present

Scientific programming workshop

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

The assignment operator is `=`

- Lax name binding: assign pretty much *anything* to a name (functions are **first class**)
- Variable names can include **Unicode characters**.
- **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
```

Silence printing in the REPL or notebooks by adding `;`

```
ħ = 6_626_070_15/1_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
```

In Julia, apart from very few speficic keywords being locked, you can pretty much redefine* predefined functions and constants, *as long as you didn't use them before*.

```
π = 3 # Archimedes of Syracuse is now crying
```

```
(+)(a,b) = "I refuse to add numbers"
3 + 2
```

A good way to think of this is that one is **shadowing** the the `(+)` methods that ship with Julia's `Base` library. You can still access them by doing `Base.(+)(3,2)`. The best, *really*, is not to redefine such holy operators as `(+)`.

Everything that exists in Julia has a certain **Type**.

Always be aware which are the underlying types of the variables.

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

```
typeof(😸) # returns Int
```

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

Basic collections

Indexing a collection is done with brackets: `collection[index]`. The `index` is typically an integer, but the indexing type can be (re)defined for any collection.

JULIA INDEXING STARTS FROM 1

Arrays

A Julia `Array` is a **mutable** and **ordered** collection of items of the same type.

The dimensionality of the Julia array is important.

- A `Vector` is an `Array` of dimension 1
- A `Matrix` is an `Array` of dimension 2.

The *element type* or *length* of an array is independent of its dimension.

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

```
myfriends = ["Karl", "Friedrich", "Vladimir", "Theodor", "Slavoj"]
years = [1818, 1820, 1870, 1903, 1949]
mixture = [1818, 1820, 1870, "Theodor", "Slavoj"]
```

The `mixture` array indeed has elements of the same type: the type `Any` : *the union of all types*. This is akin to a Python list and can't be optimised since the array can hold elements of *any* type (hence it's internally an array of pointers).

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

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

# To create a matrix

# (1) specify each entry 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]

# (2) create a 1D array and reshape it: Julia arrays are column major!
v = [1, 2, 3, 4, 5, 6, 7, 8, 9]
matrix = reshape(v, 3, 3)
```

The `:` symbol can be used to select all elements in some dimension.

```
matrix[:,1]
```

Since **arrays are mutable** their entries and size can be changed

```
fibonacci = [1, 1, 2, 3, 6]
fibonacci[5] = 5

push!(fibonacci, 8)
fibonacci
```

Read more on Multi-dimensional arrays

Ranges

A range is a sequence of of numbers, most commonly used for looping or creating equally spaced grids (the latter aka `linspace`). The syntax is

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

```
r = 0:0.01:5
```

A range is not unique to numeric data

```
rs = 'a':'z'
collect(rs)
```

Ranges **do not store all elements in memory** like `Array`s (unless `collect`ed). Their elements are generated on the fly.

Ranges are also used to index into arrays:

```
A = rand(10)
A[1:3] # gets the first 3 elements of `A`
A[end-2:end] # gets the last three elements of `A`
```

If `A` is multidimensional:

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

Exercise: basic operations with `Array`s and `Range`s

- Create 2 random vectors (with the `rand` function) of equal size and
    - Add them with the `+` operator
    - Multiply them with `*` operator.

    - Everything works ? **Why not?**
- Create a matrix of zeros (with the `zeros` function)
    - Get its `(1,1)` element
    - Set the first 3 elements of the 2nd column to `1.0`

Tuples

Tuples are ordered immutable collections of elements. Use them to group together related data not necessarily of of the same type.

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

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

NamedTuples

These are exactly like tuples but also assign a name to each field they contain.

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

```
myfavoritethings = (place="mensa", pets="cats", number=π)
```

These objects can be accessed with `[1]` like normal tuples, with they key symbol `[:key]` and also with `.key`.

```
nt[1]
nt.place
nt[:place] # returns "mensa"
```

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

Exercise: first taste of (im)mutability with `Tuple`s

- Create a 3-tuple with a `String`, a `Number` and an `Array` fields
    - Change `Number` field to twice its value. Does it work? Why (not)?
    - Change the 1st element of the `Array` field to twice its value. Does it work? Why (not)?

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 # \in
    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:

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

F(x,y,...) is evaluated with the variables x, y, etc. taking on each value in their given list of values. The result is an `N`-d dense array with dimensions that are the concatenation of the dimensions of the variable ranges `rx`, `ry`, etc.

```
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 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

```
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 `collatz` that counts the steps it takes to reach `1` following the **Collatz conjecture algorithm** (if n is odd do $n = 3n + 1$ otherwise do $n = n/2$).

```
collatz(100) == 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() == ()
swap_args(a) == (a,)
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 previously seen, this creates a function named `f1` with 1-argument locally named `x`

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

This assigns to the name `f2` a 1-argument function, with an argument locally named `x`

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

Examples:

- Directly call an anynoymous functions

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

- Incredibly useful for 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
```

The `add_1` is simply a **partial function application**: take the `+` operator, which adds 2 values, to `+1`, acting just on 1 value

```
sum(x -> x + 1, [1,2,3]) == 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'
```

Tip: You can also use `reverse`, which shares the same implementation as `swap_args` (you can see directly from the source code: `methods(reverse)` and look for the tuple implementation)

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

- Can annotate the types of our arguments:

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

ff(3, 4) # returns 12

ff(3.0, 4) # fails (method not defined!)
```

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

Multiple dispatch

(this will keep popping up)

Calling the right implementation of a function based on the arguments. Only the positional arguments (and type) are used to look up the correct method.

Happening all the time under the hood or on paper: multiplying scalars is completely different from multiplying matrices.

In Julia a function (i.e., the same `name`) may contain multiple concrete implementations (called Methods), selected via multiple dispatch.

- Question: what about functions in OOP languages?

Examples of multiple dispatch:

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

The dispatch mechanism chooses the most specific method for the input types

```
my_sum(2, 3) # returns 5
my_sum("Fuck", "COVID19") # returns a concatenated string
my_sum("Yo", 10) # errors

# Check what exactly is being called with a @which macro
@which my_sum(2,3)
```

Julia has got your back in case of ambiguities

```
f(x, y::Int) = 1
f(x::Int, y) = 2
f(2,3) # errors
```

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`

Consider all arrays to have an `AbstractArray` type.

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

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

No functions in Julia are more special than others, including operators. To add a `method` to the `(+)` operator `function`, import it from the Julia `Base` library (since it's Julia who owns `+`)

```
import Base: +
```

- Extend `(+)` to also work with `String`s (e.g., string concatenation with a space)
- Use `sum` to sum an array of `String`s

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

String concatenation (without a space in between) is already implemented, with the `(*)` operator.

- What is the identity `String` under multiplication?

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
```

Note: **A module** are workspaces with their own global scope. This is useful because it allows creation of global variables without conflicts! (When you use `REPL` you are in the `Main` module (`@__MODULE__`) so you can define anything you want without having to worry about conflicts with

- 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
```

Blocks

`begin` blocks are great as well but do not introduce a local scope

```
y = begin
    c = 3
    3c + 2
end
c # returns 3
```

so `begin` blocks find their use in multi-line definitions, e.g.,

```
f = x -> begin
    c = 2
    2c + x
end
```

To avoid polluting the global scope (in your notebooks) use `let` blocks

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

Passing by reference: mutating vs. non-mutating functions

Sit down kiddo, let's talk mutability

Mutable data can be changed in-place, i.e. literally in the place in memory where the data is stored.

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!

Mutable entities in Julia are passed by reference

When passing a mutable container, e.g., an `Array`, this is always passed by reference (i.e., a reference and not a copy of the variable is passed)

```
f(v) = (v[1] = 99)

x = [1,2,3]
f(x)
x[1] == 9
```

Pro-tip: in Julia there's a *convention* to add a `!` to the name of functions that *mutate* their arguments: `f!(v) = (v[1] = 99)`

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

Consider the very simple example

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

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.

Any operation such as `a+=2a` is just *syntatic sugar* for

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

In Julia **ALL** updating operators are not in-place

(there are ways around this, but more on that later)

Note: if you are coming from Python you may have an unhealthy relationship with `+=`-like operators: they behave like the above example, but with `Numpy` they act in-place (i.e., mutate the arrays).

In Julia, with an array, the behaviour is just like as the example with a scalar,

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

a += 2a
b
```

Meta-discussion: mutable vs immutable algorithms

Immutability doesn't really exist: immutability implies time-independence... and there's nothing really stopping time (at least until the heat-death of the universe).

The very process of storing information (that is ordering bits) requires mutation. But we can achieve immutability at least syntactically.

Tips to minimise the amount of time developing scientific code

by denying mutation and and promoting good hygiene

aka how to correct bad programming habits which hurt more than help

- Use `let` blocks to reduce global scope pollution
  - Global variables are **very** prone to be mutated since they don't have to be passed as an argument explicitely

- Pure thoughts: decompose programs into (pure) functions:
  - Same return value for the same arguments: no variation on non-local variables, (mutable) referenced arguments, etc.
  - Side-effects-free evaluation: no variation on non-local variables, (mutable) referenced arguments, etc.
  - Break software into chunks to fit into the most limited memory: human memory.

- Give functions and variables meaningful names
  - Ditch `Jupyter` for 95% of the cases: use `Pluto` notebooks to prototype

- Use tuples / structs to avoid repetition
  - `a1 = 1, a2 = 2` becomes `as = (1, 2)`

- Be defensive
  - Add `@assert` s to ensure validity of your inputs / results
  - Generate unit tests for your functions: these are as important as the problem you are ultimately solving

- Do NOT oversmart yourself:
  - avoid *premature optimisation*: write clear and concise code and only think about optimisations after unit testing
  - avoid *premature pessimisation*: take a few good minutes and sketch on paper the data structures / algorithm design before writing any code

- Abuse of your colleagues to review your code and warn you about common pitfalls

- Require of your code the same standards you require others' calculations / experiments / general care in life

Read more on good Scientific Practises

- 1
- 2
- 3

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

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

- Write the Euler integrator. Note: you can pass a type to `zeros` to create an `Array` of `0` s of some specific type: `zeros(Float64, 10)`
- Write a `unit` test for the integrator

Pro-tip: **NEVER** use the Euler method to solve any differential equation outside tutorials

Types

Types are formats for storing information.

How do we tell if `0001110001100011011101101111010101101` 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.

Supertype `Any` type

Its predefined abstract type that all types are subtypes of: `Any` is the union of all types, the entire universe of possible types.

```
# `isa` determines the type of some value
3 isa Any
[1,2,3] isa Any
(x -> 2x) isa Any
f(x) = 3x; f isa Any
```

When type annotation is ommited, the method accepts values of any type.

Everything that will follow below will be about looking at subsets of `Any` .

Abstract Types

Abstract types cannot be instantiated, and serve only to establish some conceptual hierarchy between types: these are the backbone of the type system.

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
```

You can use the function `supertypes` or `subtypes` to find these types and check hirarchies with the operator `<:`

```
Float64 <: AbstractFloat # returns True
```

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.

Composite types

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

In OOP languages, composite types also have named functions methods associated with them, and the combination is called an "object".

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
```

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

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

and access its `fields` with the traditional `.`

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

The 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 to the function `MyCar`

- These are called **outer** constructors.

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

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

- 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)
```

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

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
```

To add **mutability** to the field **values** (not types!), insert the `mutable` keyword

```
mutable struct MyMutableCar
    # ...
end
```

Exercise: In the beginning You created the heaven and the

Earth (Genesis* 1:1)

Jump directly to the 6th day of creation and create some `Animal` s that live and interact in the Garden of Eden.

Remember to embed the hierarchical relations between abtract and concrete and types using `<:`

- Create an abstract type for the `Animal` kingdom
- Create abstract types for `Reptile` s and `Mammal` s
- Make use of *multiple dispatch* and write (some) interactions between the animals

```
interaction(::Animal, ::Animal) = undefined
interaction(r::Reptile, m::Mammal) = "The reptile attacks the mammal"
```

- Create **concrete animals**, namely a 🐍 a 👱 and a 👩 (no gender is assumed) and their interactions (extra points if you stick to the historical guidelines)

```
interaction(::👩, ::👱) = "Mmmmmm, you tried these fruits before?"
```

- Create 🐸 before going for a rest, on the 7th day.

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
```

Note:

- The usual Julia `Array` s are parametric (on their type and dimension)
- Abstract types can also be parametric.

Read more

Type parameters in function signatures

Method definitions can optionally have type parameters qualifying the signature.

```
identity(p::Point{U}) where {U} = p
eltype(p::Point{U}) where {U} = U

fPoint = Point{Float64}(1.0, 2.0)
sPoint = Point{String}("x", "y")

eltype(fPoint) # returns Float64
eltype(sPoint) # returns String
```

Question: Why is the use of a keyword `where` necessary?

Pitfals when mixing type hierarchies

```
distance(p::Point{Real}) = sqrt(p.x^2 + p.y^2)

distance(fPoint) # fails: no method matching distance(::Point{Float64})
```

Question: If `Float64 <: Real` why does `Point{Float64} <: Point{Real}` yields false? Should there be any hierarchy between `Point{String}`, `Point{Float}` or `Point{Number}`?

The hierarchy wanted was established the level element types of `Point` and not at the level of `Point` s.

Hence,

```
distance(p::Point{T}) where {T <: Real} = sqrt(p.x^2 + p.y^2)
```

Think of `Point{T} where T <: Real` as the set of all concrete `Point` types for which element types are a subtype of `Real`: `Point{Float64}, Point{Int64}, Point{Int32}, ...}`

Exercise: Multiple dispatch on parametric types

On both tasks disallow behaviour having a function return `error("<error message>")`

- Diagonal dispatch: Create a function `party` that throws a 🎉 when both its arguments have the same type
- Give the `n_elements` function from a previous exercise a bit of personality and have it only operate on `Array` s with concrete types, e.g., disallow `AbstractArray{Any}` with `error("<error message>")`

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

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