In [ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook("lab00.ipynb")
```

Lab 0: Intro to Jupyter Notebooks (Optional)

Welcome to the first lab for this course! We'll share some helpful tips on using Jupyter notebooks and give a quick refresher on writing code in Python and NumPy. As with all assignments in this class, make sure you run the grader cell at the very top!

Learning Objectives

1. Learn some helpful tips on working in Jupyter notebooks
2. Review basic Python syntax
3. Review basic NumPy syntax and useful methods

Jupyter

We view/access all of our assignments and several of our lectures via Jupyter Notebooks (https://jupyter-notebook.readthedocs.io/en/stable/examples/Notebook/What%20is%20the%20Jupyter%20Notebook.html), which allow us to run code in different cells and see how they compare. You should already be familiar with Jupyter Notebooks from Data 8, but here we give some helpful tips on working in them.

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

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

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

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    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 Econ 148!')
```

```
Welcome to Econ 148!
```

Importing Libraries

In data 8, you used the `datascience` and `numpy` libraries to work with data. In econ 148, we will introduce you to several libraries, including `pandas`, `matplotlib`, `seaborn`, `sklearn`, etc. It is convention to import any libraries you use at the very top of the notebook, along with any shorter aliases (i.e. abbreviations) which make it easy to refer to them. For this lab, we mostly use `numpy`, except one question where we load in data using `pandas`. We have imported them both along with their common aliases below.

In [3]:
```
import numpy as np
import pandas as pd
```

Keyboard Shortcuts

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

Here are a few that we like:

1. `Ctrl` / `Cmd` + `Return` : *Evaluate the current cell*
2. `Shift` + `Return` : *Evaluate the current cell and move to the next*
3. `Ctrl` / `Cmd` + `+` or `-` : *Zoom in or out*
4. `Ctrl` / `Cmd` + `/` : *Comment or uncomment the selected code at once*
5. `ESC` : *command mode* (press before using any of the commands below)
    A. `a` : *create a cell above*
    B. `b` : *create a cell below*
    C. `dd` : *delete a cell*
    D. `z` : *undo the last cell operation*
    E. `m` : *convert a cell to markdown*
    F. `y` : *convert a cell to code*

Running Cells

Aside from keyboard shortcuts (specifically `Shift + Return`), you can also run a single cell by clicking the `Run` button in the top left corner of your notebook. If you hover over the button, you will also find some other options that allow you to run multiple cells. Specifically, restarting your kernel clears all saved variables and frees up memory. The `Restart Kernel and Run upto Selected Cell` option is particularly useful for situations where you believe your code is fine but you're running into strange memory issues.

Python

Python is the main programming language we'll use in the course. Work through the following cells and make sure you understand what is happening in each.

Feel free to review one or more of the following materials as a refresher:

- **Python Tutorial (https://docs.python.org/3/tutorial/index.html)**: Introduction to Python from the creators of Python
- **Composing Programs Chapter 1 (http://composingprograms.com/pages/11-getting-started.html)**: This is more of a introduction to programming with Python.
- **Advanced Crash Course (http://cs231n.github.io/python-numpy-tutorial/)**: A fast crash course which assumes some programming background.

Lists

Question 1: In the cell below, define the variable `numbers` to be equal to a list of all whole numbers from 1 to 5 (inclusive).

In [4]:
```
q1 = [1,2,3,4,5] # SOLUTION
q1
```

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

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

Question 2: Add the number `6` at the end of `q1` and store the result in `q2`.

In [121]:
```
q2 = q1 + [6] # SOLUTION
q2
```

Out[121]:
```
[1, 2, 3, 4, 5, 6]
```

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

Question 3: Remove the element `3` from `q2` and store the result in `q3`, without editing `q2`.

Hint: Be careful if using the `remove` method. Feel free to google if you're stuck!

In [123]:
```
q3 = q2[:] # SOLUTION
q3.remove(3) # SOLUTION
q3
```

Out[123]:
```
[1, 2, 4, 5, 6]
```

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

Loops and List Comprehensions

Loops help us iterate over every element in an iterable object (such as a list or an array). For example, the code below prints every single month that is stored in the list `months`.

In [126]:
```
months = ["January", "February", "March", "April", "May", "June",
          "July", "August", "September", "October", "November", "December"]
for month in months:
    print(month)
```

```
January
February
March
April
May
June
July
August
September
October
November
December
```

We can perform more complicated expressions with loops. For example, let's print every single month that doesn't start with the letter J.

In [127]:
```
for month in months:
    if not month.startswith('J'):
        print(month)
```

```
February
March
April
May
August
September
October
November
December
```

While the examples above have both used for-loops, we can also use while loops. For example, what does the code below do?

In [128]:
```
i = 0
while i < 6:
    print(months[i])
    i += 1
```

```
January
February
March
April
May
June
```

A list comprehension in Python is essentially a simple for-loop written with special syntax. For example, we can convert our first for loop (printing every month) to a list comprehension, as done below.

In [129]:
```
[print(month) for month in months];
```

```
January
February
March
April
May
June
July
August
September
October
November
December
```

We have included a `;` at the end of list comprehension to suppress the output, why? What would happen if we don't include it?

Let's make things more complicated; let's use a list comprehension to get a list containing every single month that doesn't start with the letter J.

In [130]:
```
[month for month in months if not month.startswith('J')]
```

Out[130]:
```
['February',
 'March',
 'April',
 'May',
 'August',
 'September',
 'October',
 'November',
 'December']
```

Question 4: The list `first_hundred` below contains all numbers from 1 to 100 (inclusive on both ends; try printing out each element in the list if you're unsure). Use a list comprehension to get a list of each number that is divisible by either 7 or 13, and store the result in `q4`.

In [131]:
```
first_hundred = range(1, 101)
q4 = [num for num in first_hundred if num % 7 == 0 or num % 13 == 0] # SOLUTIO
N
q4
```

Out[131]:
```
[7, 13, 14, 21, 26, 28, 35, 39, 42, 49, 52, 56, 63, 65, 70, 77, 78, 84, 91, 9
8]
```

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

Dictionaries

Dictionaries are collections of key-value pairs, where each 'key' is associated with a certain 'value'. They are defined using `{}`, and can contain as many values as you like. You can read more about them here (https://realpython.com/python-dicts/). We define a sample dictionary below.

In [133]:
```
tel = {'justin': 5414, 'luis': 1685, 'dawson': 9522}
tel
```

Out[133]:
```
{'justin': 5414, 'luis': 1685, 'dawson': 9522}
```

Here, the keys are the names of the people, and the values are the last four digits of their phone numbers. You can retrieve the value associated with any key but not vice versa. For example, the code below retrieves the last 4 digits of Peter's phone number, but you cannot find out the name of the person whose phone number ends with 9522.

In [134]:
```
tel['luis']
#tel[1685] This does not work
```

Out[134]:
```
1685
```

You can add and delete key-value pairs, as shown below.

In [135]:
```
# insert a new key value pair
tel['bennett'] = 3698

# delete an existing key value pair (two ways)
tel.pop('justin')
del tel['luis']
```

Question 5: Write code that removes the record for `bennett` from the dictionary `tel2` and adds a new record for `eric` with value 1480.

In [136]:
```
tel2 = {'justin': 5414, 'luis': 1685, 'dawson': 9522, 'bennett': 9522}
# BEGIN SOLUTION NO PROMPT
tel2.pop('bennett')
tel2['eric'] = 1480
# END SOLUTION
""" # BEGIN PROMPT
...
"""; # END PROMPT
```

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

Defining Functions

Functions take in certain values (as defined by their parameters), perform some processing with them and output some other values. You can define your own functions; an example is below:

In [140]:
```
def add2(x):
    """This docstring explains what this function does: it adds 2 to a numbe
r."""
    return x + 2
```

You can see the output of `help(function)` contains the docstring of the function.

In [141]:
```
help(add2)
```

```
Help on function add2 in module __main__:

add2(x)
    This docstring explains what this function does: it adds 2 to a number.
```

Question 6: Define the function `print_extension` to take in a dictionary (the parameter `phonebook`) and string (the parameter `name`), returning a string stating the four-digit value associated with the key `name` in `phonebook`. For example:

```
>>> print_extension(tel, 'dawson')
'The extension for dawson is 9522.'
```

Note: You don't need to handle cases where `name` doesn't exist in the `phonebook`.

In [149]:
```
# BEGIN SOLUTION NO PROMPT
def print_extension(phonebook, name):
    value = phonebook[name]
    return "The extension for " + str(name) + " is " + str(value) + "."
# END SOLUTION
""" # BEGIN PROMPT
def print_extension(phonebook, name):
    value = ...
    return ... # Return the string "The extension for [name] is [value]."
"""; # END PROMPT
print_extension(tel, 'bennett') # this should print "The extension for luis is
1685."
```

Out[149]:
```
'The extension for bennett is 3698.'
```

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

Question 7: Write a function `squares_dict` that takes in a list of numbers `l` and creates a dictionary that records the the square of each number. For example:

```
>>> print(squares_dict([0,1,2,4]))
{0: 0, 1: 1, 2: 4, 4: 16}
```

In [152]:
```
# BEGIN SOLUTION NO PROMPT
def squares_dict(l):
    squares = {} # initialize dictionary to hold the squared numbers

    for number in l:
        squares[number] = number ** 2

    return squares
# END SOLUTION
""" # BEGIN PROMPT
def squares_dict(l):
    squares = {} # initialize dictionary to hold the squared numbers

    for number in ...:
        ...

    return squares
"""; # END PROMPT
print(squares_dict([0,1,2,4]))
```

```
{0: 0, 1: 1, 2: 4, 4: 16}
```

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

NumPy

The `NumPy` library lets us do fast, simple calculations with numbers in Python.

If the following section is a little difficult, we've provided some resources for review below. Don't worry if you're not familiar with everything below, we will ensure to only use the NumPy functions referenced in data 8.

- Condensed Numpy Review (http://cs231n.github.io/python-numpy-tutorial/#numpy)
- Data 100's Numpy Review (http://ds100.org/fa17/assets/notebooks/numpy/Numpy_Review.html)
- The Official Numpy Tutorial (https://numpy.org/doc/stable/user/quickstart.html)

Arrays

The heart of NumPy lies in the arrays. NumPy arrays are similar to Python lists, but much faster if you have a large number of elements. They have two key differences compared to Python lists:

- NumPy arrays have a fixed size, you cannot directly add an element like you could in a list. Instead, you have to use a function such as `np.append`, which technically create another array.
- NumPy arrays cannot store objects of different data types and will instead convert everything to the same data type.

These are demonstrated below.

In [100]:
```
sample_list = [1,2,3] # creating a sample list
sample_array = np.array([1,2,3]) # creating a sample array
sample_list, sample_array
```

Out[100]:
```
([1, 2, 3], array([1, 2, 3]))
```

In [101]:
```
sample_list_appended = sample_list + [4] # appending to Python lists is easy
sample_array_appended = np.append(sample_array, 4) # appending to arrays requi
res using a function
sample_list_appended, sample_array_appended
```

Out[101]:
```
([1, 2, 3, 4], array([1, 2, 3, 4]))
```

In [102]:
```
sample_list_str = sample_list_appended + ['woah'] # lists can store different
data types
sample_array_str = np.append(sample_array_appended, 'woah') # arrays cannot, t
hey convert everything to a string here
sample_list_str, sample_array_str
```

Out[102]:
```
([1, 2, 3, 4, 'woah'], array(['1', '2', '3', '4', 'woah'], dtype='<U21'))
```

You can easily do math and add/subtract different arrays with the same dimensions, as shown below.

In [103]:
```
sample_array_appended = np.array([1, 2, 3, 4])
print(f"{sample_array_appended} + {1} is: {sample_array_appended + 1}")
print(f"{sample_array_appended} * {8} is: {sample_array_appended * 8}")
print(f"{sample_array_appended} + {np.arange(5,9)} is: {sample_array_appended
+ np.arange(5,9)}")
print(f"{sample_array_appended} - {np.arange(5,9)} is: {sample_array_appended
- np.arange(5,9)}")
print(f"{sample_array_appended} squared is: {sample_array_appended ** 2}")
```

```
[1 2 3 4] + 1 is: [2 3 4 5]
[1 2 3 4] * 8 is: [ 8 16 24 32]
[1 2 3 4] + [5 6 7 8] is: [ 6  8 10 12]
[1 2 3 4] - [5 6 7 8] is: [-4 -4 -4 -4]
[1 2 3 4] squared is: [ 1  4  9 16]
```

NumPy has several built-in functions you can use to work on arrays of numbers; examples shown below.

In [104]:
```
np.sum(sample_array_appended)
```

Out[104]:
```
10
```

In [105]:
```
np.mean(sample_array_appended)
```

Out[105]:
```
2.5
```

In [106]:
```
np.min(sample_array_appended), np.max(sample_array_appended)
```

Out[106]:
```
(1, 4)
```

In [107]:
```
np.std(sample_array_appended)
```

Out[107]:
```
1.118033988749895
```

You can also define matrices as 2-dimensional numpy arrays, as shown below:

In [108]:
```
first_matrix = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
first_matrix
```

Out[108]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

You can use the `.shape` attribute for seeing the dimensions (number of rows/columns) of the numpy array.

In [109]:
```
first_matrix.shape
```

Out[109]:
```
(3, 3)
```

The `np.arange` function is very helpful for defining new arrays of numbers (documentation (https://numpy.org/doc/stable/reference/generated/numpy.arange.html)). Examples are shown below.

In [110]:
```
np.arange(10) #Starts at 0, ends at 9 (non-inclusive on the ending side), gives you all whole numbers
```

Out[110]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [111]:
```
np.arange(0, 10, 2) # Starts at 0, ends at 9, gives you numbers in increments of 2
```

Out[111]:
```
array([0, 2, 4, 6, 8])
```

In [112]:
```
np.arange(3, 8, 0.5) # Starts at 3, ends at 7.5, gives you numbers in increments of 0.5
```

Out[112]:
```
array([3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5, 7. , 7.5])
```

In [113]:
```
np.arange(10, 0, -2) # Starts at 10, ends at 2, gives you numbers in increments of -2
```

Out[113]:
```
array([10,  8,  6,  4,  2])
```

Slicing

A key aspect of working with NumPy arrays and matrices is slicing, a convenient tool to extract a certain portion of an array. You can extract portions based on indices as well as true/false conditions.

If choosing to extract portions based on indices, the basic syntax often follows `array[start:stop:step]` . Examples are shown below.

In [114]:
```
sample_array_appended = np.array([1, 2, 3, 4])
sample_array_appended[:]
# Using just [:] gives you all the values in the array. You don't have to include the second :
# This happens because the default starting index is 0 and the default ending index is after the last element
# The default step size is 1
```

Out[114]:
```
array([1, 2, 3, 4])
```

In [115]:
```
sample_array_appended[0:1] # Arrays are 0-indexed, exclusive of the element at the `stop` index
```

Out[115]:
```
array([1])
```

In [116]:
```
sample_array_appended[0:len(sample_array_appended)+1:2] # gives you every other element, starting from the first
```

Out[116]:
```
array([1, 3])
```

In [117]:
```
sample_array_appended[-2:] # gives you the last 2 elements
```

Out[117]:
```
array([3, 4])
```

You can also use slicing with 2-D arrays, or matrices

In [118]:
```
first_matrix = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
first_matrix[:,:] # You're now slicing along 2 axes (rows and columns), separa
ted by a comma
```

Out[118]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [119]:
```
first_matrix[1,:]
```

Out[119]:
```
array([4, 5, 6])
```

In [120]:
```
first_matrix[:,1]
```

Out[120]:
```
array([2, 5, 8])
```

In [121]:
```
first_matrix[0:2,1:3]
```

Out[121]:
```
array([[2, 3],
       [5, 6]])
```

Finally, we can also slice based on true/false conditions.

In [122]:
```
sample_array_appended = np.array([1, 2, 3, 4])
sample_array_appended[sample_array_appended<3]
```

Out[122]:
```
array([1, 2])
```

In [123]:
```
print(sample_array_appended**2 >= 4) # The output of this is an array of true/
false values
sample_array_appended[sample_array_appended**2 >= 4] # We keep all indices in
the original array where the value is True
```

```
[False  True  True  True]
```

Out[123]:
```
array([2, 3, 4])
```

In the following questions, we will use numpy to answer some questions about the California Independent System Operator (CAISO) Daily Renewables Watch. This data records the hourly production of various renewable resources as well as a hourly breakdown of total production of resource type, both in megawatts (MW). To see how the dataset is formatted, click here (http://content.caiso.com/green/renewrpt/20180617_DailyRenewablesWatch.txt).

The following line loads in data for Question 8. Note that it uses pandas -- we'll learn more about that next week! However, the output is a numpy array.

In [124]:
```
data_q8 = pd.read_csv("data/q8.csv").values
data_q8
```

Out[124]:
```
array([['2017-08-21', 1, 971, 245, 164, 406, 2032, 0, 0],
       ['2017-08-21', 2, 971, 246, 174, 408, 2056, 0, 0],
       ['2017-08-21', 3, 971, 248, 175, 406, 1912, 0, 0],
       ['2017-08-21', 4, 972, 250, 175, 415, 1808, 0, 0],
       ['2017-08-21', 5, 972, 251, 175, 405, 1771, 0, 0],
       ['2017-08-21', 6, 973, 250, 175, 412, 1681, 0, 0],
       ['2017-08-21', 7, 972, 254, 175, 441, 1560, 174, 0],
       ['2017-08-21', 8, 971, 256, 174, 442, 1559, 2376, 0],
       ['2017-08-21', 9, 969, 259, 174, 495, 1582, 5484, 86],
       ['2017-08-21', 10, 965, 258, 169, 463, 1534, 5312, 141],
       ['2017-08-21', 11, 945, 260, 172, 451, 1372, 3503, 110],
       ['2017-08-21', 12, 939, 259, 176, 457, 1297, 6329, 201],
       ['2017-08-21', 13, 936, 256, 176, 460, 1308, 8700, 472],
       ['2017-08-21', 14, 936, 257, 176, 468, 1496, 8857, 527],
       ['2017-08-21', 15, 936, 260, 177, 471, 1584, 8732, 598],
       ['2017-08-21', 16, 936, 262, 175, 501, 1620, 7983, 395],
       ['2017-08-21', 17, 936, 260, 176, 486, 1649, 7020, 262],
       ['2017-08-21', 18, 935, 258, 174, 528, 1580, 5329, 228],
       ['2017-08-21', 19, 955, 258, 172, 610, 1644, 2193, 119],
       ['2017-08-21', 20, 964, 255, 172, 604, 1845, 114, 0],
       ['2017-08-21', 21, 967, 247, 171, 583, 2358, 0, 0],
       ['2017-08-21', 22, 970, 235, 171, 484, 2282, 0, 0],
       ['2017-08-21', 23, 970, 230, 171, 436, 1840, 0, 0],
       ['2017-08-21', 24, 971, 228, 172, 423, 1620, 0, 0]], dtype=object)
```

Question 8: Use slicing to output an array containing the hourly values of solar photovoltaic and solar thermal production. Make sure you see how the dataset is formatted (http://content.caiso.com/green/renewrpt/20180617_DailyRenewablesWatch.txt).

Hint: We have only loaded in the first dataset from the link above. What columns represent the hourly values of solar photovoltaic and solar thermal production?

In [125]:
```
q8 = data_q8[:, -2:] # SOLUTION
q8
```

Out[125]:
```
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [174, 0],
       [2376, 0],
       [5484, 86],
       [5312, 141],
       [3503, 110],
       [6329, 201],
       [8700, 472],
       [8857, 527],
       [8732, 598],
       [7983, 395],
       [7020, 262],
       [5329, 228],
       [2193, 119],
       [114, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=object)
```

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

Now, we will import another dataset that contains all the data for renewable resources generation in April 2017.

In [128]:
```
data_q9 = pd.read_csv("data/q9.csv").values
data_q9
```

Out[128]:
```
array([['2017-04-01', 1, 934, ..., 1664, 0, 0],
       ['2017-04-01', 2, 935, ..., 1595, 0, 0],
       ['2017-04-01', 3, 935, ..., 1563, 0, 0],
       ...,
       ['2017-04-30', 22, 920, ..., 3269, 0, 0],
       ['2017-04-30', 23, 917, ..., 3320, 0, 0],
       ['2017-04-30', 24, 918, ..., 3213, 0, 0]], dtype=object)
```

In [129]:
```
data_q9[0,:]
```

Out[129]:
```
array(['2017-04-01', 1, 934, 217, 172, 550, 1664, 0, 0], dtype=object)
```

Question 9: How much more electricity was produced via wind energy compared to geothermal energy (in WM) in all of April 2017? Note, each row in the dataset represents all energy produced for a given *hour*.

In [130]:
```
q9 = np.sum(data_q9[:, 6] - data_q9[:, 2]) # SOLUTION
q9
```

Out[130]:
```
723677
```

In [ ]:
```
grader.check("q9")
```

Feedback

Question 10: Please fill out this short feedback form (https://forms.gle/KDrqdNbv6m2J5nYW6) to let us know your thoughts on this lab! We really appreciate your opinions and feedback! At the end of the Google form, you should see a codeword. Assign the codeword to the variable `codeword` below.

In [132]:
```
codeword = "welcome" # SOLUTION
```

In [ ]:
```
grader.check("q10")
```

Congratulations, you are finished with lab 0 of econ 148! You are not required to submit this lab, but you may do so if you wish.

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, run_tests=True)
```