present

Scientific programming workshop

Good Scientific Practises

- https://swcarpentry.github.io/good-enough-practices-in-scientific-computing/
- https://arxiv.org/pdf/1210.0530v3.pdf

The easiest way to learn and experiment with Julia is by starting an interactive session (also known as a read-eval-print loop or "REPL").

So fire up your terminal and type `julia` or open your Julia executable

Variables

A variable, in Julia, is a name associated (or bound) to a value (this one with a specific type)

1. The assignment operator is =

- You can assign *anything* to a variable binding, functions are **first class**
- The variable names can include pretty much any Unicode character.
- **All Julia operations return a value**. This includes assignment.

```
x = 1
```

```
# \alpha + TAB
α = 3
```

```
# \:smile_cat: + TAB
😺 = 2
```

```
😺 + α # You can 'TAB' before completing the command: try \alp + TAB
```

One can also silence printing by adding `;`

```
ħ = 6_626_070_15/1_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000_000;
```

And literal numbers can multiply anything without having to put `*` inbetween, as long as the number is on the left side:

```
5x - 1.2e-5x
```

Everything that exists in Julia has a certain **Type**. (e.g. numbers can be integers, floats, rationals). This is extremely important and we should always be aware with which types we are working with.

To find the type of a thing in Julia you simply use `typeof(thing)`:

```
typeof(😺) # returns Int

typeof("thursday seminar") # returns String
```

Basic collections

Indexing a collection (like an array or a dictionary) in Julia is done with brackets: `collection[index]`. The `index` is typically an integer, although some structures can have arbitrary indices (like a dictionary), and you can define any indexing type for any collection you want.

Julia indexing starts from 1

Tuples

Tuples are ordered immutable collections of elements. They are mostly used when the elements are not of the same type with each other and are intended for small collections.

Syntax: `(item1, item2, ...)`

```
myfavoritethings = ("mensa", "cats", π)
myfavoritethings[1] # will return "mensa"
```

NamedTuples

These are exactly like tuples but also assign a name to each variable they contain. They rest between

the `Tuple` and `Dict` type in their use.

Syntax: `(key1 = val1, key2 = val2, ...)`

```
nt = (x = 5, y = "str", z = 5/3)
```

These objects can be accessed with `[1]` like normal tuples, but also with the syntax `.key`:

```
nt[1]
nt.x
nt[:x] # will return 5
```

Pro-tip: You can use @unpack from `UnPack.jl` with named tuples

Arrays

Read more on Multi-dimensional arrays

The standard Julia `Array` is a mutable and ordered collection of items of the same type. The dimensionality of the Julia array is important. A `Matrix` is an array of dimension 2. A `Vector` is an array of dimension 1. The *element type* of what an array contains is irrelevant to its dimension!

i.e. a Vector of Vectors of Numbers and a Matrix of Numbers are two totally different things!

Syntax: `[item1, item2, ...]`

```
myfriends = ["Ted", "Robyn", "Barney", "Lily", "Marshall"]
fibonacci = [1, 1, 2, 3, 5, 8, 13]
mixture = [1, 1, 2, 3, "Ted", "Robyn"]
```

As mentioned, the type of the elements of an array must be the same. Yet above we mix numbers with strings! I wasn't lying though; the above array is an **unoptimized** version that can hold **any** thing. You can see this in the "type" of the array, whicn is to the left of its dimenmsionality: `Array{TypeOfTheThings, Dimensionality}`. For `mixture` it is `Any`.

Arrays of other data structures, e.g. vectors or dictionaries, or anything, as well as multi-dimensional arrays are possible:

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

```
# If you want to make a matrix, two ways are the most common: (1) specify each entr
y one by one
matrix = [1 2 3; # elements in same row separated by space
          4 5 6; # semicolon means "go to next row"
          7 8 9]

R[1,2] # two dimensional indexing
```

Since arrays are mutable we can change their entries, or even add new ones

```
fibonacci_new = [1, 1, 2, 3, 5, 8, 13]
fibonacci_new[1] = 15

push!(fibonacci_new, 16)
fibonacci_new
```

Lastly, for multidimension arrays, the `:` symbol is useful, which means to "select all elements in this dimension".

```
y = rand(3,3)
y[:,1]
```

Ranges

Ranges are useful shorthand notations that define a "vector" (one dimensional array). This is done with the following syntax:

```
start:step:end
range(start, end; length = ...)
range(start, end; step = ...)
```

```
r = 0:0.01:5

# Ranges are not unique to numeric data, and can be used with anything that extends
their interface, e.g.
rs = 'a':'z'

collect(rs)
```

It is important to understand that ranges **do not store all elements in memory** like `Vector`s. Instead they produce the elements on the fly when necessary, and therefore are in general preferred over `Vector`s if the data is equi-spaced.

Lastly, ranges are typically used to index into arrays. One can type `A[1:3]` to get the first 3 elements of `A`, or `A[end-2:end]` to get the last three elements of `A`. If `A` is multidimensional, the same type of indexing can be done for any dimension:

```
A = rand(4, 4)
A[1:3, 1]
```

Exercise: basic operations with `Array`s

- Create 2 random vectors of equal size. Add them with the `+` operator. Now multiply them with `*`. Does it work? **Why not?**
- Create a matrix of zeros (use the `zeros` function) and set its element `(1,1)` to `1.0`. Now set the first two elements of the 3rd column to `2.0`.

Control-flow and iteration

Iteration in Julia is high-level. This means that not only it has an intuitive and simple syntax, but also iteration works with anything that can be iterated. Iteration can also be extended (more on that later).

`for` loops

A `for` loop iterates over a container and executes a piece of code, until the iteration has gone through all the elements of the container. The syntax for a `for` loop is

```
for *var* in *loop iterable*
    *loop body*
end
```

Example:

```
for n ∈ 1:5
    println(n)
```

```
end
```

`while` loops

A `while` loop executes a code block until a boolean condition check (that happens at the start of the block) becomes `false`. Then the loop terminates (without executing the block again). The syntax for a standard `while` loop is

```
while *condition*
    *loop body*
end
```

Example:

```
n = 0
while n < 5
    n += 1
    println(n)
end
```

List comprehension

Comprehensions provide a general and powerful way to construct arrays. Comprehension syntax is similar to set construction notation in mathematics:

$$A = [F(x, y, \dots) \text{for } x = rx, y = ry, \dots]$$

The meaning of this form is that F(x,y,...) is evaluated with the variables x, y, etc. taking on each value in their given list of values. Values can be specified as any iterable object, but will commonly be ranges like 1:n or 2:(n-1), or explicit arrays of values like [1.2, 3.4, 5.7]. The result is an N-d dense array with dimensions that are the concatenation of the dimensions of the variable ranges rx, ry, etc. and each F(x,y,...) evaluation returns a scalar.

```
a = [sin(x) for x in range(0.0, π, length=1000)]
a = [sin(x) for x in range(-π, +π, length=1000) if x > 0]
```

Generator Expressions

Comprehensions can also be written without the enclosing square brackets, producing an object

known as a generator. This object can be iterated to produce values on demand, instead of allocating an array and storing them in advance

```
a = (evaluate_expensive_function(x) for x in range(-π, π, length=1000))
```

Conditionals

Conditionals execute a specific code block depending on what is the outcome of a given boolean check.

with `if`

In Julia, the syntax

```
if *condition 1*
    *option 1*
elseif *condition 2*
    *option 2*
else
    *option 3*
end
```

with ternary operators

For this last block, we could instead use the ternary operator with the syntax

```
a ? b : c
```

which equates to

```
if a
    b
else
    c
end
```

Functions

A function is an object that maps a tuple of argument values to a return value. Julia functions are not pure mathematical functions, because they can alter and be affected by the global state of the program.

The **basic syntax** is

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

or, as a 1-liner

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

f(3,4) # returns 7
```

Since a function name is just a `keyword`

```
g(x,y) = f(x,y)

# or, alternatively
h = f

# evals to 'true'
f(3,4) == g(3,4) == h(3,4)
```

The `+` symbol is actually a function name as well so even though it's terribly useful to use it between operands, it is actually just another function, expecting arguments delimited by parenthesis

Operators such as `+` are called *infix* operators.

- Question: Can you name other *infix* operators?

```
+(2,3) # returns 5
```

Functions in Julia support

- optional positional arguments: **always given by their order**
- keyword arguments: **always given by their keyword** (arguments defined iafter the symbol `;`)

```
g(x, y = 5; z = 2) = x * z * y

g(5) # give x. default y, z
g(5, 3) # give x, y. default z
g(5; z = 3) # give x, z. default y
g(2, 4; z = 1.5) # give everything
g(2, 4, 2) # keyword arguments can't be specified by position
```

Exercise: Collatz conjecture

Given a positive integer, create a function that counts the steps it takes to reach `1` following the **Collatz conjecture algorithm** (if n is odd do $n = 3n + 1$ otherwise do $n = n/2$). Test it with the number 100 to get 25.

Challenge: can you do it without loops?

Protip: make a type-stable function by using `÷` , (`\div<TAB>`): In Julia `/` is the floating point devision operator and thus `n/m` is always a float number even if `n, m` are integers.*

Slurping and splatting

- Slurping: `...` combines many arguments into one argument in **function definitions**

```
count_args(x...) = length(x) # on the rhs 'x' is actually a tuple of values

count_args(3.0, 2.0, 5.0, "a") # returns 4 (... packs in defintions)
```

- Splatting: `...` splits one argument into many different arguments in **function calls**

```
add3(a,b,c) = a + b + c

x = (1.0, 2.0, 3.0)
add3(x...) # returns 3 (... unpacks on calls)
```

Exercise: Write a function `swap_args` that swaps the arguments

Example:

```
swap_args(a,b) == (b,a)
swap_args(a,b,c,d) == (d,c,b,a)
```

Pro-tip: Cover the simple cases first

Anonymous functions

Functions can also be created anonymously, without being given a name, using either of these syntaxes

```
x -> x^2 + 2x - 1
```

and

```
function (x)
    x^2 + 2x - 1
end
```

So what's the difference?

As we have seen, this creates a function `f1` with 1-argument which we can address by `x`

```
f1(x) = ...
```

This assigns to the name `f2` a 1-argument function, with an argument which can address by `x`

```
f2 = x -> undefined
```

Examples:

- We can directly call an anynoymous functions

```
(x -> x + 1)(3) # returns 4
```

- It's very useful for quick and disposable functions. Consider `sum`, which can
  - sum a container `sum([1,2,3]) = 6`
  - sum a function applied to all elements of a container

```
add_1(x) = x + 1
sum(add_1, [1,2,3]) == sum([1+1, 2+1, 3+1]) == 9

# can be done much cleanear with an anonymous function
sum(x -> x + 1, [1,2,3]) # returns 9
```

Exercise: Swap arguments when calling a function

Create a function `swap` using the previously defined `swap_args` that calls any function `f` but with its arguments swapped.

Example:

```
swapped_f = swap(f)
swapped_f(a,b) == f(b,a) # returns 'true'
```

equivalently

```
swap(f)(a,b) == f(b,a)      # returns 'true'
swap(f)(a,b,c) == f(c,b,a)  # returns 'true'
```

Anonating types

- You can almost always ignore types

But why would you? With minimal effort they bring a lot of information, possibly speed and make your programs safer

We can annotate the types of our arguments

```
ff(x::Int, y) = x * y
```

```
ff(3, 4) # returns 12

ff(3.0, 4) # fails
```

And we can write a convertion for the return type

```
function gg(x::Int, y)::Int
    return x * y
end

gg(3, 4) # returns 12

gg(3, 4.1) # errors
```

This is conceptually very different from annotating the return type. Because Julia is dynamic it's not possible to guarantee the return type... The maximum we can do is to force a type convertion. However, there **may be some hope**

Multiple dispatch

The term multiple dispatch refers to calling the right implementation of a function based on the arguments. Note that only the positional arguments are used to look up the correct method.

This is really what is happening all the time under the hood: adding Ints is very different from adding Complex numbers, for example!

In Julia, a function may contain multiple concrete implementations (called Methods), selected via multiple dispatch, whereas functions in Python have a single implementation (no polymorphism).

```
my_sum(a::Int, b::Int) = a + b
my_sum(a::String, b::String) = a * " " * b # string concatenation is achieved by
(*)
```

The dispatch mechanism is choosing the most specific method for the input types

```
my_sum(2, 3) # returns 5
my_sum("Fuck", "COVID19") # returns a concatenated string

# We can check what exactly is being called with a @which macro
```

```
@which my_sum(2,3)
```

Exercise: Add (+) for `String`s to Julia

No functions in Julia are more special than others.

That applies to common operators as well. To add a `method` to the `(+)` operator, we have to import it from the Julia `Base` library (since it's Julia who owns `+`)

```
import Base: +
```

Extend `(+)` to also work with strings (e.g., `my_syum`) and `sum` an array of `String`s.

Pro-tip: Since we don't own neither `(+)` or `String` this is called *type piracy* and should be avoided as it can lead to unexpected behaviour

Exercise: Recursive `length`

Write a function that calculates the total number of elements inside nested arrays, ultimately containing numbers

Examples:

- `n_elements([1,1,1]) == 3`
- `n_elements([[1,], [1,], [1,1], [1,1]]) == 6`
- `n_elements([[], [], [1,2]]) == 2`
- `n_elements([[[2,1], [3,4]], [[1,2],]]) == 6`
- `n_elements([[1,2,[1,2]]]) == 4`

Note: Consider all arrays to have an `AbstractArray` type

Protip: Don't oversmart yourself: start **SIMPLE** and only then move on to the edge cases

Scoping

The **scope of a variable** is the region of code within which a variable is visible. Variable scoping helps

avoid variable naming conflicts. The concept is intuitive: two functions can both have arguments called x without the two x's referring to the same thing.

- Global scope

If a variable is in the global scope (of a module) it is visible even locally

```
x = 1
f() = x
f() # will return 1
```

- Local scope

When you create a function / structure / are inside a loop a local scope is created

```
x = 1
function f()
    x = 2
    return x
end
f() # will return 2
```

Pro-tip: to avoid polluting the global scope consider the `let` blocks, which work like `begin` blocks but introducing local scopes

```
x = let
    b = 1 # temporary variable
    2b + 2
end
b # will throw an error because b is not defined!
```

Sit down kiddo, let's talk mutability

Passing by reference: mutating vs. non-mutating functions

Mutable entities in Julia are passed by reference

Mutable means that the values of your data can be changed in-place, i.e. literally in the place in memory the variable is stored in the computer. **Immutable** data cannot be changed after creation, and thus the only way to change part of immutable data is to actually make a brand new immutable object from scratch.

For example, `Vector`s are mutable

```
v = [5, 5, 5]
v[1] = 6 # change first entry of x
v
```

But e.g. `Tuple`s are immutable

```
t = (5, 5, 5)
t[1] = 6
t
```

Note that while a `Tuple` is immutable, its elements may not be!

Do Julia algebraic operators such as `+=` operate in-place?

Consider the very simple example

```
a = 1
b = a
a += 2
b # returns 1
```

Were you expecting this behaviour? The problem is that sometimes we want mutability and other times we definitely do not want it. So we have to be very precise with what operators such as `+=` mean. And in Julia **ALL** updating operators are not in-place

Of course there are ways around this, but more on that later.

So we when have an array

```
a = [1,2]
b = a
```

```
a += 2a
b
```

The operation does not change the values in `a` but **REBINDS** the name `a` to the result of `a + 2a`, which of course is a new array.

So any operation such as `+=` is just *syntatic sugar* for

```
temp = a + 2a
a = temp
```

Meta-discussion: mutable vs immutable algorithms

Immutability doesn't really exist: we know very well from physics that something immutable is something time-independent... And there's nothing really stopping time. The very process of storing information (that is ordering bits) requires mutation. But we can achieve immutability at least locally.

One of the main culprits of the insane amount of time needed to develop scientific code is unwanted or forgotten mutatation of variables.

So when we write scientific code it's best to start with such a pure way of coding (mutation free). As it will lead to way less unexpected behaviour. The trade-off will be speed of course, but that's something we can deal with later

Here are some tips to minimise this time

- Use pure functions (Thus a pure function is a computational analogue of a mathematical function):
  - Its return value is the same for the same arguments (no variation with local static variables, non-local variables, mutable reference arguments or input streams from I/O devices).
  - Its evaluation has no side effects (no mutation of local static variables, non-local variables, mutable reference arguments or I/O streams).
- Use `let` blocks to reduce global scope pollution
  - Variables in the local scope are **very** prone to be mutated since they don't have to be passed as an argument explicitely
- Do NOT oversmart yourself, write clear and concise code and think about optimisations later only after your prototype is finished

- Use Pluto notebooks to prototype (they promote a hygenic use of global scoped variables)

Read more

Exercise: Write an Euler integrator

$$y_{n+1} = y_n + \Delta t_n \ f(y_n, t_n)$$

to solve the differential equation

$$y'(t) = f(y(t), t), \qquad f(y(t), t) = \sin(t) * y(t)$$

subject to the initial condition

$$y(t = 0.0) = 1.0$$

Protip: **NEVER** use the Euler method to solve any differential equation outside tutorials

Types

Types are formats for storing information.

How do we tell if `00011100011000110111011011111010101101` represents a number, or several numbers, or a colour, or a word, or what?

Each computer language has its own way of specifying the formats of information that it can use. Those are its types.

Note: It's good practice to *CamelCase* composite types and keep normal function names lower-cased.

Abstract Types

Abstract types cannot be instantiated, and serve only as nodes in the type graph, thereby describing sets of related concrete types: those concrete types which are their descendants. We begin with abstract types even though they have no instantiation because they are the backbone of the type system: they form the conceptual hierarchy which makes Julia's type system more than just a collection of object implementations.

How the numerical hierarchy in Julia works

```
abstract type Number end
abstract type Real <: Number end
abstract type AbstractFloat <: Real end
abstract type Integer <: Real end
```

The Number type is a direct child type of Any, and Real is its child. In turn, Real has two children (it has more, but only two are shown here; we'll get to the others later): Integer and AbstractFloat, separating the world into representations of integers and representations of real numbers.

The default supertype is Any – a predefined abstract type that all objects are instances of and all types are subtypes of. In type theory, Any is commonly called "top" because it is at the apex of the type graph.

Abstract vs concrete types

Concrete types are anything that can be actually instantiated. Any value that exists in Julia code that is running always has a concrete type. On the other hand, abstract types exist only to establish hierarchical relations between the concrete types.

Composite types

AKA *structs* or *objects* in other languages, these are **collection of named fields**.

In OOP languages, composite types also have named functions associated with them, and the combination is called an "object" (but many times you can find inconsistencies where not everything behaves like objects).

In Julia, all values are objects, but functions are not bundled with the objects they operate on.

Composite types are introduced with the `struct` keyword followed by a block of field names

```
struct MyCar
```

```
    brand::String
    color
end
```

Fields with no type annotation default to `Any`, and can accordingly hold any type of value.

We can create an object of type `MyCar` by calling a function `MyCar` (which is aumomatically created)

```
my_yellow_renault = MyCar("Renault", "yellow")
```

and access the `fields` of the car with the traditional `.`

```
my_yellow_renaul.brand # returns "Renault"
```

These functions that create new instances of our composite types are called **constructors**.

Question: Did we use constructors in this class before?

Since **constructors** are just functions, it's straightforward to extend ways of creating `MyCar` objects by adding other methods named `MyCar`

```
# All cars are, by-default, black
MyCar(b) = MyCar(b, "black")

my_car1 = MyCar("Mercedes")
my_car2 = MyCar("Mercedes", "white")
```

These are called **outer** constructors. One can also add **inner** constructors, which are quite useful for enforcing constraints

```
struct MyNewCar
    brand::String
    color
    wheels::Int

    MyNewCar(b, c) = new(b, c, 4)
end
```

Checking the methods available to create an instance of `MyNewCar`

```
# We see that we can't really specify the number of wheels
methods(MyNewCar)
```

The composite types we create until now are **immutable** so we can't really change the fields

```
my_car3 = MyNewCar("Mercedes", "blue")
my_car3.color = "yellow" # fails
```

But note that if one your fields has is **mutable**, like an array, then we can still mutate *its contents*

```
my_car4 = MyNewCar("Mercedes", ["blue"])
my_car4.color = ["blue"] # fails
my_car4.color[1] = "yellow" # mutates the field contents
```

Parametric types

Julia's type system is parametric: types can take parameters.

Parametric composite types

Type parameters are introduced immediately after the type name, surrounded by curly braces

```
struct Point{T}
    x::T
    y::T
end
```

This declaration defines a new parametric composite type, `Point{T}`, holding two "coordinates" of type `T`. What, one may ask, is `T`? Well, that's precisely the point of parametric types: it can be *any* type at all.

By specifying the parametric type, we obtain an unlimited number of distinct, **usable**, concrete types

```
Point{Float64} # point whose coordinates are 64-bit floating-point values
Point{String}  # "point" whose "coordinates" are string objects
```

If this sounds way too theoretical, here's an example where *parametric* types mean life or death: with `Array`s.

An `Array{Float64}` can be stored as a contiguous memory block of 64-bit floating-point values. While an `Array{Real}` can't possibly know how large each element is going to be so it can only be stored as array of pointers to individually allocated Real numbers (which could be `Float64` but also could be `Float64^1000`)

Parametric abstract types

The abstract types can also have parameters.

Consider the abstract (parametric) type `AbstractArray`

```
typeof([1.0, 2.0]) <: AbstractArray{Float64} # returns true
```

Read more

Multiple dispatch on parametric types

Fectch the `n_elements` function from the previous exercise

Note that something like `[1,2, [1,2]]` cannot have a well-defined type since the elements are both `Int` or `Vector{Int}` . So `typeof([1,2,[1,2]])` is actually `Vector{Any}` .

Supposing we don't want to operate on such impure arrays, we could do something like

```
n_elements(a::AbstractArray{Any}) = error("I don't operate on inferior type-unstabl
e structures")

n_elements([[1,2,[1,2]]]) # will throw an error
```

Exercise: Write a composite type `Measurement` that can propagate uncertainties!

Remember that, for some measurement $m_i = \mathrm{val}_i \pm \mathrm{err}_i$, the following rules apply

- $$a\, m_i = a\, \mathrm{val}_i \pm a\, \mathrm{err}_i$$
- $$m_1 + m_2 = (\mathrm{val}_1 + \mathrm{val}_2) \pm \sqrt{\mathrm{err}_1^2 + \mathrm{err}_2^2}$$

The latter assumes that variables are **always** uncorrelated, i.e., $\sigma_{12} = 0$, which is not true for, e.g., $m_2 = m_1$. But we are going to assume so because this is just an exercise.

In order to code the algebraic relationships, we need to extend the usual operators. This is achieved by

```
import Base: +
+(a::Number, m::Measurement) = ...
```

Also add a method for `zero`, which is the standard function to zero some type. See how it works on `zero(Float64)` or `zero(Int)`

```
import Base: zero
zero(::Type{Measurement}) = ...
```

If you have time, you can also add some syntatic sugar. There's a **vast set** of binary infix operators, that is, operators you can insert between operands, such as `a + b` being actually *sugar* for `+(a,b)`. Add a function with `±` as name to be able to create `Measurements` as `a ± b`

Exercise: Change the Euler integrator such that the initial condition is now

$$y(t = 0.0) = 1.0 \pm 0.3$$