

BEE 4750 Lab 1: Julia Basics

Due Date

Friday, 9/1/23, 9:00pm

Introduction

Julia

Julia is an up-and-coming language, originally developed for scientific programming. While learning a new programming language always has its hiccups, the good news is that if you've programmed in a high-level language such as Python or MATLAB, most Julia concepts should look familiar.

If you have not successfully set up Julia, follow the instructions in [Application Exercise 1](#) and/or ask for help.

You can use other editors for this course, but our recommendation is [Visual Studio Code](#) with the [Julia extension](#), which will make life a *lot* simpler! You should have set this up as part of [Application Exercise 1](#), but if not, do so now and/or ask for help.

Jupyter Notebooks

Jupyter notebooks integrate text and equations in Markdown with Julia (or Python, or R) code. To do this, Jupyter notebooks consist of two types of “cells”: code cells and Markdown (text) cells.

Click once on this section of text. A box will appear around this text (and some areas above/below it) - all of that is within this cell.

Markdown is a text markup framework for formatting language that makes things look pretty when viewed across different platforms: web browsers, notebooks, and so forth. Text written in Markdown can also include hyperlinks, LaTeX equations, section headers, and images, among

other features (most of [the course website](#) and the lecture notes were all written in Markdown!). [Here is a basic Markdown cheat sheet](#).

What you are looking at right now is the formatted text after the Markdown is processed. To see the raw Markdown, do one of:

- press **Enter** while that cell is selected, or
- double-click on that cell.

A couple of the features you will see in this Markdown cell:

- The `---` command creates a horizontal line. This is also nice for separating sections.
- Backticks (``...``) can be used to format and highlight code, keystrokes, etc.
- The `#` sign is used to create a new section header; two `#` signs (`##`) is used to create a new subsection header; `###` creates a subsubsection, and so on.
- You can create a bulleted list by using the asterisk `*` or a dash `-` and a space.
- You can create regular text by just typing as usual.
- You can create **bold-faced text** by wrapping it with two asterisks on both sides.
- You can create *italicized text* by wrapping it with a single asterisk on both sides.
- To create a new paragraph, you must include a blank line between the old and new paragraphs.

At this point you might be wondering how to turn this cell back into the fully formatted Markdown text instead of the raw Markdown you're probably still looking at. You have a couple of options, depending on your platform, but the most consistent is to type **Shift + Enter** to **execute** the cell (this is also how to run code, but more on that later).

Additionally, you will frequently need to create new cells in your Jupyter notebooks. How you do this will depend on how you interact with the notebook, but try to figure this out now.

One tip is to think carefully about what bits of code should be in the same cell, as you typically only see output from the last command in a cell. For example, compare the following:

```
x = 5
sin(x)
```

-0.9589242746631385

with

```
x = 5
```

```
sin(x)
```

```
-0.9589242746631385
```

In Julia, you can also suppress the output of a command with a semi-colon:

```
sin(x);
```

which can help if you want to split some code out for clarity or to insert some text prior to it, but don't want to clutter the notebook with its output.

For code cells, to execute the commands within the cell, we also press **Shift+Enter**.

Finally, **make sure that you evaluate all of the code cells in order before submitting**. One bad outcome with notebooks occurs when cells are evaluated out of order, so fixed bugs and edits in previous cells do not get a chance to propagate down. You can do this with the **Run All** command in whichever interface you're using to edit your notebook.

Julia Basics

There are many tutorials and references for Julia, including a [basics overview on the class website](#). Please feel free to reference these as you work through any part of the course.

Formatting Math

It will often be helpful to include nicely-formatted mathematics in a notebook. Markdown accomodates this using LaTeX syntax. A LaTeX cheatsheet is available on the class website, and many other guides exist online.

Below is an example of a formatted equation:

$$x = 5.$$

Looking For Help

There is no shame in using Google, or other resources, for help when programming. There are many, many times when you can't quite get the syntax to work, can't quite figure out the right package or command to use, or are feeling too lazy or overwhelmed (I'm not judging either of those!) to dig through the documentation. Some good resources include:

- [Stack Overflow](#) is a treasure trove of answers;
- The [official Julia forum](#) and the [Julia Subreddit](#) are also very useful.

You are also highly encouraged to post on [Ed Discussion](#), though getting a response might be less immediate. Just be mindful that to get good answers, [you have to help people help you](#), and **make sure to give credit to any resources that were helpful!**

Exercises

Use your understanding of Julia syntax to complete the following (hopefully short) exercises. Convert your completed notebook to a PDF and submit it to the Gradescope Assignment "Lab 1".

Remember to:

- Evaluate all of your code cells, in order (using a Run All command). This will make sure all output is visible and that the code cells were evaluated in the correct order.
- Tag each of the problems when you submit to Gradescope; a 10% penalty will be deducted if this is not done.

Exercise 1

Given two numeric arrays `x` and `y`, write a function to compute their dot product if they have equal length, and return an error if not (this is useful for debugging!). Use the following code as a starting point.

```
function dot_product(x, y)
    if ... # insert test condition for equal lengths
        # compute and return dot product
    else
        throw(DimensionMismatch("length of x not equal to length of y"))
    end
end
```

Here are some tests to make sure your code works as intended. Tests like these are useful to make sure everything works as intended. One reason to split your code up into functions is

that it makes it straightforward to write tests to make sure each piece of your code works, which makes it easier to identify where errors are occurring.

```
dot_product([1 2 3], [4 5 6])
```

If you know the value you should get, you can write a more formal test using the `@assert` macro, which is a good way to “automate” checking (since you get an obvious error if the code doesn’t work as desired):

```
@assert dot_product([1 2 3], [4 5 6]) == 32
```

Let’s also make sure we get an error when the dimensions of the two vectors don’t match:

```
dot_product([1 2 3], [4 5])
```

Exercise 2

This problem involves implementing Newton’s method for computing square roots; it was shamelessly copied from MIT’s [Introduction to Computational Thinking](#). You will use a `while` loop instead of the `for` loop shown above.

Implement the following algorithm in a function `newton_sqrt`:

Given $x > 0$:

1. Take a guess a .
2. Divide x by a .
3. Update a as the average of x/a and a .
4. Repeat until x/a is within a tolerance of ε from a .
5. Return $a \approx \sqrt{x}$.

```
# insert your code here
```

Use your `newton_sqrt` function to compute $\sqrt{2}$ to within a tolerance of $\varepsilon = 0.01$.

```
# insert your code here
```

Exercise 3: Making a Plot

Write a function to compute the square of an integer x . Evaluate this function for integers between $x = -5$ and $x = 5$ and make a plot of the squared values (you can find a quick guide to making various types of plots [here](#)). Make sure to label your axes.

```
# insert your code here
```