Summary: Introduction to Julia

1. Introduction

First off, we can print text to the screen this way:

```
println("Hello, World!")
```

In a Jupyter Notebook, to execute a cell, we hit `Shift + Enter`. More info about Jupyter Notebooks here: `https://jupyter.org/`.

To suppress the output of a function (returned value), we use a semicolon at the end of a line:

```
rand(3, 3);
```

In Julia, we need not explicitly state the type of our variable when defining it. We can also reassign a variable to an entirely different type.

```
a = 42
b = "This is a string"
a = b
```

We can also add single and multi-line comments as shown:

```
# This is a comment

#=
This is a multi-line
comment
=#
```

Basic math operations are as usual. In Julia contrary to Python, we use '^' to denote exponentiation instead of '**'.

```
sum = 5 + 3
difference = 5 - 3
product = 5 * 3
quotient = 5 / 3
exponent = 5 ^ 3
modulus = 5 % 3
```

2. Strings

Strings are enclosed in double quotes (" "). They can can even be multi-line using three double quotes. Characters are enclosed in singe quotes (' ')

```
s1 = "Why would you say something"
s2 = " so controversial yet so brave?"
s3 = """
Single-line who?
"""
my_char = 'b'
```

We can use variables in strings. This is called string interpolation:

```
a = 42
s4 = "What is the meaning of life? $a."
s5 = "Half the meaning is $(a/2)."
```

We can even use variables by using string concatenation.

```
a = 42
s4 = string("What is the meaning of life?, a, ".")
```

There are 3 ways to concatenate strings:

```
s1 = "Why would you say something"
s2 = " so controversial yet so brave?"

s6 = s1 * s2
s7 = string(s1, s2)
s8 = "$s1$s2"
```

s6, s7, s8 are the same.

3. Data Structures

There are three main data structures for collections of objects in Julia: A Dictionary, an Array, and a Tuple.

A Dictionary (or Dict) is a mutable (i.e. changeable or modifiable) unordered collection of objects.

An array is a mutable and ordered collection of objects.

A tuple is an immutable (can't be changed) ordered collection of objects.

Though these data structures do have their derivatives, we will discuss only the main data structures so we stick to the fundamentals.

A. Dicts

We define dicts like this:

```
my_dict = Dict("a" => 1, "b" => 2, "c" => 3)
```

"a", "b", and "c" are called "keys" and 1, 2, and 3 are called "values".

We can add items to dicts like this:

```
my_dict["d"] = 4
```

We can access dicts like this: (the code prints '4')

```
println(my_dict["d"])
```

We can remove items from dicts using the key of the item:

```
pop!(my_dict, "d")
```

B. Arrays

We define Arrays like this:

```
my_arr = ["a", "b", "c"]
my_friends = []
```

We can access arrays like this: (the code prints `"c"`). Note that arrays and tuples in Julia start from the index 1.

```
println(my_arr[3])
```

Since arrays are mutable (changeable), their elements can be changed:

```
my_arr[2] = "B"
```

We can add elements to the end of arrays like this:

```
push!(my_arr, "d")
```

We can remove items from the end of an array like this:

```
pop!(my_arr)
```

Arrays can also be multi-dimensional, and need not contain the same type:

```
new_arr = [1, 2, 3, "a", 'b', (p, q, r)]
multi_arr = [[1, 2, 3],
             [4, 5],
             [6 ,7, 8, 9]]
```

We can initialize a random n-dimensional array like this:

```
3_dimensional_array = rand(4, 5, 6)
```

This creates a 3D matrix with 4 rows, 5 columns, and 6 stacks or layers filled with random numbers from 0 to 1. If the row-column-stack terms don't work for you, you can imagine a 4x5x6 cuboid made up of small cubes. They can be accessed at the 'i'th row, 'j'th column, and 'k'th layer/stack:

```
3_dimensional_array[i, j, k]
```

C. Tuples

We define Tuples like this:

```
my_tuple = ("a", "b", "c")
```

We can access tuples like this: (the code prints `"c"`).

```
println(my_tuple[3])
```

4. Loops and Control Flow

Loops make it easier to run code repetitively. A while loop takes in a condition, and keeps running the enclosed code while the condition is true, and a for loop iterates over a collection to run code a fixed number of times. While loops can also interate over a collection, but they need a predefined iterator and need to explicitly increment that iterator to do so.

We write while loops like this:

```
i = 0
arr = [1, 3, 5, 7]
while i < length(arr)
    println(arr[i])
    i += 1
end
```

We write for loops like this:

```
arr = [1, 3, 5, 7]
for i in arr
    println(i)
end
```

The 'in' keyword can be replaced with the '=' or '∈' symbol. We can even use double for loops to iterate through 2D arrays:

```
arr = rand(3, 3)
for i in 1:3, j in 1:3
    println(arr[i, j])
end
```

We can even create lists using for loops (called Array Comprehension):

```
sum_table = [i + j for i in 1:3, j in 1:3]
```

We can use if statements to evaluate conditions and execute code accordingly:

```
if *condition1*
    *code to run if condition1 is true*
elseif *condition2*
    *code to run if condition1 is false, but condition2 is true*
else
    *code to run if all conditions are false*
end
```

We can use a cool one-liner for binary if-statements (called Ternary Operators):

```
*cond* ? *code to run if cond is true* : *code to run if cond is false*
```

We can use short-circuit evaluation:

```
a > b && println("a is larger than b")
a < b && println("a is smaller than b")
a == b && println("a is equal to b")
```

5. Functions

Functions make it easier to reuse bits of code whenever we want them. In Julia, we define functions like this:

```
function function_name(param1, param2, param3)
    *code that may or may not use param1, param, and param3*
end
```

We can even write functions in one line:

```
function_name(param1, param2, param3) = *code that may or may not use params*
```

We can even define functions inside other code (anonymous functions):

```
arr = [1, 3, 5, 7]
arr_sqr = map(x -> x^2, arr)
```

The above code squares each element of the array and stores it in a new array.

By convention, functions that mutate the parameters provided to them should have a '!' symbol at the end of their name. The following code sorts 'abc' in-place.

```
abc = [3, 3, 4, 1, 2]
sort!(abc)
```

We can even run functions on each element in an array using broadcasting:

```
arr = [1, 2, 3, 4]
f(x) = x^2
arr_sqr = f.(arr)
```

The dot after the function name tells Julia to run the function on each of the elements of the array.

6. Packages

Packages are very useful in real life applications. They save a ton of time, and prevent people from 'reinventing the wheel'. In Julia we can install and use a package using Pkg. Once we install a package on an environment using Pkg.add(), we don't need to run it again unless we want the latest updates to that package.

```
using Pkg
Pkg.add("package_name")
using package_name
```

7. Multiple Dispatch

We can specify the types a function takes in not only to boost code performance, but also to modify existing functions for different types.

```
import Base: +

+(s1::String, s2::String) = string(s1, s2)

s1 = "Why would you say something"
s2 = " so controversial yet so brave?"
s3 = s1 + s2
```

8. Benchmarking

We use the BenchmarkTools.jl package for benchmarking. To benchmark a function, we run:

```
using BenchmarkTools

@benchmark function(param1, param2, param3)
```

This prints some information about how fast the function is.

9. Linear Algebra and Misc.

To get the help docstring for some function, just run the following. Note that this only works for some functions.

```
?print
```

To create a random n x m sized matrix with integer values from a to b,

```
A = rand(a:b, n, m)
```

To create a copy of a matrix A, do the following. Note that changes to A will not affect B

```
B = copy(A)
```

To merge columns of A and B, (kind of like an augmented matrix),

```
C = [A B]
```

To multiply and transpose matrices,

```
C = A*B
A_t = A'
```

To solve the system Ax = B for x,

```
x = A\B
```

Note: If A is tall (more rows than columns), Julia returns a least squares solution.
If A is short (more columns than rows), Julia returns a minimum norm solution
If A is singular (det(A) = 0), then Julia returns a smallest norm solution.
For advanced Linear Algebra, see the Standard Library LinearAlgebra library:
https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/

For info on Plotting in Julia, see Plots.jl: http://docs.juliaplots.org/

Footnotes:
Background by Michael Atkins on Toptal. Image is licensed under CC BY-SA 3.0 License.
https://www.toptal.com/designers/subtlepatterns/cutcube/ (License is in footer)
Fonts used: Roboto Mono, Poppins on Google Fonts, and Inconsolata (for Code Blocks) licensed under the Apache license 2.0, Open Font license and Open Font License respectively.
https://fonts.google.com/specimen/Poppins
https://fonts.google.com/specimen/Roboto+Mono
https://fonts.google.com/specimen/Inconsolata
Thanks to:
Julia Academy and Jane Herriman (for the course)