```
In [ ]: # Initialize Otter
        import otter
        grader = otter.Notebook("lab0-rev.ipynb")
```

1 Lab 0 Basics of using python notebook

Welcome to the labs of Demog 180. This lab is designed to help students who would like to have some hands-on tutorial with the Python language and use of Jupyter notebook.

Lab 0 is constructed based on lab 1 and lab 2 in Data 8.

Good resources for knowledge, tutorial and training in Python:

1. The website for Data8: http://data8.org/
   A. Datascience documentation: http://data8.org/datascience/#
   B. Textbook for Data 8: Computational and Inferential thinking https://www.inferentialthinking.com/chapters/intro.html

2. Cheatsheet for syntext: https://www.datacamp.com/community/data-science-cheatsheets.
3. The D-Lab offers a short workshop on Python Fundamentals a few times each semester. You can participate in the virtual workshop: https://dlab.berkeley.edu/events/python-fundamentals-parts-1-3/2023-09-11. You can also work on your own through the materials available the D-Lab's GitHub repository: https://github.com/dlab-berkeley/Python-Fundamentals

Lab policies:

1. If you want to practice Jupyter notebook and Python language, you can go through the materials and questions and submit lab 0.

2. For the labs in this semester, you are not required to attend the lab sessions but **required to submit all labs (except for lab 0).**

3. Labs are not graded based on the accuracy of your answers, but on effort; to get full credit, you have to make a good faith attempt at solving all of the lab. Labs are intended to help prepare you for the homework, so completing the labs will also help make the homework easier.

4. You are welcome to ask any questions during lab sections and on Ed. We strongly encourage students to help each other with labs and try new things. When posting on Ed, it is helpful if you include a screenshot of the question you need help with, what you've already tried, and a short explanation of what is confusing. For labs only, it is ok to share your code on Ed. **Do not share homework code on Ed.**

Let's get started.

Question: What is your name and hometown?

Type your answer here, replacing this text.

Solution: (Your name and hometown)

2 1. Jupyter Notebook

This webpage is called a Jupyter notebook. A notebook is a place to write programs and view their results.

2.1 1.1. Text cells

In a notebook, each rectangle containing text or code is called a cell.

Text cells (like this one) can be edited by double-clicking on them. They're written in a simple format called *Markdown* to add formatting and section headings. You don't need to learn Markdown, but you might want to.

After you edit a text cell, click the "run cell" button at the top that looks like | to confirm any changes. (Try not to delete the instructions of the lab.)

Practice This paragraph is in its own text cell. Try editing it so that this sentence is the last sentence in the paragraph above, and then click the "run cell" | button .

2.2 1.2 Code cells

Other cells contain code in the Python 3 language. Running a code cell will execute all of the code it contains.

To run the code in a code cell, first click on that cell to activate it. It'll be highlighted with a little green or blue rectangle. Next, either press | or hold down the shift key and press return or enter.

Try running this cell:

```
In [2]: print("Hello, World!")
```

```
Hello, World!
```

And this one:

```
In [3]: print("\N{WAVING HAND SIGN}, \N{EARTH GLOBE ASIA-AUSTRALIA}!")
```

```
 ,  !
```

The fundamental building block of Python code is an expression. Cells can contain multiple lines with multiple expressions. When you run a cell, the lines of code are executed in the order in which they appear. Every print expression prints a line. Run the next cell and notice the order of the output.

```
In [4]: print("First this line is printed,")
        print("and then this one.")
```

```
First this line is printed,
and then this one.
```

2.3 1.3. Writing Jupyter notebooks

You can use Jupyter notebooks for your own projects or documents. When you make your own notebook, you'll need to create your own cells for text and code.

To add a cell, click the + button in the menu bar. You can change the cell to a code cell or marktown (text) cell by clicking inside it so it's highlighted, clicking the drop-down box next to the restart ( ) button in the menu bar, and choosing "Code" or "Markdown".

Practice Add a code cell below this one. Write code in it that prints out:

```
The quick brown fox jumps over the lazy dog.
```

Run your cell to verify that it works.

2.4 1.4. The Kernel

The kernel is a program that executes the code inside your notebook and outputs the results. In the top right of your window, you can see a circle that indicates the status of your kernel. If the circle is empty ( ), the kernel is idle and ready to execute code. If the circle is filled in ( ), the kernel is busy running some code.

You may run into problems where your kernel is stuck for an excessive amount of time, your notebook is very slow and unresponsive, or your kernel loses its connection. If this happens, try the following steps: 1. At the top of your screen (between “Cell” and ”Widgets“ , click **Kernel**, then **Interrupt**. 2. If that doesn’t help, click **Kernel**, then **Restart**. If you do this, you will have to run your code cells from the start of your notebook up until where you paused your work. 3. If that doesn’t help, restart your server. First, save your work by clicking **File** at the top left of your screen, then **Save and Checkpoint**. Next, click **Control Panel** at the top right. Choose **Stop My Server** to shut it down, then **My Server** to start it back up. Then, navigate back to the notebook you were working on.

3 2. Numbers and arithmetic

The expression 3.2500 evaluates to the number 3.25. (Run the cell and see.)

```
In [5]: 3.2500
```

```
Out[5]: 3.25
```

You don’t have to `print` the value. When you run a notebook cell, if the last line has a value, then Jupyter helpfully prints out that value for you. However, it won’t print out prior lines automatically. Try the following cell and see the difference.

```
In [6]: print(2)
        3
        4
```

```
2
```

```
Out[6]: 4
```

Above, you should see that 4 is the value of the last expression, 2 is printed, and 3 is lost forever because it was neither printed nor last.

Python can also do arithmetic directly. The line in the next cell subtracts. Its value is what you’d expect. Run it.

```
In [7]: 3.25 - 1.5
```

```
Out[7]: 1.75
```

Many basic arithmetic operations are built in to Python. The textbook section on Expressions describes all the arithmetic operators used in the course.

Note that the common operator that differs from typical math notation is **, which raises one number to the power of the other. So, 2**3 stands for 23 and evaluates to 8. Using ^ to denote power will return an error.

The order of operations is what you learned in elementary school, and Python also has parentheses.

```
In [8]: 6+6*5-6*3**2*2**3/4*7
```

```
Out[8]: -720.0
```

```
In [9]: 6+(6*5-(6*3))**2*((2**3)/4*7)
```

```
Out[9]: 2022.0
```

3.1 3. Checking and commenting your code

Now that you know how to name things, you can start using the built-in *tests* to check whether your work is correct. Try not to change the contents of the test cells (cells that have code like `grader.check(...)`).

Try to answer Question 3.1, below, and then run the `grader.check(...)` cell to see if you got it right. If you haven't, this test will tell you the correct answer. Resist the urge to just copy it, and instead try to adjust your expression. (Sometimes the tests will give hints about what went wrong…)

Question 3.1. Write a Python expression in this next cell that's equal to $5 \times (3\frac{10}{11}) - 50\frac{1}{3} + 2^{.5 \times 22} - \frac{7}{33} + 5$. That's five times three and ten elevenths, minus fifty and a third, plus two to the power of half 22, minus 7 33rds plus five. By "$3\frac{10}{11}$" we mean $3 + \frac{10}{11}$, not $3 \times \frac{10}{11}$.

```
In [10]: # BEGIN SOLUTION NO PROMPT
         q1 = 5 * (3 + 10 / 11) - (50 + 1 / 3) + 2 ** (.5 * 22) - 7/33 + 5
         q1
         # END SOLUTION
         """ # BEGIN PROMPT
         # Replace the ellipses (...) with your code; try to use parentheses only when necessary.
         # Below, we'll test your answer
         q1 = ...
```

```
    q1
    """; # END PROMPT
```

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

You may have noticed these lines in the cell above:

```
# Replace the ellipses (...) with your code; try to use parentheses only when necessary.
# Below, we'll test your answer
```

That is called a *comment*. It doesn't make anything happen in Python; Python ignores anything on a line after a #. Instead, it's there to communicate something about the code to you, the human reader. Comments are extremely useful. In the words of Harold Abelson, a professor of computer science and electrical engineering at MIT, "Programs must be written for people to read, and only incidentally for machines to execute." (Abelson worked on Logo, the programming language used for the Apple II and created MIT's version of CS61A)

3.2 4. Calling functions and nesting

The most common way to combine or manipulate values in Python is by calling functions. Python comes with many built-in functions that perform common operations.

For example, the `abs` function takes a single number as its argument and returns the absolute value of that number. The absolute value of a number is its distance from 0 on the number line, so `abs(5)` is 5 and `abs(-5)` is also 5.

```
In [12]: abs(5)
```

```
Out[12]: 5
```

```
In [13]: abs(-5)
```

```
Out[13]: 5
```

After calling the function `abs`, you get a value 5 from this **expression** abs(-5). You can assign this value to a name by an equal sign so that we could use this value in other contexts. This process takes an **assignment statement**. An assignment statement works as we're **evaluating the right-hand side** of the equals sign and **assigning it to the left-hand side**.

The **statements** and **expressions** are the building blocks of Python code.

Question 4.1. Use the `abs` function to calculate how much shorter is A, 1.24 meters tall, from average human height, 1.68 meters tall. Assign the result to the answer variable called q2.

Check your answer.

```
In [14]: # BEGIN SOLUTION NO PROMPT
         q2 = abs(1.24 - 1.68)
         q2
         # END SOLUTION
         """ # BEGIN PROMPT
         q2 = ...
         q2
         """; # END PROMPT
```

```
In [ ]: grader.check("q2")
```

An additional layer of nesting: the function `max` takes two values and return the bigger one, and the `min` function returns the smallest of values. Call functions to answer the following question.

Question 4.2. Among A (1.24m), B (1.53m) and C (1.72m), which one deviates the most from the average height (1.68m)? Assign the biggest distance from average as the answer q3.

```
In [17]: # BEGIN SOLUTION NO PROMPT
         dis_A = abs(1.24 - 1.68)
         dis_B = abs(1.52 - 1.68)
         dis_C = abs(1.72 - 1.68)

         q3 = max(dis_A, dis_B, dis_C)
         # END SOLUTION
         """ # BEGIN PROMPT
         dis_A = abs(1.24 - 1.68)
         dis_B = abs(1.52 - 1.68)
         dis_C = abs(1.72 - 1.68)

         q3 = max(...)
         """; # END PROMPT
```

```
In [ ]: grader.check("q3")
```

4 5. Importing code and functions

Most programming involves work that is very similar to work that has been done before. Since writing code is time-consuming, it's good to rely on others' published code when you can. Rather than copy-pasting, Python allows us to **import** other code, creating a **module** that contains all of the names created by that code.

Python includes many useful modules that are just an `import` away. We'll look at the `math` module as a first example. The `math` module is extremely useful in computing mathematical expressions in Python.

Suppose we want to very accurately compute the area of a circle with radius 5 meters. For that, we need the constant π, which is roughly 3.14. Conveniently, the `math` module has `pi` defined for us:

```
In [19]: import math
         radius = 5
         area_of_circle = radius**2 * math.pi
```

`pi` is defined inside `math`, and the way that we access names that are inside modules is by writing the module's name, then a dot, then the name of the thing we want:

```
<module name>.<name>
```

In order to use a module at all, we must first write the statement `import <module name>`. That statement creates a module object with things like `pi` in it and then assigns the name `math` to that module. Above we have done that for `math`. Note that you only need to import a module once

Question 5.1 The module `math` also provides the name `e` for the base of the natural logarithm, which is roughly 2.71. Compute $e^{\pi} - \pi$, giving it the name `near_twenty`.

```
In [20]: # BEGIN SOLUTION NO PROMPT
         near_twenty = math.e ** math.pi - math.pi
         near_twenty
         # END SOLUTION
         """ # BEGIN PROMPT
         near_twenty = ...
         near_twenty
         """; # END PROMPT
```

```
In [ ]: grader.check("q4")
```

Modules can provide other named things, including **functions**. For example, `math` provides the name `sin` for the sine function. Having imported `math` already, we can write `math.sin(3)` to compute the sine of 3.

(Note that this sine function considers its argument to be in radians, not degrees. 180 degrees are equivalent to π radians.)

Question 5.2 A $\frac{\pi}{4}$-radian (45-degree) angle forms a right triangle with equal base and height, pictured below. If the hypotenuse (the radius of the circle in the picture) is 1, then the height is $\sin(\frac{\pi}{4})$.

Compute that using `sin` and `pi` from the `math` module. Give the result the name `sine_of_pi_over_four`.

(Source: Wolfram MathWorld)

```
In [22]: # BEGIN SOLUTION NO PROMPT
         sine_of_pi_over_four = math.sin(math.pi/4)
         sine_of_pi_over_four
         # END SOLUTION
         """ # BEGIN PROMPT
         sine_of_pi_over_four = ...
         sine_of_pi_over_four
         """; # END PROMPT
```

```
In [ ]: grader.check("q5")
```

You can try the other functions in the `math` module: `log`,`factorial`,`sqrt`,etc.

There's many variations of how we can import methods from outside sources. For example, we can import just a specific method from an outside source, we can rename a library we import, and we can import every single method from a whole library.

```
In [24]: #Importing just cos and pi from math.
         #Notice that we don't have to use "math." before cos and pi
         from math import cos, pi
         print(cos(pi))
         #We do have to use it infront of other methods from math, though
         math.log(pi) # Note that the log in math module take the natural logrithm (base e) of a number
```

```
-1.0
```

```
Out[24]: 1.1447298858494002
```

```
In [25]: #We can nickname math as something else, if we don't want to type math
         import math as m
         m.log(m.pi)
```

```
Out[25]: 1.1447298858494002
```

```
In [26]: #Lastly, we can import everything from math
         from math import *
         log(pi)
```

```
Out[26]: 1.1447298858494002
```

5 6. Defining functions

Sometimes the functions written by other people are not convenient enough to serve our purposes. In such case, we can define new functions to simply the coding process.

Let's write a very simple function that converts a proportion to a percentage by multiplying it by 100. For example, the value of `to_percentage(.5)` should be the number 50. (No percent sign)

A function definition has a few parts.

`def` It always starts with `def` (short for **def**ine):

```
def
```

Name Next comes the name of the function. Let's call our function `to_percentage`.

```
def to_percentage
```

Signature Next comes something called the *signature* of the function. This tells Python how many arguments your function should have, and what names you'll use to refer to those arguments in the function's code. `to_percentage` should take one argument, and we'll call that argument `proportion` since it should be a proportion.

```
def to_percentage(proportion)
```

We put a colon after the signature to tell Python it's over.

```
def to_percentage(proportion):
```

Documentation Functions can do complicated things, so you should write an explanation of what your function does. For small functions, this is less important, but it's a good habit to learn from the start. Conventionally, Python functions are documented by writing a triple-quoted string:

```
def to_percentage(proportion):
    """Converts a proportion to a percentage."""
```

Body Now we start writing code that runs when the function is called. This is called the *body* of the function. We can write anything we could write anywhere else. First let's give a name to the number we multiply a proportion by to get a percentage.

```
def to_percentage(proportion):
    """Converts a proportion to a percentage."""
    factor = 100
```

`return` The special instruction `return` in a function's body tells Python to make the value of the function call equal to whatever comes right after `return`. We want the value of `to_percentage(.5)` to be the proportion .5 times the factor 100, so we write:

```
def to_percentage(proportion):
    """Converts a proportion to a percentage."""
    factor = 100
    return proportion * factor
```

Note that `return` inside a function gives the function a value, while `print`, which we have used before, is a function which has no `return` value and just prints a certain value out to the console. The two are **very** different. However, you can use print() if you just want to print out a value and you don't need to store the value in a variable.

```
In [27]: # Define and run the to_percentage function
         def to_percentage(proportion):
             """Converts a proportion to a percentage."""
             factor = 100
             return proportion * factor
         to_percentage(0.18)
```

```
Out[27]: 18.0
```

Here's something important about functions: the names assigned within a function body are only accessible within the function body. Once the function has returned, those names are gone. So even though you defined `factor = 100` inside the body of the `to_percentage` function up above and then called `to_percentage`, you cannot refer to `factor` anywhere except inside the body of `to_percentage`:

```
In [28]: # If you uncomment the line with 'factor' below, you should see an error when you run it.
         # (If you don't, you might have defined factor somewhere above.)

         #factor
```

Question 6.1 Define a function called `abs_log` that takes the absolute value of the log of a positive number. Use this function on the number 0.2021.

```
In [29]: # BEGIN SOLUTION NO PROMPT
         def abs_log(positive):
             return abs(math.log(positive))

         abs_log(0.2021)
         # END SOLUTION
         """ # BEGIN PROMPT
         def abs_log(positive):
             return abs(math.log(positive))

         abs_log(0.2021)
         """; # END PROMPT
```

```
In [ ]: grader.check("q6")
```

6 7. Ifelse and loops

6.1 7.1 Conditional Statements

A conditional statement is composed of a sequence of conditions that allow Python to choose from different alternatives based on whether some condition is true.

Here is a basic example.

```
def sign(x):
    if x > 0:
        return 'Positive'
    else:
        return 'Negative'
```

If the input `x` is greater than `0`, we return the string `'Positive'`. Otherwise, we return `'Negative'`.

If we want to test multiple conditions at once, we use the following general format.

```
if <if expression>:
    <if body>
elif <elif expression 0>:
    <elif body 0>
elif <elif expression 1>:
    <elif body 1>
...
else:
    <else body>
```

Only the body for which the conditional expression is true will be evaluated. Each `if` and `elif` expression is evaluated and considered in order, starting at the top. As soon as a true value is found, the corresponding body is executed, and the rest of the conditional statement is skipped. If none of the `if` or `elif` expressions are true, then the `else body` is executed.

For more examples and explanation, refer to the section on conditional statements here.

Question 7.1 Improve your `abs_log` function that it returns the absolute value of the log of a positive number, and returns an error message saying “Error: the input number should be positive.” Use this function again on the number 0.

```
In [31]: def abs_log_improve(number):
             if number >0:
                 return abs(math.log(positive))
             else:
                 return 'Error: the input number should be positive.' # SOLUTION
```

```
In [ ]: grader.check("q7")
```

6.2 7.2 Loops

Using a `for` statement, we can perform a task multiple times. This is known as iteration. Here is an example that prints out the color of the rainbow.

```
In [33]: from datascience import * # you need the function make_array in datascience module
         rainbow = make_array("red", "orange", "yellow", "green", "blue", "indigo", "violet") #make_arr

         for color in rainbow:
             print(color)
```

```
red
orange
```

```
yellow
green
blue
indigo
violet
```

Here is an example of printing numbers 2 integers apart from the starting number to the end point.

```
In [34]: for x in range(1,11,2): #range(start, end, step), the end point is open-ended aka not included
             print(x)
```

```
1
3
5
7
9
```

Question 7.2 Try to define a function `sd` that takes a list of numbers and calculate the standard deviation of this list of numbers. The standard deviation is a measure that is used to quantify the amount of variation of a set of data values. The formula is: $sd = \sqrt{\frac{\sum(x-\bar{x})^2}{n}}$. Note that $\bar{x}$ is the mean and n is the number of numbers in the list.

Apply this function to an array:(2,4,6,8,10).

Hint1: you need a mean value to calculate the deviation for each number

Hint2: you need to sum up all the results produced by the loop.

```
In [35]: # BEGIN SOLUTION NO PROMPT
         import numpy as np #import the module that contains the mean function

         def sd(array):
             mean_value = np.mean(array)
             sum_dev = 0
             for number in array:
                 dev = number - mean_value
                 sum_dev = sum_dev + dev**2
             return math.sqrt(sum_dev/len(array))

         q8 = sd(make_array(2,4,6,8,10)) # SOLUTION
         # END SOLUTION
         """ # BEGIN PROMPT
         import numpy as np #import the module that contains the mean function
```

```
def sd(array):
    ...


q8 =  ...
"""; # END PROMPT
```

```
In [ ]: grader.check("q8")
```

7 Submitting your work

7.1 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!**

This lab is optional, so you don't have to submit it. But if you want to do so (for practice), please upload the .zip file to Gradescope.

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