```
# Initialize Otter
import otter
grader = otter.Notebook("lab0-gettingstarted.ipynb")
```

Lab 0: Intro to Jupyter notebooks, review of Python and NumPy

Welcome to the first lab of PSTAT 100! This lab is meant to help you familiarize yourself with Jupyter Notebooks and review Python and NumPy.

Objectives

- Keyboard shortcuts, running cells, and viewing documentation in Jupyter Notebooks
- Review functions, lists, and loops
- Review NumPy arrays: indexing, attributes, and operations on arrays

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** and do not copy them from others.

By submitting your work in this course, whether it is homework, a lab assignment, or a quiz/exam, you agree and acknowledge that **this submission is your own work and that you have read the policies regarding Academic Integrity**: https://studentconduct.sa.ucsb.edu/academic-integrity. The Office of Student Conduct has policies, tips, and resources for proper citation use, recognizing actions considered to be cheating or other forms of academic theft, and studentsâ€™ responsibilities. You are required to read the policies and to abide by them.

If you collaborate with others, we ask that you indicate their names on your submission.

Your name and collaborators

Name: Marissa Santiago

Collaborators:

1. 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.

Important Note: 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 currectly answer the question. There may be other/additional 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.

Python

Python is the main programming language we'll use in the course. We expect that you are familiar with it, so we will not be covering general Python syntax. If any of the below 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.

You need to make sure that you are comfortable with the following basic concepts (from CS 8): * Basic datatypes: int, float, string, bool * Variables and assignment statements * Basic arithmetic, including exponentiation `**` and modulo `%` (i.e., the remainder from the division); precedence rules * Boolean values, operators for Boolean logic (`and`, `or`, `not`) * Nested expressions, compound expressions, built-in methods (e.g., `round`, `max`) * Conditional statements / nested conditionals * Defining and running functions, properly using input parameters and distinguishing them from the input arguments * `import`ing modules and specific functions * Documenting code through the use of comments (in-line and block comments) * String concatenation, indexing, slicing, methods (e.g., `upper`/`lower`, `replace`, `strip`) * Loops: `while` and `for`; how to avoid infinite loops * Lists: indexing, slicing, methods (e.g., `append`, `sort`) * Dictionaries: syntax, accessing keys and values, adding new elements * Generating random numbers

Question 1a

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

$$\sum_{i=1}^{n} \left(i^3 + 5i^3\right)$$

```
def summation(n):
    """Compute the summation i^3 + 5 * i^3 for 1 <= i <= n."""

    return sum([i**3+5*i**3 for i in range(1, n+1)])
```

```
grader.check("q1_a")
```

results:

result:

result:

result:

Use your function to compute the sum for...

```
# i = 2
summation(2)
```

```
54
```

```
# i = 20
summation(20)
```

```
264600
```

List comprehension

In Python, normally you can fill a list with elements using a for loop as seen in the example below.

```
four_squares = []
# Add square numbers from 1 to 10000 inclusive to the list "squares" if they end in the digit "4"
for i in range(1,101):
    if (i**2)%10 == 4:
        four_squares.append(i**2)
print(four_squares)
```

```
[4, 64, 144, 324, 484, 784, 1024, 1444, 1764, 2304, 2704, 3364, 3844, 4624, 5184, 6084, 6724, 7744, 846
```

Alternatively, you can create this same list in a single line of code by moving the for-loop and if-statement inside the loop's creation. This is called a **list comprehension**.

The syntax for a list comprehension is this:

$$**value**** ** for - loop **** ** condition **$$

* **value** is the value you want to put into the list * **for-loop** is the for-loop that iterates through a list or a range * **condition** is the if-statement that determines if the **value** is allowed to be inserted into the list

For more information, you can read this beginners tutorial on list comprehensions

```
# value: i**2
# for-loop: for i in range(1,101)
# condition: if (i**2)%10 == 4
four_squares_list_comprehension = [i**2 for i in range(1,101) if (i**2)%10 == 4]
```

```
four_squares_list_comprehension
```

```
[4,
 64,
 144,
 324,
 484,
 784,
 1024,
 1444,
 1764,
 2304,
 2704,
 3364,
 3844,
 4624,
 5184,
 6084,
 6724,
 7744,
 8464,
 9604]
```

Aligning lists with `zip()`

If you need to line up 2 or more lists, Python has a built-in function called **zip()**. This allows you to loop through more than one list at a time such that you get values in each list that have the same index in the other lists. This can also be used within for-loops to make them even more powerful.

```
a = [1,2,3,4]
b = [2,3,4,5]
# Print a[0]*b[0], a[1]*b[1],...
for x, y in zip(a,b):
    print(x*y)
```

```
2
6
12
20
```

Question 1b

Write a function `list_sum` that computes **the square** of *each* value in `list_1`, **the cube** of *each* value in `list_2`, and returns a list containing the element-wise sum of these results. Assume that `list_1` and `list_2` have the same number of elements. Try to use a list comprehension to write it all on one line.

```
def 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.
    """
    assert len(list_1) == len(list_2), "both args must have the same number of elements"
    return ([x**2 + y**3 for x,y in zip(list_1,list_2)])
```

```
grader.check("q1_b")
```

passed!

2. NumPy

NumPy NumPy (pronounced "NUM-pie") is the numerical computing module, which we will be using a lot in this course. Here's a quick recap of NumPy. For more review, read the following materials.

- **NumPy Quick Start Tutorial**
- **Stanford CS231n NumPy Tutorial**

Question 2a

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.

Let's use `np.array` to create an array. It takes a sequence, such as a list or range (remember that list elements are included between the square brackets `[` and `]`).

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

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

```
grader.check("q2_a")
```

passed!

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

Indexing

NumPy arrays are integer-indexed by position, with the first element indexed as position 0. Elements can be retrieved by enclosing the desired positions in brackets `[]`.

```
arr[3] #index starts at 0
```

```
4
```

To retrieve consecutive positions, specify the starting index and the ending index separated by `:` – *e.g.*, `arr[from:to]`. This syntax is non-inclusive of the left endpoint; notice below that the starting index is *not* included in the output.

```
arr[2:4]
```

```
array([3, 4])
```

Attributes

NumPy arrays have several attributes that can be retrieved by name using syntax of the form `arr.attr`. Some useful attributes are:

- `.shape`, a tuple with the length of each array dimension
- `.size`, the length of the first array dimension
- `.dtype`, the data type of the entries (float, integer, etc.)

```
arr.shape
```

```
(5,)
```

```
arr.size
```

```
5
```

```
arr.dtype
```

```
dtype('int64')
```

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

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

```
[1, '3']
```

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

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

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

```
array([5. , 8.3])
```

Operations on arrays

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:

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

```
[1, 2, 3, 4, 5, 6]
```

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

```
array([5, 7, 9])
```

Question 2b

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

```
# for reproducibility - setting the seed will result in the same random draw each time
np.random.seed(42)

# draw 60 uniformly random integers between 0 and 1
random_arr = np.random.rand(60)

# solution here
valid_values = random_arr[2*random_arr**4>1]
```

```
grader.check("q2_b")
```

passed!

Question 2c

Use NumPy to recreate your answer to Question 1b (that computes the **square** of each value in list_1, the **cube** of each value in list_2, and returns a list containing the element-wise sum of these results). The input parameters will both be lists, so you will need to convert the lists into arrays before performing your operations.

Hint: Use 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.

```
def 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. The inputs do not need to be NumPy arrays.

    Return a NumPy array.
    """
    assert len(list_1) == len(list_2), "both args must have the same number of elements"
    list_1 = np.array(list_1)
    list_2 = np.array(list_2)
    return list_1**2 + list_2**3
```

```
grader.check("q2_c")
```

passed!

You might have been told that Python is slow, but array arithmetic is carried out very fast, even for large arrays.

For ten numbers, `list_sum` and `array_sum` both take a similar amount of time.

```
sample_list_1 = list(range(10))
sample_array_1 = np.arange(10)
```

```
%%time
list_sum(sample_list_1, sample_list_1)
```

```
CPU times: user 13 µs, sys: 0 ns, total: 13 µs
Wall time: 16.5 µs
```

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

```
%%time
array_sum(sample_array_1, sample_array_1)
```

```
CPU times: user 122 µs, sys: 28 µs, total: 150 µs
Wall time: 84.6 µs
```

```
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 `list_sum` executes faster than `array_sum`! However, we will commonly be working with much larger datasets (run the following blocks a few times to get a better understanding for the average timings):

```
sample_list_2 = list(range(100000))
sample_array_2 = np.arange(100000)
```

```
%%time
list_sum(sample_list_2, sample_list_2)
pass # The pass hides the output
```

```
CPU times: user 58.5 ms, sys: 3.54 ms, total: 62 ms
Wall time: 214 ms
```

```
%%time
array_sum(sample_array_2, sample_array_2)
pass
```

```
CPU times: user 2.04 ms, sys: 2.03 ms, total: 4.07 ms
Wall time: 64.7 ms
```

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.

A note on `np.arange` and `np.linspace`

Usually we use `np.arange` to return an array that steps from `a` to `b` with a fixed step size `s`. While this is fine in some cases, we sometimes prefer to use `np.linspace(a, b, N)`, which divides the interval `[a, b]` into N equally spaced points.

`np.arange(start, stop, step)` produces an array with all the numbers starting at `start`, incremendted up by `step`, stopping **before** `stop` is reached. For example, the value of `np.arange(1, 6, 2)` is an array with elements 1, 3, and 5 – it starts at 1 and counts up by 2, then stops before 6. `np.arange(4, 9, 1)` is an array with elements 4, 5, 6, 7, and 8. (It doesn't contain 9 because `np.arange` stops *before* the stop value is reached.)

`np.linspace` always includes **both end points** while `np.arange` will **not** include the second end point `b`. For this reason, especially when we are plotting ranges of values we tend to prefer `np.linspace`.

Notice how the following two statements have different parameters but return the same result.

```
np.arange(-5, 6, 1.0)
```

```
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.,  5.])
```

```
np.linspace(-5, 5, 11)
```

```
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.,  5.])
```

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

```
grader.check_all()
```

```
q1_a results:
    q1_a - 1 result:
        Trying:
            n = 1;
        Expecting nothing
        ok
        Trying:
            type(summation(n)) == np.intc
        Expecting:
            True
        **********************************************************************
        Line 2, in q1_a 0
        Failed example:
            type(summation(n)) == np.intc
        Expected:
            True
        Got:
            False

    q1_a - 2 result:
        Test case passed!

    q1_a - 3 result:
        Test case passed!

q1_b results: All test cases passed!

q2_a results: All test cases passed!

q2_b results: All test cases passed!

q2_c results: All test cases passed!
```