```
In [ ]: # Initialize Otter
        import otter
        grader = otter.Notebook("lab01.ipynb")
```

1 Lab 01

Welcome to the first lab of Data 100! This lab is meant to help you familiarize yourself with JupyterHub, review Python and `numpy`, and introduce you to `matplotlib`, a Python visualization library.

To receive credit for a lab, answer all questions correctly and submit before the deadline.

This lab is due Monday, June 27 at 11:59 PM.

1.0.1 Collaboration Policy

Data science is a collaborative activity. While you may talk with others about the labs, we ask that you **write your solutions individually**. If you do discuss the assignments with others please **include their names** below. (That's a good way to learn your classmates' names.)

Collaborators: *list collaborators here*

1.1 Part 1: Jupyter Tips

1.1.1 Viewing Documentation

To output the documentation for a function, use the `help` function.

```
In [1]: help(print)
```

```
Help on built-in function print in module builtins:

print(…)
    print(value, …, sep=' ', end='\n', file=sys.stdout, flush=False)
```

```
Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file:  a file-like object (stream); defaults to the current sys.stdout.
sep:   string inserted between values, default a space.
end:   string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
```

You can also use Jupyter to view function documentation inside your notebook. The function must already be defined in the kernel for this to work.

Below, click your mouse anywhere on the `print` block below and use `Shift` + `Tab` to view the function's documentation.

```
In [2]: print('Welcome to Data 100.')
```

```
Welcome to Data 100.
```

1.1.2 Importing Libraries and Magic Commands

In Data 100, we will be using common Python libraries to help us process data. By convention, we import all libraries at the very top of the notebook. There are also a set of standard aliases that are used to shorten the library names. Below are some of the libraries that you may encounter throughout the course, along with their respective aliases.

```
In [3]: import pandas as pd
        import numpy as np
        import matplotlib.pyplot as plt
        plt.style.use('fivethirtyeight')
        %matplotlib inline
```

`%matplotlib inline` is a Jupyter magic command that configures the notebook so that Matplotlib displays any plots that you draw directly in the notebook rather than to a file, allowing you to view the plots upon executing your code. (Note: In practice, this is no longer necessary, but we're showing it to you now anyway.)

Another useful magic command is `%%time`, which times the execution of that cell. You can use this by writing it as the first line of a cell. (Note that `%%` is used for *cell magic commands* that apply to the entire cell, whereas `%` is used for *line magic commands* that only apply to a single line.)

```
In [4]: %%time
```

```
lst = []
for i in range(100):
    lst.append(i)
```

```
CPU times: user 13 µs, sys: 4 µs, total: 17 µs
Wall time: 20.7 µs
```

1.1.3 Keyboard Shortcuts

Even if you are familiar with Jupyter, we strongly encourage you to become proficient with keyboard shortcuts (this will save you time in the future). To learn about keyboard shortcuts, go to **Help –> Keyboard Shortcuts** in the menu above.

Here are a few that we like: 1. `Ctrl` + `Return` (or `Cmd` + `Return` on Mac): *Evaluate the current cell* 1. `Shift` + `Return`: *Evaluate the current cell and move to the next* 1. `ESC` : *command mode* (may need to press before using any of the commands below) 1. `a` : *create a cell above* 1. `b` : *create a cell below* 1. `dd` : *delete a cell* 1. `z` : *undo the last cell operation* 1. `m` : *convert a cell to markdown* 1. `y` : *convert a cell to code*

1.2 Part 2: Prerequisites

It's time to answer some review questions. Each question has a response cell directly below it. Most response cells are followed by a test cell that runs automated tests to check your work. Please don't delete questions, response cells, or test cells. You won't get credit for your work if you do.

If you have extra content in a response cell, such as an example call to a function you're implementing, that's fine. Also, feel free to add cells between the question cells and test cells (or the next cell, for questions without test cases). Any extra cells you add will be considered part of your submission. Finally, when you finish an assignment, make sure to "restart and run all cells" to ensure everything works properly.

Note that for labs, ontime submissions that pass all the test cases will receive full credit. However for homeworks, test cells don't always confirm that your response is correct. They are meant to give you some useful feedback, but it's your responsibility to ensure your response answers the question correctly. There may be other tests that we run when scoring your notebooks. We **strongly recommend** that you check your solutions yourself rather than just relying on the test cells.

1.2.1 Python

Python is the main programming language we'll use in the course. We expect that you've taken CS 61A, Data 8, or an equivalent class, so we will not be covering general Python syntax. If any of the following exercises are challenging (or if you would like to refresh your Python knowledge), please review one or more of the following materials.

- **Python Tutorial**: Introduction to Python from the creators of Python.
- **Composing Programs Chapter 1**: This is more of a introduction to programming with Python.
- **Advanced Crash Course**: A fast crash course which assumes some programming background.

1.2.2 NumPy

NumPy is the numerical computing module introduced in Data 8, which is a prerequisite for this course. Here's a quick recap of NumPy. For more review, read the following materials.

- **NumPy Quick Start Tutorial**
- **DS100 NumPy Review**
- **Stanford CS231n NumPy Tutorial**
- **The Data 8 Textbook Chapter on NumPy**

1.2.3 Question 1

The core of NumPy is the array. Like Python lists, arrays store data; however, they store data in a more efficient manner. In many cases, this allows for faster computation and data manipulation.

In Data 8, we used `make_array` from the `datascience` module, but that's not the most typical way. Instead, use `np.array` to create an array. It takes a sequence, such as a list or range.

Below, create an array `arr` containing the values 1, 2, 3, 4, and 5 (in that order).

```
In [5]: arr = np.array([1, 2, 3, 4, 5]) # SOLUTION
```

```
In [ ]: grader.check("q1")
```

In addition to values in the array, we can access attributes such as shape and data type. A full list of attributes can be found here.

```
In [8]: arr[3]
```

```
Out[8]: 4
```

```
In [9]: arr[2:4]
```

```
Out[9]: array([3, 4])
```

```
In [10]: arr.shape
```

```
Out[10]: (5,)
```

```
In [11]: arr.dtype
```

```
Out[11]: dtype('int64')
```

Arrays, unlike Python lists, cannot store items of different data types.

```
In [12]: # A regular Python list can store items of different data types
         [1, '3']
```

```
Out[12]: [1, '3']
```

```
In [13]: # Arrays will convert everything to the same data type
         np.array([1, '3'])
```

```
Out[13]: array(['1', '3'], dtype='<U21')
```

```
In [14]: # Another example of array type conversion
         np.array([5, 8.3])
```

```
Out[14]: array([5. , 8.3])
```

Arrays are also useful in performing *vectorized operations*. Given two or more arrays of equal length, arithmetic will perform element-wise computations across the arrays.

For example, observe the following:

```
In [15]: # Python list addition will concatenate the two lists
         [1, 2, 3] + [4, 5, 6]
```

```
Out[15]: [1, 2, 3, 4, 5, 6]
```

```
In [16]: # NumPy array addition will add them element-wise
         np.array([1, 2, 3]) + np.array([4, 5, 6])
```

```
Out[16]: array([5, 7, 9])
```

1.2.4 Question 2

Question 2a Write a function `summation` that evaluates the following summation for $n \geq 1$:

$$\sum_{i=1}^{n} i^3 + 3i^2$$

Note: You should not use `for` loops in your solution. Check the NumPy documentation. If you're stuck, try a search engine! Searching the web for examples of how to use modules is very common in data science. You may find `np.arange` helpful for this question!

```
In [17]: def summation(n):
             """Compute the summation i^3 + 3 * i^2 for 1 <= i <= n."""
             # BEGIN SOLUTION
             if n < 1:
                 raise ValueError("n must be greater than or equal to 1")
             return sum((np.arange(1, n + 1) ** 3) + (3 * np.arange(1, n + 1) ** 2))
             # END SOLUTION
```

```
In [ ]: grader.check("q2a")
```

Question 2b Write a function `elementwise_array_sum` that computes the square of each value in `list_1`, the cube of each value in `list_2`, then returns a list containing the element-wise sum of these results. Assume that `list_1` and `list_2` have the same number of elements, do not use for loops.

The input parameters will both be **Python lists**, so you may need to convert the lists into arrays before performing your operations. The output should be a **NumPy array.**

```
In [21]: def elementwise_array_sum(list_1, list_2):
             """Compute x^2 + y^3 for each x, y in list_1, list_2.

             Assume list_1 and list_2 have the same length.

             Return a NumPy array.
             """
             assert len(list_1) == len(list_2), "both args must have the same number of elements"
             # BEGIN SOLUTION
             # Solution 1
             return np.square(list_1) + np.power(list_2, 3)
             # Solution 2
             return np.array(list_1) ** 2 + np.array(list_2) ** 3
             # END SOLUTION
```

```
In [ ]: grader.check("q2b")
```

You might have been told that Python is slow, but array arithmetic is carried out very fast, even for large arrays. Below is an implementation of the above code that does not use NumPy arrays.

```
In [29]: def elementwise_list_sum(list_1, list_2):
             """Compute x^2 + y^3 for each x, y in list_1, list_2.

             Assume list_1 and list_2 have the same length.
             """

             return [x ** 2 + y ** 3 for x, y in zip(list_1, list_2)]
```

For ten numbers, `elementwise_list_sum` and `elementwise_array_sum` both take a similar amount of time.

```
In [30]: sample_list_1 = list(range(10))
         sample_array_1 = np.arange(10)
```

```
In [31]: %%time
         elementwise_list_sum(sample_list_1, sample_list_1)
```

```
CPU times: user 12 µs, sys: 4 µs, total: 16 µs
Wall time: 18.6 µs
```

```
Out[31]: [0, 2, 12, 36, 80, 150, 252, 392, 576, 810]
```

```
In [32]: %%time
         elementwise_array_sum(sample_array_1, sample_array_1)
```

```
CPU times: user 42 µs, sys: 13 µs, total: 55 µs
Wall time: 60.6 µs
```

```
Out[32]: array([  0,   2,  12,  36,  80, 150, 252, 392, 576, 810])
```

The time difference seems negligible for a list/array of size 10; depending on your setup, you may even observe that `elementwise_list_sum` executes faster than `elementwise_array_sum`! However, we will commonly be working with much larger datasets:

```
In [33]: sample_list_2 = list(range(100000))
         sample_array_2 = np.arange(100000)
```

```
In [34]: %%time
         elementwise_list_sum(sample_list_2, sample_list_2)
         ; # The semicolon hides the output
```

```
CPU times: user 66.2 ms, sys: 7.12 ms, total: 73.3 ms
Wall time: 71.6 ms
```

```
Out[34]: '# The semicolon hides the output'
```

```
In [35]: %%time
         elementwise_array_sum(sample_array_2, sample_array_2)
         ;
```

```
CPU times: user 2.5 ms, sys: 1.17 ms, total: 3.67 ms
Wall time: 2.2 ms
```

```
Out[35]: ''
```

With the larger dataset, we see that using NumPy results in code that executes over 50 times faster! Throughout this course (and in the real world), you will find that writing efficient code will be important; arrays and vectorized operations are the most common way of making Python programs run quickly.

Question 2c Recall the formula for population variance below:

$$\sigma^2 = \frac{\sum_{i=1}^{N}(x_i - \mu)^2}{N}$$

Complete the functions below to compute the population variance of `population`, an array of numbers. For this question, **do not use built in NumPy functions, such as `np.var`.** Again, avoid using `for` loops!

```
In [36]: def mean(population):
             """
             Returns the mean of population (mu)

             Keyword arguments:
             population -- a numpy array of numbers
             """
             # Calculate the mean of a population
             return sum(population) / len(population) # SOLUTION

         def variance(population):
             """
             Returns the variance of population (sigma squared)

             Keyword arguments:
             population -- a numpy array of numbers
             """
             # Calculate the variance of a population
             # BEGIN SOLUTION
             m = mean(population)
             return sum((population - m) ** 2) / len(population)
             # END SOLUTION
```

```
In [ ]: grader.check("q2c")
```

Question 2d Given the array `random_arr`, assign `valid_values` to an array containing all values x such that $2x^4 > 1$.

Note: You should not use `for` loops in your solution. Instead, look at `numpy`'s documentation on Boolean Indexing.

```
In [41]: np.random.seed(42)
         random_arr = np.random.rand(60)
         valid_values = random_arr[(2 * random_arr ** 4 > 1)] # SOLUTION
```

```
In [ ]: grader.check("q2d")
```

1.3 Part 3: Plotting

Here we explore plotting using `matplotlib` and `numpy`.

1.3.1 Question 3

Consider the function $f(x) = x^2$ for $-\infty < x < \infty$.

Question 3a Find the equation of the tangent line to f at $x = 0$.

Use LaTeX to type your solution, such that it looks like the serif font used to display the math expressions in the sentences above.

HINT: You can click any text cell to see the raw Markdown syntax.

Type your answer here, replacing this text.

SOLUTION:

Just by visualizing the graph of f you can see this is the horizontal line $y = 0$.

Question 3b Find the equation of the tangent line to f at $x = 8$. Please use LaTeX to type your solution.

Type your answer here, replacing this text.

SOLUTION:

The tangent line passes through $(8, f(8))$ and has slope $\frac{d}{dx}f(x) = 2x$ evaluated at $x = 8$. So its equation is $y - 64 = 16(x - 8)$ which is $y = 16x - 64$.

Question 3c Write code to plot the function f, the tangent line at $x = 8$, and the tangent line at $x = 0$.

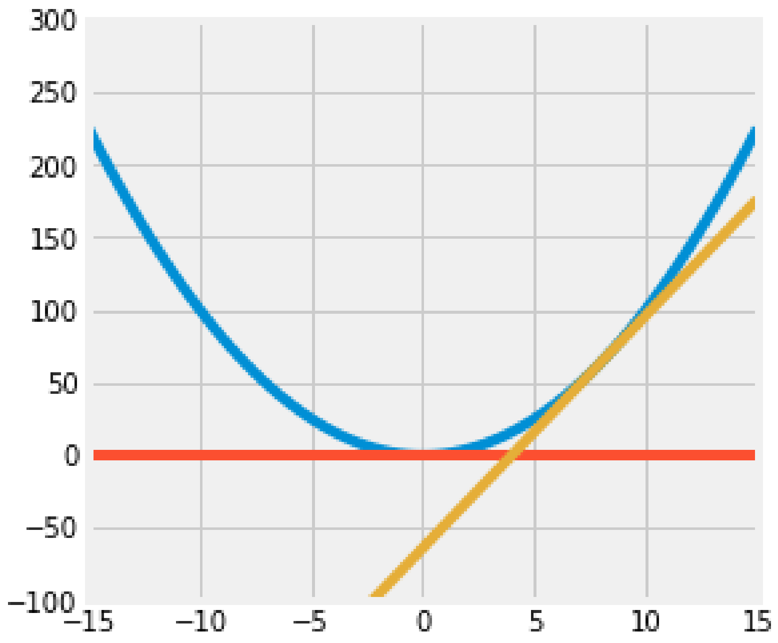
Set the range of the x-axis to (-15, 15) and the range of the y-axis to (-100, 300) and the figure size to (4,4).

Your resulting plot should look like this (it's okay if the colors in your plot don't match with ours, as long as they're all different colors):

You should use the `plt.plot` function to plot lines. You may find the following functions useful:

- `plt.plot(..)`
- `plt.figure(figsize=..)`
- `plt.ylim(..)`
- `plt.axhline(..)`

```
In [45]: def f(x):
             return x ** 2 # SOLUTION

         def df(x):
             return 2 * x # SOLUTION

         def plot(f, df):
             # BEGIN SOLUTION
             x = np.linspace(-15, 15, 100)

             plt.figure(figsize = (4, 4))
             plt.plot(x, f(x))
             plt.plot(x, np.zeros(x.shape[0]))
             plt.plot(x, df(8) * (x - 8) + f(8))

             plt.xlim(-15, 15)
             plt.ylim(-100, 300)
             # END SOLUTION

         plot(f, df)
```



1.3.2 Question 4 (Ungraded)

Data science is a rapidly expanding field and no degree program can hope to teach you everything that will be helpful to you as a data scientist. So it's important that you become familiar with looking up documentation and learning how to read it.

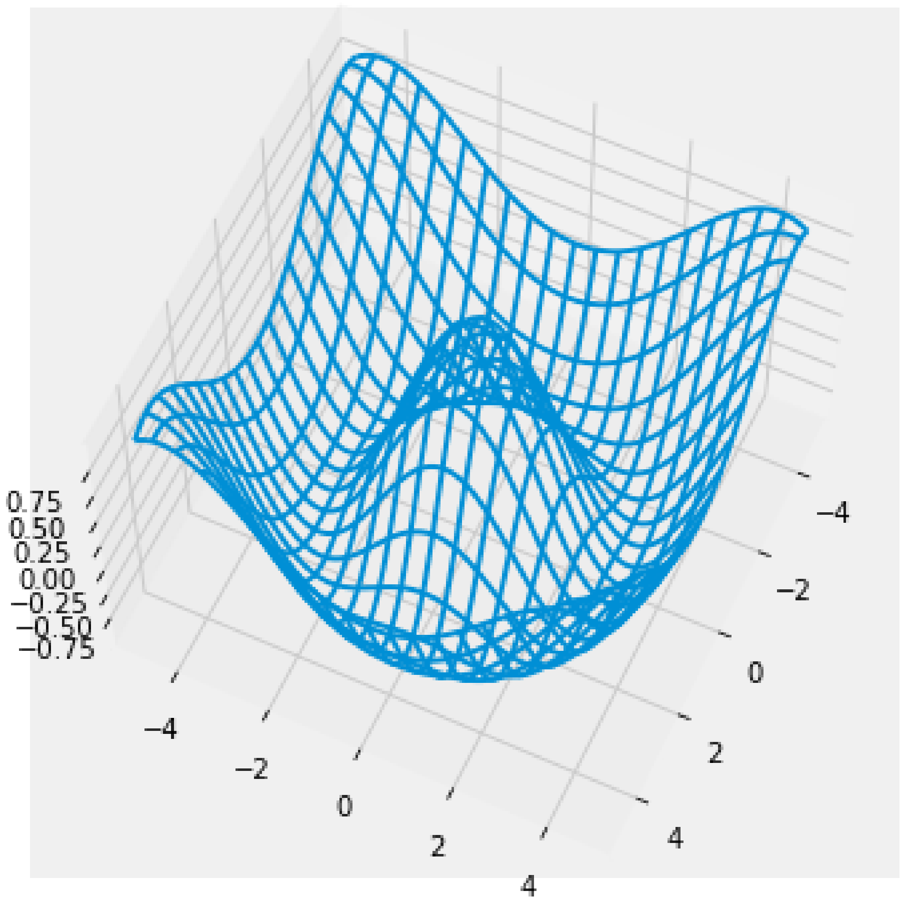
Below is a section of code that plots a three-dimensional "wireframe" plot. You'll see what that means when you draw it. Replace each `# Your answer here` with a description of what the line above does, what the arguments being passed in are, and how the arguments are used in the function. For example,

```
np.arange(2, 5, 0.2)
# This returns an array of numbers from 2 to 5 with an interval size of 0.2
```

Hint: The `Shift` + `Tab` tip from earlier in the notebook may help here. Remember that objects must be defined in order for the documentation shortcut to work; for example, all of the documentation will show for method calls from `np` since we've already executed `import numpy as np`. However, since `z` is not yet defined in the kernel, `z.reshape(x.shape)` will not show documentation until you run the line `z = np.cos(squared)`.

```
In [46]: from mpl_toolkits.mplot3d import axes3d
```

```
u = np.linspace(1.5 * np.pi, -1.5 * np.pi, 100)
# Your answer here
[x, y] = np.meshgrid(u, u)
# Your answer here
squared = np.sqrt(x.flatten() ** 2 + y.flatten() ** 2)
z = np.cos(squared)
# Your answer here
z = z.reshape(x.shape)
# Your answer here

fig = plt.figure(figsize = (6, 6))
ax = fig.add_subplot(111, projection = '3d')
# Your answer here
ax.plot_wireframe(x, y, z, rstride = 5, cstride = 5, lw = 2)
# Your answer here
ax.view_init(elev = 60, azim = 25)
# Your answer here
plt.savefig("figure1.png")
# Your answer here
```



1.3.3 Question 5 (Ungraded)

Do you think that eating french fries with mayonnaise is a crime?
Tell us what you think in the following Markdown cell. :)

To double-check your work, the cell below will rerun all of the autograder tests.

```
In [ ]: grader.check_all()
```

1.4 Submission

Make sure you have run all cells in your notebook in order before running the cell below, so that all images/graphs appear in the output. The cell below will generate a zip file for you to submit. **Please save before exporting!**

```
In [ ]: # Save your notebook first, then run this cell to export your submission.
        grader.export(pdf=False)
```