# hello_world

June 22, 2021

## 1 Getting Started with Jupyter Notebooks

Hi! Welcome to Jupyter notebooks. These documents allow you to write notes, add images, display figures and plots, and **write source code** all within a single document. We will almost exclusively work within these notebooks for the week. So it is important to become familiar with the essential features of these documents.

You are probably viewing this file within your web browser and there is another tab title "Home", which contains a list of files and folders that you can select to open. You will want to keep both of these open so that you can navigate your files and create new ones of your own.

Let's get started with some of the basics that we will need for the course. Before we get started let's make sure that everything is working okay. Execute the line of Julia code below by either putting your cursor in the cell and typing "Shift+Enter" or clicking on the run command above.

**Resources**

- Julia By Example

[ ]:
```
# Simple test to make sure everything is working...
yourName   = readline()
println("Hi $yourName ! Welcome to Julia (v $VERSION).")
```

**Did it work?**

If you see a statement that looks like this

```
Hi Your Name ! Welcome to Julia (v X.X.X)
```

then you are successfully up and running! If not, then please ask for some assistance because something might not be working like it should be.

### 1.1 Importing Packages

We installed a couple of packages that we would like to use throughout the week. A package is a collection of code written in the native language, in this case Julia, that we can load into our notebooks and use their intended functionallity. Here we would like to load DFTK.jl (density functional toolkit), which we will use to perform some calculations later on this week.

Importing packages is really in Julia. Try executing the commands below. These are pretty heavy packages meaning it might take a few seconds to load them into our notebook.

[ ]:
```julia
# using tells Julia that we would like to load this package
using DFTK
using Plots
```

Next to the cell you should see `In [*]` the * denotes the notebook is excuting this cell of code currently. Once the * changes into an integer this means the cell has finished executing the code within this cell. If no errors occur this means we have successfully installed and loaded the packages need for the week. If an error is returned, please ask for some assistance.

## 1.2 Simple Functions

Functions are useful when you would like to write some code to do one thing generically on various different types of input. We will use functions to make our code cleaner to read as well as automate some of our tasks. In the next couple of blocks we introduce the syntax of the Julia functions through example. The basic architecture of Julia function looks like this

```julia
function NameOfYourFunction(argument1, argument2, ...)
         # code to perform some kind of action with the arguments given



         return #return the output from the operations you just performed
end
```

Now that we know what a skeleton of a function looks like, let's try to write our own. Consider that you are working on your math homework. Your homework has asked you to solve for the roots of many quadratic equations, which have the form

$$0 = ax^2 + bx + c,$$

which we know has a closed form solution

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Rather, than crunching the numbers by hand let's write a function that can do this for us and output the answer.

[ ]:
```julia
function solve_quadratic(a, b, c)
    # solve_quadratic - finds the roots of the quadratic equation: ax^2 + bx +␣
 ↪c = 0
    # Input: a - coefficient for x^2 term,
    #        b - coefficient for x term,
    #        c - coefficient for constant term
    # Output: x1, x2 - the two roots from the quadratic equation.
    okay = false
    try
        sqrt(b*b - 4*a*c)
        okay = true
    catch
        println("Something went wrong with the sqrt. Possibly complex number␣
↪encountered.")
    end
    if okay
        x_minus = (-b - sqrt(b*b - 4*a*c))/(2*a)
        x_plus = (-b + sqrt(b*b - 4*a*c))/(2*a)
        println("Roots of 0 = $a x^2 + $b x+ $c are: $x_minus, $x_plus")
        return x_plus, x_minus
    end
end;
```

[ ]:
```
solve_quadratic(2, 3, 1);
```

## 1.3 Exercise

Write a function that will tell you all you need to know about a circle given it's radius. The function should print out the circle's radius, diameter, area, and circumference as well as return all of these.

```
function circleProp(r)
        # circleProp - determines the properties of a circle
        # Input: r - the radius of the circle

        # calculate diameter

        # calculate area

        # calculate circumference
        return r, d, area, circumference
 end
```

Since you need to use the number $\pi$ in your solution, it is useful to know an interesting property of the Julia language. It has this number stored inside of itself and it can be accessed by either calling `pi` or typing "" + pi"tab". This will render the Greek letter pi. Try it out!

[ ]:
```
# Example of Greek letter' in Julia
println( )
println(pi)
```

## 1.4 Arrays

We will need to work with arrays a fair amount as well. Arrays are simply lists to hold onto information for us. They can hold onto integers, numbers, or strings (i.e., words). Below are a few examples of different kinds of arrays.

[ ]:
```
array = [1, 2, 3] # a simple array
```

```
[ ]: array = collect(1:100) # note that if you don't want something to output use a␣
      ↪";" at the end of the line
     array = collect(1:100);
```

```
[ ]: empty_array = []
     # now add to your empty array
     append!(empty_array, 1.0)
     println(empty_array)
     append!(empty_array, 100.0)
     println(empty_array)
```

```
[ ]: # To access an element of an array use square brackets next to the name of the␣
      ↪array and provide the index of
     # the element that you would like to access
     println("The first element in empty_array is $(empty_array[1])")
     println("The last element in array is $(array[end])")
```

## 1.5 Dictionaries

Dictionaries are a special kind of object, which like a language dictionary stores keys and definitions. In an english dictionary, the words are the keys and the definitions are the values. We will use dictionaries to store related information into one object then call the information out of this using the names of the keys. Let's look into the syntax of Julia dictionaries.

```
[ ]: a1 = Dict(1=>"one", 2=>"two")
     println(a1[1])
```

```
[ ]: person = Dict("height" => 5.6, "weight" => 130, "name" => "Sally", "eye" =>␣
      ↪"brown", "age" => 22)
     println(person["name"])
     println(person["age"])
     println(person["eye"])
```

## 1.6 4 Loops and Conditionals

The most widely used part of any programming language are for loops and conditional statements. The basic structure of for loop in Julia is

```
for index in array_to_loop_through
        # perform some operation on index
end
```

Conditional statements are sections of code which check if a certain condition is true or false and knows what to do if the condition is true and what to do if it is false. A if/else statement looks like this

```
condition = true

if condition
```

```
        # do something
    else
        # do something different (or just don't do anything)
    end
```

## 1.7 Exercise

Write a for loop to square all the elements in an array of numbers 1 to 25.

## 1.8 Exercise

Write a for loop with a conditional statement inside that checks whether the number is even or odd. If the number is even square it. If the number is odd do not square it. Note that Julia has an iseven function. Look the function up in the Julia documentation. This is a good habit to learn because when writing code, we will always be needing to look up the syntax and arguments of functions that we would like to use.