read, modify, and maintain
- There are elements that make your code more efficient
    - Using less time or space (memory)

Intro to programming

If you will be doing computational work, there are:

1. Language-independent coding basics you should know
   - Arrays are stored in memory in particular ways
2. Language-independent best practices you should use
   - Indent to convey program structure, naming conventions
3. Language-dependent idiosyncracies that matter for function, speed, etc
   - Julia: type stability; R: vectorize

Intro to programming

Learning these early will:

1. Make coding a lot easier
2. Reduce total programmer time
3. Reduce total computer time
4. Make your code understandable by someone else or your future self
5. Make your code flexible

A broad view of programming

Your goal is to make a **program**

A program is made of different components and sub-components

The most basic component is a **statement**, more commonly called a **line of code**

A broad view of programming

Here is an example of a pseudoprogram:

```
deck = ["4 of hearts", "King of clubs", "Ace of spades"]
shuffled_deck = shuffle(deck)
first_card = shuffled_deck[1]
println("The first drawn card was " * shuffled_deck ".")
```

This program is very simple:

1. Create a deck of cards

A broad view of programming

Here is an example of a pseudoprogram:

```
deck = ["4 of hearts", "King of clubs", "Ace of spades"]
shuffled_deck = shuffle(deck)
first_card = shuffled_deck[1]
println("The first drawn card was " * shuffled_deck ".")
```

This program is very simple:

1. Create a deck of cards
2. Shuffle the deck

A broad view of programming

Here is an example of a pseudoprogram:

```
deck = ["4 of hearts", "King of clubs", "Ace of spades"]
shuffled_deck = shuffle(deck)
first_card = shuffled_deck[1]
println("The first drawn card was " * shuffled_deck ".")
```

This program is very simple:

1. Create a deck of cards
2. Shuffle the deck
3. Draw the top card

A broad view of programming

Here is an example of a pseudoprogram:

```
deck = ["4 of hearts", "King of clubs", "Ace of spades"]
shuffled_deck = shuffle(deck)
first_card = shuffled_deck[1]
println("The first drawn card was " * shuffled_deck ".")
```

This program is very simple:

1. Create a deck of cards
2. Shuffle the deck
3. Draw the top card
4. Print it

A broad view of programming

```
deck = ["4 of hearts", "King of clubs", "Ace of spades"]
shuffled_deck = shuffle(deck)
first_card = shuffled_deck[1]
println("The first drawn card was " * shuffled_deck ".")
```

What are the parentheses and why are they different from square brackets?

How does shuffle work?

What’s `println`?

It’s important to know that a **good program has understandable code**

Julia specifics

We will discuss coding in the context of Julia but a lot of this ports to Python, MATLAB, etc[1]

We will review

1. Types
2. Iterating
3. Broadcasting/vectorization
4. Scope
5. Generic functions
6. Multiple dispatch

[1] See https://cheatsheets.quantecon.org

1. Types

Types: boolean

All languages have some kind of **variable types** like *integers* or *arrays*

The first type you will often use is a boolean (`Bool`) variable that takes on a value of `true` or `false`:

```
x = true
```

```
## true
```

```
typeof(x)
```

```
## Bool
```

Types: boolean

We can save the boolean value of actual statements in variables this way:

```julia
@show y = 1 > 2
```

```
## y = 1 > 2 = false
```

```
## false
```

`@show` is a Julia macro for showing the operation.

- You can think of a macro as a shortcut name that calls a bunch of other things to run

Quick detour: logical operators

Logical operators work like you'd think

`==` (equal equal) tests for equality

```
1 == 1
```

```
## true
```

`!=` (exclamation point equal) tests for inequality

```
2 != 2
```

```
## false
```

Quick detour: logical operators

You can also test for approximate equality with ≈ (type `\approx<TAB>`)

```
1.00000001 ≈ 1
```

```
## true
```

Now back to types

Types: numbers

Two other data types you will use frequently are integers

```
typeof(1)
```

```
## Int64
```

and floating point numbers

```
typeof(1.0)
```

```
## Float64
```

- 64 means 64 bits of storage for the number, which is probably the default on your machine

Types: numbers

You can always instantiate alternative floating point number types

```
converted_int = convert(Float32, 1.0);
typeof(converted_int)
```

```
## Float32
```

Types: numbers

Math works like you would expect:

```
a = 2
```

```
## 2
```

```
b = 1.0
```

```
## 1.0
```

```
a * b
```

```
## 2.0
```

```
a^2
```

```
## 4
```

Types: numbers

```
2a - 4b
```

```
## 0.0
```

```
@show 4a + 3b^2
```

```
## 4a + 3 * b ^ 2 = 11.0
```

```
## 11.0
```

In Julia, you dont need `*` in between numeric literals (numbers) and variables

Types: strings

Strings store sequences of characters

You implement them with double quotations:

```
x = "Hello World!";
typeof(x)
```

```
## String
```

Note that `;` is used to suppress output for that line of code. Unlike some other languages, in Julia you don't need to add `;` after every command

Types: strings

It's easy to work with strings. Use `$` to interpolate a variable/expression

```
x = 10; y = 20; println("x + y =  $(x+y).")
```

```
## x + y =  30.
```

Use `*` to concatenate strings

```
a = "Aww"; b = "Yeah!!!"; println(a * " " * b)
```

```
## Aww Yeah!!!
```

You probably won't use strings too often unless you're working with text data or printing output. Note that `;` can also be used to type multiple commands in the same line. I'm doing it make it fit in this slide, but you should avoid it

Types: containers

Containers are types that store collections of data

The most basic container is the `Array` which is denoted by square brackets

```
a1 = [1 2; 3 4]; typeof(a1)
```

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

Arrays are **mutable**, which means you can change their values

```
a1[1,1] = 5; a1
```

```
## 2×2 Matrix{Int64}:
##  5  2
##  3  4
```

You reference elements in a container with square brackets

Types: containers

An alternative to the `Array` is the `Tuple`, which is denoted by parentheses

```
a2 = (1, 2, 3, 4); typeof(a2)
```

```
## NTuple{4, Int64}
```

`a2` is a `Tuple` of 4 `Int64`s. Tuples have no dimension

Types: containers

Tuples are **immutable** which means you **can't** change their values

```
try
  a2[1,1] = 5;
catch
  println("Error, can't change value of a tuple.")
end
```

```
## Error, can't change value of a tuple.
```

Types: containers

Tuples don't need parentheses (but it's probably best practice for clarity)

```
a3 = 5, 6; typeof(a3)
```

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

Types: containers

Tuples can be **unpacked**

```
a3_x, a3_y = a3;
a3_x
```

```
## 5
```

```
a3_y
```

```
## 6
```

This is basically how functions return output when you call them

Types: containers

But an alternative and more efficient container is the `NamedTuple`

```
nt = (x = 10, y = 11); typeof(nt)
```

```
## NamedTuple{(:x, :y), Tuple{Int64, Int64}}
```

```
nt.x
```

```
## 10
```

```
nt.y
```

```
## 11
```

Another way of accessing `x` and `y` inside the NamedTuple is

```
nt[:x]; nt[:y];
```

Types: containers

A `Dictionary` is the last main container type. They are like arrays but are indexed by keys (names) instead of numbers

```
d1 = Dict("class" => "AAAA999", "grade" => 97);
typeof(d1)
```

```
## Dict{String, Any}
```

`d1` is a dictionary where the key are strings and the values are any kind of type

Types: containers

Reference specific values you want in the dictionary by referencing the key

```
d1["class"]
```

```
## "AAAA999"
```

```
d1["grade"]
```

```
## 97
```

Types: containers

If you just want all the keys or all the values, you can use these base functions

```
keys_d1 = keys(d1)
```

```
## KeySet for a Dict{String, Any} with 2 entries. Keys:
##   "class"
##   "grade"
```

```
values_d1 = values(d1)
```

```
## ValueIterator for a Dict{String, Any} with 2 entries. Values:
##   "AAAA999"
##   97
```

2. Iteration

Iterating

As in other languages we have loops at our disposal:

`for` loops iterate over containers

```
for count in 1:10
  random_number = rand()
  if random_number > 0.2
    println("We drew a $random_number.")
  end
end
```

```
## We drew a 0.4119812206733202.
## We drew a 0.8861602863109513.
## We drew a 0.5577220906481677.
## We drew a 0.8616660603970644.
## We drew a 0.5097188389267755.
## We drew a 0.6948735181415873.
## We drew a 0.6028942225229486.
## We drew a 0.280022700798886.
## We drew a 0.928033423563621.
```

Iterating

`while` loops iterate until a logical expression is false

```
while rand() > 0.5
  random_number = rand()
  if random_number > 0.2
    println("We drew a $random_number.")
  end
end
```

Iterating

An `Iterable` is something you can loop over, like arrays

```
actions = ["codes well", "skips class"];
for action in actions
    println("Charlie $action")
end
```

```
## Charlie codes well
## Charlie skips class
```

Iterating

The type `Iterator` is a particularly convenient subset of Iterables

These include things like the dictionary keys:

```
for key in keys(d1)
  println(d1[key])
end
```

```
## AAAA999
## 97
```

Iterating

Iterating on `Iterator`s is more *memory efficient* than iterating on arrays

Here's a **very** simple example. The top function iterates on an `Array`, the bottom function iterates on an `Iterator`:

```
function show_array_speed()
  m = 1
  for i = [1, 2, 3, 4, 5, 6]
    m = m*i
  end
end;

function show_iterator_speed()
  m = 1
  for i = 1:6
    m = m*i
  end
end;
```

Iterating

```
using BenchmarkTools
@btime show_array_speed()
```

```
##   25.703 ns (1 allocation: 112 bytes)
```

```
@btime show_iterator_speed()
```

```
##   1.800 ns (0 allocations: 0 bytes)
```

The `Iterator` approach is faster and allocates no memory

`@btime` is a macro from `BenchmarkTools` that shows you the elasped time and memory allocation

Neat looping

A nice thing about Julia vs MATLAB: your loops can be much neater because you don't need to index when you just want the container elements

```
f(x) = x^2;
x_values = 0:20:100;
for x in x_values
  println(f(x))
end
```

```
## 0
## 400
## 1600
## 3600
## 6400
## 10000
```

Neat looping

This loop directly assigns the elements of `x_values` to `x` instead of having to do something clumsy like `x_values[i]`

`0:20:100` creates something called a `StepRange` (a type of `Iterator`) which starts at `0`, steps up by `20` and ends at `100`

Neat looping

You can also pull out an index and the element value by enumerating

```
f(x) = x^2;
x_values = 0:20:100;
for (index, x) in enumerate(x_values)
  println("f(x) at value $index is $(f(x)).")
end
```

```
## f(x) at value 1 is 0.
## f(x) at value 2 is 400.
## f(x) at value 3 is 1600.
## f(x) at value 4 is 3600.
## f(x) at value 5 is 6400.
## f(x) at value 6 is 10000.
```

`enumerate` basically assigns an index vector

Neat looping

There is also a lot of Python-esque functionality to loop without indexes

For example: `zip` lets you loop over multiple different iterables at once

```
last_name = ("Lincoln", "Bond", "Walras");
first_name = ("Abraham", "James", "Leon");

for (first_idx, last_idx) in zip(first_name, last_name)
  println("The name's $last_idx, $first_idx $last_idx.")
end
```

```
## The name's Lincoln, Abraham Lincoln.
## The name's Bond, James Bond.
## The name's Walras, Leon Walras.
```

Neat looping

Nested loops can also be made very neatly

```
for x in 1:3, y in 3:-1:1
  println("$x minus $y is $(x-y)")
end
```

```
## 1 minus 3 is -2
## 1 minus 2 is -1
## 1 minus 1 is 0
## 2 minus 3 is -1
## 2 minus 2 is 0
## 2 minus 1 is 1
## 3 minus 3 is 0
## 3 minus 2 is 1
## 3 minus 1 is 2
```

The first loop is the *outer* loop, the second loop is the *inner* loop

Comprehensions: the neatest looping

Comprehensions are an elegant way to use iterables that makes your code cleaner and more compact

```
squared = [y^2 for y in 1:2:11]
```

```
## 6-element Vector{Int64}:
##     1
##     9
##    25
##    49
##    81
##   121
```

This created a 1-dimension `Array` using one line

Comprehensions: the neatest looping

We can also use nested loops for comprehensions

```
squared_2 = [(y+z)^2 for y in 1:2:11, z in 1:6]
```

```
## 6×6 Matrix{Int64}:
##    4    9   16   25   36   49
##   16   25   36   49   64   81
##   36   49   64   81  100  121
##   64   81  100  121  144  169
##  100  121  144  169  196  225
##  144  169  196  225  256  289
```

This created a 2-dimensional `Array`

Use this (and the compact nested loop) sparingly since it's hard to follow

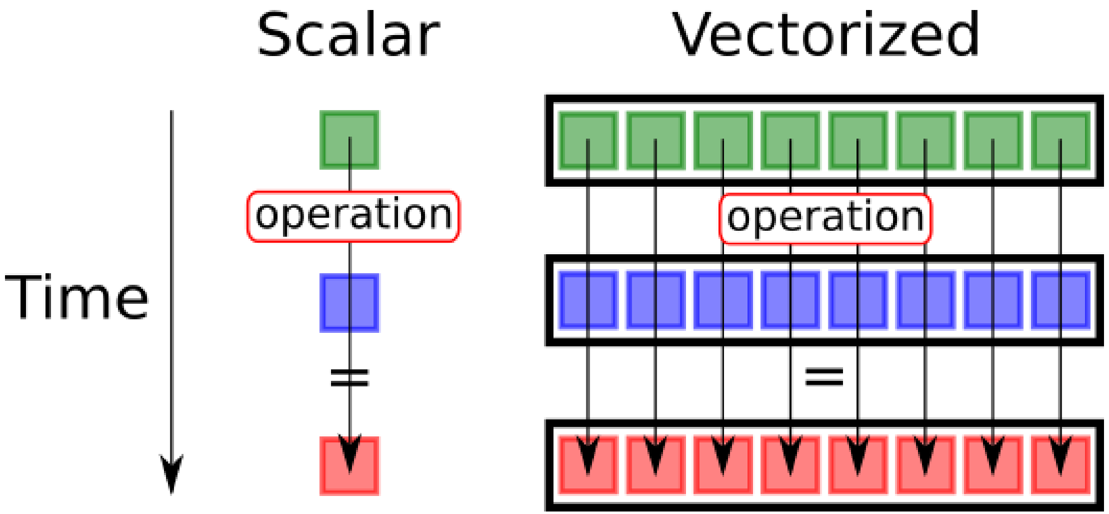
3. Broadcating/Vectorization

Vectorization

Iterated operations element by element is usually an inefficient approach

Another way is to do operations over an entire array. This is called **vectorization**

- It's faster because your processor can do some operations over multiple values with one instruction
- We'll get a better idea next lecture when we review the basics of computer architecture



Dot syntax: broadcasting/vectorization

Vectorizing operations is easy in Julia: just use *dot syntax* (like in MATLAB)

```
g(x) = x^2;
squared_2 = g.(1:2:11)
```

```
## 6-element Vector{Int64}:
##    1
##    9
##   25
##   49
##   81
##  121
```

This is actually called **broadcasting** in Julia

Dot syntax: broadcasting/vectorization

When broadcasting, you might want to consider **pre-allocating** arrays

Vectorization creates *temporary allocations*: temporary arrays in the middle of the process that aren't actually needed for the final product

Julia can do broadcasting in a nicer, faster way by fusing operations together and avoiding these temporary allocations

Dot syntax: broadcasting/vectorization

Let's write two functions that do the same thing:

```
function show_vec_speed(x)
  out = [3x.^2 + 4x + 7x.^3 for i = 1:1]
end
function show_fuse_speed(x)
  out = @. [3x.^2 + 4x + 7x.^3 for i = 1:1]
end
```

- The top one is just a normal, non-vectorized call
- The `@.` in the bottom one vectorizes everything in one swoop: the function call, the operation, and the assignment to a variable

Dot syntax: broadcasting/vectorization

First, precompile* the functions

```
x = rand(10^6);
show_vec_speed(x);
show_fuse_speed(x);
```

* *Just-in-time compilation* (JIT) is one of the tricks Julia does to make things run faster. It translates your code to processor language the first time you run it and uses the translated version every time you call it again. Here, we run the functions once so that compiling doesn't add to our measure of running time in the next slide.

Dot syntax: broadcasting/vectorization

Then, let's run and time it

```
@btime show_vec_speed(x)
```

```
##   14.013 ms (13 allocations: 45.78 MiB)
```

```
## 1-element Vector{Vector{Float64}}:
##  [12.555760435556753, 0.23956846007105262, 1.2973527673016338, 0.47285419773325577, 3.598250992189733,
```

```
@btime show_fuse_speed(x)
```

```
##   2.049 ms (3 allocations: 7.63 MiB)
```

```
## 1-element Vector{Vector{Float64}}:
##  [12.555760435556753, 0.23956846007105262, 1.2973527673016338, 0.47285419773325577, 3.598250992189733,
```

Full vectorization using `@.` is about 5--10x faster with 1/6 of the memory allocation

Dot syntax: broadcasting/vectorization

Let's see another example

```
h(y,z) = y^2 + sin(z); # function to evaluate
y = 1:2:1e6+1;         # input y
z = rand(length(y));   # input z
```

Dot syntax: broadcasting/vectorization

Here we are vectorizing the *function call* only

```
# precompile h
h_out_1 = h.(y,z);
```

```
@btime h_out_1 = h.(y,z) # evaluate h.(y,z) and measure time
```

```
##   4.317 ms (4 allocations: 3.81 MiB)
```

```
## 500001-element Vector{Float64}:
##    1.0494430944513036
##    9.036345937290246
##   25.13197933161584
##   49.65937822628343
##   81.46963139078271
##  121.5813811982197
##  169.75211809991453
##  225.77032620714914
##  289.3833981245439
##  361.43657560356434
```

Dot syntax: broadcasting/vectorization

Here we are vectorizing the *function call* **and** *assignment*. With pre-allocated memory and vectorized assignment, we get an additional performance gain

```
h_out_2 = similar(h_out_1) # This pre-allocates memory for an object of the same type and size
```

```
@btime h_out_2 .= h.(y,z)
```

```
##   3.908 ms (2 allocations: 128 bytes)
```

```
## 500001-element Vector{Float64}:
##    1.0494430944513036
##    9.036345937290246
##   25.13197933161584
##   49.65937822628343
##   81.46963139078271
##  121.5813811982197
##  169.75211809991453
##  225.77032620714914
##  289.3833981245439
```

Dot syntax: broadcasting/vectorization

Here we are again vectorizing the *function call* **and** *assignment*. But the `@.` syntax helps us write clear code because we only need to use it once instead of adding `.`'s everywhere

```
h_out_3 = similar(h_out_1)
```

```
@btime @. h_out_3 = h(y,z)
```

```
##   3.922 ms (2 allocations: 128 bytes)
```

```
## 500001-element Vector{Float64}:
##     1.0494430944513036
##     9.036345937290246
##    25.13197933161584
##    49.65937822628343
##    81.46963139078271
##   121.5813811982197
##   169.75211809991453
##   225.77032620714914
```

4. Scope

Scope

The **scope** of a variable name determines when it is valid to refer to that variable

- E.g.: if you create a variable inside a function, can you reference that variable outside the function?
- You can think of scope as different contexts within your program

The two basic scopes are **local** and **global**

Scope can be a frustrating concept to grasp at first. But understanding how scopes work can save you a lot of debugging time

Let's walk through some simple examples to see how it works

Scope

First, functions have their own **local scope**

```
ff(xx) = xx^2;
yy = 5;
ff(yy)
```

```
## 25
```

`xx` isn't bound to any values outside the function `ff`

- It is only used inside the function

Scope

Locally scoped functions allow us to do things like:

```
xx = 10;
fff(xx) = xx^2;
fff(5)
```

```
## 25
```

Although `xx` was declared equal to 10 *outside the function*, the function still evaluated `xx` within its own scope at 5 (the value passed as argument)

Scope

But, this type of scoping also has (initially) counterintuitive results like:

```
zz = 0;
for ii = 1:10
  zz = ii
end
println("zz = $zz")
```

```
## zz = 0
```

What happened?

Scope

What happened?

The `zz` *outside* the for loop has a different scope: it's in the **global scope**

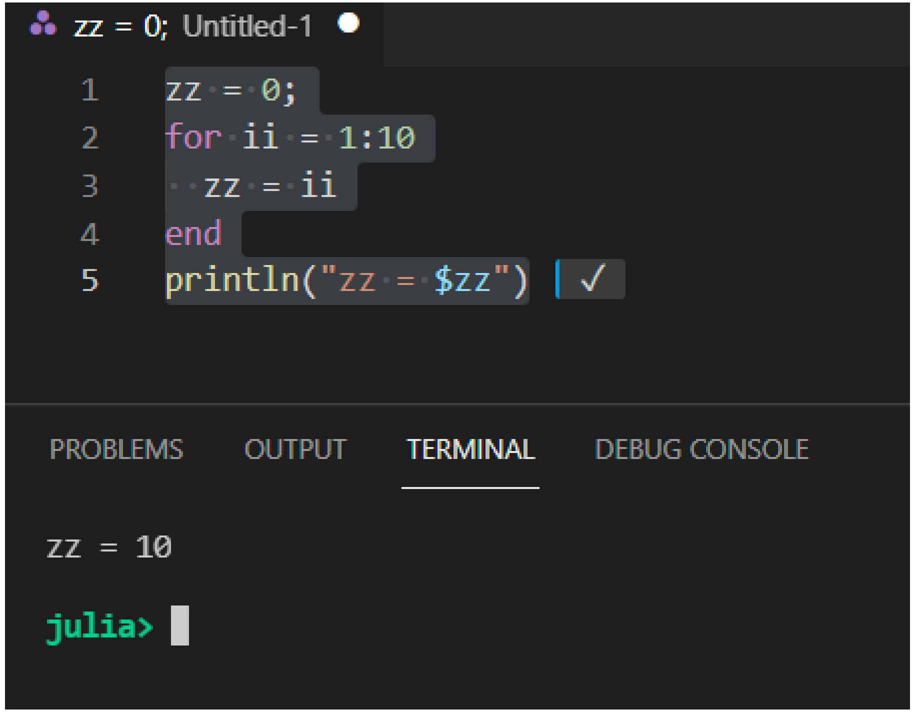
The global scope is the outermost scope, outside all functions and loops

The `zz` *inside* the for loop has a scope *local* to the loop

Since the outside `zz` has global scope, the locally scoped variables in the loop can't change it

Scope

But hold on. If you copy and paste the previous code and run it in REPL, it will actually return 10, not 0. * Was it all a lie?!



*Thanks, Chad, for pointing that out.

Scope

Actually, there are two types of local scope: **soft** and **hard**

Here is how Julia `1.7` applies them

Construct	Scope type	Allowed within
`module`, `baremodule`	global	global
`struct`	local (soft)	global
`for`, `while`, `try`	local (soft)	global, local
`macro`	local (hard)	global
functions, `do` blocks, `let` blocks, comprehensions, generators	local (hard)	global, local

Scope

When you assign `x = 10`

- If `x` is already defined in the local scope: the existing local `x` is assigned
- Otherwise
    - In **hard local scope**: a new local `x` is created and assigned
    - In **soft local scope**, it depends on whether a global `x` is defined...
        - If there is no global `x`: a new local `x` is created and assigned
        - If there is a global `x`: the assignment is *ambiguous*...
            - In *non-interactive* context (running a file): a new local `x` is created and assigned
            - In *interactive* context (REPL, notebooks): the global `x` is assigned

Scope

So here is why we get different results:

- The `for` loop written in global (e.g.: outside of a function) has **soft local scope**
- When I run the code in a file to generate these slides, that `for` loop is in a *non-interactive* context $\rightarrow$ a new local `zz` is created and assigned
- When I run it in VS Code/REPL, it's in an *interactive* context $\rightarrow$ the global `zz` is assigned

(This is a bit confusing, I know...)

Scope

Generally, you want to avoid global scope because it can cause conflicts, slowness, etc. But you can use `global` to force it if you want something to have global scope

```
zz = 0;
for ii = 1:10
  global zz
  zz = ii
end
println("zz = $zz")
```

```
## zz = 10
```

Scope

Local scope kicks in whenever you have a new block keyword (i.e. you indented something) except for `if`

Global variables inside a local scope are inherited for reading, not writing

```
x, y = 1, 2;
function foo()
  x = 2         # assignment introduces a new local
  return x + y # y refers to the global
end;
foo()
```

```
## 4
```

```
x
```

```
## 1
```

Scope

We can fix looping issues with global scope by using a wrapper function that doesn't do anything but change the parent scope so it is not global

```
zzz = 1;
function wrapper()
  zzz = 0;
  for iii = 1:10
    zzz = iii
  end
  println("zzz = $zzz")
end
```

```
## wrapper (generic function with 1 method)
```

```
wrapper()
```

```
## zzz = 10
```

5. Generic programming

Generic functions

If you use Julia to write code for research you should aim to write **generic functions**

These functions

- are flexible: e.g. can deal with someone using an `Int` instead of a `Float`
- have high performance, speed comparable to C

Generic functions

Functions are made generic by paying attention to types and making sure types are **stable**

Type stability: Given an input into a function, operations on that input should maintain the type so Julia *knows* what its type will be throughout the full function call

This allows Julia to compile type-specialized versions of the functions, which will yield higher performance

Type stability sounds like mandating types (like what C and Fortran do, unlike what R and Python do). So how do we make it flexible?

Generic functions: type stability

These two functions look the same, but are they?

```
function t1(n)
  s = 0
  t = 1
  for i in 1:n
     s += s/i
     t = div(t, i)
  end
  return t
end
```

```
function t2(n)
  s  = 0.0
  t = 1
  for i in 1:n
     s += s/i
     t = div(t, i)
  end
  return t
end
```

Generic functions: type stability

No! t1 is *not type stable*

`t1` starts with `s` as an `Int64`. But then we have `s += s/i` which means it must hold a `Float64`

It must be converted to `Float` so it is not type stable

Generic functions: type stability

We can see this when calling the macro `@code_warntype` where it reports `t1` at some point handles `s` that has type `Union{Float64,Int64}`, either `Float64` or `Int64`

Julia now can't assume `s`'s type and produce pure integer or floating point code. This leads to **performance degradation**

```
MethodInstance for t1(::Int64)
  from t1(n) in Main at Untitled-1:40
Arguments
  #self#::Core.Const(t1)
  n::Int64
Locals
  @_3::Union{Nothing, Tuple{Int64, Int64}}
  t::Int64
  s::Union{Float64, Int64}
  i::Int64
```

```
MethodInstance for t2(::Int64)
  from t2(n) in Main at Untitled-1:50
Arguments
  #self#::Core.Const(t2)
  n::Int64
Locals
  @_3::Union{Nothing, Tuple{Int64, Int64}}
  t::Int64
  s::Float64
  i::Int64
```

Concrete vs abstract types

A **concrete type** is one that can be instantiated

- E.g.: `Float64`, `Bool`, `Int32`

An **abstract type** cannot

- E.g.: `Real`, `Number`, `Any`

Concrete vs abstract types

Abstract types are used for organizing types

You can check where types are in the hierarchy (with the subtype operator `<:`)

```
@show Float64 <: Real
```

```
## Float64 <: Real = true
```

```
## true
```

```
@show Array <: Real
```

```
## Array <: Real = false
```

```
## false
```

Concrete vs abstract types

You can see the type hierarchy with the supertypes and subtypes commands

```
using Base: show_supertypes
show_supertypes(Float64)
```

```
## Float64 <: AbstractFloat <: Real <: Number <: Any
```

Creating new types

We can actually create new composite types using `struct`

```
struct FoobarNoType # This will be immutable by default
  a
  b
  c
end
```

This creates a new type called `FoobarNoType`

Creating new types

We can generate a variable of type `FoobarNoType` using its **constructor** which will have the same name

```
newfoo = FoobarNoType(1.3, 2, "plzzz");
typeof(newfoo)
```

```
## FoobarNoType
```

```
newfoo.a
```

```
## 1.3
```

Creating new types

Custom types are a *handy and elegant way of organizing your program*

- You can define a type `ModelParameters` to contain all your model parameters
- Each variable you instantiate represents a single scenario
- Then, instead of having a function call

```
RunMyModel(param1, param2, param3, param4, param5);
```

- You call

```
RunMyModel(modelParameters);
```

Creating new types

You should always declare types for the fields of a new composite type

You can declare types with the double colon

```
struct FoobarType # This will be immutable by default
  a::Float64
  b::Int
  c::String
end
```

Creating new types

```
newfoo_typed = FoobarType(1.3, 2, "plzzz");
typeof(newfoo_typed)
```

```
## FoobarType
```

```
newfoo.a
```

```
## 1.3
```

This lets the compiler generate efficient code because it knows the types of the fields when you construct a `FoobarType`

Declaring abstract types isn't good enough: you need to declare concrete types. But how do we keep it flexible, then?

Creating new types

Parametric types are what help deliver flexibility

We can create types that hold different types of fields by declaring subsets of abstract types

```
struct FooParam{t1 <: Real, t2 <: Real, t3 <: AbstractArray{<:Real}}
  a::t1
  b::t2
  c::t3
end
newfoo_para = FooParam(1.0, 7, [1., 4., 6.])
```

```
## FooParam{Float64, Int64, Vector{Float64}}(1.0, 7, [1.0, 4.0, 6.0])
```

The curly brackets declare all the different type subsets we will use in `FooParam`

This actually delivers high-performance code!

Delivering flexibility

We want to make sure types are stable but code is flexible

Ex: if want to preallocate an array to store data, how do we know how to declare it's type?

We don't need to!

Delivering flexibility

```
using LinearAlgebra                    # necessary for I
function sametypes(x)
  y = similar(x)                       # preallocates an array that is `similar` to x
  z = I                                # creates a scalable identity matrix
  q = ones(eltype(x), length(x))       # one is a type generic array of ones, fill creates the array of
  y .= z * x + q
  return y
end
```

```
## sametypes (generic function with 1 method)
```

```
x = [5.5, 7.0, 3.1];
y = [7, 8, 9];
```

Delivering flexibility

We did not declare any types but the function is type stable

```
sametypes(x)
sametypes(y)
```

```
MethodInstance for sametypes(::Vector{Float64})
  from sametypes(x) in Main at Untitled-1:69
Arguments
  #self#::Core.Const(sametypes)
  x::Vector{Float64}
Locals
  q::Vector{Float64}
  z::UniformScaling{Bool}
  y::Vector{Float64}
```

```
MethodInstance for sametypes(::Vector{Int64})
  from sametypes(x) in Main at Untitled-1:69
Arguments
  #self#::Core.Const(sametypes)
  x::Vector{Int64}
Locals
  q::Vector{Int64}
  z::UniformScaling{Bool}
  y::Vector{Int64}
```

There's a lot of other functions out there that help with writing flexible, type-stable code

6. Multiple dispatch

Multiple dispatch

Why type stability really matters: multiple dispatch

This means that the same function name can perform different operations depending on the type of the inputs it receives

In practice, a function specifies different **methods**, each of which operates on a specific set of types

Multiple dispatch

When you write a function that is type stable, you are actually writing many different methods, each of which are optimized for certain types

If your function isn't type stable, the optimized method may not be used

This is why Julia can achieve C speed: it compiles optmized code for each type and doesn't need to waste time "guessing" a variable's type

Multiple dispatch

/ has 118 different methods for division depending on the input types! These are 103 specialized sets of codes

```
methods(/)
```

```
## # 118 methods for generic function "/":
## [1] /(a, b::ChainRulesCore.AbstractThunk) in ChainRulesCore at C:\Users\Diego\.julia\packages\ChainRule
## [2] /(x::Union{Int128, Int16, Int32, Int64, Int8, UInt128, UInt16, UInt32, UInt64, UInt8}, y::Union{Int
## [3] /(x::Union{Integer, Complex{<:Union{Integer, Rational}}}, y::Rational) in Base at rational.jl:345
## [4] /(x::Union{Int16, Int32, Int8, UInt16, UInt32, UInt8}, y::BigInt) in Base.GMP at gmp.jl:545
## [5] /(c::Union{UInt16, UInt32, UInt8}, x::BigFloat) in Base.MPFR at mpfr.jl:433
## [6] /(c::Union{Int16, Int32, Int8}, x::BigFloat) in Base.MPFR at mpfr.jl:445
## [7] /(c::Union{Float16, Float32, Float64}, x::BigFloat) in Base.MPFR at mpfr.jl:457
## [8] /(A::Union{LinearAlgebra.AbstractTriangular, StridedMatrix}, D::Diagonal) in LinearAlgebra at C:\PR
## [9] /(X::StridedArray{P}, y::P) where P<:Dates.Period in Dates at C:\PROGRA~1\JULIA-~1.1\share\julia\st
## [10] /(X::StridedArray{P}, y::Real) where P<:Dates.Period in Dates at C:\PROGRA~1\JULIA-~1.1\share\juli
## [11] /(x::Union{SparseArrays.SparseVector{Tv, Ti}, SubArray{Tv, 1, <:SparseArrays.AbstractSparseVector{
## [12] /(A::Tridiagonal, B::Number) in LinearAlgebra at C:\PROGRA~1\JULIA-~1.1\share\julia\stdlib\v1.7\Li
## [13] /(z::Complex, x::Real) in Base at complex.jl:346
## [14] /(A::SparseArrays.AbstractSparseMatrixCSC, D::Diagonal) in SparseArrays at C:\PROGRA~1\JULIA-~1.1\
## [15] /(D::Diagonal, x::Number) in LinearAlgebra at C:\PROGRA~1\JULIA-~1.1\share\julia\stdlib\v1.7\Linea
```

Some concluding words on programming

There is really only one way to effectively get better at programming: **PRACTICE**

Yes, reading *can help*, especially by making you aware of tools and resources. But it's no substitute for actually solving problems with the computer

PS1 will be posted after we finish the next lecture. This is a good time to get familiarized with Julia and sharpen your skills

Some concluding words on programming

How to get started with your practice?

My suggestion of an intuitive way: **practice writing programs to solve problems you would know how to solve by hand**

- The computer follows a strict logic that very often is different from yours
- Learning how to tell the computer to follow instructions and get to a destination you already know is a great way of learning

My personal favorite: Project Euler

Some concluding words on programming

> *Project Euler is a series of challenging mathematical/computer programming problems that will require more than just mathematical insights to solve. Although mathematics will help you arrive at elegant and efficient methods, the use of a computer and programming skills will be required to solve most problems.*

Example of problems

1. If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23. *Find the sum of all the multiples of 3 or 5 below 1000.*
2. The prime factors of 13195 are 5, 7, 13 and 29. *What is the largest prime factor of the number 600851475143 ?*

You can type in your answer and it will tell you if it's correct

Some concluding words on programming

More on coding practices and efficiency:

- See JuliaPraxis for best practices for naming, spacing, comments, etc
- See more Performance tips from Julia Documentation

Course roadmap

This concludes Unit 1. Up next

1. Intro to Scientific Computing
2. **Numerical operations and representations**
    1. **Numerical arithmetic** $\leftarrow$
    2. Numerical differentiation and integration
    3. Function approximation
3. Systems of equations
4. Optimization
5. Structural estimation