

04. Lists and Loops [demonstration]

March 21, 2020

1 Lists and Loops [demonstration]

The code below shows some ways of using python **lists** (known as **arrays** in many languages). Note how these lists are "sliced" from index i to j using `my_list[i:j]` notation.

```
In [1]: # How to print a list structure
my_list = [1, 2, 3, "a", "b", "c"]
print("my_list is:", my_list)

my_list is: [1, 2, 3, 'a', 'b', 'c']
```

```
In [2]: # Prints each element of a list individually
print("Looping through a list...")
for item in my_list:
    print("item is", item)
```

```
Looping through a list...
item is 1
item is 2
item is 3
item is a
item is b
item is c
```

```
In [3]: # Prints the number of elements in a list
print("The len function is important!")
num_elements = len(my_list)
print("my_list has", num_elements, "elements")

# Could also be done without the intermediate variable
print("The len function is important!")
print("my_list has", len(my_list), "elements")
```

```
The len function is important!
my_list has 6 elements
```

```
The len function is important!
my_list has 6 elements
```

```
In [4]: # Using range to loop through a list by index
    print("A less great way to loop through a list...")
        for index in range(len(my_list)):
            item = my_list[index] # accessing an element in a list!
            print("item is", item)
```

```
A less great way to loop through a list...
item is 1
item is 2
item is 3
item is a
item is b
item is c
```

```
In [5]: # Looping through a partial list
    print("Slicing a list from beginning...")
        for item in my_list[:3]:
            print("item is", item)
```

```
Slicing a list from beginning...
item is 1
item is 2
item is 3
```

```
In [6]: # Looping through a partial list again
    print("Slicing a list to the end...")
        for item in my_list[3:]:
            print("item is", item)
```

```
Slicing a list to the end...
item is a
item is b
item is c
```

```
In [7]: # Looping through a partial list again
    print("Slicing a list in the middle...")
        for item in my_list[2:4]:
            print("item is", item)
```

```
Slicing a list in the middle...
item is 3
item is a
```

```
In [8]: print("Enumerating a list...")
    for i, item in enumerate(my_list):
        print("item number", i, "is", item)
```

```
Enumerating a list...
item number 0 is 1
item number 1 is 2
item number 2 is 3
item number 3 is a
item number 4 is b
item number 5 is c
```

```
In [9]: print("Another way to enumerate using a list 'method'...")
    for item in my_list:
        index = my_list.index(item)
        print("item", item, "has index", index)
```

```
Another way to enumerate using a list 'method'...
item 1 has index 0
item 2 has index 1
item 3 has index 2
item a has index 3
item b has index 4
item c has index 5
```

```
In [ ]:
```

03. For Loops [demonstration]

March 21, 2020

1 For Loops [demonstration]

The code below demonstrates two ways of looping through the numbers 0,1,2,3,4,5

```
In [1]: print("1 - demonstrating for loop")
    # Python allows you loop over a list
    for i in [0,1,2,3,4,5]:
        print(i)
```

```
1 - demonstrating for loop
0
1
2
3
4
5
```

```
In [2]: print("2 - demonstrating for loop with range()")
    # If you want to loop until a specified number, use range(number)
    for i in range(6):
        print(i)
```

```
2 - demonstrating for loop with range()
0
1
2
3
4
5
```

1.0.1 The range function

The `range` function is one you'll probably use a lot - often in conjunction with a `for` loop. The code below shows how you can use the `range` function to print out a multiplication table.

Read through the code below. What is the biggest number you expect to see printed?

```
In [4]: # This is an example of a nested loop
# Nested loops are loops inside of loops
# the left_num will increment slower than the right_num
# How many times will right_num perform full loops?
for left_num in range(5):
    for right_num in range(5):
        product = left_num * right_num
        print(left_num, "x", right_num, "=", product)

0 x 0 = 0
0 x 1 = 0
0 x 2 = 0
0 x 3 = 0
0 x 4 = 0
1 x 0 = 0
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
2 x 0 = 0
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
3 x 0 = 0
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
4 x 0 = 0
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
```

```
In [ ]:
```

10. Python Control Flow [demonstation]

March 21, 2020

1 Control Flow in Python

The code below demonstrates how the flow of program execution can be controlled using statements like `if`, `elif` and `else`.

```
In [1]: Y = 7
        # Check that 5 < Y and Y < 10
        if 5 < Y < 10:
            print("Y is between 5 and 10")
        else:
            print("Y is not between 5 and 10")

Y is between 5 and 10
```

```
In [2]: X = 4
        if X < 5:
            print("X is a small number")
        elif X < 20:
            print("X is a medium sized number")
        else:
            print("X is a big number")

X is a small number
```

```
In [3]: if True:
        print("True is always True!")

True is always True!
```

```
In [5]: if False:
        print("This will never be printed")
```

```
In [ ]:
```

02. Data Types [demonstration]

March 21, 2020

1 Data Types [demonstration]

Explore how **strings**, **numbers**, and **booleans** are defined in Python.

Read through the code and try to predict what each cell's output will look like *before* you run that cell.

```
In [5]: # STRINGS
#
# In this section you define several variables. These are all defined
# to be strings. Note that there are several ways to define a string.
#
print("STRINGS") # Prints the word STRINGS as the output

# Declaring strings with double quotes
my_string_1 = "hello"

# Declaring strings with single quotes
my_string_2 = 'world'

# Use either three double quotes or three single quotes
# note in this case you can have single quotes of either type
# inside the string. This is a powerful feature!
my_multiline_string = """"
Dear World,
Hello. I am a multiline python string.
I'm enclosed in triple quotes. I'd write
them here, but that would end the string!
I know! I'll use a slash as an escape character.

Triple quotes look like this: \"\"\""

Sincerely,
Python
"""
```

```
newline_character = "\n"
print("Now run the cell below to see these strings printed!")
```

STRINGS

Now run the cell below to see these strings printed!

```
In [6]: print(my_string_1, my_string_2)
        print(my_multiline_string)
        print(newline_character)
        print("-----")
        print(newline_character)
```

hello world

Dear World,

Hello. I am a multiline python string.
I'm enclosed in triple quotes. I'd write
them here, but that would end the string!

I know! I'll use a slash as an escape character.

Triple quotes look like this: """

Sincerely,
Python

```
In [7]: # NUMBERS AND BOOLEANS
```

```
print("NUMBERS")
my_float      = 0.5
my_integer    = 7
my_negative   = -3.5
my_fraction   = 1/2
```

```
# what do you think THIS line of code will assign to the variable
# does_half_equal_point_five?
does_half_equal_point_five = (my_fraction == my_float)
```

NUMBERS

```
In [8]: print("The absolute value of", my_negative, "is", abs(my_negative))
      print(my_integer, "squared is equal to", my_integer ** 2)
      print("Does", my_fraction, "equal", my_float, "?", does_half_equal_point_five)
```

The absolute value of -3.5 is 3.5

7 squared is equal to 49

Does 0.5 equal 0.5 ? True

In []:

05. List Comprehensions [demonstration]

March 22, 2020

1 List Comprehensions [demonstration]

One of Python's most "Pythonic" features is the **list comprehension**. It's a way of creating a list in a compact and, once you get used to them, readable way. See the demonstrations below.

```
In [1]: # The preferred 'pythonic' way to create a list
numbers_0_to_9 = [x for x in range(10)]
print("Numbers 0 to 9", numbers_0_to_9)
```

Numbers 0 to 9 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```
In [2]: # The above is equivalent to the following just more compact:
numbers_0_to_9 = []
for x in range(10):
    numbers_0_to_9.append(x)
print("Numbers 0 to 9", numbers_0_to_9)
```

Numbers 0 to 9 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```
In [3]: # You can also choose to do computation / flow control when generating
# lists
squares = [x * x for x in range(10)]
print("Squares      ", squares)
```

Squares [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

```
In [4]: # note - this example uses the "modulo" operator
odds = [x for x in range(10) if x % 2 == 1]
print("Odds        ", odds)
```

Odds [1, 3, 5, 7, 9]

1.0.1 Advanced List Comprehensions [optional]

If you're already an advanced programmer, you may be interested in some of these more advanced list comprehensions. They're one of the things that experienced Python users learn to love about Python.

This example also uses a data type called a namedtuple which is similar to a struct data type in other languages.

```
In [2]: # Import the module collections and be able to access namedtuple
        # without having to write collections.namedtuple each time.
        from collections import namedtuple
        # Create a tuple of class 'Person'
        Person = namedtuple("Person", ["name", "age", "gender"])

        # Build a list of people by using Person
        people = [
            Person("Andy", 30, "m"),
            Person("Ping", 1, "m"),
            Person("Tina", 32, "f"),
            Person("Abby", 14, "f"),
            Person("Adah", 13, "f"),
            Person("Sebastian", 42, "m"),
            Person("Carol", 68, "f"),
        ]

        # first, let's show how this namedtuple works.
        # Get the first entry in the list (which is Andy from above)
        andy = people[0]

        # From here, you can access andy like it was a class with attributes
        # name, age, and gender (as seen from the namedtuple)
        print("name: ", andy.name)
        print("age: ", andy.age)
        print("gender:", andy.gender)

name: Andy
age: 30
gender: m
```

```
In [3]: # now let's show what we can do with a list comprehension
        male_names = [person.name for person in people if person.gender=="m"]
        print("Male names:", male_names)

Male names: ['Andy', 'Ping', 'Sebastian']
```

```
In [4]: # Create a list of names where there age
        teen_names = [p.name for p in people if 13 <= p.age <= 18 ]
        print("Teen names:", teen_names)
```

```
Teen names: ['Abby', 'Adah']
```

```
In [ ]:
```

06. Python's random Library [demonstration]

March 22, 2020

1 Python's random Library [demonstration]

The code below demonstrates how a library (like `random`) can be **imported** and then used.
Run the cells below in order.

```
In [1]: # `random` is a Python "module". It provides functionality associated
# with randomness / probability. For now you don't need to know exactly
# what is happening when we write `import random as rd`, instead focus
# on how we USE this module.
import random as rd

In [2]: # Call the FUNCTION named random 3 times and assign the result of each
# call to a variable.
a = rd.random()
b = rd.random()
c = rd.random()
print("a is", a)
print("b is", b)
print("c is", c)

# random isn't the only function in the random module.
# Do a google search for "python random library" or
# go here: https://docs.python.org/3/library/random.html
# to learn more.

print("Now keep pressing Ctrl+Enter to run this cell many times!")
print("Notice how the values of a,b, and c keep changing!")

a is 0.0823743364915106
b is 0.6680213156071539
c is 0.4499209318035706
Now keep pressing Ctrl+Enter to run this cell many times!
Notice how the values of a,b, and c keep changing!
```

In []:

08. Functions [demonstration]

March 22, 2020

1 Functions [demonstration]

You just saw code that looks like the code below. Hopefully you found that by changing the value of num_trials you could also change how close the overall "heads percentage" was to 50%. By running more and more trials, you would get closer and closer to a consistent 50%.

```
In [ ]: # Import the random module and reference it as rd
         import random as rd

         num_trials = 100 # Sets the number of flips
         heads = 0 # A counter for the number of heads
         tails = 0 # A counter for the number of tails
         p_heads = 0.5 # The probability for heads

         # Simulate coin flips up to the num_trials specified
         for i in range(num_trials):
             # Collect a random number between [0, 1]
             random_number = rd.random()
             # If the number is less than heads count it as heads
             # Otherwise, count it as tails
             if random_number < p_heads:
                 heads = heads + 1
             else:
                 tails += 1
```

Now, we're going to **encapsulate** this code by putting it into a function we define. The code below shows how a function is **defined** and **called** in Python.

As you read through the code below, pay attention to the following:

1. **Whitespace / indentation.** Many programming languages use braces {}, parentheses (), brackets <>, etc... to delimit a block of code. Python does not!
2. **Defining vs. Calling a function.** All the code between lines 3-14 is the *definition* of simulate_coin_flips and line 16 shows how you *call* simulate_coin_flips.

```
In [1]: # Import the random module and reference it as rd
         import random as rd

         def simulate_coin_flips(num_trials):
```

```

    ...
A function to simulate coin flips

Args:
    num_trials (int): The number of coin flip
                      trials to simulate

Returns:
    int: The percentage of heads from the trials
    ...

heads = 0 # A counter for the number of heads
tails = 0 # A counter for the number of tails
p_heads = 0.5 # The probability for heads

# Simulate coin flips up to the num_trials specified
for i in range(num_trials):
    # Collect a random number between [0,1]
    random_number = rd.random()
    # If the number is less than heads count it as heads
    # Otherwise, count it as tails
    if random_number < p_heads:
        heads = heads + 1
    else:
        tails += 1
    # Calculate the percentage of heads based on the number of
    # heads and trials
percent_heads = heads / num_trials
return percent_heads

percentage = simulate_coin_flips(200) # calling the function
print(percentage) # Import the random module and reference it as rd

```

0.495

You now have a new tool you can use! If you want to try calling this function with different values for num_trials, you can.

In [2]: `print(simulate_coin_flips(100))`

0.55

In [3]: `print(simulate_coin_flips(10000))`

0.4978

In [4]: `print(simulate_coin_flips(1000000))`

0.49974

In []:

09. Simulating Probabilities [demonstration]

March 22, 2020

1 Simulating Probabilities [demonstration]

The code below shows one method for simulating dice rolls. Read through it and try to understand how it works.

What does the data stored in `roll_counts` represent?

```
In [1]: # Import the random module and reference it as rd
        import random as rd

def simulate_dice_rolls(N):
    """
    Simulates dice rolls

    Args:
        N (int): The number of trials

    Returns:
        list: roll counts [1,6]
    """
    # Create a list to track the 6 options for the roll
    roll_counts = [0,0,0,0,0,0]
    for i in range(N):
        # Randomly select a value from the list (1 to 6)
        roll = rd.choice([1,2,3,4,5,6])
        # Recall indices start at 0 so we need to decrement
        index = roll - 1
        roll_counts[index] = roll_counts[index] + 1
    return roll_counts

def show_roll_data(roll_counts):
    """
    Shows the dice roll data

    Args:
        roll_counts (list): The roll counts stored in the list
    """
```

```

Returns:
    list: roll counts [1,6]
"""

# Gets the number of sides of the dice and prints
# the side of the die.
# enumerate creates the position of the die and the
# list value
for dice_side, frequency in enumerate(roll_counts):
    print(dice_side + 1, "was rolled", frequency, "times")

roll_data = simulate_dice_rolls(1000)
show_roll_data(roll_data)

1 was rolled 182 times
2 was rolled 149 times
3 was rolled 179 times
4 was rolled 176 times
5 was rolled 157 times
6 was rolled 157 times

```

1.0.1 Basic Data Visualization [optional]

This section is optional but you may find it interesting.

You'll learn more about this throughout the Nanodegree, but now is a great time to look at one data visualization tool called a histogram.

```

In [2]: # Matplotlib is one of the most common plotting packages in Python
        # to use it more succinctly, you can call it
        from matplotlib import pyplot as plt
        # This line is needed
        %matplotlib inline

def visualize_one_die(roll_data):
    """
    Visualizes the dice rolls

Args:
    roll_data (int): roll counts in a list from [1,6]

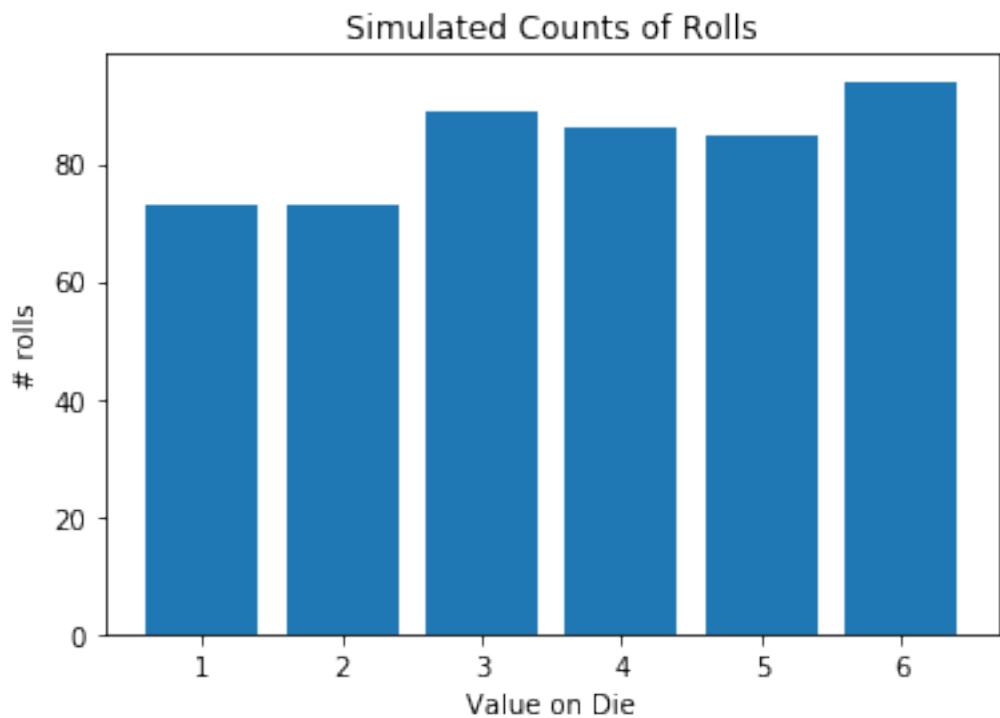
Returns:
    None - shows a plot with the x-axis is the dice values
            and the y-axis as the frequency for t
"""

roll_outcomes = [1,2,3,4,5,6]
fig, ax = plt.subplots()
ax.bar(roll_outcomes, roll_data)
ax.set_xlabel("Value on Die")

```

```
    ax.set_ylabel("# rolls")
    ax.set_title("Simulated Counts of Rolls")
    plt.show()

roll_data = simulate_dice_rolls(500)
visualize_one_die(roll_data)
```



In []:

Programming Bayes' Rule, exercise

March 22, 2020

1 Bayes' Rule

You know that Bayes' rule can be described as a way to improve a prior belief by incorporating observed data, related to this belief (like test data or sensor measurements). The rule is written as:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

Where A is the event and B is some observed, related data.

In this next quiz, given only three probabilities: `p_A`, `p_B_given_A`, and `p_notB_given_notA`, which can be written in notation as:

$$\begin{aligned} &P(A), P(B|A) \\ &P(\neg B|\neg A) \end{aligned}$$

You will be asked to write a function to calculate the posterior probability

$$P(A|B)$$

Note: This is a challenging question, so be sure to test your code or even use a pen and paper to work through this exercise!

1.0.1 TODO: Complete bayes

Complete this function so that it returns the posterior probability for any set of inputs.

```
In [2]: # Given three input probabilities, complete this function
      # so that it returns the posterior probability
```

```
def bayes(p_A, p_B_given_A, p_notB_given_notA):

    ## TODO: Calculate the posterior probability
    ## and change this value
    p_B = p_B_given_A * p_A + (1 - p_notB_given_notA) * (1 - p_A)
    posterior = p_B_given_A * p_A / p_B

    return posterior
```

```

## TODO: Change these values, run your code with them, and use print
## statements to see the output of your function and get feedback
p_A = 0.2
p_B_given_A = 0.9
p_notB_given_notA = 0.5

posterior = bayes(p_A, p_B_given_A, p_notB_given_notA)
print('Your function returned that the posterior is: ' + str(posterior))

```

Your function returned that the posterior is: 0.3103448275862069

1.0.2 Testing Cell

Run this cell and it will compare the output of your function with the correct, expected output.
Your code should pass all tests and work for any valid input values.

```

In [3]: # Test code - do not change
import solution

test_p_A = 0.4
test_p_B_given_A = 0.7
test_p_notB_given_notA = 0.9

# This line calls your function and stores the output
posterior = bayes(test_p_A, test_p_B_given_A, test_p_notB_given_notA)
correct_posterior = solution.bayes(test_p_A, test_p_B_given_A, test_p_notB_given_notA)

# Assertion, comparison test
try:

    assert(abs(posterior - correct_posterior) < 0.0001)
    print('That\'s right!')
except:
    print ('Your code returned that the posterior is: ' +str(posterior)
          + ', which does not match the correct value.')

```

That's right!

In []:

Arrays, demonstration

March 22, 2020

1 Arrays

Arrays are data structures that can contain multiple values, similar to lists in Python, but they only contain one type of data i.e. all integers or all characters, and they are often used to represent a robot's environment. This is easiest to see in an example. Say we have a self-driving car driving on a one-lane road, and near the end of this road is a stop sign, as in the below image.

1.0.1 Road Array

We can treat this road as an array, and break it up into grid cells for a robot to understand; each grid cell will contain information about the road that we can use to help our car navigate!

In Python code, the road will contain two character values: 'r' for road and 's' for stop sign. Then we can create a program that iterates through this array, checks for the stop sign (the 's') and makes the car stop at the cell *right in front of* (in this case, one cell before) the stop sign.

Note that in this demonstration, we'll be using the `numpy` library, which is a Python library that helps us create and manipulate arrays; it's commonly used in a variety of applications including self-driving cars, and we'll see it come up again and again in code.

```
In [1]: import numpy as np
```

```
# A one-lane road, represented by an array
# Here is a 1x7 road
road = np.array(['r', 'r', 'r', 'r', 'r', 's', 'r'])
```

1.0.2 Print out some information about the array

```
In [2]: # Print out some information about this road
print('The length of this array is: ' + str(len(road)))
```

The length of this array is: 7

1.0.3 Read the values in an array

```
In [3]: # Access the first index and read its value
value = road[0]
print('\n')
print('Value at index [0] = ' + str(value))
```

```

# Read the last item in the array
# A negative index moves from the end of the list backwards!
value_end = road[-1]
print('\n')
print('Value at index [-1] = ' +str(value_end))

# Compare first and last values
equal = (value == value_end)
print('\n')
print('Are the first and last values equal? ' +str(equal))

```

Value at index [0] = r

Value at index [-1] = r

Are the first and last values equal? True

2 Array Iteration

Iterating, or looping, through an array is a useful way of reading all the information it contains sequentially. The following code demonstrates how to iterate through an entire array and how to iterate until you find a certain location.

In [4]: import numpy as np

```

# A 1x7 road
road = np.array(['r', 'r', 'r', 'r', 'r', 's', 'r'])

# Iterate through the array
length = len(road)
for index in range(0, length):
    # Find and store the value at each index
    value = road[index]
    # Print a new line and the value
    print('road['+str(index)+'] = '+str(value))

road[0] = r
road[1] = r
road[2] = r
road[3] = r
road[4] = r
road[5] = s

```

```
road[6] = r
```

```
In [5]: # Iterate and exit the loop (return) once you reach index 3 - the middle
for index in range(0, length):
    # Check if index is equal to 3
    print(str(index))
    if index == 3:
        print('We\'ve reached the middle of the road and we\'re leaving the loop!')
        break
```

```
0
1
2
3
```

```
We've reached the middle of the road and we're leaving the loop!
```

```
In [ ]:
```

Array Iteration and Stopping, exercise

March 22, 2020

1 Iteration and Stopping

Consider the same road as in the last demonstration. Now that we know how to read information from an array that represents the road, we want to act on this information! It will be up to you to tell the car to stop if it's right in front of the stop sign.

1.0.1 The road

In [3]: `import numpy as np`

```
# The 1x7 road
road = np.array(['r', 'r', 'r', 'r', 'r', 's', 'r'])
```

1.0.2 TODO: Complete `find_stop_index`

Complete this function so that it returns the index of the cell that is *right before* the stop sign (ex. 0, if the stop sign is at index 1)!

```
In [10]: # This function takes in the road and determines where to stop
def find_stop_index(road):
    ## TODO: Iterate through the road array
    ## TODO: Check if a stop sign ('s') is found in the array
    ## If one is, break out of your iteration
    ## and return the value of the index that is *right before* the stop sign
    ## Change the stop_index value
    for i in range(len(road)-1):
        if road[i+1] != 's':
            stop_index = i - 1
    return stop_index
```

1.0.3 Testing Cell

Run this cell and it will compare the output of your function with the correct, expected output.

Assertions

This cell is using something called `assertions` in Python, which are statements that check the validity of code. In this case, the first assertion checks that the output of your function: `stop`, is equal to the expected output: `correct_stop` and then prints out feedback!

Your code should pass both tests and work for any 1D road.

```
In [11]: # Test code - do not change
import array_solution

# This line calls the stop function and stores the output
stop = find_stop_index(road)
correct_stop = array_solution.stop_test(road)

# Assertion, comparison test
try:
    assert(stop == correct_stop)
    print('That\'s right!')
except:
    print ('Your code returned that the stop index is: ' +str(stop)
          + ', which does not match the correct value: ' +str(correct_stop))

# Test 2
stop2 = find_stop_index(array_solution.test_road)
correct_stop2 = array_solution.stop_test(array_solution.test_road)

try:
    assert(stop2 == correct_stop2)
    print('You passed the second test!')
except:
    print ('For test 2, your code returned that the stop index is: ' +str(stop2)
          + ', which does not match the correct value.')
```

That's right!
You passed the second test!

In []:

2D Arrays and the Robot World, demonstration

March 22, 2020

1 The Robot World

A robot, much like you, perceives the world through its "senses." For example, self-driving cars use video, radar, and Lidar, to observe the world around them. As cars gather data, they build up a 3D world of observations that tells the car where it is, where other objects (like trees, pedestrians, and other vehicles) are, and where it should be going!

In this section, we'll be working with a 2D representation of the world for simplicity, and because two dimensions are often all you'll need to solve a certain problem.

1.0.1 2D Arrays

This demonstration shows how to represent a robot's world as a 2D array of observations, in this case, we'll work with a simple example that uses integer variables to represent whether there is (1) or is not (0) a tree present at that location on the grid.

For example, a robot that sees a 2x4 world may see this 2D array:

```
[[0, 0, 0, 1],  
 [1, 0, 0, 0]]
```

Which indicates that there are only two trees: one at the top rightmost point in its world, and one at the leftmost bottom. In fact the coordinates or indices of these tree locations are [0, 3] for the top-right corner and [1, 0] for the bottom-left. This has to do with the way indices are counted in Python. A graphical representation is shown below.

Note that in this demonstration, we'll be using the `numpy` library again, which is a Python library that helps us create and manipulate arrays.

1.0.2 Define the 2D world

In [1]: `import numpy as np`

```
# A simple robot world can be defined by a 2D array  
# Here is a 6x5 (num_rows x num_cols) world  
world = np.array([[0, 0, 0, 1, 0],  
                  [0, 0, 0, 1, 0],  
                  [0, 1, 1, 0, 0],  
                  [0, 0, 0, 0, 1],  
                  [1, 0, 0, 1, 0],  
                  [0, 1, 1, 1, 0]])
```

```

[1, 0, 0, 0, 0] ])

# Visualize the world
print(world)

[[0 0 0 1 0]
 [0 0 0 1 0]
 [0 1 1 0 0]
 [0 0 0 0 1]
 [1 0 0 1 0]
 [1 0 0 0 0]]

```

In [2]: # Print out some information about the world

```

print('The shape of this array is: ' + str(world.shape))
print('Notice that the number of rows come before the number of columns!')
print('It\'s height is: ' + str(world.shape[0]) +
    ', and it\'s width is: ' + str(world.shape[1]))

```

The shape of this array is: (6, 5)

Notice that the number of rows come before the number of columns!

It's height is: 6, and it's width is: 5

1.0.3 Read and compare values in a 2D array

```

In [3]: # Access a location and read its value
value = world[3][0]
print('\n')
print('Value at index [3, 0] = ' + str(value))

# Read the first three items in the 3rd row
row = 2
column_index = 0
value_left = world[row, column_index]
value_middle = world[row, column_index + 1]
value_right = world[row, column_index + 2]

print('\nThe first three values in row 2 : ' + str(value_left) + ', '
    + str(value_middle) + ', '
    + str(value_right) )

# Compare the first two values and print the result
compare = world[0][0] == world[0][1]
print('\nDo the first two values match? ' + str(compare))

```

```
Value at index [3, 0] = 0
```

```
The first three values in row 2 : 0, 1, 1
```

```
Do the first two values match? True
```

1.0.4 Change an Array and Plant a Tree!

```
In [5]: # Define a function to plant a tree
# and change the value of the array in that location
def plant_tree(y, x):
    # check that the indices are in the boundaries of the array dimensions
    if(y < world.shape[0] and x < world.shape[1]):
        world[y,x] = 1
        print('\n' + str(world)) # prints a newline and the current world

    # Call the function at the location x = 3, and y = 2
    # You can call this multiple times in a row
    plant_tree(0, 2)
```

```
[[0 0 1 1 0]
 [0 0 0 1 0]
 [0 1 1 0 0]
 [0 0 0 0 1]
 [1 0 0 1 0]
 [1 0 0 0 0]]
```

```
In [ ]:
```

2D Iteration, demonstration

March 22, 2020

1 2D Iteration

In this demonstration, you'll see how to iterate through a 2D array used nested loops, and we'll be using our iterative skills to locate trees in this world!

1.0.1 Create the world

In [1]: `import numpy as np`

```
# A 6x5 robot world
world = np.array([
    [0, 0, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 0, 0],
    [0, 0, 0, 0, 1],
    [1, 0, 0, 1, 0],
    [1, 0, 0, 0, 0]
])

# Print out some information about the world
print(world)
print('\nThe shape of this array is: ' + str(world.shape))

[[0 0 0 1 0]
 [0 0 0 1 0]
 [0 1 1 0 0]
 [0 0 0 0 1]
 [1 0 0 1 0]
 [1 0 0 0 0]]
```

The shape of this array is: (6, 5)

1.0.2 Iterate through the items in the world

```
In [2]: # This function uses nested for loops and knowledge
        # about the shape of the array to print out each item with its index
def iterate2D(world):
    # y-dimension (rows)
    for i in range(0, world.shape[0]):
```

```

# x-dimension (columns)
for j in range(0, world.shape[1]):
    print('Index ['+str(i)+'][ '+str(j)+'] = ' +str(world[i][j]))

# Call the iterate function
print('\n')
iterate2D(world)

Index [0] [0] = 0
Index [0] [1] = 0
Index [0] [2] = 0
Index [0] [3] = 1
Index [0] [4] = 0
Index [1] [0] = 0
Index [1] [1] = 0
Index [1] [2] = 0
Index [1] [3] = 1
Index [1] [4] = 0
Index [2] [0] = 0
Index [2] [1] = 1
Index [2] [2] = 1
Index [2] [3] = 0
Index [2] [4] = 0
Index [3] [0] = 0
Index [3] [1] = 0
Index [3] [2] = 0
Index [3] [3] = 0
Index [3] [4] = 1
Index [4] [0] = 1
Index [4] [1] = 0
Index [4] [2] = 0
Index [4] [3] = 1
Index [4] [4] = 0
Index [5] [0] = 1
Index [5] [1] = 0
Index [5] [2] = 0
Index [5] [3] = 0
Index [5] [4] = 0

```

1.0.3 Find the first tree, 1, in the world

```

In [3]: # This function is similar to our iterate2D function,
        # But looks for the first tree in the array and prints its location [x][y]
def first_tree(world):
    # iterates through all indices starting at the top-left [0][0]

```

```
for i in range(0, world.shape[0]):  
    for j in range(0, world.shape[1]):  
        # check if a tree is found  
        if(world[i][j] == 1):  
            # if so, print the index and leave the loop with a return statement  
            print('First tree found at location: ['+str(i)+']['+str(j)+']')  
            return  
  
# Call the first_tree function  
print('\n')  
first_tree(world)
```

First tree found at location: [0] [3]

In []:

why use numpy arrays

March 22, 2020

1 Why use numpy arrays?

Numpy is one of those tools that will keep showing up as you learn about self driving cars. This is because numpy arrays tend to be:

1. **compact** (they don't take up as much space in memory as a Python list).
2. **efficient** (computations usually run quicker on numpy arrays than Python lists).
3. **convenient** (which we'll talk about more now).

```
In [1]: # consider this 2d python grid (list of lists)
grid = [
    [0, 1, 5],
    [1, 2, 6],
    [2, 3, 7],
    [3, 4, 8]
]

# It's easy to print, for example, row number 0:
print(grid[0])

[0, 1, 5]
```

```
In [2]: # but how would you print COLUMN 0? In numpy, this is easy

import numpy as np

np_grid = np.array([
    [0, 1, 5],
    [1, 2, 6],
    [2, 3, 7],
    [3, 4, 8]
])

# The ':' usually means "*all values"
print(np_grid[:,0])
```

```
[0 1 2 3]
```

```
In [3]: # What if you wanted to change the shape of the array?
```

```
# For example, we can turn the 2D grid from above into a 1D array  
# Here, the -1 means automatically fit all values into this 1D shape  
np_1D = np.reshape(np_grid, (1, -1))  
  
print(np_1D)
```

```
[[0 1 5 1 2 6 2 3 7 3 4 8]]
```

```
In [ ]: # We can also create a 2D array of zeros or ones  
# which is useful for car world creation and analysis
```

```
# Create a 5x4 array  
zero_grid = np.zeros((5, 4))  
  
print(zero_grid)
```

```
In [ ]:
```

discrete_probability_exercise

March 22, 2020

1 Discrete Probability Exercise

You have two dice. These are fair, six-sided dice with the values one through six.

You take the dice, and you roll them. Then you take the sum of the two numbers. For example, if you roll the dice and get 5 and 4, then the sum would be 9.

Now, you're curious to know what sum or sums are most likely to occur. So you decide to repeat your experiment two-thousand times and then analyze the results.

You roll the dice over and over again. Each time you write down what sum you got.

2 Results of your experiment

Here are the results of your experiment. The left hand column shows the sum and the right hand column shows how many times that sum appeared over your two-thousand trials:

Dice Sum	Count
2	54
3	111
4	163
5	222
6	277



alt text

Dice Sum	Count
7	336
8	276
9	220
10	171
11	111
12	59

Your task is to analyze these results and turn them into a discrete probability distribution.

Follow along the Ipython notebook completing each exercise and demo. We've provided answers to the exercises in the next part of the lesson.

3 Exercise: Create a List

In order to analyze these results, you'll need to put them into some sort of data structure. For this case, use a Python list.

A Python list is just what it sounds like: a list of values. If you had the values 1 5 19 25 and 30, here is how you could put them in a Python list:

```
listexample = [1, 5, 19, 25, 30]
```

In [29]: #####

```
# TODO: Create a Python list of the values in the 'Count' column.  
# Your list should start with 54 and follow the same order  
# as the data in the column: 54, 111, 163, etc.  
####
```

```
count_data = [54, 111, 163, 222, 277, 336, 276, 220, 171, 111, 59]
```

4 Demo: Print Your List

Run the code cell below to print out your list. This cell uses a for loop to iterate through every value in your list. Pay attention to the syntax because you'll be writing your own for loop soon.

In [30]: # Run this code cell. You do not need to change anything

```
# A for loop to print out every value in the count_data list  
# The len() function determines the size of the list  
# The range() function creates an integer  
#         list from 0 to len(count_data).  
  
for i in range(len(count_data)):  
    print(count_data[i])
```

54
111

```
163  
222  
277  
336  
276  
220  
171  
111  
59
```

5 Exercise: Sum the Counts

Let's double check how many trials you actually did. In this exercise, you'll calculate the sum of your count_data list. This sum represents the total number of times you rolled the dice.

Finding the sum of a Python list is relatively straightforward. The syntax is:

```
sum(list_variable)

In [31]: ###
# TODO: calculate the sum of the count_data list
#       and put the sum in the total_count variable
####

total_count = sum(count_data)

# This will print out the result
print(total_count)
```

2000

6 Demo: Visualization of the Data

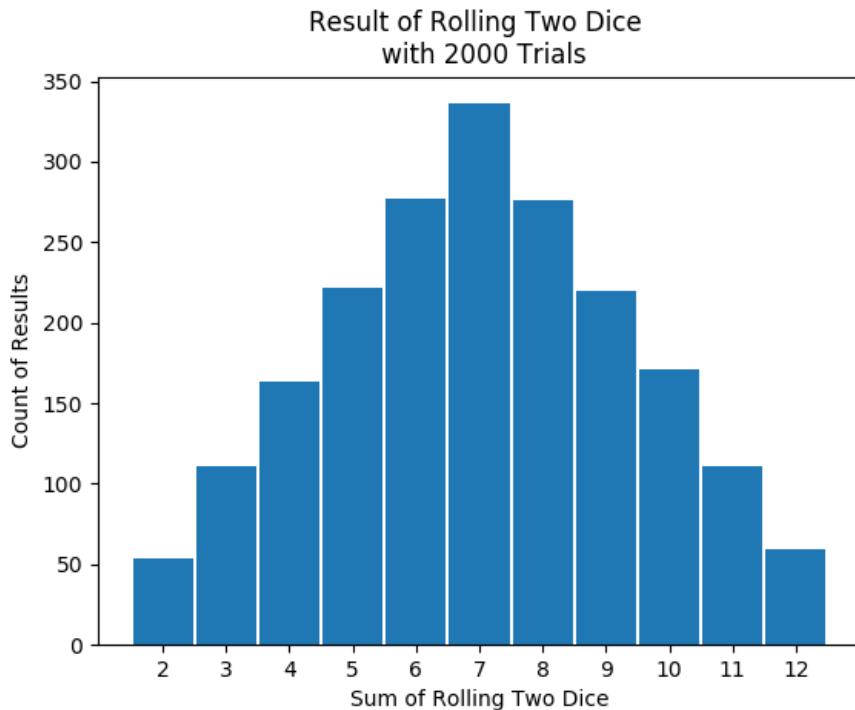
If you look at the data, you can already tell which case has the highest probability of occurring: 7.

And it also looks like the probability distribution is symmetrical with 6 and 8 having the same probability, 5 and 9, etc. Here is a visualization of the data:

But is this visualization a discrete probability distribution? Here is a reminder of three important characteristics of discrete probability distributions.

- For all values on the x-axis, the y value is greater than or equal to 0.
- For each x, the probability $p(x)$ is equal to the y value
- The sum of all y values is 1

It looks like this visualization violates the second and third criteria. The sum of the y-values is 2000 not 1. And the y values represent counts not probability.



Visualization

7 Exercise: Calculate a Discrete Probability Distribution

How can you convert the data into a probability distribution? Currently, the sum of all the y-values is 2000, but the sum needs to equal 1.

What happens if you divide each y-value by the total count of 2000?

Like 54/2000, 111/2000, 163/2000, etc.?

It turns out that for discrete variables, dividing each y-value by the total count will give you a discrete probability distribution. Dividing each value by the total is called normalization.

In this exercise, you'll use a for loop to divide each value in your list by the total count. You'll put the results in a new list called `discrete_probability`.

Here is some example code to give you an idea of how to create the new list that will hold the normalized counts:

```
# a python list
mylist = [1, 2, 3, 4, 5]

# an empty python list
newlist = []

# for loop
for i in range(len(mylist)):
    newlist.append(mylist[i])
```

In [32]: ####

```

# INSTRUCTIONS: Use a for loop to iterate through the
#      count_data list
#
# For each value in the list, divide the value by the total_count
# variable.
#
# You will need to append the results to a new list
#
#####

normalized_counts = []

# TODO: Write a for loop to iterate through the count_data list.
#       Use the for loop example given previously to help
#       get yourself started

for i in range(0, len(count_data)):

    # TODO: Inside the for loop, divide each value in
    # count_data by the total_count variable and append
    # the result to the normalized_counts variable.

    normalized_counts.append(count_data[i] / total_count)

print('Here are the normalized counts: ')
print(normalized_counts)

```

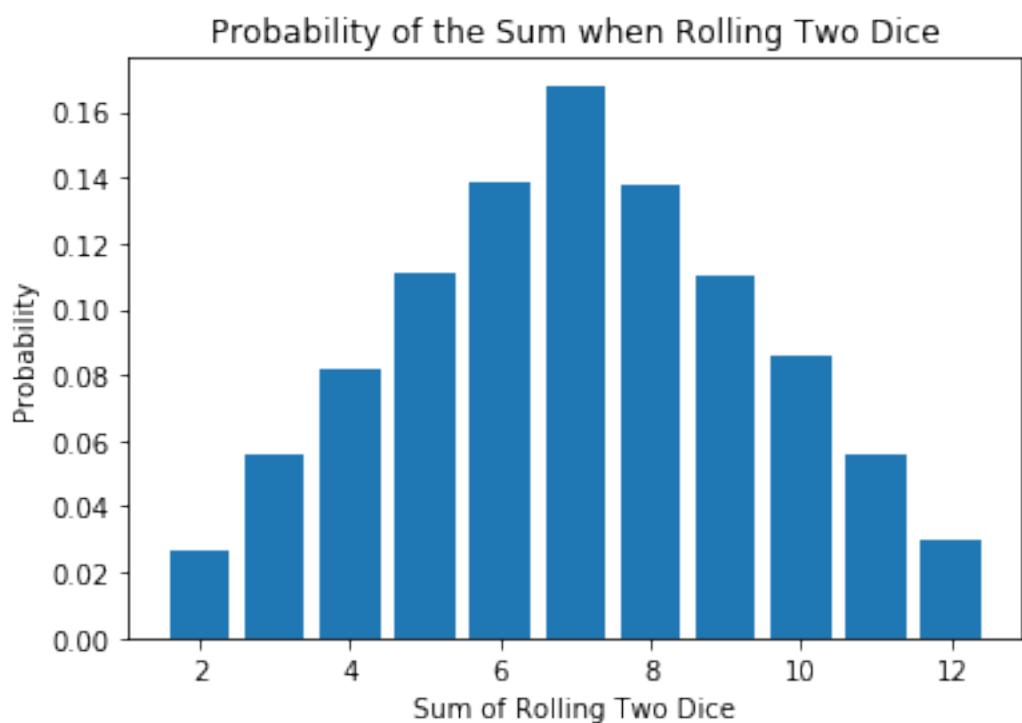
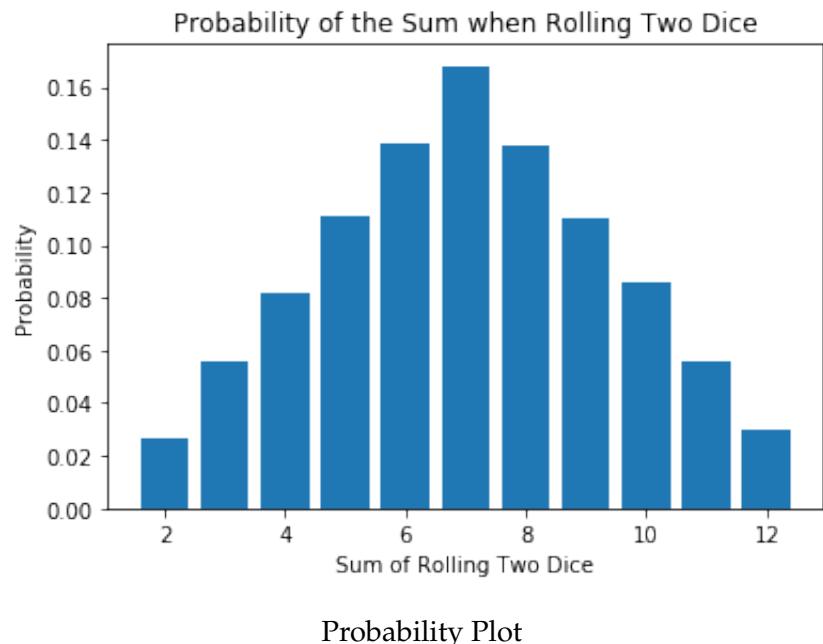
Here are the normalized counts:

[0.027, 0.0555, 0.0815, 0.111, 0.1385, 0.168, 0.138, 0.11, 0.0855, 0.0555, 0.0295]

8 Demo: Plot Your Results

We have written code for you that will plot your results. Run the cell below to see the plot. If you've normalized the counts correctly, your output should look like this plot

```
In [33]: import plot
        %matplotlib inline
        plot.plot_probability(normalized_counts)
```



9 Check your Results

Run the code below to check your results. If the code cell prints out "Awesome work!", then your results were what we expected. We've also provided answer code in the next part of the lesson.

```
In [34]: assert count_data == [54, 111, 163, 222, 277, 336, 276, 220, 171, 111, 59], "The count_"
         assert total_count == 2000, "The total_count variable is not correct."
         assert normalized_counts == [54/2000, 111/2000, 163/2000, 222/2000, 277/2000, 336/2000,
                                         print('Awesome work! Your answers are what we expected.')
```

Awesome work! Your answers are what we expected.

```
In [ ]:
```

1_math_in_python_demonstration

March 22, 2020

1 Math in Python [demonstration]

The code below demonstrates some common mathematical operations in python as well as usage of Python's `isinstance` function.

1.0.1 Math in Python

Run the cell below to see examples of using math in Python. The math library contains many methods including a method that ouputs the value of pi.

```
In [1]: import math

radius      = 10.0
diameter    = 2 * radius
circumference = 2 * math.pi * radius
area        = math.pi * radius ** 2

print("Radius is", radius)
print("Diameter is", diameter)
print("Circumference is", circumference)
print("Area is", area)

Radius is 10.0
Diameter is 20.0
Circumference is 62.83185307179586
Area is 314.1592653589793
```

1.0.2 `isinstance()` function

In the next code cell, you will see how the `isinstance()` function works.

The `isinstance()` function checks to see if a variable holds a certain type of value like an integer, float, string, etc.

```
In [2]: sqr_root_2 = math.sqrt(2)
is_sqr_root_2_an_integer = isinstance(sqr_root_2, int)
print("Is square root two an integer?", is_sqr_root_2_an_integer)
```

```
is_sqr_root_2_a_float = isinstance(sqr_root_2, float)
print("Is square root two an integer?", is_sqr_root_2_a_float)
```

```
Is square root two an integer? False
Is square root two a float? True
```

In []:

2_uniform_distribution

March 22, 2020

1 Uniform Probability Distribution

1.0.1 Your Task

Let's get started! In this exercise, you'll write a function for calculating continuous uniform probability distributions.

1.0.2 Continuous Uniform Distribution

From the lesson videos, the wheel of fortune and spinning bottle examples both had continuous uniform distributions. You spin, and the arrow or bottle had an equal chance of stopping anywhere between 0 and 360 degrees.

1.0.3 Instructions

For this first programming exercise, write a function that calculates the probability that the arrow will stop between two angles. You'll want this function to work with any continuous uniform distribution not just a wheel or bottle.

As an example: the probability that the bottle stops between 20 and 30 degrees is $P = (30 - 20) / (360 - 0)$.

The function has four inputs: * `low_range` representing the first angle of interest * `high_range` representing the second angle * `minimum` representing the minimum value of the distribution (0 for the spinning example) * `maximum` representing the maximum value of the distribution (360 for the spinning example)

The output should be the probability that the arrow stops between `low_range` and `high_range`.

Assume `low_range < high_range`.

```
In [1]: def probability_uniform(low_range, high_range, minimum, maximum):  
  
    ## TODO: Calculate the probability of an event occurring  
    ## between low_range and high_range.  
    ## Assume the user has given valid inputs such that low_range < high_range.  
    ##     minimum < maximum  
    ##  
  
    probability = (high_range - low_range) / (maximum - minimum)  
  
    return probability
```

2 Test Your Results

Run the code cell below to test your results. If you get an assertion error, that means your answer was not what we expected.

```
In [2]: ## TODO: Test your results by running this cell.  
## If the cell produces no output, your answer was as expected  
  
assert "{0:.2f}".format(probability_uniform(15, 305, 0, 360)) == '0.81'  
assert "{0:.2f}".format(probability_uniform(1, 5, 0, 10)) == '0.40'  
assert "{0:.2f}".format(probability_uniform(55, 70, 20, 300)) == '0.05'  
print('Great work! Your code outputs the expected results.')
```

Great work! Your code outputs the expected results.

```
In [ ]:
```

3_function_improvements

March 22, 2020

1 Problems with your probability_uniform Function

Your `probability_uniform` function should work at this point. But, you might run into a couple of problems.

1. What happens if you call your function like this: `probability_range(35, 20, 0, 360)` ?
2. What happens if you input an angle that is outside the possibilities of the bottle's possible outcomes like `probability_range(-25, 390, 0, 360)`?
3. What if you call your function like this: `probability_range('a', 'b', 0, 360)`?

When writing functions, it's important to think of edge cases or incorrect user input.

2 Your Task

Your task is to improve your function. Let's make these changes one step at a time.

3 Task One

Make sure the function outputs a valid probability when `low_range` is greater than `high_range`.

There is more than one way to go about this. You could:

1. Calculate the absolute value of `high_range - low_range`
2. Use if statements to compare `low_range` and `high_range` to see which one is greater

Here is a [link](#) to using Python's absolute value function.

Fill out the TODOs below.

```
In [17]: def probability_range_improved(low_range, high_range, minimum, maximum):  
  
    # TODO: calculate and return the probability  
    # even if low range is greater than high range.  
    # Use the abs() function or if statements  
  
    probability = abs(high_range - low_range) / (maximum - minimum)  
    return probability
```

Run the cell below to see if your function works as expected.

```
In [18]: assert "{0:.2f}".format(probability_range_improved(25, 700, 5, 800)) == '0.85'  
        assert "{0:.2f}".format(probability_range_improved(700, 25, 5, 800)) == '0.85'  
        print('Nice work!')
```

Nice work!

4 Task Two

Check the inputs to the function to make sure they are not strings. If the user inputted a string, the function should return None.

This exercise might seem trivial, but if you try to do something like 'my_string'/2 in Python, you will get an error. Debugging the errors and avoiding them is a key programming skill.

Hint: Use the Python `isinstance()` function. If you're not sure how to use this function, search for it online.

Hint: Use your code from Task One to calculate the probability

```
In [19]: def probability_range_improved(low_range, high_range, minimum, maximum):  
  
    # TODO: check if any of the inputs are strings.  
    # hint: the python function isinstance() will be useful  
    if(isinstance(low_range, str) or isinstance(high_range, str) or isinstance(minimum,  
        # print a message to the user and return none  
        print('Inputs should be numbers not string')  
        return None  
  
    probability = abs(high_range - low_range) / (maximum - minimum)  
    return probability
```

Run the next cell to check your results.

```
In [20]: assert probability_range_improved('a', 0, -100, 500) == None  
        assert probability_range_improved(5, 'b', -100, 500) == None  
  
        assert "{0:.2f}".format(probability_range_improved(25, 700, 5, 800)) == '0.85'  
        assert "{0:.2f}".format(probability_range_improved(700, 25, 5, 800)) == '0.85'  
  
        print('Well done!')
```

```
Inputs should be numbers not string  
Inputs should be numbers not string  
Well done!
```

5 Task Three

Check that the user has only inputted `low_range` and `high_range` values that are in between the allowed minimum and maximum. If an input is out of the allowed range, return None.

Hint: Use your code from Task One to calculate the probability

```
In [28]: def probability_range_improved(low_range, high_range, minimum, maximum):

    # TODO check that low_range is between minimum and maximum
    if (low_range < minimum or low_range > maximum):
        # print a message to the user and return none
        print('Your low range value must be between minimum and maximum')
        return None

    if (high_range < minimum or high_range > maximum):
        # print a message to the user and return none
        print('The high range value must be between minimum and maximum')
        return None

    probability = abs(high_range - low_range) / (maximum - minimum)
    return probability
```

Run the next code cell to check your results.

```
In [29]: assert probability_range_improved(-100, 300, 100, 500) == None
assert probability_range_improved(105, 700, 100, 500) == None

assert "{0:.2f}".format(probability_range_improved(25, 700, 5, 800)) == '0.85'
assert "{0:.2f}".format(probability_range_improved(700, 25, 5, 800)) == '0.85'

print('Awesome!')
```

```
Your low range value must be between minimum and maximum
The high range value must be between minimum and maximum
Awesome!
```

6 Task Four

Take all of your work from task one, two and three. Put them into one final function.

```
In [30]: def probability_range_improved(low_range, high_range, minimum, maximum):

    if (isinstance(low_range, str) or isinstance(high_range, str)):
        # print a message to the user and return none
        print('Inputs should be numbers not string')
        return None

    if (low_range < minimum or low_range > maximum):
        # print a message to the user and return none
        print('Your low range value must be between minimum and maximum')
        return None

    if (high_range < minimum or high_range > maximum):
```

```

# print a message to the user and return none
print('The high range value must be between minimum and maximum')
return None

# TODO: calculate and return the probability
# even if low range is greater than high range
probability = abs(high_range - low_range) / (maximum - minimum)
return probability

```

Run the cell below. If there are no AssertionErrors, then your code runs as expected. In Python, assert checks whether a statement resolves to True or False

```

In [31]: assert probability_range_improved('a', 0, -100, 500) == None
assert probability_range_improved(0, 'b', -100, 500) == None
assert probability_range_improved(-100, 300, 100, 500) == None
assert probability_range_improved(105, 700, 100, 500) == None
assert "{0:.2f}".format(probability_range_improved(25, 700, 5, 800)) == '0.85'
assert "{0:.2f}".format(probability_range_improved(700, 25, 5, 800)) == '0.85'
print('You got the results we were looking for!')

```

```

Inputs should be numbers not string
Inputs should be numbers not string
Your low range value must be between minimum and maximum
The high range value must be between minimum and maximum
You got the results we were looking for!

```

```
In [ ]:
```

```
In [ ]:
```

plotting_in_python

March 22, 2020

1 Demo: Making Visualizations in Python

The next part of the lesson is about visualizing probability distributions.

Python has a very useful but somewhat complicated library for creating visualizations called [matplotlib](#).

One of the best ways to learn matplotlib is by looking at examples. If you search the internet for "matplotlib bar chart" or "matplotlib scatter plot", you will find many code examples that you can learn from.

To get you started, we are going to show you a few plots using matplotlib. These examples will help you for the next few parts of the coding lessons.

2 Example: ScatterPlot

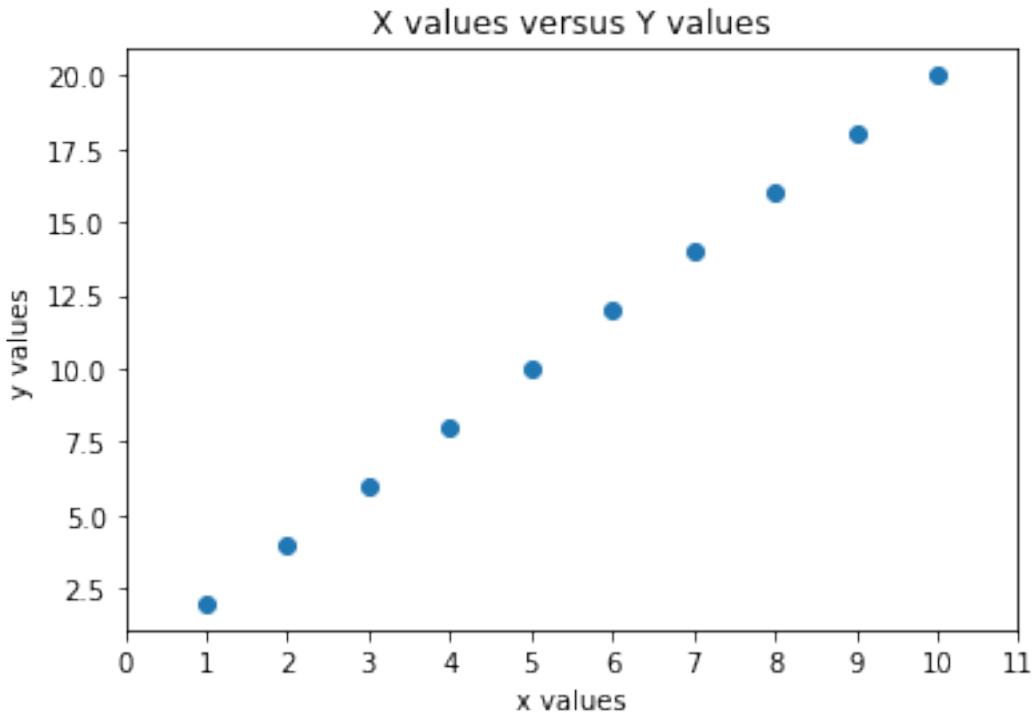
One of the most common plots is an x-y scatterplot. The code below will give you a sense for how matplotlib works. You'll see that you build up a plot piece by piece.

We are using some made-up data for the x and y positons of the points. Run the code below, and then we'll explain what each line is doing.

```
In [1]: import matplotlib.pyplot as plt
        %matplotlib inline

x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

plt.scatter(x, y)
plt.xlabel('x values')
plt.ylabel('y values')
plt.title('X values versus Y values')
plt.xticks([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
plt.show()
```



3 Explanation of the Code

```
import matplotlib.pyplot as plt
```

The import statement makes the matplotlib library available to your program. 'as plt' means that we can refer to the library as plt instead of by its full name.

```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Here we assign values to be plotted. If you think in terms of (x, y), then the points would be (1, 2) then (2, 4) then (3, 6), etc. However, matplotlib expects the x values and y values to be in separate lists.

```
plt.scatter(x, y)
```

The plt.scatter(x,y) line tells Python to create a scatter plot.

```
plt.xlabel('x values')
plt.ylabel('y values')
plt.title('X values versus Y values')
```

These lines put labels on the x-axis, y-axis and gives the chart a title.

```
plt.xticks([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```

This line manually sets the x-tick marks.

```
plt.show()
```

And finally, plt.show() outputs the chart

4 Example: Bar Chart

What if we take the same x and y values but instead create a bar chart? Only two things changed in the code below.

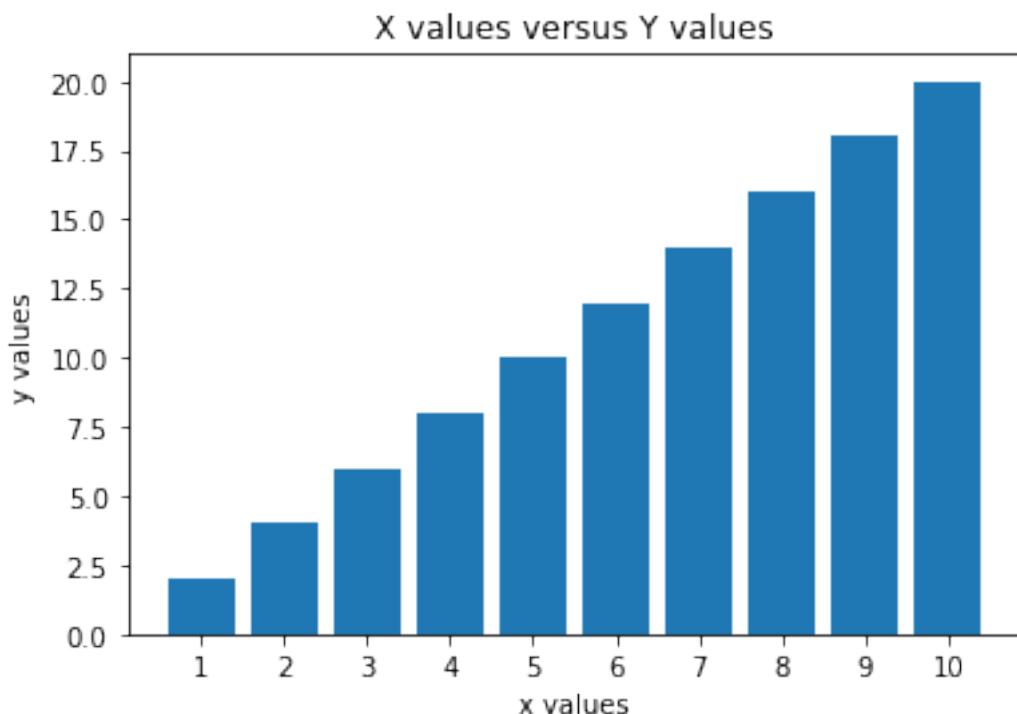
Instead of a scatter plot, we've created a bar chart. plt.bar(x,y)

And we modified the x tick marks so that 0 and 11 were not included. Run the code below to see what happens.

```
In [2]: import matplotlib.pyplot as plt
%matplotlib inline

x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

plt.bar(x, y)
plt.xlabel('x values')
plt.ylabel('y values')
plt.title('X values versus Y values')
plt.xticks([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
plt.show()
```



5 Example: Bar Chart Again

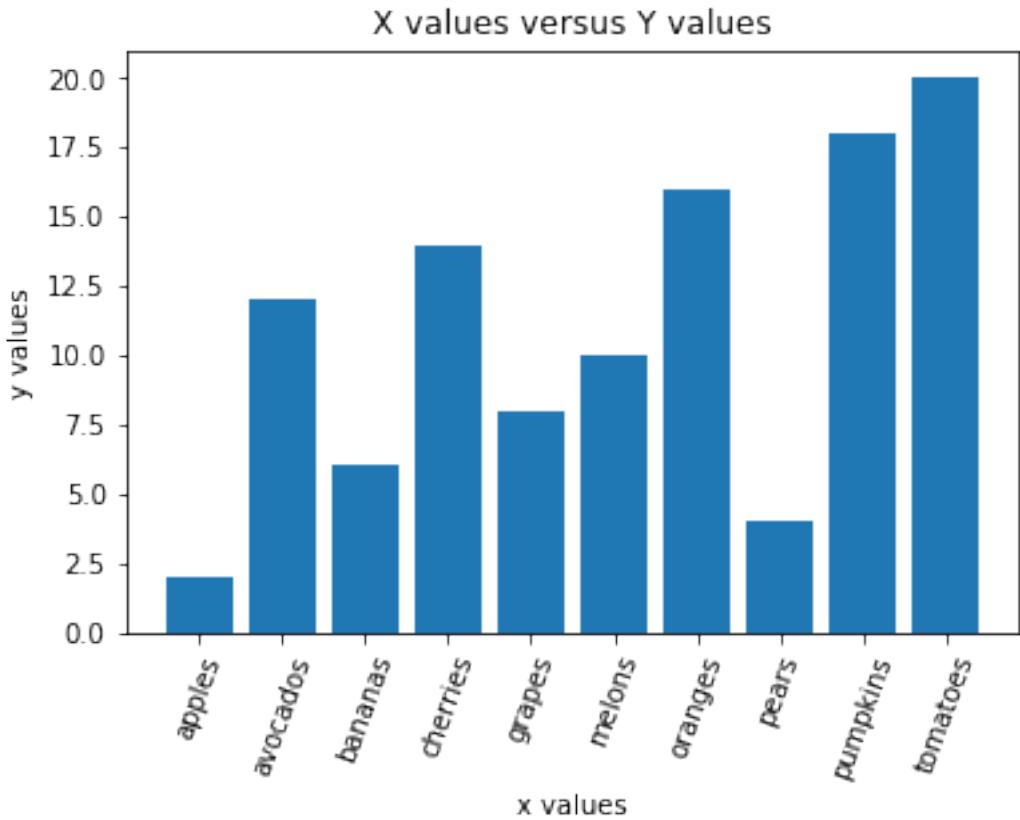
If you're familiar with bar charts, then you might remember that the x-axis is actually a discrete variable. Take a look at the code below to see how the visualization changes. The major change in the code was that the x values are now strings instead of numerical.

Run this code cell below.

```
In [3]: import matplotlib.pyplot as plt
%matplotlib inline

x = ['apples', 'pears', 'bananas',
      'grapes', 'melons', 'avocados', 'cherries', 'oranges', 'pumpkins',
      'tomatoes']
y = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

plt.bar(x, y)
plt.xlabel('x values')
plt.ylabel('y values')
plt.title('X values versus Y values')
plt.xticks(rotation=70)
plt.show()
```



Matplotlib sorts the x values alphabetically.

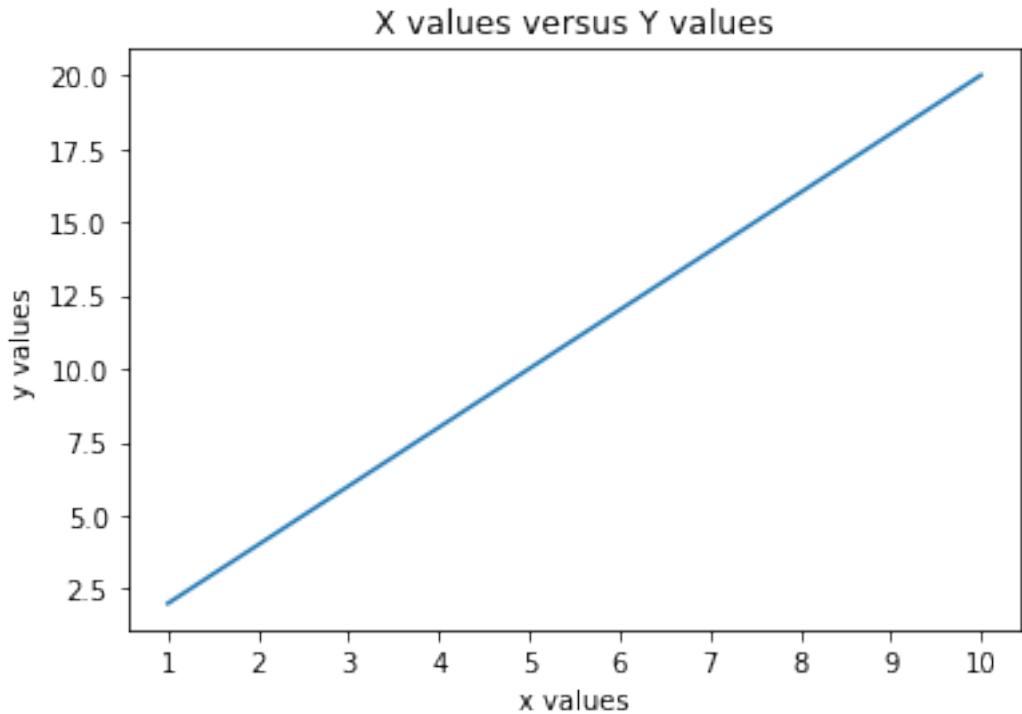
6 Example: Line Chart

For the final example, run the code cell below. It outputs a line plot.

```
In [4]: import matplotlib.pyplot as plt
%matplotlib inline

x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
y = [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]

plt.plot(x, y)
plt.xlabel('x values')
plt.ylabel('y values')
plt.title('X values versus Y values')
plt.xticks([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
plt.show()
```



The only line of code that changed was `plt.plot(x, y)` instead of `plt.bar(x, y)`.
Matplotlib automatically outputs a line chart when you call `plt.plot()`.
Now that you are familiar with matplotlib, it's time to make some visualizations!

In []:

5_visualize_discrete_distribution

March 22, 2020

1 Instructions

Write a function that, given a list of x-axis intervals, relative probabilities and a total probability, calculates the height of each bar. Remember that the sum of the area for all bars should be the total probability.

Here is an example input and output:

- * a vehicle accident is 5 times more likely from 5am to 10am versus midnight to 5am.
- * a vehicle accident is 3 times more likely from 10am to 4pm versus midnight to 5am.
- * a vehicle accident is 6 times more likely from 4pm to 9pm versus midnight to 5am.
- * a vehicle accident is 1/2 as likely from 9pm to midnight versus midnight to 5am.

* The probability of getting in an accident on any given day is .05

The inputs would look like this.

For the hours, you can use 24 hour time hour_intervals = [0, 5, 10, 16, 21, 24]

relative_probabilities = [1, 5, 3, 6, 0.5]

total_probability = 0.05

The output would be the height of each bar:

```
[0.0006451612903225806,  
 0.0032258064516129032,  
 0.0016129032258064516,  
 0.003870967741935484,  
 0.0005376344086021505]
```

At the end of the exercise, your visualization should look like this:

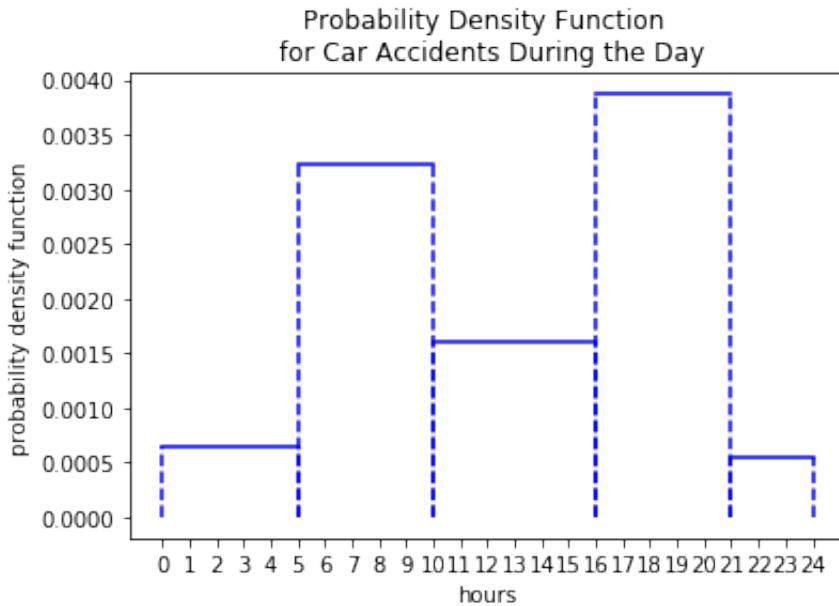
2 Hints

- Summing the area of all the bars equals the total probability, which in this case is 0.05.
- The relative probabilities and total probability can be used to find the exact area of each bar.
If the area of the first bar is A, then the area of the second bar is 5A, the third bar is 3A, etc.
- Once you know the area of each bar, you can divide each area by its width to calculate the bar height.

2.1 Function Inputs and Outputs

The bar_heights() function below has three inputs

- * intervals - representing the x-axis intervals for each bar
- * relative_probabilities - representing the relative probabilities for each interval
- * total_probability - representing the total area of all the bars



piece-wise continuous visualization

The `bar_heights()` function has one output * heights - a list of each height for each interval in the probability density function

Fill in the TODOs to get the function working

```
In [10]: def bar_heights(intervals, relative_probabilities, total_probability):

    heights = []

    #TODO: sum the relative probabilities
    total_relative_prob = sum(relative_probabilities)

    for i in range(0, len(relative_probabilities)):

        #TODO: Looping through the relative_probabilities list,
        #      take one probability at a time and
        #      calculate the area of each bar. Think about how you can
        #      calculate the area of a bar knowing the total_probability,
        #      relative probability, and the sum of the relative probabilities.

        #HINT: It's possible to do this in one line of code

        bar_area = (relative_probabilities[i] / total_relative_prob) * total_probabiliti

        # TODO: Calculate the height of the bar and append the value to the
        # heights list. Remember that the area of each bar
        # is the width of the bar times the height of the bar

        #HINT: It's possible to do this in one line of code
```

```

        heights.append(bar_area / (intervals[i+1] - intervals[i]))

    return heights

```

Run the next cell to test out your function

```
In [11]: print(bar_heights([0, 5, 10, 16, 21, 24], [1, 5, 3, 6, 0.5], 0.05))
```

```
[0.0006451612903225806, 0.0032258064516129032, 0.0016129032258064516, 0.003870967741935484, 0.00
```

2.1.1 Visualize Results

Once the bar_heights function is working, here is some code to visualize your results.

```

In [12]: import matplotlib.pyplot as plt
          %matplotlib inline

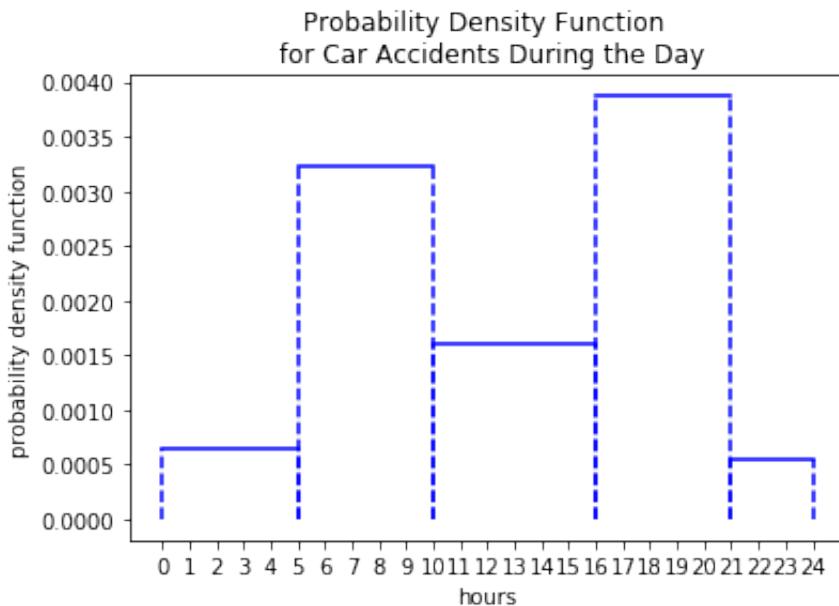
hour_intervals = [0, 5, 10, 16, 21, 24]
probability_intervals = [1, 5, 3, 6, 1/2]
accident_probability = 0.05

heights = bar_heights(hour_intervals, probability_intervals, accident_probability)

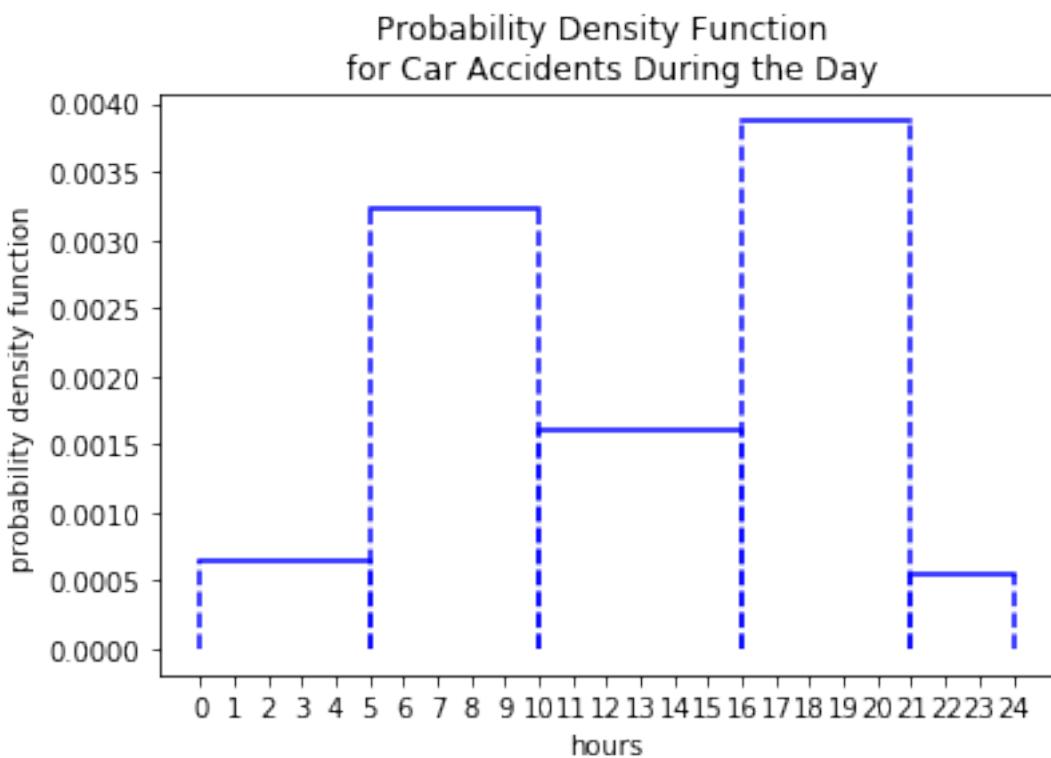
for i in range(len(probability_intervals)):
    plt.plot([hour_intervals[i], hour_intervals[i+1]], [heights[i], heights[i]], color='black')
    plt.plot([hour_intervals[i], hour_intervals[i]], [0, heights[i]], '--', color='blue')
    plt.plot([hour_intervals[i+1], hour_intervals[i+1]], [0, heights[i]], '--', color='red')

plt.xticks(range(0,25,1))
plt.xlabel('hours')
plt.ylabel('probability density function')
plt.title('Probability Density Function \n for Car Accidents During the Day')
plt.show()

```



piece-wise continuous visualization



When you run the code cell above, the visualization should look like this:

In []:

6_1D_selfdrivingcar

March 22, 2020

1 Robot World 1-D

1.0.1 Introduction

In the nanodegree, you are going to see robot probability distributions represented two different ways: * with discrete probability distributions * and with continuous probability distributions

Discrete probability distributions are used when you track a robot's movement across a map divided into square grids. Each grid is a discrete location where the robot could be located.

In this exercise, you'll work with a probability distribution representing the uncertainty in a robot's location.

1.1 Robot Initialization

Imagine you have a robot living in a 1-D world. The robot lives on a grid with nine different spaces. The robot can only move forwards or backwards. If the robot falls off the grid, it will loop back around to the other side.

The robot has a map so that it knows there are only nine spaces. But the robot does not know where it is on the map. Here is the 1-D map.

When the robot first turns on, the probability that the robot is on any one of these spaces is $1/9$; the implication is that the robot has an equal probability of being at any of the spaces on the grid.

2 Exercise 1 - Initial Probability

Now, write a function that receives the number of spaces in the robot's world and then returns a list containing the initial probability for each space on the grid.

So in the example given so far, there would be a list with nine probabilities. And every value in the list would be $1/9$. Remember, because the robot does not know where it is at first, the probability of being in any space is the same.



1-D Robot World

Python's `list.append()` method might be useful.

```
In [7]: import matplotlib.pyplot as plt
%matplotlib inline

import numpy as np

def initialize_robot(grid_size):

    grid = []

    # TODO: for each space on the map grid, store the initial probability
    # in the grid list. For example, if there are eight spaces on the grid,
    # the grid list should have eight entries where each entry represents
    # the initial probability of the robot being in that space.
    for i in range(grid_size):
        prob = 1 / grid_size
        grid.append(prob)

    return grid
```

Run the cell below to make sure that your function outputs the correct results.

```
In [8]: # Result should be a list with 9 elements all having value 1/9
assert initialize_robot(9) == [1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9]

# Result should be a list with 4 elements all having value 1/4
assert initialize_robot(4) == [1/4, 1/4, 1/4, 1/4]

print('Hooray! You just initialized a discrete probability distribution')
```

Hooray! You just initialized a discrete probability distribution

3 Exercise 2 - Grid Probability

Now, write a function called `grid_probability` that outputs the probability that the robot is at a specific point on the grid. The input to this function will be:

- the output of the previous function (ie a list representing a 1-D map of probabilities)
- the grid number where you want to know the probability

So if you wanted to know the current probability that the robot is at the fifth tile on the grid, you would call the function like:

```
grid_probability(my_grid, 4).
```

Why would the function input be 4 instead of 5? Think about how Python accesses values in a list. Typing `mylist[0]` gives you the first element in the list. Typing `mylist[1]` gives you the second element in the list.

```
In [20]: def grid_probability(grid, position):

    #####
    # TODO: Use an if statement to make sure that the position input
    # does not go beyond the size of the list. Say the list has five elements
    # and your code tries to access grid[5] or grid[6]. That will lead to an
    # error.

    if(position > len(grid)):
        return None

    # TODO: If the position input is legitimate, then return the probability
    # stored at that position. If the position input is not legitimate, then
    # return None
    #####
    return grid[position]
```

Run the cell below to test the results of your code. If the grid_probability function works as expected, the code cell should print out "Awesome work!".

```
In [21]: assert grid_probability([.1, .1, .2, .1, .5], 2) == 0.2
assert grid_probability([.1, .1, .2, .1, .5], 7) == None
print('Awesome work!')
```

Awesome work!

4 Exercise 3 - Visualize Robot World

Next, write a function that outputs a bar chart showing the probabilities of each grid space.

Remember that we are working with a discrete probability distribution; the robot location can only take on certain values ie square 1, square 2, square 3, square 4, etc.

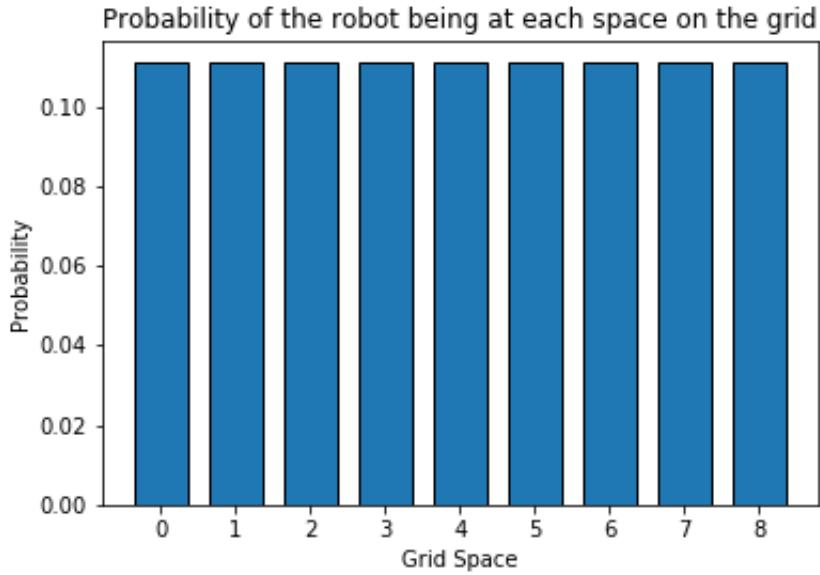
The grid number would be on the x-axis. For a discrete probability distribution, the y-axis represents probability.

The input to the function is a list with the probability that the robot is at each point on the grid. Your result should look something like this:

```
In [18]: import matplotlib.pyplot as plt
%matplotlib inline

import numpy as np

def output_map(grid):
    ###
```



Uniform Probability Distribution

```

# TODO: Start by creating a list to represent the x-axis labels.
# For example, if the grid variable has length 5, x_labels would contain
# a list [0, 1, 2, 3, 4].
#
# HINT: Python's built in len() and range() functions might
#       be useful. If you are not sure how to use these
#       functions, look them up in a search engine.
#       For example, in google, look up "Python len".
####

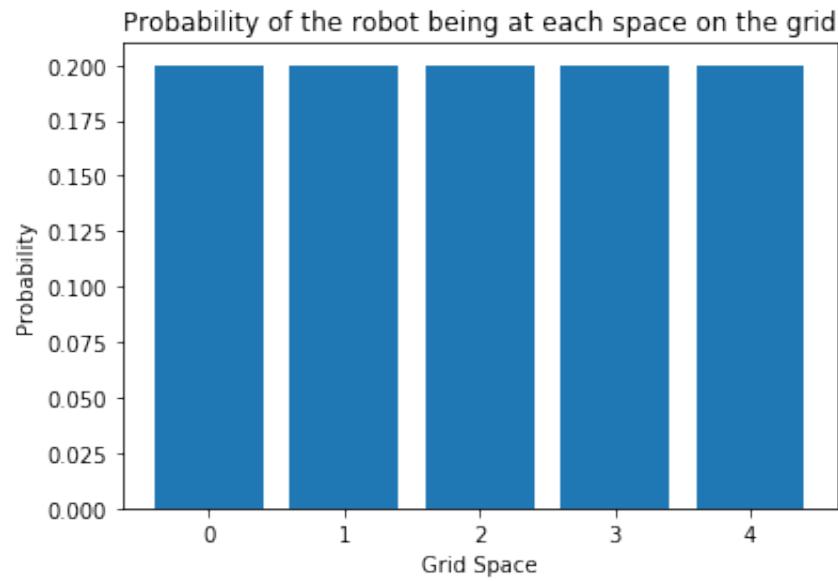
x = []
y = []
for i in range(len(grid)):
    x.append(i)
    y.append(1 / len(grid))

###
# TODO: Using matplotlib, output a bar chart of the results.
# Notice that we have already imported the matplotlib library
# at the top of this code cell.

# If you are unsure how to make a bar chart, go back to the
# "Plotting in Python demonstration" to see an example.

# Make sure your plot has an xlabel, a ylabel, and a title
# Don't forget that the last line needs to be plt.show() so
# that your visualization prints out to the screen.

```

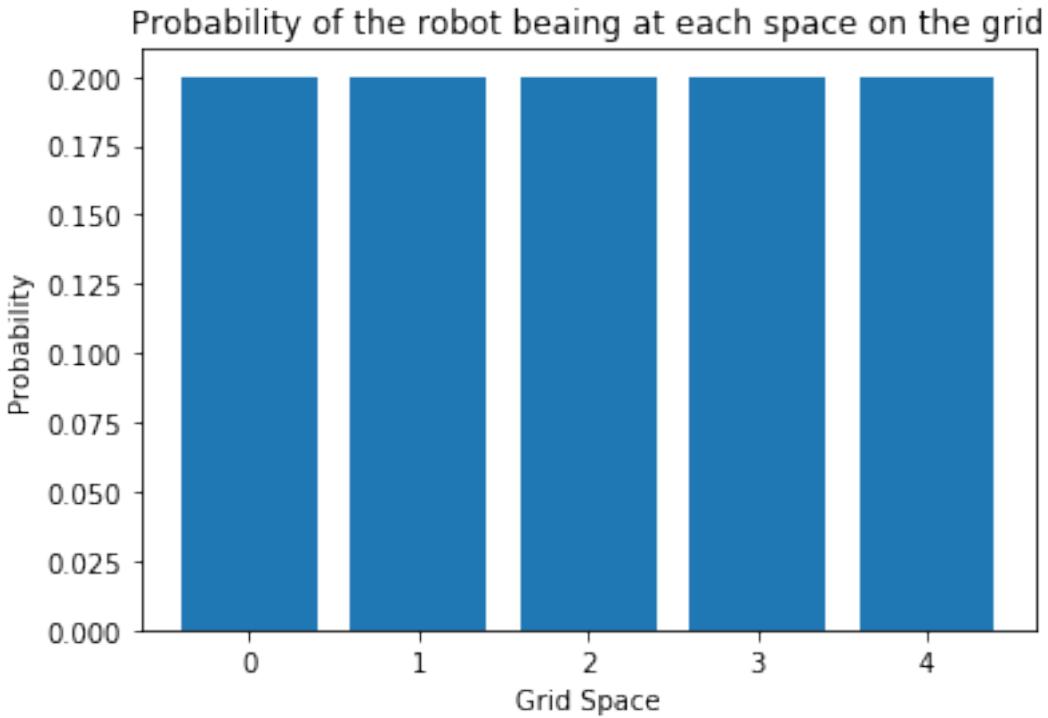


Robot 1D Visualization

```
plt.bar(x, y)
plt.xlabel('Grid Space')
plt.ylabel('Probability')
plt.title('Probability of the robot being at each space on the grid')
```

Run the code cell below to see the results of your visualization. Your results should look something like this:

In [19]: `output_map([.2, .2, .2, .2, .2])`



4.0.1 Exercise 4 - Updating Probabilities

This final problem is meant to be the most challenging one. You'll have to think about how for loops and list indexing work.

As the robot moves around and senses its surroundings, it will update its belief about where it is on the map. You'll learn about this in the localization lesson that comes up next.

Instructions For now, write a function that updates the probabilities for each grid space. The function has these two inputs:

- a list containing the probabilities that the robot is at each spot on the map-grid
- a list of lists containing the new probabilities. Each element in the list has two entries: the first entry is a point on the map grid, and the second entry is the new probability.

Example Input and Output Here is some example input:

The robot is initialized with a five-grid map, so the probabilities are in a list

- `robot_grid = [0.2, 0.2, 0.2, 0.2, 0.2]`

The robot figures out that it actually has a 0.4 probability of being at the first spot `robot_grid[0]` and 0.15 probability of being at all the other spots. So the second input looks like this

- `updates = [[4, 0.15], [0, .4], [3, 0.15], [1, 0.15], [2, 0.15]]`

Notice that this second input is not in the order of the map grid. The first value is for grid 5 (indexed as 4), then grid 1 (indexed as 0), then grid 2 (indexed as 3), etc.

The updates variable could also look like this and not contain information about the entire grid: * `updates = [3, 0.1], [4, 0.2]`

Therefore, think about how you can use the information in the updates variable to correctly change the values in the robot_grid.

The output of the function would be the updated list of probabilities:

- [0.4, 0.15, 0.15, 0.15, 0.15]

Hints Juggling all of this information in your head might prove difficult. Consider taking out a pencil and paper to work through the problem.

In [46]: `def update_probabilities(grid, updates):`

```
###  
# TODO: write a for loop that goes through the updates list  
# and replaces the probabilities in the grid variable.  
#  
# Here are a few HINTS:  
#     You can change a value in the grid variable like this:  
#         grid[0] = .5 where the 0 represents the first grid space  
#  
#     To access values in a list of lists, you need two brackets.  
#  
#     For example, say  
#     updates = [[5, 0.15], [0, .4], [3, 0.15], [1, 0.15], [2, 0.15]]  
#  
#     updates[0] will give you access to the first element in the list,  
#     which is [5, 0.15].  
#  
#     updates[0][0] gives you access to the first element of [5, 0.15]  
#     or in other words the value 5. updates[0][1] gives you access  
#     to the value 0.15.  
#  
###  
  
for i in range(len(updates)):  
    grid[updates[i][0]] = updates[i][1]  
  
return grid
```

Run the code cell below to test your `update_probabilities` function

In [47]: `assert update_probabilities([0.2, 0.2, 0.2, 0.2, 0.2], [[0, .4], [1, 0.15], [2, 0.15],
assert update_probabilities([0.2, 0.2, 0.2, 0.2, 0.2], [[1, 0.15], [0, .4], [4, 0.15],
assert update_probabilities([0.2, 0.2, 0.2, 0.2, 0.2], [[0, .25], [4, 0.15]]) == [0.25,
print('Everything looks good!')`

Everything looks good!

In []:

7_2D_self-drivingcar_demo

March 22, 2020

1 Self Driving Car 2D World [Demo]

Now you'll use a 2-dimensional grid instead of a 1-dimensional grid. But you're going to do the same four things that you did for the 1-D case:

- * write a function that initializes probabilities across the grid
- * write a function that outputs the probability that the robot is at a specific point on the grid
- * write a function to visualize the probabilities of the grid (this function is provided for you)
- * write a function to update probabilities on the grid

2 2D Grid Example

Now, your robot lives in a two-dimensional world. Your robot can move left, right, forwards or backwards. Here is a bird's eye view of an example 7 by 7 grid.

3 Review of a 1D Grid in Python

In the 1D case, you made a list in Python with the probabilities like this:

```
probability_list = [.2, .2, .2, .2, .2]
```

And then you accessed probabilities like this:

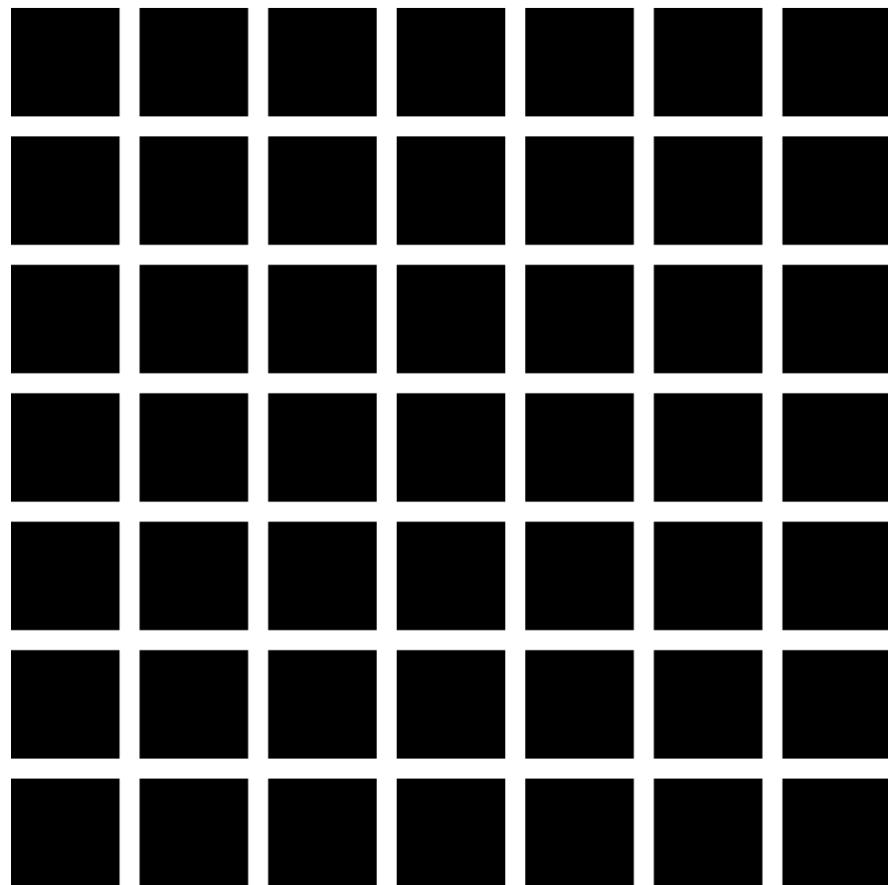
```
probability_list[0]
probability_list[1]
probability_list[2]
probability_list[3]
probability_list[4]
```

This image below shows what the grid looks like and the associated index for each space.

4 2D Grid in Python

Now, with a 2D grid, each space needs two indices: one index is for the row and the other index for the column.

The first index represents the row, and the second index represents the column. In Python, a 2D grid could be represented with a nested list like this.



alt text



alt text

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]

alt text

```
probability_grid = [[1/15, 1/15, 1/15, 1/15, 1/15],  
                   [1/15, 1/15, 1/15, 1/15, 1/15],  
                   [1/15, 1/15, 1/15, 1/15, 1/15]]
```

To access these values, you need the row number and the column number. The values in the first row are

```
probability_grid[0][0]  
probability_grid[0][1]  
probability_grid[0][2]  
probability_grid[0][3]  
probability_grid[0][4]
```

The values in the second row are

```
probability_grid[1][0]  
probability_grid[1][1]  
probability_grid[1][2]  
probability_grid[1][3]  
probability_grid[1][4]
```

And the third row

```
probability_grid[2][0]  
probability_grid[2][1]  
probability_grid[2][2]  
probability_grid[2][3]  
probability_grid[2][4]
```

5 Code Examples

Now, run the code cell below. It contains a few examples of how to use nested for loops. These code examples should help you with the coding exercises.

5.0.1 Example 1

Print out the values in a nested list

```
In [1]: nested_list = [[5, 3, 9, 7, 6],  
                      [2, 7, 2, 1, 9],  
                      [8, 5, 3, 1, 8]]  
  
def print_list(nested_list):  
    print('print out list values one at a time')  
    for i in range(len(nested_list)):  
        for j in range(len(nested_list[0])):  
            print(nested_list[i][j])  
  
print_list(nested_list)  
  
print out list values one at a time  
5  
3  
9  
7  
6  
2  
7  
2  
1  
9  
8  
5  
3  
1  
8
```

5.0.2 Example 2

Print out the values in a nested list with some nicer formatting

```
In [2]: def print_formatted_list(nested_list):  
    # print a blank line  
    print('\nprint out list values with formatting')  
    # print out list values with formatting  
    for i in range(len(nested_list)):  
        for j in range(len(nested_list[0])):  
            # if statement makes sure the last number in a row does not have a comma  
            if j != len(nested_list[0]) - 1:  
                print(str(nested_list[i][j]) + ", " , end="")  
            else:  
                print(str(nested_list[i][j]), end="")
```

```
        print()
print()

print_formatted_list(nested_list)

print out list values with formatting
5, 3, 9, 7, 6
2, 7, 2, 1, 9
8, 5, 3, 1, 8
```

5.0.3 Example Three

This example shows you how to create a 2D list and append new rows to the list.

```
In [3]: twodlist = []
number_rows = 5

for i in range(number_rows):
    twodlist.append([5, 2, 1, 8])

print_formatted_list(twodlist)

print out list values with formatting
5, 2, 1, 8
5, 2, 1, 8
5, 2, 1, 8
5, 2, 1, 8
5, 2, 1, 8
```

5.0.4 Example Four

For the last example, here is a 2D list created with a nested for loop. Notice how the code first creates a new row and then appends the row to the 2D list.

```
In [4]: twodlist = []
row = []
number_rows = 5
number_columns = 6

for i in range(number_rows):
    for j in range(number_columns):
        row.append(i)

    twodlist.append(row)
```

```
twodlist.append(row)
row = []

print_formatted_list(twodlist)

print out list values with formatting
0, 0, 0, 0, 0, 0
1, 1, 1, 1, 1, 1
2, 2, 2, 2, 2, 2
3, 3, 3, 3, 3, 3
4, 4, 4, 4, 4, 4
```

6 Exercises

Go on to the next part of the lesson to start the exercises!

7_2D_self-drivingcar_exercises

March 22, 2020

1 Self Driving Car 2D World

Here are your tasks for this set of exercises:

- * write a function that initializes probabilities across the grid
- * write a function that outputs the probability that the robot is at a specific point on the grid
- * write a function to visualize the probabilities of the grid (this function is provided for you)
- * write a function to update probabilities on the grid

This set of exercises are the same as the 1D world. But now you'll need to use nested for loops and nested lists, which can be tricky.

2 Exercise 1

Write a function that initializes probabilities across the 2D grid. Remember that initially, the probabilities are equal across the entire grid. And all of the probabilities need to add up to one.

So if your grid was 5 by 4, you'd have 20 spaces; the initial probability associated with each space would be $1/20$.

Here are the inputs and outputs of the function:

Inputs * the number of rows in the grid * the number of columns in the grid

Outputs * a nested list containing the probabilities for each grid cell

```
In [11]: def initial_grid(rows, columns):  
  
    # TODO: initialize an empty list in a variable called grid  
    grid = []  
  
    # TODO: initialize an empty row in a variable called row  
    row = []  
  
    # TODO: calculate the initial probability  
    probability = 1 / (rows * columns)  
  
    # TODO: write a nested for loop that appends values to the grid variable  
    # HINT: You first need to fill in a row with values and then append the row to the  
    #       Then you'll need to set the row variable to an empty list.  
    #       The logic of all this can be tough to think through.  
    #       If you get stuck, see the demonstrations in the previous part of the lesson
```

```

for i in range(rows):
    for j in range(columns):
        row.append(probability)
    grid.append(row)
    row = []

return grid

```

Run the code cell below to test your results

```

In [12]: assert initial_grid(5, 4) == [[0.05, 0.05, 0.05, 0.05],
                                         [0.05, 0.05, 0.05, 0.05],
                                         [0.05, 0.05, 0.05, 0.05],
                                         [0.05, 0.05, 0.05, 0.05],
                                         [0.05, 0.05, 0.05, 0.05]]

assert initial_grid(2, 5) == [[0.1, 0.1, 0.1, 0.1, 0.1],
                             [0.1, 0.1, 0.1, 0.1, 0.1]]

assert initial_grid(2, 2) == [[0.25, 0.25],
                            [0.25, 0.25]]

print('Hooray!')

```

Hooray!

3 Exercise 2

Write a function that outputs the probability that the robot is at a specific point on the grid.

Here are the function inputs and outputs:

Inputs

- a 2D grid given as a nested list
- a row number
- a column number

Outputs * the probability at the input row, column

HINT

Remember that the first grid cell in row one, column one is accessed with [0] [0] not [1] [1].

```
In [13]: def probability(grid, row, column):
```

```

# TODO: return the probability that the robot is at the space representing by the row and column
return grid[row][column]

```

```
In [14]: assert probability([[.25, .1],
                           [.45, .2]],
                           1, 1) == 0.2

        assert probability([[.05, .1, .1],
                           [.04, .3, .02],
                           [.01, .02, .02],
                           [.005, .012, .06],
                           [.09, .07, .103]], 3, 2) == 0.06

        assert probability([[.05, .1, .1],
                           [.04, .3, .02],
                           [.01, .023, .017],
                           [.005, .012, .06],
                           [.09, .07, .103]], 2, 2) == .017

        print('You passed all the assertion tests.')

You passed all the assertion tests.
```

4 DEMO

Run the code cell below to visualize the probabilities of the grid. This function is provided for you. We are providing this code for you since we haven't talked about how to graphically represent probability distributions in 2D.

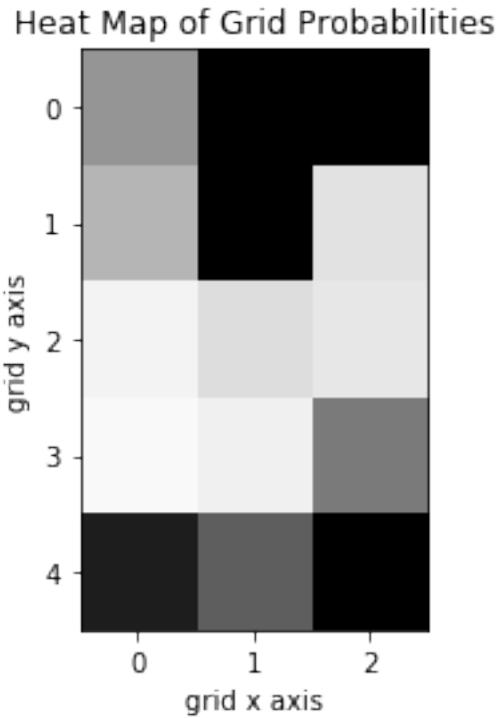
Run the code cells below to see the output. The code shows a heat map. Whereas a bar chart represented probability on the y-axis, a heat map can represent probability with color.

```
In [15]: import matplotlib.pyplot as plt
        %matplotlib inline

        def graph_grid(grid):
            plt.imshow(grid, cmap='Greys', clim=(0,.1))
            plt.title('Heat Map of Grid Probabilities')
            plt.xlabel('grid x axis')
            plt.ylabel('grid y axis')
            plt.legend()
            plt.show()

In [16]: grid = [[.05, .1, .1],
               [.04, .3, .02],
               [.01, .023, .017],
               [.005, .012, .06],
               [.09, .07, .103]]

graph_grid(grid)
```



5 Exercise 3

Write a function to update probabilities on the grid. Like the 1-D case, your function will receive a coordinate and the updated probability. The updated probability coordinates will not be in any particular order; for example, the list might contain grid cell (2,5) followed by grid cell (1,7).

Here are the inputs and outputs of the function.

Inputs * A 2D nested list containing the probabilities that the robot is at each cell * A nested list containing grid cell coordinates and a new probability

Ouput * A 2D nested list, representing the grid, which contains the updated robot grid probabilities

Let's say you had a grid with the following probabilities:

```
grid = [[.05, .1, .1],
        [.04, .3, .02],
        [.01, .023, .017],
        [.005, .012, .06],
        [.09, .07, .103]]
```

You receive an update list with the following values

```
update_list = [
    [[4,2], 0.012],
    [[2,2], 0.108],
```

```
[[0,1], 0.004],  
[[3,0], 0.101]  
]
```

The grid cell at [4,2] currently has a probability value of 0.103. Now, you'll replace the value with 0.012.

The grid cell at [2,3] has the probability 0.017.

Don't get intimidated by the update_list variable! It's just a nested list similar to what you've already seen. But the update_list variable is a list of list of lists.

For example, [4,3] represents the grid cell in row five column three, and 0.012 represents the new probability.

In the next cell, you'll find some example code with a couple of different ways to access the values in the list. Study the examples and use the examples to help you program the update_probability() function.

5.0.1 Nested For Loops Example

In [17]: # Example of nested lists

```
update_list = [  
    [[4,2], 0.012],  
    [[2,2], 0.108],  
    [[0,1], 0.004],  
    [[3,0], 0.101]  
]  
  
# Code for accessing the first element in the list  
print(update_list[0])  
print(update_list[0][0])  
print(update_list[0][0][0])  
print(update_list[0][0][1])  
print(update_list[0][1])  
  
# Output of for loop  
print('\noutput of for loop')  
for element in update_list:  
    print(element)  
  
print('\noutput rows and columns with probability')  
for element in update_list:  
    row, col = element[0]  
    probability = element[1]  
    print('row ', row, 'col ', col, 'probability ', probability)  
  
[[4, 2], 0.012]  
[4, 2]  
4  
2
```

```
0.012
```

```
output of for loop
[[4, 2], 0.012]
[[2, 2], 0.108]
[[0, 1], 0.004]
[[3, 0], 0.101]

output rows and columns with probability
row 4 col 2 probability 0.012
row 2 col 2 probability 0.108
row 0 col 1 probability 0.004
row 3 col 0 probability 0.101
```

5.0.2 Complete the Exercise

Now complete the exercise, use the update_list to update the probabilities in the grid variable.

```
In [32]: def update_probability(grid, update_list):
```

```
#### TODO:
# Use the update_list probabilities to update the probabilities in the grid variable
# For example, if the grid is

# grid = [[.05, .1, .1],
#          [.04, .3, .02],
#          [.01, .023, .017],
#          [.005, .012, .06],
#          [.09, .07, .103]]

# update_list = [[4, 2], 0.012]

# So the element in row 5, column 3 (remember Python does zero indexing) would be
# changed from 0.103 to 0.012

for element in update_list:
    x, y = element[0]
    grid[x][y] = element[1]

return grid
```

Now run the code below to test out your implementation

```
In [33]: grid = [[.05, .1, .1],
               [.04, .3, .02],
               [.01, .023, .017],
               [.005, .012, .06],
               [.09, .07, .103]]
```

```
update_list = [
    [[4,2], 0.012],
    [[2,2], 0.108],
    [[0,1], 0.004],
    [[3,0], 0.101]
]

assert update_probability(grid, update_list) == [[0.05, 0.004, 0.1],
[0.04, 0.3, 0.02],
[0.01, 0.023, 0.108],
[0.101, 0.012, 0.06],
[0.09, 0.07, 0.012]]

print('Nicely done')
```

Nicely done

In []:

plotting_gaussians

March 22, 2020

1 Plotting Gaussians

In this exercise, you'll use Python to calculate the Gaussian probability density function and then plot the results.

Besides matplotlib, the exercise also uses a Python library called numpy. Numpy, <http://www.numpy.org/> makes it much easier to work with arrays and matrices in Python.

This exercise does not focus on numpy and how to use it. But we'll provide enough context so that you can use it in your code.

2 Exercise 1

Write a function for calculating the probability density function of a Gaussian. The function has three inputs and one output:

inputs * mu, which is the average * sigma, which is the standard deviation * a list of x values
outputs * probability density function output

As a reminder, here is the probability density function for a Gaussian distribution:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Using numpy To calculate the square, square root or an exponent in Python, you could use the math library; however, instead you are going to use the numpy library. Study the code below to compare the math library and the numpy library and then run the code cell.

```
In [1]: import math
        import numpy as np

        print('\nExample of taking the square')
        print(math.pow(2,2))
        print(np.square(2))

        print('\nExample of taking the cube')
        print(math.pow(2,3))
        print(np.power(2,3))

        print('\nExample of taking the square root')
```

```
print(math.sqrt(4))
print(np.sqrt(4))

print('\nExample of taking the exponent')
print(math.exp(3))
print(np.exp(3))
```

Example of taking the square

```
4.0
4
```

Example of taking the cube

```
8.0
8
```

Example of taking the square root

```
2.0
2.0
```

Example of taking the exponent

```
20.085536923187668
20.0855369232
```

Using numpy with lists The numpy library lets you run mathematical expressions on elements of a list. The math library cannot do this. Study the examples below and then run the code cell.

```
In [2]: print('\nExample of squaring elements in a list')
print(np.square([1, 2, 3, 4, 5]))

print('\nExample of taking the square root of a list')
print(np.sqrt([1, 4, 9, 16, 25]))

print('\nExamples of taking the cube of a list')
print(np.power([1, 2, 3, 4, 5], 3))
```

Example of squaring elements in a list
[1 4 9 16 25]

Example of taking the square root of a list
[1. 2. 3. 4. 5.]

Examples of taking the cube of a list
[1 8 27 64 125]

Using numpy in a function Here is one last code example before you write your code. The example shows how to use numpy in a function.

```
In [3]: def numpy_example(x):
    return np.exp(x)

x = [1, 2, 3, 4, 5]
print(numpy_example(x))

[ 2.71828183  7.3890561 20.08553692 54.59815003 148.4131591 ]
```

Write your code below Now, write the code for the probability density function. Besides the numpy sqrt, power, and exp methods, you might also want to use the np.pi method, which outputs the value for pi.

```
In [5]: def gaussian_density(x, mu, sigma):
    # TODO: Return the probability density function for the
    # Gaussian distribution.
    return (1/np.sqrt(2 * np.pi * np.square(sigma))) * np.exp(-np.square(x-mu) / (2 * np.pi * np.square(sigma)))
```

Read through and run the code cell below to check your results. We've also provided a solution in the next lesson node titled "Plotting Gaussians in Python[Solution]".

In the code cell below, we've used the numpy linspace method, which has three inputs. The linspace method essentially creates a list of values. In the example below, np.linspace(0, 100, 11) creates a list of values from 0 to 100 with 11 elements. In other words (0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100). See the [linspace documentation](#).

```
In [6]: # Run this code sell to check your results
```

```
# numpy linspace creates a list of values called an array
x = np.linspace(0, 100, 11)

### Expected Answer when running the code cell
answer = np.array(([ 1.48671951e-07,   1.33830226e-05,   4.43184841e-04,
                    5.39909665e-03,   2.41970725e-02,   3.98942280e-02,
                    2.41970725e-02,   5.39909665e-03,   4.43184841e-04,
                    1.33830226e-05,   1.48671951e-07])))

# Call our function with `gaussian_density(x, 50, 10)` and compare to the answer above
# `assert_almost_equal` is more reliable with floating point numbers than `assert_array_
np.testing.assert_almost_equal(gaussian_density(x, 50, 10), answer, decimal=7)
print("Test passed!")
```

Test passed!

We've also put solution code in the next part of the lesson "Plotting Gaussians in Python [Solutions]"

3 Exercise 2

Write a function called `plot_gaussian` that creates a plot of a Gaussian function.

In the programming probability exercises, we gave examples about how to make plots in Python. We'll give some guidelines here, but you might need to go back to those exercises and study the examples.

Or alternatively, read through the [matplotlib documentation](#).

Here are the function inputs and outputs:

Inputs

- `x` - a numpy linspace array
- `mu` - an average value
- `sigma` - a standard deviation

Outputs

This function does not need a return statement; the function will print out a visualization.

```
In [16]: import matplotlib.pyplot as plt

def plot_gaussian(x, mu, sigma):
    # TODO: Use x, mu and sigma to calculate the probability density
    # function. Put the results in the y variable.
    # You can use your gaussian_density() function
    # from the first exercise.

    y = gaussian_density(x, mu, sigma)

    # TODO: Plot the results in a line chart. See the first example
    # in the pyplot tutorial for help:
    # https://matplotlib.org/xkcd/users/pyplot_tutorial.html

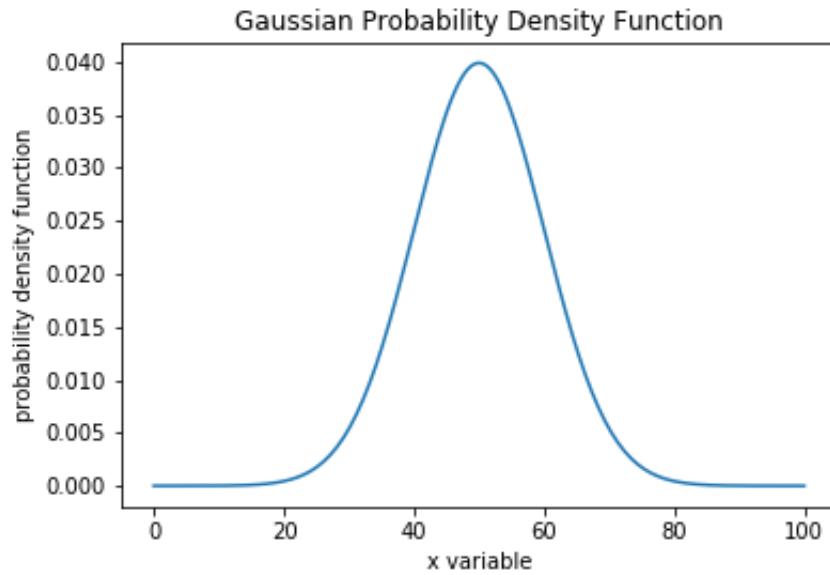
    plt.plot(x, y)
    plt.xlabel('x variable')
    plt.ylabel('probability density function')
    plt.title('Gaussian Probability Density Function')
    plt.show()

    # Make sure to label the x axis, y axis and give the chart
    # a title.
    return None
```

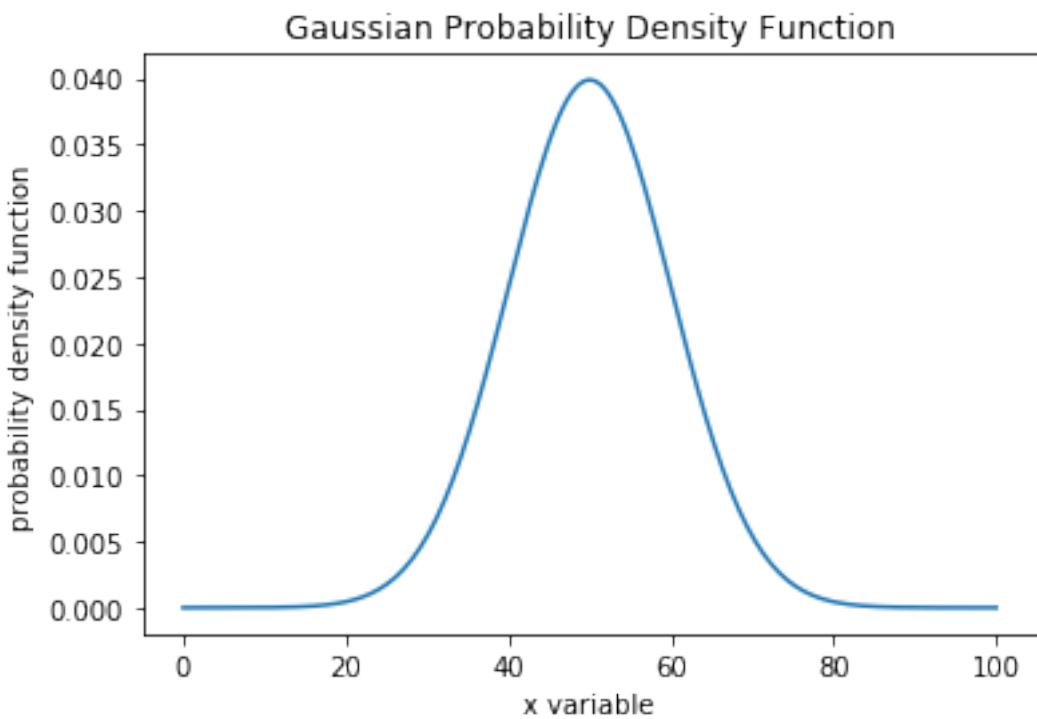
Run the code below to see the results. Your visualization should look like this:

```
In [17]: # Run this code cell to see the results
```

```
x = np.linspace(0, 100, 200)
plot_gaussian(x, 50, 10)
```



Gaussian results



See the next part of the lesson for solutions to the coding exercises.

In the next part of the lesson, we're going to talk about how to calculate probabilities from the probability density functions.

In []:

area_under_curve

March 23, 2020

1 Area Under the Curve

You are going to use Python to calculate probabilities associated with a Gaussian distribution. In other words, you'll learn to calculate the area under the probability density function.

2 SciPy Library

Python has a library called [SciPy](#), which is used for running scientific and mathematical computations. You are specifically going to use a part of the library that implements Gaussian distributions.

3 Demo: Probability Density Function

The code cell below calculates a Gaussian probability density function two ways. First, we're using the density function from the previous exercises. Then, we compare the results using the SciPy library's implementation.

You'll see that the results are exactly the same.

```
In [1]: from scipy.stats import norm
        import numpy as np

        # our solution to calculate the probability density function
        def gaussian_density(x, mu, sigma):
            return (1/np.sqrt(2*np.pi*np.power(sigma, 2.))) * np.exp(-np.power(x - mu, 2.) / (2 * np.power(sigma, 2.)))

        print("Probability density function our solution: mu = 50, sigma = 10, x = 50")
        print(gaussian_density(50, 50, 10))
        print("\nProbability density function SciPy: mu = 50, sigma = 10, x = 50")
        print(norm(loc = 50, scale = 10).pdf(50))

Probability density function our solution: mu = 50, sigma = 10, x = 50
0.0398942280401

Probability density function SciPy: mu = 50, sigma = 10, x = 50
0.0398942280401
```

You might wonder why the Gaussian distribution is called "norm" in the SciPy library; it's because the Gaussian distribution is also called the normal distribution.

Also, note that to initialize the distribution, the loc keyword is the mean and the scale keyword is the standard deviation.

4 Calculating Probability

The area under the probability density function represents probability. Your job will be to write a function that calculates the probability between two x-values. For example, using the winter San Francisco temperature example, what is the probability that the temperature is between 30 degrees and 50 degrees?

The SciPy library has a function that calculates the area under the curve for you. It's called cdf ([cumulative density function](#)). You can use the cdf SciPy method in a similar way to the pdf method. Run the code cell below to see an example.

```
In [2]: norm(loc = 50, scale = 10).cdf(50)
```

```
Out[2]: 0.5
```

Why is the output 0.5? The cdf method gives you the area under the curve from $x = -\infty$ through the input, which in this case was 50. The area under the curve is 0.5 meaning there is a 50% chance that the temperature is between $-\infty$ and 50 degrees.

Run the code cell below to see a visualization of the area under the curve from $-\infty$ to 50.

```
In [13]: import matplotlib.pyplot as plt
%matplotlib inline

#####
# The plot_fill function plots a probability density function and also
# shades the area under the curve between x_prob_min and x_prob_max.
# INPUTS:
# x: x-axis values for plotting
# x_prob_min: minimum x-value for shading the visualization
# x_prob_max: maximum x-value for shading the visualization
# y_lim: the highest y-value to show on the y-axis
# title: visualization title
#
# OUTPUTS:
# prints out a visualization
#####

def plot_fill(x, x_prob_min, x_prob_max, y_lim, title):
    # Calculate y values of the probability density function
    # Note that the pdf method can accept an array of values from numpy linspace.
    y = norm(loc = 50, scale = 10).pdf(x)

    # Calculate values for filling the area under the curve
    x_fill = np.linspace(x_prob_min, x_prob_max, 1000)
```

```

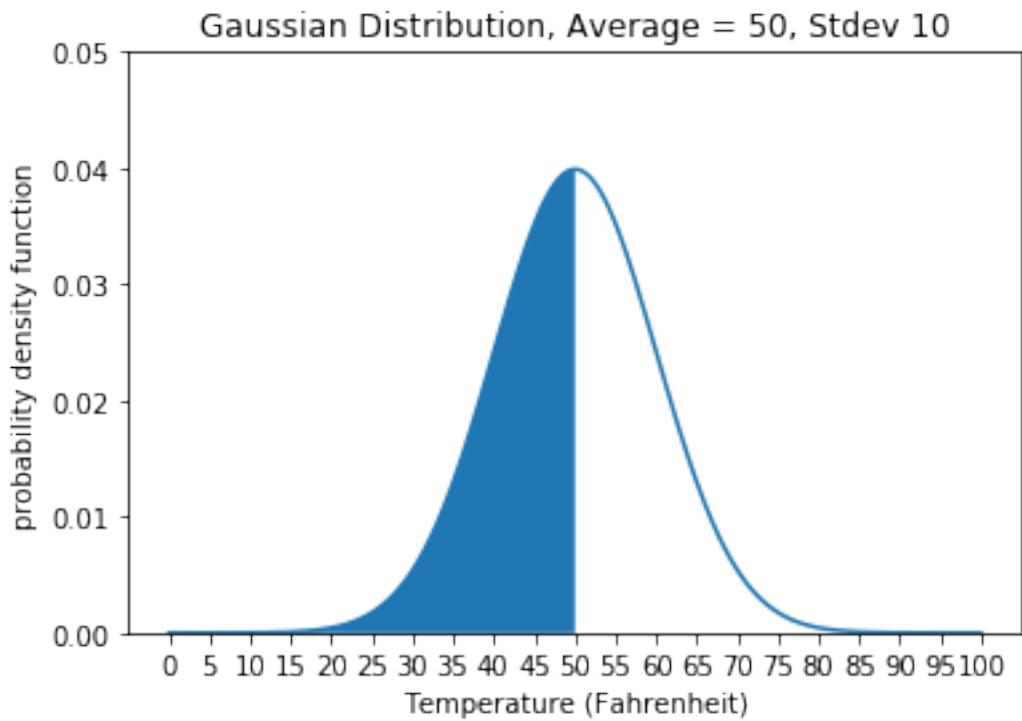
y_fill = norm(loc = 50, scale = 10).pdf(x_fill)

# Plot the results
plt.plot(x, y)
plt.fill_between(x_fill, y_fill)
plt.title(title)
plt.ylim(0, y_lim)
plt.xticks(np.linspace(0, 100, 21))
plt.xlabel('Temperature (Fahrenheit)')
plt.ylabel('probability density function')
plt.show()

average = 50
stdev = 10
y_lim = 0.05
x = np.linspace(0, 100, 1000)

plot_fill(x, 0, 50, y_lim,
           'Gaussian Distribution, Average = ' + str(average) + ', Stdev ' + str(stdev))

```



5 More Examples

Here are a few more examples of the cdf method. The code cell below prints out the probability that the temperature is between:

- * -infinity and 25
- * -infinity and 75
- * -infinity and 125
- * -infinity and +infinity

```
In [14]: print(norm(loc = 50, scale = 10).cdf(25))
          print(norm(loc = 50, scale = 10).cdf(75))
          print('%.20f' % norm(loc = 50, scale = 10).cdf(125)) # '%.20f' prints out 20 decimal pl
          print(norm(loc = 50, scale = 10).cdf(float('inf')))

0.00620966532578
0.993790334674
0.99999999999996813660
1.0
```

What if you wanted to know the probability of the temperature being between 25 and +infinity? Or 75 and +infinity?

Well, you know that the area under the curve from -infinity to +infinity is equal to 1. And the cdf function can return the probability from -infinity to 25.

So if you wanted to know the probability from 25 to +infinity, you could do the following calculation:

```
In [15]: print(1 - norm(loc = 50, scale = 10).cdf(25))

0.993790334674
```

Notice that for this particular Gaussian distribution, the probability that temperature is between -infinity and 75 is the same as the probability from 25 to +infinity. This is due to the symmetry of the Gaussian distribution: 25 and 75 are both 25 away from the mean of 50.

6 Exercise

What if you wanted to know the probability that the temperature is between 25 and 75? Based on what you've seen so far, you have all the information you need to write a function that does this calculation.

Run the code cell below to get a hint for how to do this. The code cell outputs visualizations for the area under the curve from -infinity to 25, then -infinity to 75, and finally between 25 and 75.

Remember that the cdf method calculates area under the curve from -infinity up to the number you specify. How can you use this information to calculate the probability **between** 25 and 75? Remember that the probability between 25 and 75 is equal to the area under the curve between 25 and 75.

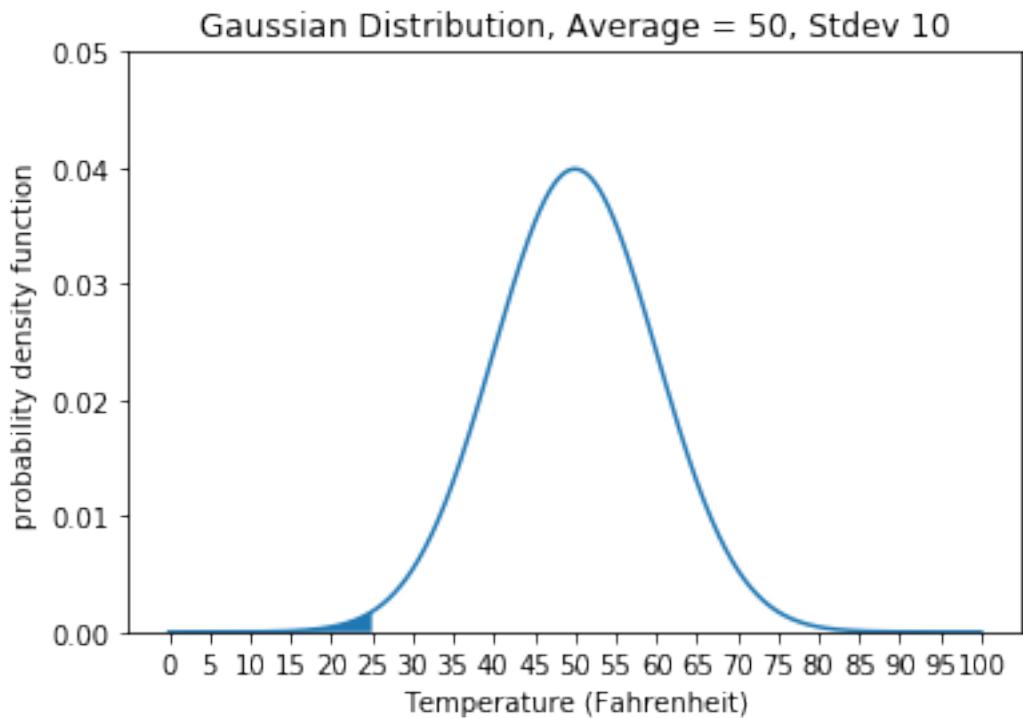
```
In [16]: average = 50
          stdev = 10
          y_lim = 0.05
```

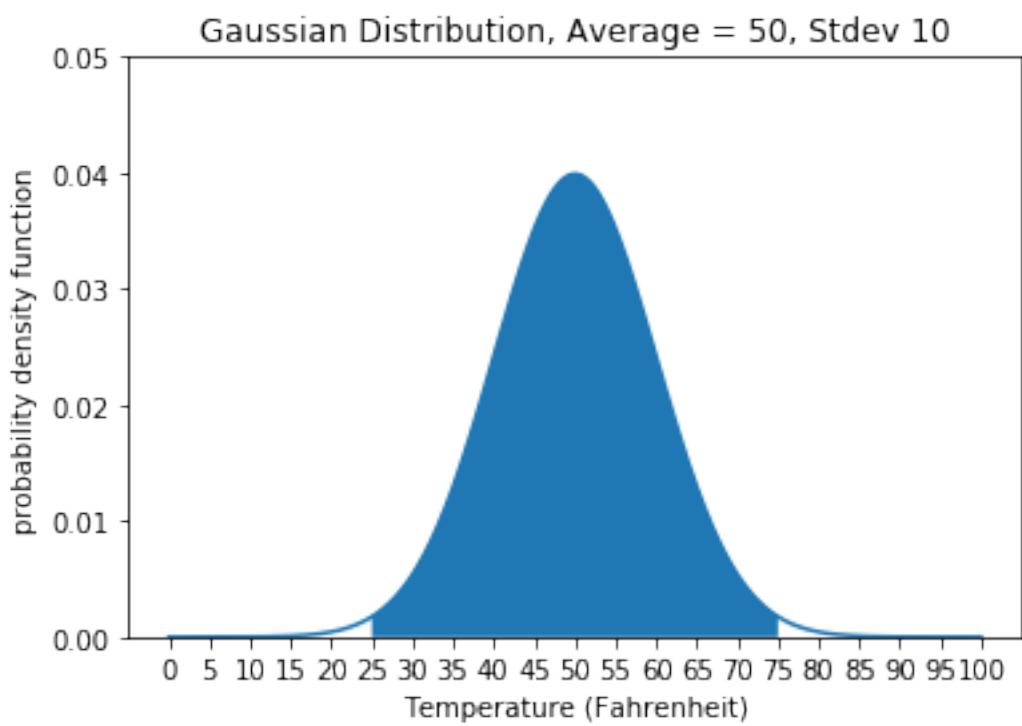
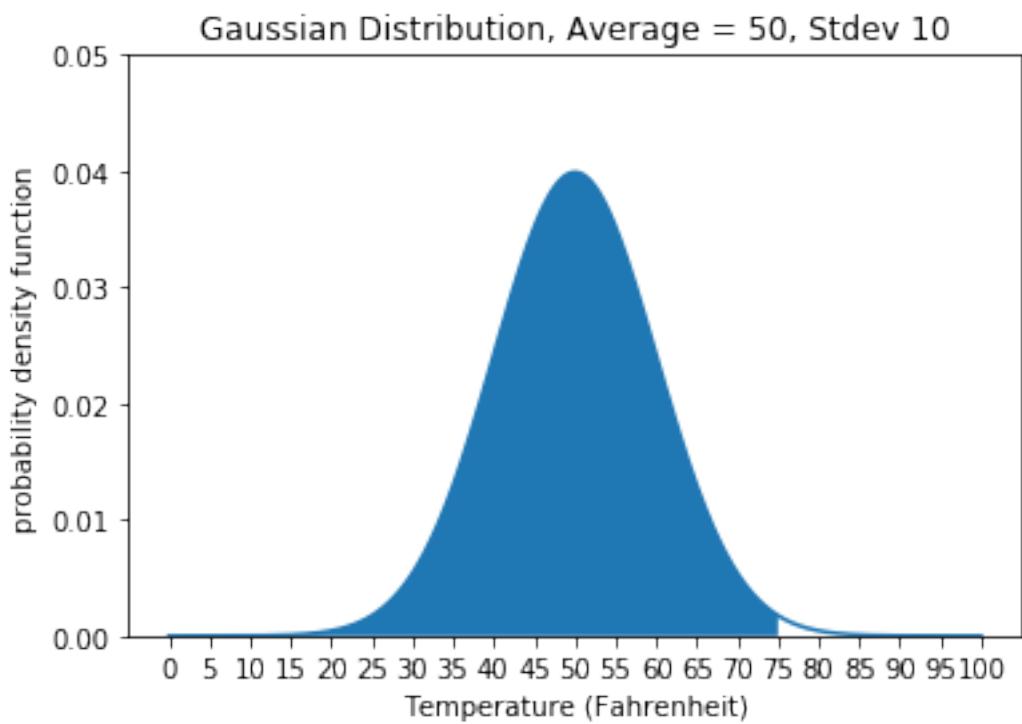
```
x = np.linspace(0, 100, 1000)

plot_fill(x, 0, 25, y_lim,
           'Gaussian Distribution, Average = ' + str(average) + ', Stdev ' + str(stdev))

plot_fill(x, 0, 75, y_lim,
           'Gaussian Distribution, Average = ' + str(average) + ', Stdev ' + str(stdev))

plot_fill(x, 25, 75, y_lim,
           'Gaussian Distribution, Average = ' + str(average) + ', Stdev ' + str(stdev))
```





Fill in your code below. The function has four inputs and one output. Here are the inputs:

Inputs * mean - Gaussian distribution mean * stdev - Gaussian distribution standard deviation

* x_low - low end of the probability range * x_high - high end of the probability range

Output * probability that the x value is between x_low and x_high

We've provided a solution in the next node of the lesson title "Calculating Area Under the Curve in Python [Solution]".

```
In [29]: def gaussian_probability(mean, stdev, x_low, x_high):
    # TODO: return the Gaussian distribution probability
    # that the x-value is between x_low and x_high

    # Use the SciPy library norm.cdf method
    return norm(loc = mean, scale = stdev).cdf(x_high) - norm(loc = mean, scale = stdev).cdf(x_low)
```

Run the code cell below to test your results. A solution is provided in the next part of the lesson.

```
In [30]: assert float('{0:.8f}'.format(gaussian_probability(50, 10, 25, 75))) == 0.98758067
        assert float('{0:.2f}'.format(gaussian_probability(50, 10, float('-inf'), float('inf')))) == 1.0
        assert float('{0:.8f}'.format(gaussian_probability(50, 10, 20, 60))) == 0.83999485
        print('All tests passed')
```

All tests passed

```
In [ ]:
```

central_limit_theorem

March 23, 2020

1 Central Limit Theorem

In this section, we are going to show you how the central limit theorem works. We'll:

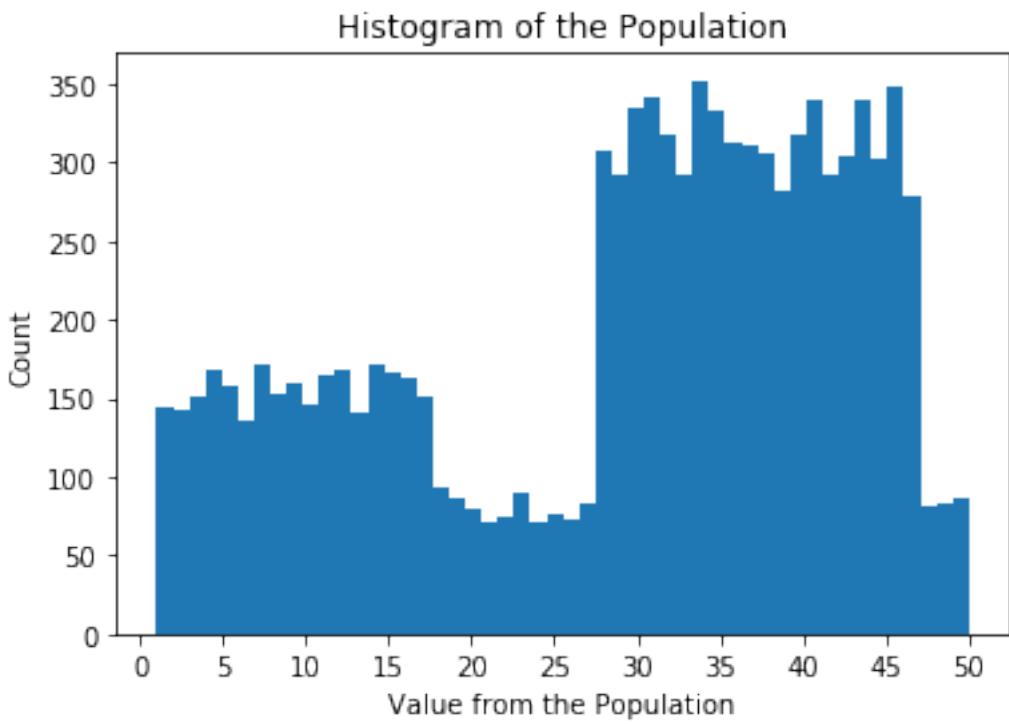
- * generate random samples from a population
- * take the means of the samples
- * visualize the resulting means

You'll see that although the population does not follow a Gaussian distribution, the resulting distribution of the sample means does look Gaussian.

To start things off, run the code cell below. This cell will run a helper function that creates the population data, then visualizes the population data and calculates the mean of the population data. There are 10,000 data points in the population.

If you run the cell multiple times, you'll notice that the distribution changes slightly; however, the general shape remains the same.

```
In [1]: import helpers  
        import numpy as np  
        %matplotlib inline  
  
population_data = helpers.distribution(50, 10000, 100)  
helpers.histogram_visualization(population_data)  
print('population mean ', np.mean(population_data))
```



```
population mean 29.0344
```

2 Taking Samples from the Population

The next code cell will randomly choose N data points from the population. These N data points will be called a sample. We are using the numpy library's random.choice method to randomly choose N values, which you can read about [here](#).

Run the code cell below to see some example output. The code randomly draws 10 data points from the population to make a sample of size 10.

```
In [2]: def random_sample(population_data, sample_size):
    return np.random.choice(population_data, size = sample_size)

random_sample(population_data, 10)
```

```
Out[2]: array([ 5, 39, 45, 12,  7, 15, 37, 50,  7, 50])
```

3 Calculating the Sample Mean

We'll next use the numpy library to calculate the mean of each randomly generated sample.

```
In [3]: def sample_mean(sample):
    return np.mean(sample)

    # take a sample from the population
example_sample = random_sample(population_data, 10)

    # calculate the mean of the sample and output the results
sample_mean(example_sample)

Out[3]: 36.200000000000003
```

4 Central Limit Theorem Results

Now, we'll use the `random_sample()` function and the `sample_mean()` function to show how the central limit theorem works.

The code below contains a for loop that draws a random sample of size N, then takes the mean of the sample, and stores the means in a list. Each iteration of the for loop will have a different random sample. Study the code below and then run the cell.

```
In [4]: """
# Code for showing how the central limit theorem works.
# The function inputs:
# population - population data
# n - sample size
# iterations - number of times to draw random samples

def central_limit_theorem(population, n, iterations):
    sample_means_results = []
    for i in range(iterations):
        # get a random sample from the population of size n
        sample = random_sample(population, n)

        # calculate the mean of the random sample
        # and append the mean to the results list
        sample_means_results.append(sample_mean(sample))
    return sample_means_results

print('Means of all the samples ')
central_limit_theorem(population_data, 10, 10000)
```

Means of all the samples

```
Out[4]: [29.19999999999999,
26.69999999999999,
24.5,
35.700000000000003,
35.5,
```

28.800000000000001,
28.899999999999999,
29.5,
33.0,
19.399999999999999,
27.600000000000001,
29.800000000000001,
31.800000000000001,
30.100000000000001,
29.5,
25.399999999999999,
34.299999999999997,
30.800000000000001,
35.0,
35.799999999999997,
26.600000000000001,
32.600000000000001,
39.600000000000001,
34.0,
30.100000000000001,
34.100000000000001,
23.399999999999999,
31.300000000000001,
34.700000000000003,
27.399999999999999,
26.899999999999999,
25.899999999999999,
32.100000000000001,
29.899999999999999,
24.199999999999999,
29.699999999999999,
30.399999999999999,
30.0,
34.100000000000001,
32.600000000000001,
30.399999999999999,
23.0,
36.299999999999997,
25.100000000000001,
27.600000000000001,
26.399999999999999,
33.100000000000001,
25.5,
20.600000000000001,
22.100000000000001,
31.0,
25.800000000000001,
19.100000000000001,

28.5,
33.29999999999997,
21.10000000000001,
24.10000000000001,
23.69999999999999,
27.19999999999999,
34.39999999999999,
31.19999999999999,
23.30000000000001,
31.19999999999999,
21.69999999999999,
21.39999999999999,
29.19999999999999,
27.89999999999999,
29.5,
29.19999999999999,
24.0,
32.2999999999997,
23.89999999999999,
13.19999999999999,
28.89999999999999,
29.30000000000001,
29.60000000000001,
27.30000000000001,
33.70000000000003,
37.39999999999999,
24.30000000000001,
36.39999999999999,
37.20000000000003,
27.30000000000001,
28.10000000000001,
27.69999999999999,
28.19999999999999,
24.60000000000001,
34.2999999999997,
25.30000000000001,
19.0,
26.19999999999999,
32.60000000000001,
28.10000000000001,
29.39999999999999,
21.30000000000001,
27.89999999999999,
23.80000000000001,
31.30000000000001,
28.80000000000001,
27.19999999999999,
33.5,

32.200000000000003,
24.800000000000001,
28.899999999999999,
35.200000000000003,
28.5,
31.699999999999999,
41.0,
27.399999999999999,
40.899999999999999,
34.0,
35.899999999999999,
31.100000000000001,
28.5,
26.899999999999999,
32.100000000000001,
33.700000000000003,
32.299999999999997,
23.600000000000001,
25.5,
28.199999999999999,
35.899999999999999,
36.5,
29.5,
29.699999999999999,
33.600000000000001,
31.199999999999999,
34.899999999999999,
27.5,
26.800000000000001,
34.200000000000003,
34.899999999999999,
28.800000000000001,
35.200000000000003,
25.199999999999999,
30.899999999999999,
32.899999999999999,
27.600000000000001,
30.399999999999999,
30.0,
21.0,
26.600000000000001,
20.199999999999999,
28.5,
30.800000000000001,
22.100000000000001,
27.600000000000001,
22.199999999999999,
30.800000000000001,

31.199999999999999,
25.699999999999999,
28.699999999999999,
27.199999999999999,
29.0,
26.5,
24.800000000000001,
29.600000000000001,
36.899999999999999,
27.800000000000001,
33.799999999999997,
28.5,
29.800000000000001,
31.0,
22.699999999999999,
28.800000000000001,
30.100000000000001,
40.5,
25.199999999999999,
32.0,
33.799999999999997,
38.100000000000001,
33.399999999999999,
35.5,
32.299999999999997,
32.799999999999997,
28.399999999999999,
29.300000000000001,
28.300000000000001,
23.100000000000001,
26.600000000000001,
30.0,
36.399999999999999,
28.899999999999999,
30.300000000000001,
28.199999999999999,
25.0,
24.399999999999999,
21.600000000000001,
30.800000000000001,
24.199999999999999,
24.600000000000001,
23.399999999999999,
25.300000000000001,
23.100000000000001,
24.399999999999999,
27.600000000000001,
21.899999999999999,

30.699999999999999,
24.100000000000001,
29.300000000000001,
28.899999999999999,
30.100000000000001,
26.199999999999999,
24.300000000000001,
32.5,
29.5,
24.399999999999999,
23.300000000000001,
33.600000000000001,
31.0,
33.0,
29.0,
31.5,
35.200000000000003,
29.100000000000001,
25.300000000000001,
29.199999999999999,
32.100000000000001,
39.5,
30.600000000000001,
24.899999999999999,
32.600000000000001,
29.100000000000001,
26.399999999999999,
32.799999999999997,
27.399999999999999,
34.399999999999999,
33.700000000000003,
30.199999999999999,
25.399999999999999,
25.5,
20.300000000000001,
27.199999999999999,
29.5,
29.800000000000001,
24.5,
26.899999999999999,
36.700000000000003,
33.200000000000003,
29.5,
30.899999999999999,
34.0,
29.5,
35.200000000000003,
24.100000000000001,

29.699999999999999,
28.899999999999999,
27.800000000000001,
28.800000000000001,
29.899999999999999,
28.399999999999999,
28.699999999999999,
34.5,
30.899999999999999,
34.600000000000001,
28.699999999999999,
30.600000000000001,
28.199999999999999,
25.300000000000001,
30.100000000000001,
32.899999999999999,
36.79999999999997,
28.199999999999999,
28.300000000000001,
32.700000000000003,
32.0,
39.100000000000001,
25.199999999999999,
27.399999999999999,
27.699999999999999,
24.300000000000001,
27.399999999999999,
24.600000000000001,
18.0,
24.0,
30.699999999999999,
30.5,
25.300000000000001,
30.0,
30.100000000000001,
28.600000000000001,
33.899999999999999,
28.0,
29.899999999999999,
36.899999999999999,
21.199999999999999,
30.5,
24.399999999999999,
32.899999999999999,
35.100000000000001,
34.399999999999999,
32.5,
30.600000000000001,

24.800000000000001,
28.5,
31.199999999999999,
29.0,
27.100000000000001,
26.199999999999999,
26.0,
23.899999999999999,
29.0,
30.300000000000001,
39.600000000000001,
28.199999999999999,
14.300000000000001,
26.899999999999999,
34.600000000000001,
29.199999999999999,
34.600000000000001,
20.199999999999999,
30.899999999999999,
28.800000000000001,
30.699999999999999,
21.899999999999999,
27.399999999999999,
29.5,
30.300000000000001,
31.600000000000001,
31.600000000000001,
27.300000000000001,
33.200000000000003,
25.699999999999999,
29.399999999999999,
26.899999999999999,
36.600000000000001,
29.300000000000001,
26.899999999999999,
24.800000000000001,
33.5,
30.300000000000001,
32.5,
23.5,
26.899999999999999,
26.199999999999999,
31.300000000000001,
30.699999999999999,
28.699999999999999,
35.399999999999999,
26.800000000000001,
28.0,

36.0,
33.5,
27.199999999999999,
36.5,
24.800000000000001,
26.800000000000001,
31.199999999999999,
28.100000000000001,
22.899999999999999,
36.700000000000003,
26.100000000000001,
30.199999999999999,
29.399999999999999,
33.799999999999997,
20.300000000000001,
24.600000000000001,
27.800000000000001,
27.0,
30.300000000000001,
26.699999999999999,
33.299999999999997,
27.699999999999999,
23.5,
32.600000000000001,
24.699999999999999,
29.5,
35.200000000000003,
35.100000000000001,
29.199999999999999,
20.5,
18.899999999999999,
41.200000000000003,
34.899999999999999,
26.800000000000001,
33.100000000000001,
23.199999999999999,
30.800000000000001,
29.300000000000001,
21.699999999999999,
36.399999999999999,
30.199999999999999,
27.199999999999999,
28.399999999999999,
29.399999999999999,
35.700000000000003,
33.700000000000003,
33.799999999999997,
31.300000000000001,

21.800000000000001,
25.399999999999999,
22.899999999999999,
17.699999999999999,
28.800000000000001,
26.300000000000001,
30.699999999999999,
19.899999999999999,
32.600000000000001,
31.0,
34.5,
30.399999999999999,
28.899999999999999,
36.200000000000003,
28.899999999999999,
29.699999999999999,
26.100000000000001,
24.199999999999999,
28.199999999999999,
29.600000000000001,
24.800000000000001,
31.0,
25.5,
29.5,
28.199999999999999,
24.600000000000001,
30.0,
29.300000000000001,
29.0,
24.899999999999999,
24.699999999999999,
24.300000000000001,
33.0,
32.79999999999997,
32.899999999999999,
25.5,
31.199999999999999,
23.800000000000001,
32.399999999999999,
26.199999999999999,
37.100000000000001,
29.800000000000001,
37.200000000000003,
30.600000000000001,
34.0,
28.399999999999999,
30.600000000000001,
36.100000000000001,

26.899999999999999,
26.600000000000001,
24.600000000000001,
24.199999999999999,
29.800000000000001,
29.399999999999999,
21.0,
26.399999999999999,
21.699999999999999,
35.399999999999999,
30.699999999999999,
35.0,
30.600000000000001,
24.5,
36.29999999999997,
28.5,
29.100000000000001,
34.899999999999999,
32.29999999999997,
29.699999999999999,
26.199999999999999,
29.600000000000001,
28.5,
29.399999999999999,
29.899999999999999,
17.5,
35.600000000000001,
26.600000000000001,
30.699999999999999,
26.5,
31.300000000000001,
35.29999999999997,
34.899999999999999,
32.70000000000003,
32.5,
29.600000000000001,
28.100000000000001,
28.5,
26.100000000000001,
14.800000000000001,
20.899999999999999,
36.0,
32.399999999999999,
33.79999999999997,
36.0,
27.100000000000001,
29.899999999999999,
19.399999999999999,

23.0,
23.69999999999999,
29.5,
26.89999999999999,
29.0,
29.60000000000001,
31.80000000000001,
34.89999999999999,
26.39999999999999,
33.5,
27.19999999999999,
27.60000000000001,
31.19999999999999,
33.0,
32.0,
29.0,
30.19999999999999,
25.89999999999999,
31.30000000000001,
23.19999999999999,
23.5,
27.0,
30.80000000000001,
29.30000000000001,
24.80000000000001,
26.5,
35.39999999999999,
28.39999999999999,
25.39999999999999,
26.5,
30.0,
26.69999999999999,
27.30000000000001,
28.80000000000001,
30.5,
30.5,
24.5,
35.0,
24.39999999999999,
31.69999999999999,
27.39999999999999,
39.39999999999999,
27.69999999999999,
26.39999999999999,
21.5,
21.19999999999999,
28.30000000000001,
31.5,

30.600000000000001,
33.600000000000001,
28.19999999999999,
27.0,
36.0,
28.300000000000001,
30.89999999999999,
32.20000000000003,
25.39999999999999,
31.80000000000001,
21.60000000000001,
37.70000000000003,
30.5,
34.60000000000001,
25.60000000000001,
32.20000000000003,
33.60000000000001,
31.30000000000001,
28.0,
24.89999999999999,
23.5,
32.29999999999997,
25.60000000000001,
27.80000000000001,
31.80000000000001,
21.39999999999999,
27.69999999999999,
38.39999999999999,
30.0,
35.89999999999999,
36.39999999999999,
29.39999999999999,
32.60000000000001,
28.0,
26.80000000000001,
26.0,
29.69999999999999,
34.70000000000003,
33.29999999999997,
28.89999999999999,
36.39999999999999,
21.89999999999999,
33.29999999999997,
30.19999999999999,
24.5,
24.89999999999999,
29.80000000000001,
21.80000000000001,

34.600000000000001,
31.5,
34.20000000000003,
27.60000000000001,
28.0,
33.0,
32.39999999999999,
30.0,
27.30000000000001,
33.39999999999999,
25.19999999999999,
22.89999999999999,
26.89999999999999,
29.30000000000001,
33.10000000000001,
29.39999999999999,
30.30000000000001,
22.39999999999999,
22.5,
28.80000000000001,
29.30000000000001,
37.29999999999997,
28.30000000000001,
32.60000000000001,
35.39999999999999,
25.5,
34.5,
31.30000000000001,
32.70000000000003,
31.19999999999999,
27.80000000000001,
31.30000000000001,
36.0,
35.79999999999997,
26.80000000000001,
33.39999999999999,
27.80000000000001,
23.39999999999999,
20.30000000000001,
32.0,
32.0,
33.5,
29.30000000000001,
34.10000000000001,
28.39999999999999,
26.5,
24.19999999999999,
29.10000000000001,

31.0,
28.600000000000001,
22.10000000000001,
22.30000000000001,
32.10000000000001,
27.10000000000001,
38.399999999999999,
24.5,
24.899999999999999,
36.20000000000003,
19.699999999999999,
31.10000000000001,
27.399999999999999,
29.5,
33.10000000000001,
27.399999999999999,
26.5,
33.0,
21.80000000000001,
25.60000000000001,
27.60000000000001,
24.199999999999999,
29.10000000000001,
22.60000000000001,
41.899999999999999,
34.29999999999997,
21.899999999999999,
29.5,
34.0,
33.399999999999999,
32.399999999999999,
24.0,
32.5,
29.5,
36.60000000000001,
35.10000000000001,
23.0,
26.80000000000001,
26.0,
26.899999999999999,
30.199999999999999,
30.199999999999999,
29.699999999999999,
19.80000000000001,
24.60000000000001,
25.899999999999999,
33.60000000000001,
33.399999999999999,

33.200000000000003,
28.399999999999999,
33.600000000000001,
22.300000000000001,
24.300000000000001,
33.100000000000001,
19.100000000000001,
28.5,
34.29999999999997,
29.699999999999999,
32.29999999999997,
28.300000000000001,
25.800000000000001,
33.399999999999999,
28.800000000000001,
22.199999999999999,
32.700000000000003,
32.899999999999999,
27.399999999999999,
34.100000000000001,
26.199999999999999,
24.399999999999999,
30.199999999999999,
29.0,
30.800000000000001,
28.100000000000001,
27.199999999999999,
33.600000000000001,
26.100000000000001,
27.300000000000001,
31.899999999999999,
23.5,
28.5,
28.300000000000001,
29.600000000000001,
34.200000000000003,
33.700000000000003,
38.399999999999999,
32.399999999999999,
34.600000000000001,
18.300000000000001,
17.100000000000001,
33.79999999999997,
29.300000000000001,
28.100000000000001,
21.100000000000001,
36.700000000000003,
25.699999999999999,

33.100000000000001,
23.399999999999999,
25.600000000000001,
36.100000000000001,
36.899999999999999,
29.5,
28.0,
29.399999999999999,
27.300000000000001,
34.600000000000001,
27.199999999999999,
33.100000000000001,
28.199999999999999,
34.0,
17.800000000000001,
32.399999999999999,
24.199999999999999,
22.699999999999999,
34.399999999999999,
32.79999999999997,
19.699999999999999,
35.100000000000001,
30.699999999999999,
35.0,
31.300000000000001,
23.100000000000001,
31.300000000000001,
23.5,
27.800000000000001,
20.800000000000001,
38.29999999999997,
22.300000000000001,
27.699999999999999,
23.600000000000001,
23.0,
34.0,
31.5,
31.300000000000001,
24.399999999999999,
31.899999999999999,
30.300000000000001,
31.199999999999999,
29.5,
30.100000000000001,
25.5,
26.300000000000001,
31.0,
21.699999999999999,

35.700000000000003,
18.5,
27.199999999999999,
22.899999999999999,
26.699999999999999,
30.699999999999999,
32.399999999999999,
32.700000000000003,
28.800000000000001,
31.199999999999999,
32.0,
29.199999999999999,
23.399999999999999,
27.199999999999999,
34.799999999999997,
28.300000000000001,
31.5,
30.699999999999999,
20.0,
37.399999999999999,
27.399999999999999,
33.100000000000001,
26.699999999999999,
32.700000000000003,
22.100000000000001,
28.100000000000001,
25.300000000000001,
31.5,
30.399999999999999,
27.0,
32.899999999999999,
27.899999999999999,
28.0,
23.5,
26.699999999999999,
31.0,
35.200000000000003,
32.700000000000003,
31.199999999999999,
32.600000000000001,
22.300000000000001,
31.800000000000001,
25.5,
28.600000000000001,
25.0,
31.399999999999999,
29.899999999999999,
28.399999999999999,

32.100000000000001,
33.100000000000001,
20.800000000000001,
21.199999999999999,
31.800000000000001,
26.600000000000001,
33.100000000000001,
30.800000000000001,
29.0,
29.800000000000001,
34.899999999999999,
30.199999999999999,
35.700000000000003,
20.199999999999999,
29.600000000000001,
32.5,
36.700000000000003,
20.600000000000001,
29.199999999999999,
32.5,
31.100000000000001,
33.799999999999997,
29.300000000000001,
30.699999999999999,
30.5,
29.199999999999999,
36.0,
28.100000000000001,
29.300000000000001,
27.899999999999999,
28.5,
36.799999999999997,
31.199999999999999,
36.0,
32.0,
28.399999999999999,
37.600000000000001,
29.5,
34.200000000000003,
25.0,
31.399999999999999,
17.800000000000001,
26.600000000000001,
31.199999999999999,
27.899999999999999,
26.600000000000001,
22.5,
30.699999999999999,

32.299999999999997,
30.39999999999999,
29.89999999999999,
33.0,
36.0,
30.300000000000001,
35.29999999999997,
29.10000000000001,
36.10000000000001,
21.0,
25.60000000000001,
28.80000000000001,
32.0,
32.0,
31.69999999999999,
36.10000000000001,
24.5,
23.60000000000001,
22.39999999999999,
22.69999999999999,
29.19999999999999,
34.39999999999999,
26.5,
30.69999999999999,
25.69999999999999,
28.30000000000001,
20.19999999999999,
26.0,
21.0,
27.5,
26.5,
20.39999999999999,
31.39999999999999,
27.89999999999999,
36.5,
18.10000000000001,
27.39999999999999,
33.29999999999997,
23.60000000000001,
31.60000000000001,
30.0,
24.89999999999999,
22.80000000000001,
19.80000000000001,
28.19999999999999,
27.89999999999999,
19.5,
25.10000000000001,

24.89999999999999,
33.100000000000001,
24.300000000000001,
26.39999999999999,
29.5,
28.69999999999999,
22.5,
30.39999999999999,
19.300000000000001,
28.69999999999999,
27.19999999999999,
34.20000000000003,
35.60000000000001,
28.0,
30.10000000000001,
31.89999999999999,
25.80000000000001,
31.5,
26.5,
36.89999999999999,
26.69999999999999,
25.30000000000001,
28.69999999999999,
26.69999999999999,
26.19999999999999,
31.19999999999999,
18.10000000000001,
31.89999999999999,
26.80000000000001,
23.60000000000001,
27.80000000000001,
31.80000000000001,
32.5,
32.10000000000001,
29.39999999999999,
24.10000000000001,
27.60000000000001,
27.39999999999999,
22.5,
34.5,
28.10000000000001,
29.19999999999999,
29.89999999999999,
31.30000000000001,
29.0,
26.60000000000001,
20.0,
30.69999999999999,

```
19.100000000000001,
17.800000000000001,
24.39999999999999,
29.5,
18.0,
25.0,
20.800000000000001,
30.89999999999999,
26.800000000000001,
23.100000000000001,
32.100000000000001,
32.100000000000001,
33.5,
28.39999999999999,
28.0,
27.0,
30.39999999999999,
27.69999999999999,
31.5,
34.20000000000003,
24.80000000000001,
33.29999999999997,
29.30000000000001,
30.60000000000001,
29.39999999999999,
25.80000000000001,
24.30000000000001,
25.80000000000001,
31.39999999999999,
32.0,
22.60000000000001,
28.19999999999999,
27.0,
27.30000000000001,
27.0,
...]
```

5 Visualize the Results - Sample Size = 30

The next cell calculates the means of ten-thousand samples each of size 30 and then visualizes the sample means using a histogram. Notice that the results roughly have the shape of a Gaussian distribution.

```
In [5]: import matplotlib.pyplot as plt

def visualize_results(sample_means):
```

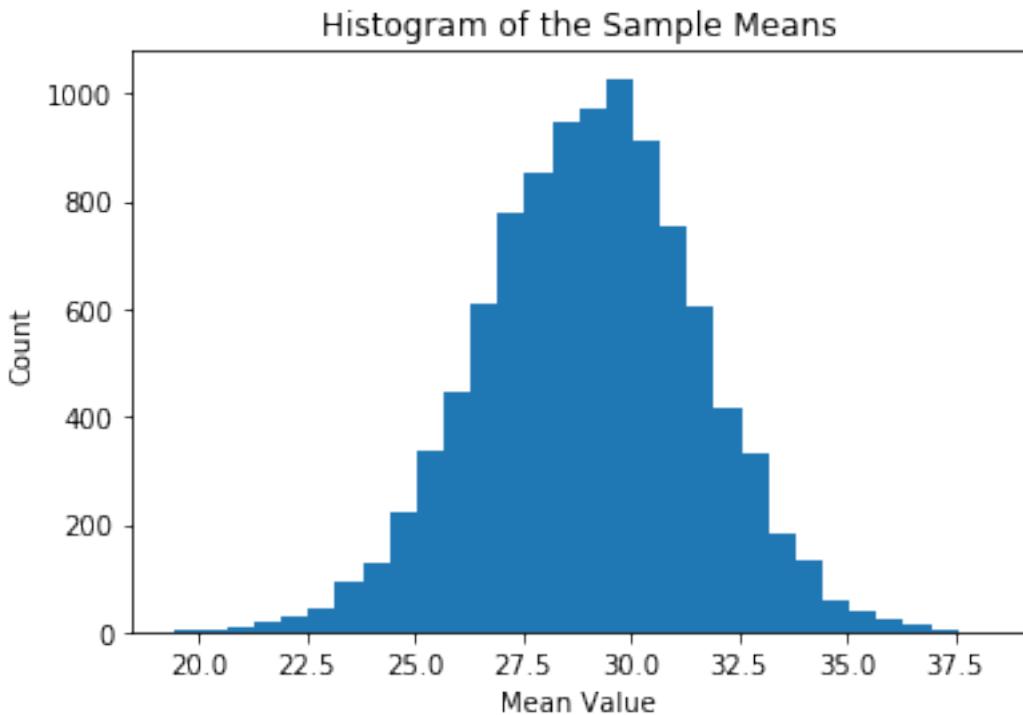
```

plt.hist(sample_means, bins = 30)
plt.title('Histogram of the Sample Means')
plt.xlabel('Mean Value')
plt.ylabel('Count')

# Take random sample and calculate the means
sample_means_results = central_limit_theorem(population_data, 30, 10000)

# Visualize the results
visualize_results(sample_means_results)

```



So we started off with a population that definitely did not have a Gaussian distribution. But by taking samples of the distribution and calculating the sample means, we end up with something that looks like a Gaussian distribution.

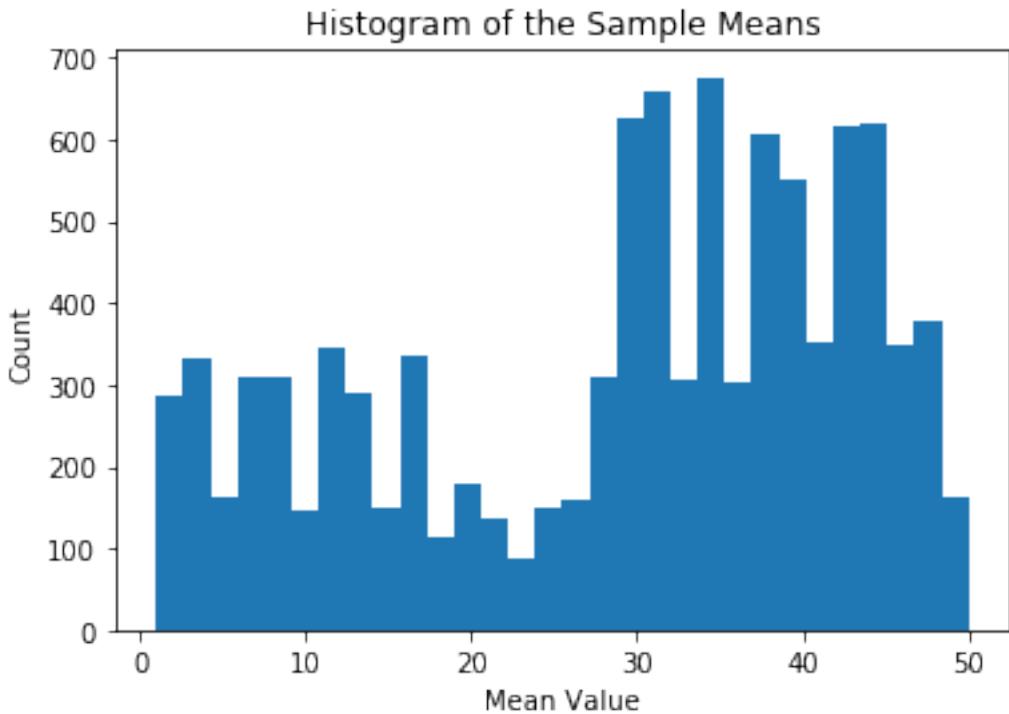
6 Visualize the Results - Sample Size = 1

One part of the central limit theorem states that the sample size needs to be large enough. A general rule of thumb is that the sample size should be greater than or equal to 30. Let's try using different sample sizes to see what the results look like.

An exaggerated case would be to use a sample size of 1. This should give us a similar distribution to the original population. Run the code below to see the results.

```
In [6]: # Take random sample and calculate the means
sample_means_results = central_limit_theorem(population_data, 1, 10000)
```

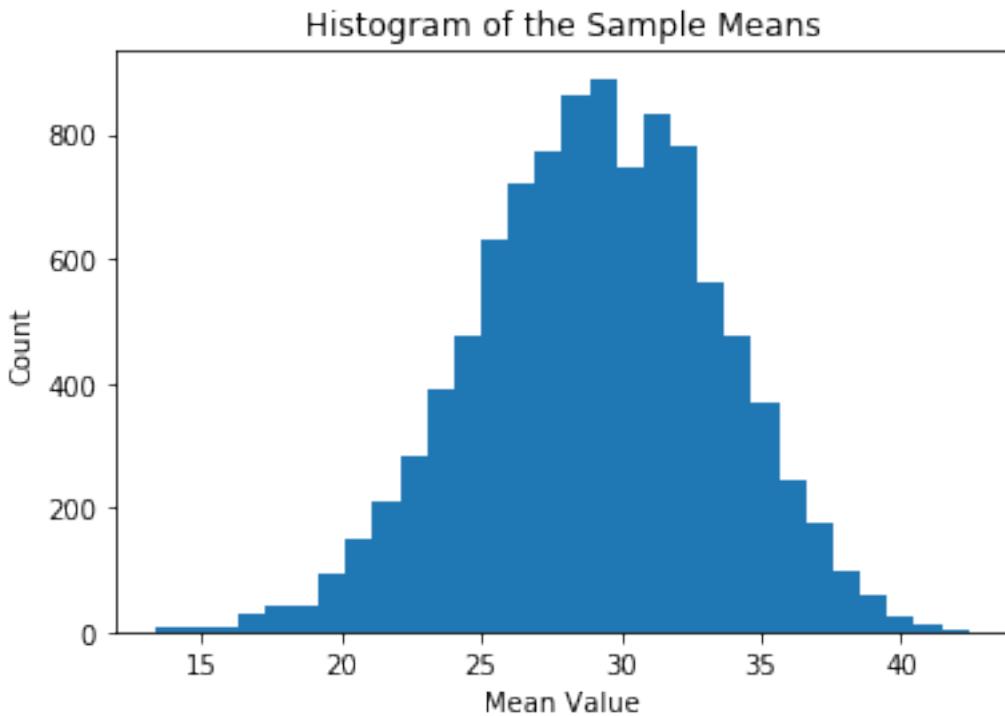
```
# Visualize the results  
visualize_results(sample_means_results)
```



7 Visualize the Results - Sample Size = 10

Now, let's use the minimum recommended sample size of 30 and see what happens.

```
In [7]: # Take random sample and calculate the means  
sample_means_results = central_limit_theorem(population_data, 10, 10000)  
  
# Visualize the results  
visualize_results(sample_means_results)
```



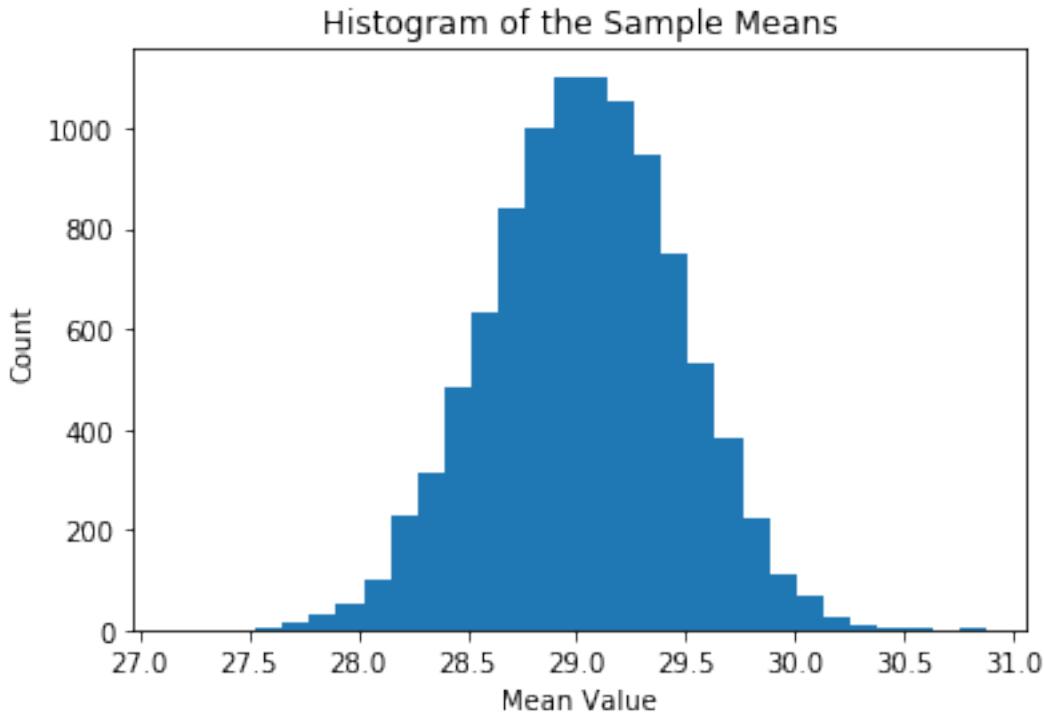
With a sample size of 10, the distribution of the sample means does look somewhat Gaussian.

8 Visualize the Results - Sample Size = 1000

Let's go even further and use a larger sample size: in this case 1000.

```
In [8]: # Take random sample and calculate the means
        sample_means_results = central_limit_theorem(population_data, 1000, 10000)

        # Visualize the results
        visualize_results(sample_means_results)
```



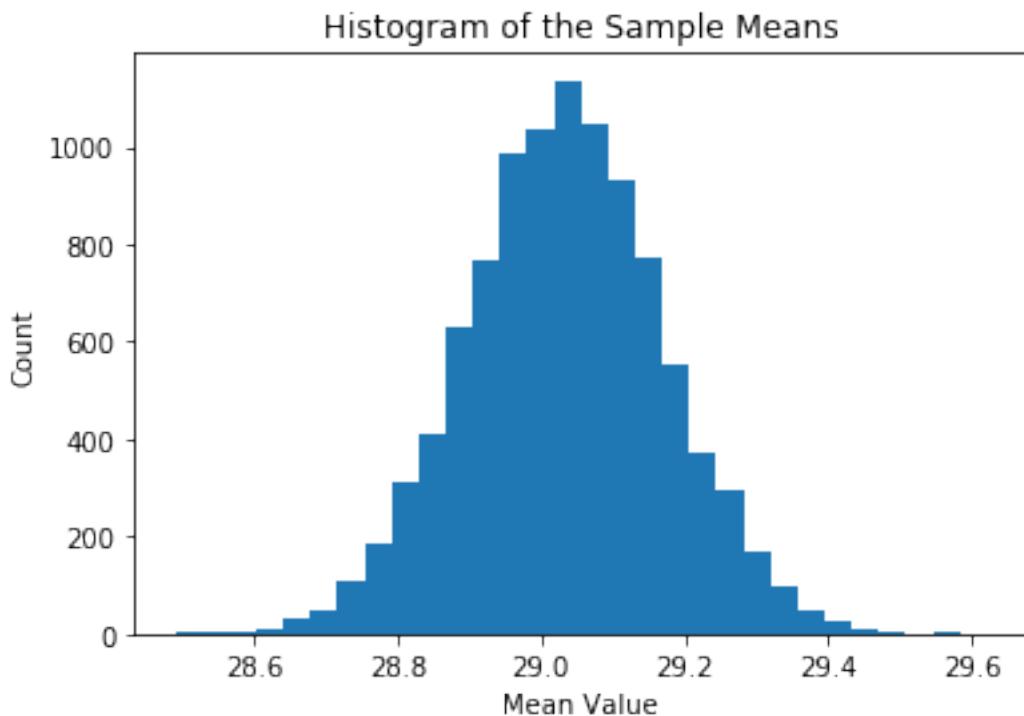
This result not only looks Gaussian, but you can see another trend. The spread of the data (ie the standard deviation) is decreasing as the sample size increases.

9 Visualize the Results - Sample Size = 10000

What happens if the sample size equals the population size? Because we're taking [random samples with replacement](#), it's very unlikely that one of the samples would be exactly the population; however, the standard deviation should decrease even further since each sample will likely be similar to the population.

```
In [9]: # Take random sample and calculate the means
sample_means_results = central_limit_theorem(population_data, 10000, 10000)

# Visualize the results
visualize_results(sample_means_results)
```



10 Conclusion

Notice as well that the center of these distributions is near the original population mean.

Think about collecting data in the real world. If you wanted to find the distribution of human height around the world, you could measure every single person's height and analyze the results. If you took the mean of your results, you would have the true average of human height; however, measuring the entire world population is not feasible.

Instead, you could take a sample of heights. If you only measured thirty people, your sample means is likely to be far from the population mean. However, if you measured 2 billion randomly chosen people, the sample mean will probably be closer to the population mean. The larger your sample, the more likely your sample mean will match the true population mean.

writeup

March 23, 2020

1 Two Dimensional Histogram Filter - Your First Feature (and your first bug).

Writing code is important. But a big part of being on a self driving car team is working with a **large** existing codebase. On high stakes engineering projects like a self driving car, you will probably have to earn the trust of your managers and coworkers before they'll let you make substantial changes to the code base.

A typical assignment for someone new to a team is to make progress on a backlog of bugs. So with that in mind, that's what you will be doing for your first project in the Nanodegree.

You'll go through this project in a few parts:

1. **Explore the Code** - don't worry about bugs at this point. The goal is to get a feel for how this code base is organized and what everything does.
2. **Implement a Feature** - write code that gets the robot moving correctly.
3. **Fix a Bug** - Implementing motion will reveal a bug which hadn't shown up before. Here you'll identify what the bug is and take steps to reproduce it. Then you'll identify the cause and fix it.

1.1 Part 1: Exploring the code

In this section you will just run some existing code to get a feel for what this localizer does.

You can navigate through this notebook using the arrow keys on your keyboard. You can run the code in a cell by pressing **Ctrl + Enter**

Navigate through the cells below. In each cell you should

1. Read through the code. It's okay to not understand everything at this point.
2. Make a guess about what will happen when you run the code.
3. Run the code and compare what you see with what you expected.
4. When you get to a **TODO** read the instructions carefully and complete the activity.

```
In [1]: # This code "imports" code from some of the other files we've written
        # in this directory. Specifically simulate.py and helpers.py
        import simulate as sim
        import helpers
        import localizer

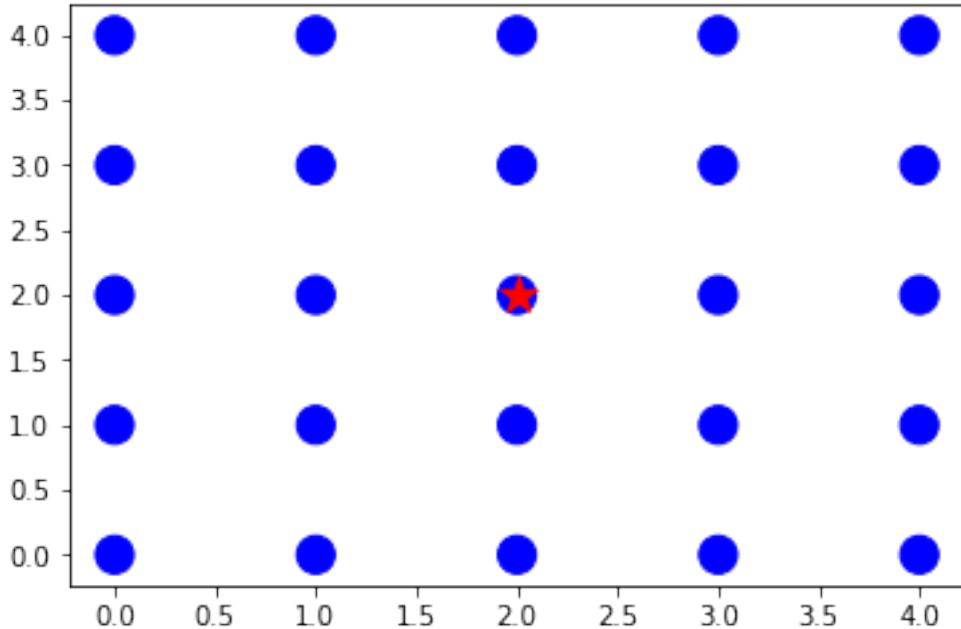
        # Don't worry too much about this code for now...
```

```

from __future__ import division, print_function
%load_ext autoreload
%autoreload 2

In [2]: # This code defines a 5x5 robot world as well as some other parameters
# which we will discuss later. It then creates a simulation and shows
# the initial beliefs.
R = 'r'
G = 'g'
grid = [
    [R,G,G,R,R] ,
    [G,G,R,G,R] ,
    [G,R,G,G,G] ,
    [R,R,G,R,G] ,
    [R,G,R,G,R] ,
]
blur = 0.05
p_hit = 200.0
simulation = sim.Simulation(grid, blur, p_hit)
simulation.show_beliefs()

```



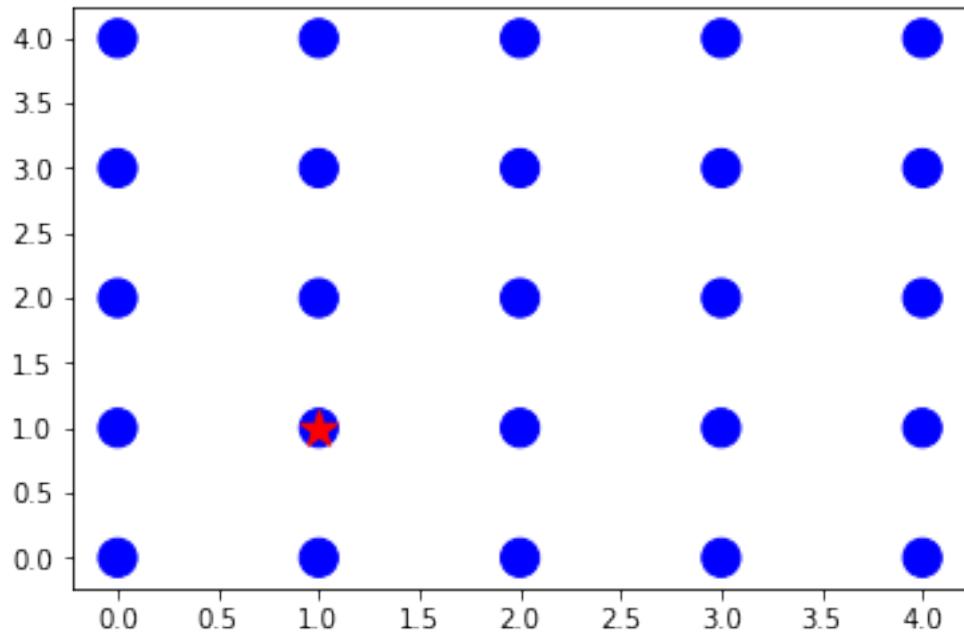
Run the code below multiple times by repeatedly pressing Ctrl + Enter.
After each run observe how the state has changed.

```

In [3]: simulation.run(1)
        simulation.show_beliefs()

```

NOTE! The robot doesn't have a working sense function at this point.



What do you think this call to `run` is doing? Look at the code in `simulate.py` to find out (remember - you can see other files in the current directory by clicking on the jupyter logo in the top left of this notebook).

Spend a few minutes looking at the `run` method and the methods it calls to get a sense for what's going on.

What am I looking at? The red star shows the robot's true position. The blue circles indicate the strength of the robot's belief that it is at any particular location.

Ideally we want the biggest blue circle to be at the same position as the red star.

```
In [4]: # We will provide you with the function below to help you look
# at the raw numbers.
```

```
def show_rounded_beliefs(beliefs):
    for row in beliefs:
        for belief in row:
            print("{:0.3f}".format(belief), end=" ")
        print()

# The {:0.3f} notation is an example of "string
# formatting" in Python. You can learn more about string
# formatting at https://pyformat.info/
```

```
In [5]: show_rounded_beliefs(simulation.beliefs)
```

```
0.040 0.040 0.040 0.040 0.040  
0.040 0.040 0.040 0.040 0.040  
0.040 0.040 0.040 0.040 0.040  
0.040 0.040 0.040 0.040 0.040  
0.040 0.040 0.040 0.040 0.040
```

1.2 Part 2: Implement a 2D sense function.

As you can see, the robot's beliefs aren't changing. No matter how many times we call the simulation's sense method, nothing happens. The beliefs remain uniform.

1.2.1 Instructions

1. Open `localizer.py` and complete the `sense` function.
2. Run the code in the cell below to import the `localizer` module (or reload it) and then test your `sense` function.
3. If the test passes, you've successfully implemented your first feature! Keep going with the project. If your tests don't pass (they likely won't the first few times you test), keep making modifications to the `sense` function until they do!

```
In [41]: reload(localizer)  
def test_sense():  
    R = 'r'  
    _ = 'g'  
  
    simple_grid = [  
        [_, _, _],  
        [_, R, _],  
        [_, _, _]  
    ]  
  
    p = 1.0 / 9  
    initial_beliefs = [  
        [p, p, p],  
        [p, p, p],  
        [p, p, p]  
    ]  
  
    observation = R  
  
    expected_beliefs_after = [  
        [1/11, 1/11, 1/11],  
        [1/11, 3/11, 1/11],  
        [1/11, 1/11, 1/11]  
    ]
```

```

p_hit  = 3.0
p_miss = 1.0
beliefs_after_sensing = localizer.sense(
    observation, simple_grid, initial_beliefs, p_hit, p_miss)

if helpers.close_enough(beliefs_after_sensing, expected_beliefs_after):
    print("Tests pass! Your sense function is working as expected")
    return

elif not isinstance(beliefs_after_sensing, list):
    print("Your sense function doesn't return a list!")
    return

elif len(beliefs_after_sensing) != len(expected_beliefs_after):
    print("Dimensionality error! Incorrect height")
    return

elif len(beliefs_after_sensing[0] ) != len(expected_beliefs_after[0]):
    print("Dimensionality Error! Incorrect width")
    return

elif beliefs_after_sensing == initial_beliefs:
    print("Your code returns the initial beliefs.")
    return

total_probability = 0.0
for row in beliefs_after_sensing:
    for p in row:
        total_probability += p
    if abs(total_probability-1.0) > 0.001:

        print("Your beliefs appear to not be normalized")
        return

print("Something isn't quite right with your sense function")

test_sense()

```

Tests pass! Your sense function is working as expected

1.3 Integration Testing

Before we call this "complete" we should perform an **integration test**. We've verified that the sense function works on it's own, but does the localizer work overall?

Let's perform an integration test. First you should execute the code in the cell below to prepare the simulation environment.

In [42]: `from simulate import Simulation`

```

import simulate as sim
import helpers
reload(localizer)
reload(sim)
reload(helpers)

R = 'r'
G = 'g'
grid = [
    [R,G,G,G,R,R,R],
    [G,G,R,G,R,G,R],
    [G,R,G,G,G,G,R],
    [R,R,G,R,G,G,G],
    [R,G,R,G,R,R,R],
    [G,R,R,R,G,R,G],
    [R,R,R,G,R,G,G],
]
]

# Use small value for blur. This parameter is used to represent
# the uncertainty in MOTION, not in sensing. We want this test
# to focus on sensing functionality
blur = 0.1
p_hit = 100.0
simulation = sim.Simulation(grid, blur, p_hit)

```

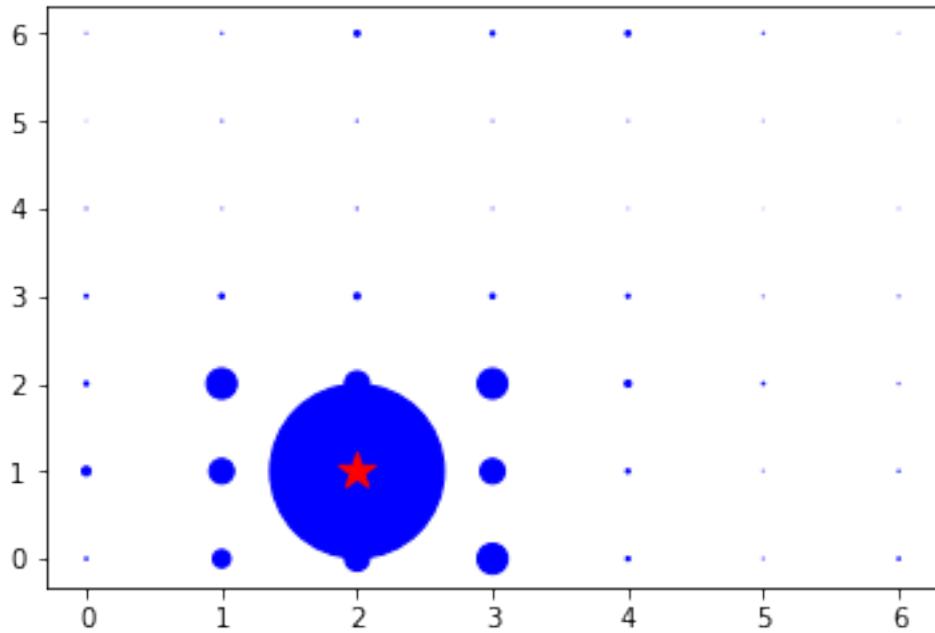
In [62]: # Use control+Enter to run this cell many times and observe how
the robot's belief that it is in each cell (represented by the
size of the corresponding circle) changes as the robot moves.
The true position of the robot is given by the red star.

```

# Run this cell about 15-25 times and observe the results
simulation.run(1)
simulation.show_beliefs()

# If everything is working correctly you should see the beliefs
# converge to a single large circle at the same position as the
# red star. Though, if your sense function is implemented correctly
# and this output is not converging as expected.. it may have to do
# with the `move` function bug; your next task!
#
# When you are satisfied that everything is working, continue
# to the next section

```



1.4 Part 3: Identify and Reproduce a Bug

Software has bugs. That's okay.

A user of your robot called tech support with a complaint

"So I was using your robot in a square room and everything was fine. Then I tried loading in a map for a rectangular room and it drove around for a couple seconds and then suddenly stopped working. Fix it!"

Now we have to debug. We are going to use a systematic approach.

1. Reproduce the bug
2. Read (and understand) the error message (when one exists)
3. Write a test that triggers the bug.
4. Generate a hypothesis for the cause of the bug.
5. Try a solution. If it fixes the bug, great! If not, go back to step 4.

1.4.1 Step 1: Reproduce the bug

The user said that **rectangular environments** seem to be causing the bug.

The code below is the same as the code you were working with when you were doing integration testing of your new feature. See if you can modify it to reproduce the bug.

```
In [72]: from simulate import Simulation
        import simulate as sim
        import helpers
```

```

reload(localizer)
reload(sim)
reload(helpers)

R = 'r'
G = 'g'

grid = [
    [R,G,G,G,R,R,R],
    [G,G,R,G,R,G,R],
    [G,R,G,G,G,G,R],
    [R,R,G,R,G,G,G],
]
blur = 0.001
p_hit = 100.0
simulation = sim.Simulation(grid, blur, p_hit)

# remember, the user said that the robot would sometimes drive around for a bit...
# It may take several calls to "simulation.run" to actually trigger the bug.
simulation.run(1)
simulation.show_beliefs()

```

IndexErrorTraceback (most recent call last)

```

<ipython-input-72-505b0031fba1> in <module>()
22 # remember, the user said that the robot would sometimes drive around for a bit...
23 # It may take several calls to "simulation.run" to actually trigger the bug.
--> 24 simulation.run(1)
25 simulation.show_beliefs()

```

```

/home/workspace/simulate.pyc in run(self, num_steps)
103                     self.sense()
104                     dy, dx = self.random_move()
--> 105                     self.move(dy,dx)

```

```

/home/workspace/simulate.pyc in move(self, dy, dx)
55                     self.true_pose = (new_y, new_x)
56                     beliefs = deepcopy(self.beliefs)
--> 57                     new_beliefs = localizer.move(dy, dx, beliefs, self.blur)
58                     self.beliefs = new_beliefs
59

```

```
/home/workspace/localizer.pyc in move(dy, dx, beliefs, blurring)
48             new_j = (j + dx) % height
49             # pdb.set_trace()
---> 50             new_G[int(new_i)][int(new_j)] = cell
51     return blur(new_G, blurring)
```

```
IndexError: list index out of range
```

```
In [73]: simulation.run(1)
```

```
IndexErrorTraceback (most recent call last)
```

```
<ipython-input-73-019c317be5a5> in <module>()
----> 1 simulation.run(1)
```

```
/home/workspace/simulate.pyc in run(self, num_steps)
103             self.sense()
104             dy, dx = self.random_move()
--> 105             self.move(dy,dx)
```

```
/home/workspace/simulate.pyc in move(self, dy, dx)
55             self.true_pose = (new_y, new_x)
56             beliefs = deepcopy(self.beliefs)
---> 57             new_beliefs = localizer.move(dy, dx, beliefs, self.blur)
58             self.beliefs = new_beliefs
59
```

```
/home/workspace/localizer.pyc in move(dy, dx, beliefs, blurring)
48             new_j = (j + dx) % height
49             # pdb.set_trace()
---> 50             new_G[int(new_i)][int(new_j)] = cell
51     return blur(new_G, blurring)
```

```
IndexError: list index out of range
```

1_vector_coding

March 24, 2020

1 Vectors in Python

In the following exercises, you will work on coding vectors in Python.

Assume that you have a state vector

x_0

representing the x position, y position, velocity in the x direction, and velocity in the y direction of a car that is driving in front of your vehicle. You are tracking the other vehicle.

Currently, the other vehicle is 5 meters ahead of you along your x-axis, 2 meters to your left along your y-axis, driving 10 m/s in the x direction and 0 m/s in the y-direction. How would you represent this in a Python list where the vector contains $\langle x, y, vx, vy \rangle$ in exactly that order?

1.0.1 Vector Assignment: Example 1

In [1]: *## Practice working with Python vectors*

```
## TODO: Assume the state vector contains values for <x, y, vx, vy>
## Currently, x = 5, y = 2, vx = 10, vy = 0
## Represent this information in a list
x0 = [5, 2, 10, 0]
```

1.0.2 Test your code

Run the cell below to test your code.

The test code uses a Python assert statement. If you have a code statement that resolves to either True or False, an assert statement will either: * do nothing if the statement is True * throw an error if the statement is False

A Python assert statement will output an error if the answer was not as expected. If the answer was as expected, then nothing will be outputted.

```
In [2]: ### Test Cases
### Run these test cases to see if your results are as expected
### Running this cell should produce no output if all assertions are True

assert x0 == [5, 2, 10, 0]
```

1.0.3 Vector Assignment: Example 2

The vehicle ahead of you has now moved farther away from you. You know that the vehicle has moved 3 meters forward in the x-direction, 5 meters forward in the y-direction, has increased its x velocity by 2 m/s and has increased its y velocity by 5 m/s.

Store the change in position and velocity in a list variable called xdelta

```
In [3]: ## TODO: Assign the change in position and velocity to the variable  
## xdelta. Remember that the order of the vector is x, y, vx, vy
```

```
xdelta = [3, 5, 2, 5]
```

```
In [4]: ### Test Case
```

```
### Run this test case to see if your results are as expected  
### Running this cell should produce no output if all assertions are True
```

```
assert xdelta == [3, 5, 2, 5]
```

1.0.4 Vector Math: Addition

Calculate the tracked vehicle's new position and velocity. Here are the steps you need to carry this out:

- initialize an empty list called x1
- add xdelta to x0 using a for loop
- store your results in x1 as you iterate through the for loop using the append method

```
In [7]: ## TODO: Add the vectors together element-wise. For example,  
## element-wise addition of [2, 6] and [10, 3] is [12, 9].  
## Place the answer in the x1 variable.  
##  
## Hint: You can use a for loop. The append method might also  
## be helpful.
```

```
x1 = []  
for i in range(len(x0)):  
    x1.append(x0[i] + xdelta[i])
```

```
In [8]: ### Test Case
```

```
### Run this test case to see if your results are as expected  
### Running this cell should produce no output if all assertions are True  
assert x1 == [8, 7, 12, 5]
```

1.0.5 Vector Math: Scalar Multiplication

You have your current position in meters and current velocity in meters per second. But you need to report your results at a company meeting where most people will only be familiar with working in feet rather than meters. Convert your position vector x1 to feet and feet/second.

This will involve scalar multiplication. The process for coding scalar multiplication is very similar to vector addition. You will need to:

- * initialize an empty list
- * use a for loop to access each element in the vector
- * multiply each element by the scalar
- * append the result to the empty list

```
In [10]: ## TODO: Multiply each element in the x1 vector by the conversion
## factor shown below and store the results in the variable s.
## Use a for loop

meters_to_feet = 1.0 / 0.3048
x1feet = []
for i in range(len(x1)):
    x1feet.append(x1[i] * meters_to_feet)

In [11]: ### Test Cases
### Run this test case to see if your results are as expected
### Running this cell should produce no output if all assertions are True
x1feet_sol = [8/.3048, 7/.3048, 12/.3048, 5/.3048]

assert(len(x1feet) == len(x1feet_sol))
for response, expected in zip(x1feet, x1feet_sol):
    assert(abs(response-expected) < 0.001)
```

1.0.6 Vector Math: Dot Product

The tracked vehicle is currently at the state represented by

$$\mathbf{x}_1 = [8, 7, 12, 5]$$

Where will the vehicle be in two seconds?

You could actually solve this problem very quickly using Matrix multiplication, but we have not covered that yet. Instead, think about the x-direction and y-direction separately and how you could do this with the dot product.

Solving with the Dot Product You know that the tracked vehicle at \mathbf{x}_1 is 8m ahead of you in the x-direction and traveling at 12m/s. Assuming constant velocity, the new x-position after 2 seconds would be

$$8 + 12 * 2 = 32$$

The new y-position would be

$$7 + 5 * 2 = 17$$

You could actually solve each of these equations using the dot product:

$$x_2 = [8, 7, 12, 5] \cdot [1, 0, 2, 0] = 8 \times 1 + 7 \times 0 + 12 \times 2 + 5 \times 0 = 32$$

$$y_2 = [8, 7, 12, 5] \cdot [0, 1, 0, 2] = 8 \times 0 + 7 \times 1 + 12 \times 0 + 5 \times 2 = 17$$

Since you are assuming constant velocity, the final state vector would be

$$\mathbf{x}_2 = [32, 17, 12, 5]$$

Coding the Dot Product Now, calculate the state vector \mathbf{x}_2

but with code. You will need to calculate the dot product of two vectors. Rather than writing the dot product code for the x-direction and then copying the code for the y-direction, write a function that calculates the dot product of two Python lists.

Here is an outline of the steps:

- * initialize an empty list
- * initialize a variable with value zero to accumulate the sum
- * use a for loop to iterate through the vectors. Assume the two vectors have the same length
- * accumulate the sum as you multiply elements together

You will see in the starter code that \mathbf{x}_2 is already being calculated for you based on the results of your dotproduct function

```
In [32]: ## TODO: Fill in the dotproduct() function to calculate the
## dot product of two vectors.
## 

## Here are the inputs and outputs of the dotproduct() function:
##     INPUTS: vector, vector
##     OUTPUT: dot product of the two vectors
##
##
## The dot product involves multiplying the vectors element
## by element and then taking the sum of the results
##
## For example, the dot product of [9, 7, 5] and [2, 3, 4] is
## 9*2+7*3 +5*4 = 59
##
## Hint: You can use a for loop. You will also need to accumulate
## the sum as you iterate through the vectors. In Python, you can accumulate
## sums with syntax like w = w + 1

x2 = []

def dotproduct(vectora, vectorb):

    # variable for accumulating the sum
    result = 0

    # TODO: Use a for loop to multiply the two vectors
    # element by element. Accumulate the sum in the result variable

    for i in range(len(vectora)):
        result += vectora[i] * vectorb[i]
    return result

x2 = [dotproduct([8, 7, 12, 5], [1, 0, 2, 0]),
      dotproduct([8, 7, 12, 5], [0, 1, 0, 2]),
```

```
12,  
5]
```

```
In [33]: ### Test Case  
### Run this test case to see if your results are as expected  
### Running this cell should produce no output if all assertions are True  
assert x2 == [32, 17, 12, 5]
```

```
In [ ]:
```

2_matrices_in_python

March 24, 2020

0.1 Coding Matrices

Here are a few exercises to get you started with coding matrices. The exercises start off with vectors and then get more challenging

0.1.1 Vectors

```
In [18]: ### TODO: Assign the vector <5, 10, 2, 6, 1> to the variable v  
v = [5, 10, 2, 6, 1]
```

The v variable contains a Python list. This list could also be thought of as a 1x5 matrix with 1 row and 5 columns. How would you represent this list as a matrix?

```
In [19]: ### TODO: Assign the vector <5, 10, 2, 6, 1> to the variable mv  
### The difference between a vector and a matrix in Python is that  
### a matrix is a list of lists.
```

```
### Hint: See the last quiz on the previous page
```

```
mv = [v]
```

How would you represent this vector in its vertical form with 5 rows and 1 column? When defining matrices in Python, each row is a list. So in this case, you have 5 rows and thus will need 5 lists.

As an example, this is what the vector

$$<5, 7>$$

would look like as a 1x2 matrix in Python:

```
matrix1by2 = [  
    [5, 7]  
]
```

And here is what the same vector would look like as a 2x1 matrix:

```
matrix2by1 = [  
    [5],  
    [7]  
]
```

```
In [32]: ### TODO: Assign the vector <5, 10, 2, 6, 1> to the variable vT
        ### vT is a 5x1 matrix
vT = [[5], [10], [2], [6], [1]]
```

0.1.2 Assigning Matrices to Variables

```
In [33]: ### TODO: Assign the following matrix to the variable m
        ### 8 7 1 2 3
        ### 1 5 2 9 0
        ### 8 2 2 4 1

m = [[8, 7, 1, 2, 3],
      [1, 5, 2, 9, 0],
      [8, 2, 2, 4, 1]
     ]
```

0.1.3 Accessing Matrix Values

```
In [34]: ### TODO: In matrix m, change the value
            ###       in the second row last column from 0 to 5
            ### Hint: You do not need to rewrite the entire matrix
m[1][4] = 5
```

0.1.4 Looping through Matrices to do Math

Coding mathematical operations with matrices can be tricky. Because matrices are lists of lists, you will need to use a for loop inside another for loop. The outside for loop iterates over the rows and the inside for loop iterates over the columns.

Here is some pseudo code

```
for i in number of rows:
    for j in number of columns:
        mymatrix[i][j]
```

To figure out how many times to loop over the matrix, you need to know the number of rows and number of columns.

If you have a variable with a matrix in it, how could you figure out the number of rows? How could you figure out the number of columns? The `len` function in Python might be helpful.

0.1.5 Scalar Multiplication

```
In [35]: ### TODO: Use for loops to multiply each matrix element by 5
            ###       Store the answer in the r variable. This is called scalar
            ###       multiplication
            ###
            ### HINT: First write a for loop that iterates through the rows
            ###       one row at a time
            ###
            ###       Then write another for loop within the for loop that
```

```

#### iterates through the columns
#####
If you used the variable i to represent rows and j
##### to represent columns, then m[i][j] would give you
##### access to each element in the matrix
#####
Because r is an empty list, you cannot directly assign
##### a value like r[i][j] = m[i][j]. You might have to
##### work on one row at a time and then use r.append(row).
r = []
row = []
for i in range(len(m)):
    for j in range(len(m[0])):
        row.append(m[i][j] * 5)
    r.append(row)
    row = []

```

0.1.6 Printing Out a Matrix

```

In [36]: #### TODO: Write a function called matrix_print()
          #### that prints out a matrix in
          #### a way that is easy to read.
          #### Each element in a row should be separated by a tab
          #### And each row should have its own line
          #### You can test our your results with the m matrix

          #### HINT: You can use a for loop within a for loop
          #### In Python, the print() function will be useful
          #### print(5, '\t', end = '') will print out the integer 5,
          #### then add a tab after the 5. The end = '' makes sure that
          #### the print function does not print out a new line if you do
          #### not want a new line.

          #### Your output should look like this
          #### 8   7   1   2   3
          #### 1   5   2   9   5
          #### 8   2   2   4   1

def matrix_print(matrix):
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            print(matrix[i][j], '\t', end = '')
        print(matrix[i][j])
    return

m = [
    [8, 7, 1, 2, 3],
    [1, 5, 2, 9, 5],

```

```

[8, 2, 2, 4, 1]
]

matrix_print(m)

8      7      1      2      3      3
1      5      2      9      5      5
8      2      2      4      1      1

```

0.1.7 Test Your Results

In [37]: *### You can run these tests to see if you have the expected
results. If everything is correct, this cell has no output*

```

assert v == [5, 10, 2, 6, 1]
assert mv == [
    [5, 10, 2, 6, 1]
]

assert vT == [
    [5],
    [10],
    [2],
    [6],
    [1]]

assert m == [
    [8, 7, 1, 2, 3],
    [1, 5, 2, 9, 5],
    [8, 2, 2, 4, 1]
]

assert r == [
    [40, 35, 5, 10, 15],
    [5, 25, 10, 45, 25],
    [40, 10, 10, 20, 5]
]

```

0.1.8 Print Out Your Results

In [38]: *### Run this cell to print out your answers*

```

print(v)
print(mv)
print(vT)
print(m)
print(r)

```

```
[5, 10, 2, 6, 1]
[[5, 10, 2, 6, 1]]
[[5], [10], [2], [6], [1]]
[[8, 7, 1, 2, 3], [1, 5, 2, 9, 5], [8, 2, 2, 4, 1]]
[[40, 35, 5, 10, 15], [5, 25, 10, 45, 25], [40, 10, 10, 20, 5]]
```

In []:

3_matrix_addition

March 24, 2020

1 Matrix Addition

In this exercises, you will write a function that accepts two matrices and outputs their sum. Think about how you could do this with a for loop nested inside another for loop.

```
In [2]: ### TODO: Write a function called matrix_addition that
        ### calculate the sum of two matrices
        ###
        ### INPUTS:
        ###     matrix A - an m x n matrix
        ###     matrix B - an m x n matrix
        ###
        ### OUTPUT:
        ###     matrixSum - sum of matrix A + matrix B

def matrix_addition(matrixA, matrixB):

    # initialize matrix to hold the results
    matrixSum = []

    # matrix to hold a row for appending sums of each element
    row = []

    # TODO: write a for loop within a for loop to iterate over
    # the matrices

    # TODO: As you iterate through the matrices, add matching
    # elements and append the sum to the row variable

    # TODO: When a row is filled, append the row to matrixSum.
    # Then reinitialize row as an empty list

    for i in range(len(matrixA)):
        for j in range(len(matrixA[0])):
            row.append(matrixA[i][j] + matrixB[i][j])
        matrixSum.append(row)
        row = []
```

```

    return matrixSum

### When you run this code cell, your matrix addition function
### will run on the A and B matrix.

A = [
    [2,5,1],
    [6,9,7.4],
    [2,1,1],
    [8,5,3],
    [2,1,6],
    [5,3,1]
]

B = [
    [7, 19, 5.1],
    [6.5, 9.2, 7.4],
    [2.8, 1.5, 12],
    [8, 5, 3],
    [2, 1, 6],
    [2, 33, 1]
]

matrix_addition(A, B)

Out[2]: [[9, 24, 6.1],
          [12.5, 18.2, 14.8],
          [4.8, 2.5, 13],
          [16, 10, 6],
          [4, 2, 12],
          [7, 36, 2]]

```

1.0.1 Vectors versus Matrices

What happens if you run the cell below? Here you are adding two vectors together. Does your code still work?

```
In [3]: matrix_addition([4, 2, 1], [5, 2, 7])
```

```
TypeError
```

```
Traceback (most recent call last)
```

```

<ipython-input-3-61a622f3c730> in <module>()
----> 1 matrix_addition([4, 2, 1], [5, 2, 7])

```

```

<ipython-input-2-fcc67a1130f0> in matrix_addition(matrixA, matrixB)
 27
 28     for i in range(len(matrixA)):
---> 29         for j in range(len(matrixA[0])):
 30             row.append(matrixA[i][j] + matrixB[i][j])
 31     matrixSum.append(row)

TypeError: object of type 'int' has no len()

```

Why did this error occur? Because your code assumes that a matrix is a two-dimensional grid represented by a list of lists. But a horizontal vector, which can also be considered a matrix, is a one-dimensional grid represented by a single list.

What happens if you store a vector as a list of lists like $[[4, 2, 1]]$ and $[[5, 2, 7]]$? Does your function work? Run the code cell below to find out.

```
In [4]: matrix_addition([[4, 2, 1]], [[5, 2, 7]])
```

```
Out[4]: [[9, 4, 8]]
```

1.0.2 Test your Code

Run the cell below. If there is no output, then your results are as expected.

```

In [5]: assert matrix_addition([
 1, 2, 3],
 [[4, 5, 6]]) == [[5, 7, 9]]

assert matrix_addition([
 [4], [
 5])) == [[9]]

assert matrix_addition([[1, 2, 3],
 [4, 5, 6],
 [[7, 8, 9],
 [10, 11, 12]]) == [[8, 10, 12],
 [14, 16, 18]]

```

```
In [ ]:
```

4_matrix_multiplication

March 24, 2020

1 Matrix Multiplication

Matrix multiplication involves quite a few steps in terms of writing code. But remember that the basics of matrix multiplicaiton involve taking a row in matrix A and finding the dot product with a column in matrix B.

So you are going to write a function to extract a row from matrix A, extract a column from matrix B, and then calculate the dot product of the row and column.

Then you can use these functions to output the results of multiplying two matrices together.

Here is a general outline of the code that you will be writing. Assume you are calculating the product of

$$A \times B$$

- Write a nested for loop that iterates through the m rows of matrix A and the p columns of matrix B
- Initialize an empty list that will hold values of the final matrix
- Starting with the first row of matrix A, find the dot product with the first column of matrix B
- Append the result to the empty list representing a row in the final matrix
- Now find the dot product between the first row of matrix A and the second column of matrix B
- Append the result to the row list
- Keep going until you get to the last column of B
- Append the row list to the output variable. Reinitialize the row list so that it is empty.
- Then start on row two of matrix A. Iterate through all of the columns of B taking the dot product
- etc...

2 Breaking the Process down into steps

Rather than writing all of the matrix multiplication code in one function, you are going to break the process down into several functions:

get_row(matrix, row_number)

Because you are going to need the rows from matrix A, you will use a function called `get_row` that takes in a matrix and row number and then returns a row of a matrix. We have provided this function for you.

get_column(matrix, column_number)

Likewise, you will need the columns from matrix B. So you will write a similar function that receives a matrix and column number and then returns a column from matrix B.

dot_product(vectorA, vectorB)

You have actually already written this function in a previous exercise. The dot_product function calculates the dot product of two vectors.

matrix_multiply(matrixA, matrixB)

This is the function that will calculate the product of the two matrices. You will need to write a nested for loop that iterates through the rows of A and columns of B. For each row-column combination, you will calculate the dot product and then append the result to the output matrix.

3 get_row

The first function is the get_row function. We have provided this function for you.

The get_row function has two inputs and one output.

INPUTS * matrix * row number

OUTPUT * a list, which represents one row of the matrix

In Python, a matrix is a list of lists. If you have a matrix like this one:

```
m = [
    [5, 9, 11, 2],
    [3, 2, 99, 3],
    [7, 1, 8, 2]
]
```

then row one would be accessed by

```
m[0]
```

row two would be

```
m[1]
```

and row three would be

```
m[2]
```

```
In [1]: ## TODO: Run this code cell to load the get_row function
## You do not need to modify this cell
```

```
def get_row(matrix, row):
    return matrix[row]
```

4 Getting a Column from a Matrix

Since matrices are stored as lists of lists, it's relatively simple to extract a row. If A is a matrix, then

```
A[0]
```

will output the first row of the matrix.

```
A[1]
```

outputs the second row of the matrix.

But what if you want to get a matrix column? It's not as convenient. To get the values of the first column, you would need to output:

```
A[0][0]
A[1][0]
A[2][0]
...
A[m][0]
```

For matrix multiplication, you will need to have access to the columns of the B matrix. So write a function called `get_column` that receives a matrix and a column number indexed from zero. The function then outputs a vector as a list that contains the column. For example

```
get_column([
    [1, 2, 4],
    [7, 8, 1],
    [5, 2, 1]
],
1)
```

would output the second column

```
[2, 8, 2]
```

5 `get_column`

The `get_column` function is similar to the `get_row` function except now you will return a column.

Here are the inputs and outputs of the function

INPUTS * matrix * column number

OUTPUT * a list, which represents a column of the matrix

Getting a matrix column is actually more difficult than getting a matrix row.

Take a look again at this example matrix:

```
m = [
    [5, 9, 11, 2],
    [3, 2, 99, 3],
    [7, 1, 8, 2]
]
```

What if you wanted to extract the first column as a list [5, 3, 7]. You can't actually get that column directly like you could with a row.

You'll need to think about using a for statement to iterate through the rows and grab the specific values that you want for your column list.

```
In [6]: ### TODO: Write a function that receives a matrix and a column number.  
###           the output should be the column in the form of a list
```

```
### Example input:  
# matrix = [  
#     [5, 9, 11, 2],  
#     [3, 2, 99, 3],  
#     [7, 1, 8, 2]  
# ]  
#  
# column_number = 1  
  
### Example output:  
# [9, 2, 1]  
#  
def get_column(matrix, column_number):  
    column = []  
    for i in range(len(matrix)):  
        column.append(matrix[i][column_number])  
    return column  
  
In [7]: ### TODO: Run this code to test your get_column function  
assert get_column([[1, 2, 4],  
                  [7, 8, 1],  
                  [5, 2, 1]], 1) == [2, 8, 2]  
  
assert get_column([[5]], 0) == [5]
```

5.0.1 Dot Product of Two Vectors

As part of calculating the product of a matrix, you need to do calculate the dot product of a row from matrix A with a column from matrix B. You will do this process many times, so why not abstract the process into a function?

If you consider a single row of A to be a vector and a single row of B to also be a vector, you can calculate the dot product.

Remember that for matrix multiplication to be valid, A is size $m \times n$ while B is size $n \times p$. The number of columns in A must equal the number of rows in B, which makes taking the dot product between a row of A and column of B possible.

As a reminder, the dot product of $\langle a_1, a_2, a_3, a_4 \rangle$ and $\langle b_1, b_2, b_3, b_4 \rangle$ is equal to $a_1*b_1 + a_2*b_2 + a_3*b_3 + a_4*b_4$

```
In [12]: ### TODO: Write a function called dot_product() that  
###           has two vectors as inputs and outputs the dot product of the  
###           two vectors. First, you will need to do element-wise  
###           multiplication and then sum the results.
```

```

### HINT: You wrote this function previously in the vector coding
### exercises

def dot_product(vector_one, vector_two):
    sum = 0
    for i in range(len(vector_one)):
        sum += vector_one[i] * vector_two[i]
    return sum

```

In [13]: *### TODO: Run this cell to test your results*

```

assert dot_product([4, 5, 1], [2, 1, 5]) == 18
assert dot_product([6], [7]) == 42

```

5.0.2 Matrix Multiplication

Now you will write a function to carry out matrix multiplication between two matrices.

If you have an $m \times n$ matrix and an $n \times p$ matrix, your result will be $m \times p$.

Your strategy could involve looping through an empty $n \times p$ matrix and filling in each elements value.

```

In [16]: ### TODO: Write a function called matrix_multiplication that takes
###           two matrices, multiplies them together and then returns
###           the results
###
###           Make sure that your function can handle matrices that contain
###           only one row or one column. For example,
###           multiplying two matrices of size (4x1)x(1x4) should return a
###           4x4 matrix

def matrix_multiplication(matrixA, matrixB):

    ### TODO: store the number of rows in A and the number
    ###           of columns in B. This will be the size of the output
    ###           matrix
    ### HINT: The len function in Python will be helpful
    m_rows = len(matrixA)
    p_columns = len(matrixB[0])

    # empty list that will hold the product of AxB
    result = []
    row_result = []

    ### TODO: Write a for loop within a for loop. The outside
    ###           for loop will iterate through m_rows.
    ###           The inside for loop will iterate through p_columns.

```

```

for i in range(m_rows):
    for j in range(p_columns):

        ### TODO: As you iterate through the m_rows and p_columns,
        ###       use your get_row function to grab the current A row
        ###       and use your get_column function to grab the current
        ###       B column.

        row_A = get_row(matrixA, i)
        column_B = get_column(matrixB, j)

        ### TODO: Calculate the dot product of the A row and the B column

        ### TODO: Append the dot product to an empty list called row_result.
        ###       This list will accumulate the values of a row
        ###       in the result matrix

        row_result.append(dot_product(row_A, column_B))

        ### TODO: After iterating through all of the columns in matrix B,
        ###       append the row_result list to the result variable.
        ###       Reinitialize the row_result to row_result = [].
        ###       Your for loop will move down to the next row
        ###       of matrix A.
        ###       The loop will iterate through all of the columns
        ###       taking the dot product
        ###       between the row in A and each column in B.

        result.append(row_result)
        row_result = []

        ### TODO: return the result of Ax B

    return result

```

In [17]: *### TODO: Run this code cell to test your results*

```

assert matrix_multiplication([[5], [2]], [[5, 1]]) == [[25, 5], [10, 2]]
assert matrix_multiplication([[5, 1]], [[5], [2]]) == [[27]]
assert matrix_multiplication([[4]], [[3]]) == [[12]]
assert matrix_multiplication([[2, 1, 8, 2, 1], [5, 6, 4, 2, 1]], [[1, 7, 2], [2, 6, 3]])

```

In []:

5_matrix_transpose

March 24, 2020

1 Transpose of a Matrix

In this set of exercises, you will work with the transpose of a matrix.

Your first task is to write a function that takes the transpose of a matrix. Think about how to use nested for loops efficiently.

The second task will be to write a new matrix multiplication function that takes advantage of your matrix transposition function.

In [2]: *### TODO: Write a function called transpose() that
takes in a matrix and outputs the transpose of the matrix*

```
def transpose(matrix):
    matrix_transpose = []
    row = []
    for j in range(len(matrix[0])):
        for i in range(len(matrix)):
            row.append(matrix[i][j])
        matrix_transpose.append(row)
        row = []

    return matrix_transpose
```

In [3]: *### TODO: Run the code in the cell below. If there is no
output, then your answers were as expected*

```
assert transpose([[5, 4, 1, 7], [2, 1, 3, 5]]) == [[5, 2], [4, 1], [1, 3], [7, 5]]
assert transpose([[5]]) == [[5]]
assert transpose([[5, 3, 2], [7, 1, 4], [1, 1, 2], [8, 9, 1]]) == [[5, 7, 1, 8], [3, 1,
```

1.0.1 Matrix Multiplication

Now that you have your transpose function working, write a matrix multiplication function that takes advantage of the transpose.

As part of the matrix multiplication code, you might want to re-use your dot product function from the matrix multiplication exercises. But you won't need your get_row and get_column functions anymore because the transpose essentially takes care of turning columns into row vectors.

Remember that if matrix A is mxn and matrix B is nxp, then the resulting product will be m xp.

```
In [16]: ### TODO: Write a function called matrix_multiplication() that
###           takes in two matrices and outputs the product of the two
###           matrices

### TODO: Copy your dot_product() function here so that you can
###           use it in your matrix_multiplication function

def dot_product(vectorA, vectorB):
    result = 0

    for i in range(len(vectorA)):
        result += vectorA[i] * vectorB[i]

    return result

def matrix_multiplication(matrixA, matrixB):
    product = []
    new_row = []

    ## TODO: Take the transpose of matrixB and store the result
    ##       in a new variable

    matrixB_T = transpose(matrixB)

    ## TODO: Use a nested for loop to iterate through the rows
    ##       of matrix A and the rows of the tranpose of matrix B

    for i in range(len(matrixA)):
        for j in range(len(matrixB_T)):

            ## TODO: Calculate the dot product between each row of matrix A
            ##       with each row in the transpose of matrix B

            dp = dot_product(matrixA[i], matrixB_T[j])
            new_row.append(dp)

            ## TODO: As you calculate the results inside your for loops,
            ##       store the results in the product variable

            product.append(new_row)
            new_row = []

    ## TODO:
    return product
```

```
In [17]: ### TODO: Run the code in the cell below. If there is no
###           output, then your answers were as expected
```

```
assert matrix_multiplication([[5, 3, 1],  
                             [6, 2, 7]],  
                             [[4, 2],  
                              [8, 1],  
                              [7, 4]]) == [[51, 17],  
                                            [89, 42]]  
  
assert matrix_multiplication([[5]], [[4]]) == [[20]]  
  
assert matrix_multiplication([[2, 8, 1, 2, 9],  
                             [7, 9, 1, 10, 5],  
                             [8, 4, 11, 98, 2],  
                             [5, 5, 4, 4, 1]],  
                             [[4],  
                              [2],  
                              [17],  
                              [80],  
                              [2]]) == [[219], [873], [8071], [420]]
```

```
assert matrix_multiplication([[2, 8, 1, 2, 9],  
                             [7, 9, 1, 10, 5],  
                             [8, 4, 11, 98, 2],  
                             [5, 5, 4, 4, 1]],  
                             [[4, 1, 2],  
                              [2, 3, 1],  
                              [17, 8, 1],  
                              [1, 3, 0],  
                              [2, 1, 4]]) == [[61, 49, 49], [83, 77, 44], [329, 404, 39]]
```

In []:

6_inverse_matrix

March 24, 2020

1 Matrix Inverse

In this exercise, you will write a function to calculate the inverse of either a 1x1 or a 2x2 matrix.

```
In [10]: ### TODO: Write a function called inverse_matrix() that
### receives a matrix and outputs the inverse
###
### You are provided with start code that checks
### if the matrix is square and if not, throws an error
###
### You will also need to check the size of the matrix.
### The formula for a 1x1 matrix and 2x2 matrix are different,
### so your solution will need to take this into account.
###
### If the user inputs a non-invertible 2x2 matrix or a matrix
### of size 3 x 3 or greater, the function should raise an
### error. A non-invertible
### 2x2 matrix has ad-bc = 0 as discussed in the lesson
###
### Python has various options for raising errors
raise RuntimeError('this is the error message')
raise NotImplementedError('this functionality is not implemented')
raise ValueError('The denominator of a fraction cannot be zero')

def inverse_matrix(matrix):

    inverse = []

    if len(matrix) != len(matrix[0]):
        raise ValueError('The matrix must be square')

    ## TODO: Check if matrix is larger than 2x2.
## If matrix is too large, then raise an error

    if len(matrix) > 2:
        raise NotImplementedError('This functionality is not implemented, too large')
```

```

## TODO: Check if matrix is 1x1 or 2x2.
## Depending on the matrix size, the formula for calculating
## the inverse is different.
# If the matrix is 2x2, check that the matrix is invertible

if len(matrix) == 1:
    inverse.append([1 / matrix[0][0]])
else:
    row = []
    a = matrix[0][0]
    b = matrix[0][1]
    c = matrix[1][0]
    d = matrix[1][1]
    if(a * d - b * c == 0):
        raise ValueError('The Matrix is not invertible')
    else:
        factor = 1 / (a * d - b * c)
        inverse = [
            [d, -b],
            [-c, a]
        ]
        for i in range(2):
            for j in range(2):
                inverse[i][j] = factor * inverse[i][j]

## TODO: Calculate the inverse of the square 1x1 or 2x2 matrix.

return inverse

```

In [11]: ## TODO: Run this cell to check your output. If this cell does
not output anything your answers were as expected.

```

assert inverse_matrix([[100]]) == [[0.01]]
assert inverse_matrix([[4, 5], [7, 1]]) == [[-0.03225806451612903, 0.16129032258064516]
[0.22580645161290322, -0.12903225806451613]]

```

In [12]: ### Run this line of code and see what happens. Because ad = bc, this
matrix does not have an inverse
inverse_matrix([[4, 2], [14, 7]])

ValueError

Traceback (most recent call last)

```
<ipython-input-12-a57ce83b3ae1> in <module>()
    1 ### Run this line of code and see what happens. Because ad = bc, this
    2 ### matrix does not have an inverse
--> 3 inverse_matrix([[4, 2], [14, 7]])
```

```
<ipython-input-10-1bc35c040c80> in inverse_matrix(matrix)
    47     d = matrix[1][1]
    48     if(a * d - b * c == 0):
--> 49         raise ValueError('The Matrix is not invertible')
    50     else:
    51         factor = 1 / (a * d - b * c)
```

```
ValueError: The Matrix is not invertible
```

```
In [13]: ### Run this line of code and see what happens. This is a 3x3 matrix
inverse_matrix([[4, 5, 1], [2, 9, 7], [6, 3, 9]])
```

```
-----
```

```
NotImplementedError Traceback (most recent call last)
```

```
<ipython-input-13-b67f26ad65f4> in <module>()
    1 ### Run this line of code and see what happens. This is a 3x3 matrix
--> 2 inverse_matrix([[4, 5, 1], [2, 9, 7], [6, 3, 9]])
```

```
<ipython-input-10-1bc35c040c80> in inverse_matrix(matrix)
    30
    31     if len(matrix) > 2:
--> 32         raise NotImplementedError('This functionality is not implemented, too large'
    33
    34
```

```
NotImplementedError: This functionality is not implemented, too large
```

```
In [ ]:
```

kalman_filter_demo

March 25, 2020

1 Kalman Filter and your Matrix Class

Once you have a working matrix class, you can use the class to run a Kalman filter!

You will need to put your matrix class into the workspace: * Click above on the "JUPYTER" logo. * Then open the matrix.py file, and copy in your code there. * Make sure to save the matrix.py file. * Then click again on the "JUPYTER" logo and open this file again.

You can also download this file kalman_filter_demo.ipynb and run the demo locally on your own computer.

Once you have our matrix class loaded, you are ready to go through the demo. Read through this file and run each cell one by one. You do not need to write any code in this Ipython notebook.

The demonstration has two different sections. The first section creates simulated data. The second section runs a Kalman filter on the data and visualizes the results.

1.0.1 Kalman Filters - Why are they useful?

Kalman filters are really good at taking noisy sensor data and smoothing out the data to make more accurate predictions. For autonomous vehicles, Kalman filters can be used in object tracking.

1.0.2 Kalman Filters and Sensors

Object tracking is often done with radar and lidar sensors placed around the vehicle. A radar sensor can directly measure the distance and velocity of objects moving around the vehicle. A lidar sensor only measures distance.

Put aside a Kalman filter for a minute and think about how you could use lidar data to track an object. Let's say there is a bicyclist riding around in front of you. You send out a lidar signal and receive the signal back. The lidar sensor tells you that the bicycle is 10 meters directly ahead of you but gives you no velocity information.

By the time your lidar device sends out another signal, maybe 0.05 seconds will have passed. But during those 0.05 seconds, your vehicle still needs to keep track of the bicycle. So your vehicle will predict where it thinks the bicycle will be. But your vehicle has no bicycle velocity information.

After 0.05 seconds, the lidar device sends out and receives another signal. This time, the bicycle is 9.95 meters ahead of you. Now you know that the bicycle is traveling -1 meter per second towards you. For the next 0.05 seconds, your vehicle will assume the bicycle is traveling -1 m/s towards you. Then another lidar signal goes out and comes back, and you can update the position and velocity again.

1.0.3 Sensor Noise

Unfortunately, lidar and radar signals are noisy. In other words, they are somewhat inaccurate. A Kalman filter helps to smooth out the noise so that you get a better fix on the bicycle's true position and velocity.

A Kalman filter does this by weighing the uncertainty in your belief about the location versus the uncertainty in the lidar or radar measurement. If your belief is very uncertain, the Kalman filter gives more weight to the sensor. If the sensor measurement has more uncertainty, your belief about the location gets more weight than the sensor measurement.

2 Part 1 - Generate Data

The next few cells in the Ipython notebook generate simulation data. Imagine you are in a vehicle and tracking another car in front of you. All of the data you track will be relative to your position.

In this simulation, you are on a one-dimensional road where the car you are tracking can only move forwards or backwards. For this simulated data, the tracked vehicle starts 5 meters ahead of you traveling at 100 km/h. The vehicle is accelerating at -10 m/s^2 . In other words, the vehicle is slowing down.

Once the vehicle stops at 0 km/h, the car stays idle for 5 seconds. Then the vehicle continues accelerating towards you until the vehicle is traveling at -10 km/h. The vehicle travels at -10 km/h for 5 seconds. Don't worry too much about the trajectory of the other vehicle; this will be displayed for you in a visualization.

You have a single lidar sensor on your vehicle that is tracking the other car. The lidar sensor takes a measurement once every 50 milliseconds.

Run the code cell below to start the simulator and collect data about the tracked car. Notice the line `import matrix as m`, which imports your matrix code from the final project. You will not see any output yet when running this cell.

```
In [1]: %matplotlib inline

import pandas as pd
import math
import matplotlib.pyplot as plt
import matplotlib
import datagenerator
import matrix as m

matplotlib.rcParams.update({'font.size': 16})

# data_groundtruth() has the following inputs:
# Generates Data
# Input variables are:
# initial position meters
# initial velocity km/h
# final velocity (should be a negative number) km/h
# acceleration (should be a negative number) m/s^2
# how long the vehicle should idle
# how long the vehicle should drive in reverse at constant velocity
```

```

# time between lidar measurements in milliseconds

time_groundtruth, distance_groundtruth, velocity_groundtruth, acceleration_groundtruth = np.loadtxt('lidar_simulated_data.txt', delimiter=',', skiprows=1)

data_groundtruth = pd.DataFrame(
    {'time': time_groundtruth,
     'distance': distance_groundtruth,
     'velocity': velocity_groundtruth,
     'acceleration': acceleration_groundtruth
})

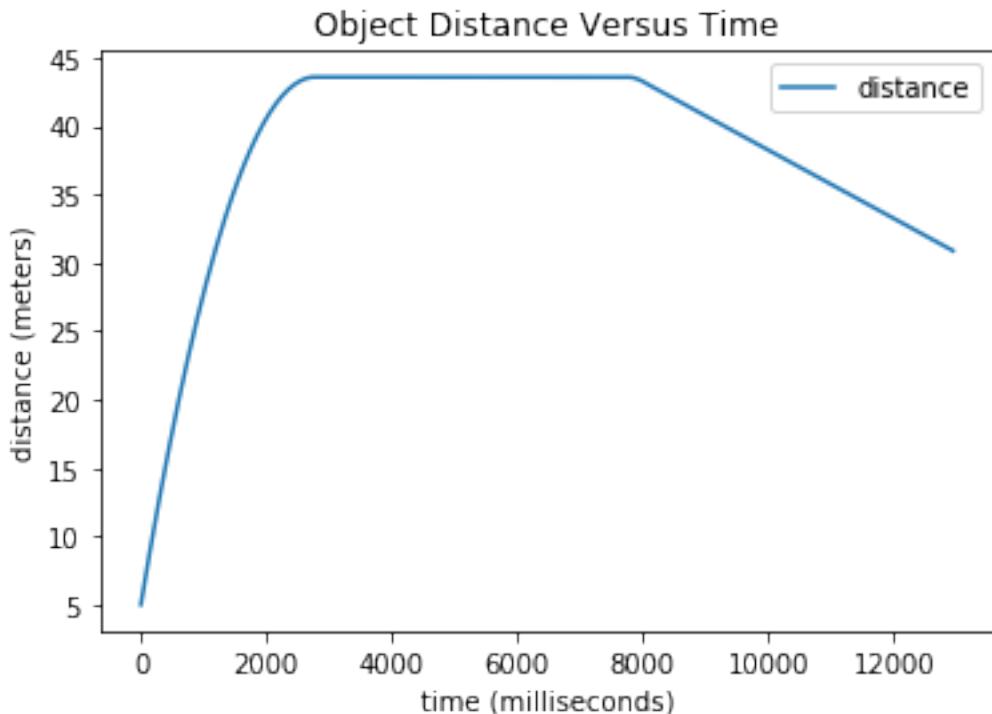
```

2.0.1 Visualizing the Tracked Object Distance

The next cell visualizes the simulating data. The first visualization shows the object distance over time. You can see that the car is moving forward although decelerating. Then the car stops for 5 seconds and then drives backwards for 5 seconds.

```
In [2]: ax1 = data_groundtruth.plot(kind='line', x='time', y='distance', title='Object Distance')
ax1.set(xlabel='time (milliseconds)', ylabel='distance (meters)')
```

```
Out[2]: [Text(0,0.5,'distance (meters)'), Text(0.5,0,'time (milliseconds)')]
```

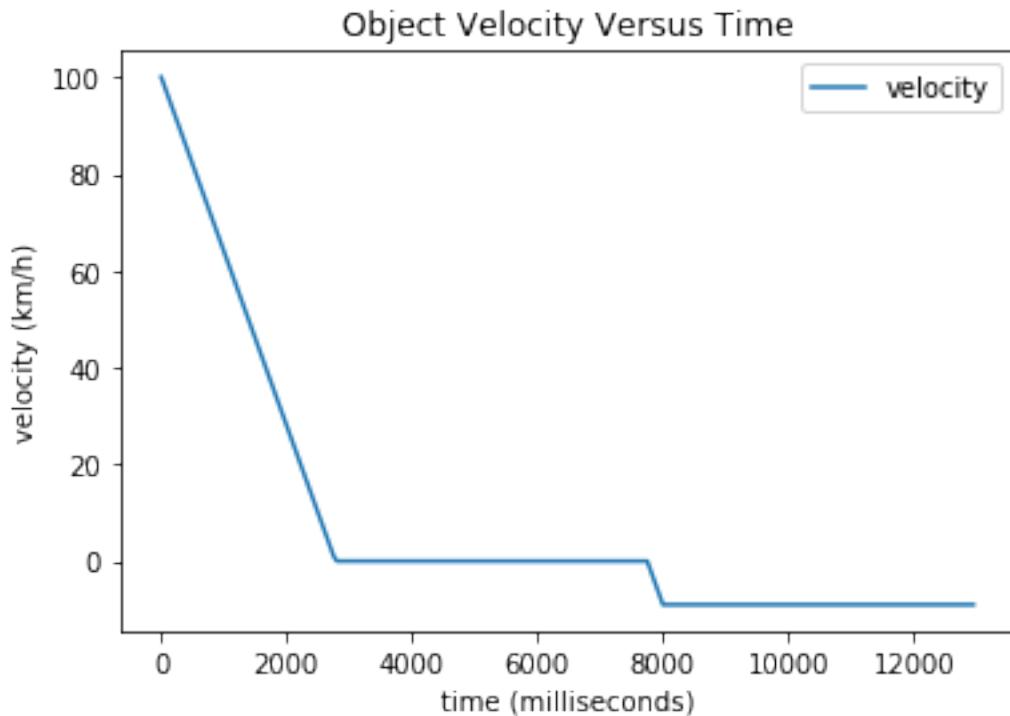


2.0.2 Visualizing Velocity Over Time

The next cell outputs a visualization of the velocity over time. The tracked car starts at 100 km/h and decelerates to 0 km/h. Then the car idles and eventually starts to decelerate again until reaching -10 km/h.

```
In [4]: ax2 = data_groundtruth.plot(kind='line', x='time', y='velocity', title='Object Velocity')
ax2.set(xlabel='time (milliseconds)', ylabel='velocity (km/h)')
```

```
Out[4]: [Text(0,0.5,'velocity (km/h)'), Text(0.5,0,'time (milliseconds)')]
```

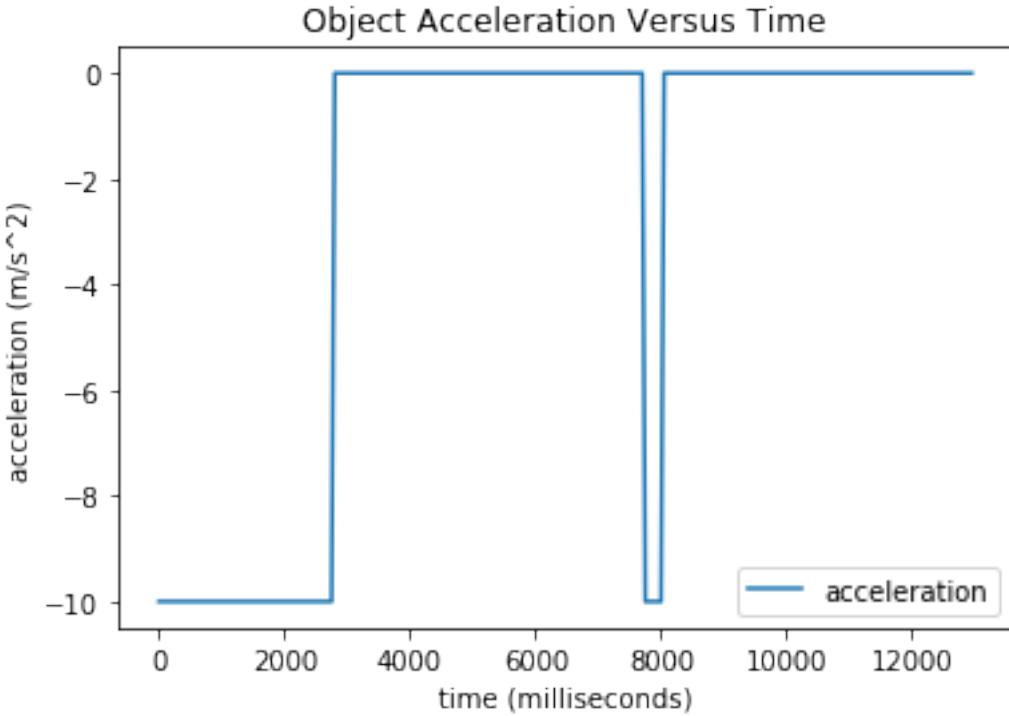


2.0.3 Visualizing Acceleration Over Time

This cell visualizes the tracked cars acceleration. The vehicle decelerates at 10 m/s^2 . Then the vehicle stops for 5 seconds and briefly accelerates again.

```
In [5]: data_groundtruth['acceleration'] = data_groundtruth['acceleration'] * 1000 / math.pow(60,2)
ax3 = data_groundtruth.plot(kind='line', x='time', y='acceleration', title='Object Acceleration')
ax3.set(xlabel='time (milliseconds)', ylabel='acceleration (m/s^2)')
```

```
Out[5]: [Text(0,0.5,'acceleration (m/s^2)'), Text(0.5,0,'time (milliseconds)')]
```



2.0.4 Simulate Lidar Data

The following code cell creates simulated lidar data. Lidar data is noisy, so the simulator takes ground truth measurements every 0.05 seconds and then adds random noise.

```
In [6]: # make lidar measurements
lidar_standard_deviation = 0.15
lidar_measurements = datagenerator.generate_lidar(distance_groundtruth, lidar_standard_deviation)
lidar_time = time_groundtruth
```

2.0.5 Visualize Lidar Measurements

Run the following cell to visualize the lidar measurements versus the ground truth. The ground truth is shown in red, and you can see that the lidar measurements are a bit noisy.

```
In [7]: data_lidar = pd.DataFrame(
    {'time': time_groundtruth,
     'distance': distance_groundtruth,
     'lidar': lidar_measurements
    })

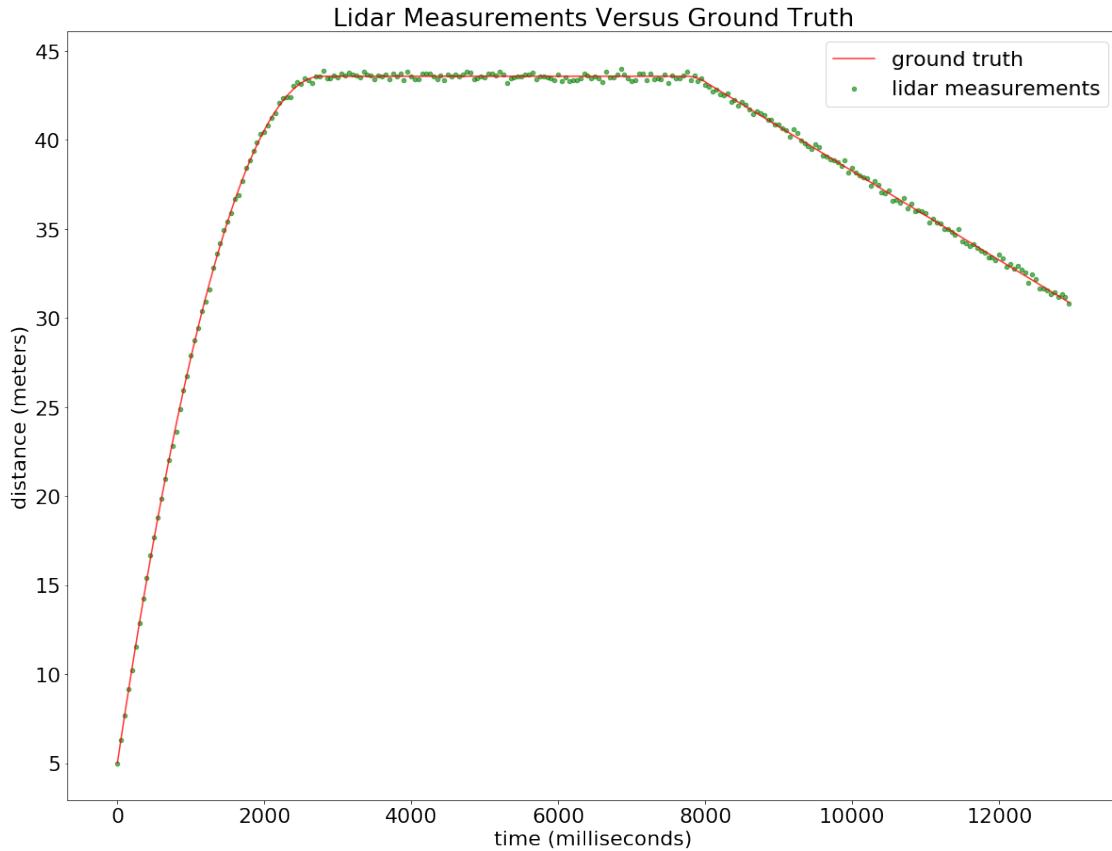
matplotlib.rcParams.update({'font.size': 22})

ax4 = data_lidar.plot(kind='line', x='time', y='distance', label='ground truth', figsize=(12, 8))
```

```

title = 'Lidar Measurements Versus Ground Truth', color='red')
ax5 = data_lidar.plot(kind='scatter', x ='time', y ='lidar', label='lidar measurements',
ax5.set(xlabel='time (milliseconds)', ylabel='distance (meters)')
plt.show()

```



3 Part 2 - Using a Kalman Filter

The next part of the demonstration will use your matrix class to run a Kalman filter. This first cell initializes variables and defines a few functions.

The following cell runs the Kalman filter using the lidar data.

In [8]: # Kalman Filter Initialization

```

initial_distance = 0
initial_velocity = 0

x_initial = m.Matrix([[initial_distance], [initial_velocity * 1e-3 / (60 * 60)]])
P_initial = m.Matrix([[5, 0],[0, 5]])

acceleration_variance = 50

```

```

lidar_variance = math.pow(lidar_standard_deviation, 2)

H = m.Matrix([[1, 0]])
R = m.Matrix([[lidar_variance]])
I = m.identity(2)

def F_matrix(delta_t):
    return m.Matrix([[1, delta_t], [0, 1]])

def Q_matrix(delta_t, variance):
    t4 = math.pow(delta_t, 4)
    t3 = math.pow(delta_t, 3)
    t2 = math.pow(delta_t, 2)

    return variance * m.Matrix([(1/4)*t4, (1/2)*t3], [(1/2)*t3, t2]]))


```

3.0.1 Run the Kalman filter

The next code cell runs the Kalman filter. In this demonstration, the prediction step starts with the second lidar measurement. When the first lidar signal arrives, there is no previous lidar measurement with which to calculate velocity. In other words, the Kalman filter predicts where the vehicle is going to be, but it can't make a prediction until time has passed between the first and second lidar reading.

The Kalman filter has two steps: a prediction step and an update step. In the prediction step, the filter uses a motion model to figure out where the object has traveled in between sensor measurements. The update step uses the sensor measurement to adjust the belief about where the object is.

In [9]: # Kalman Filter Implementation

```

x = x_initial
P = P_initial

x_result = []
time_result = []
v_result = []

for i in range(len(lidar_measurements) - 1):

    # calculate time that has passed between lidar measurements
    delta_t = (lidar_time[i + 1] - lidar_time[i]) / 1000.0

    # Prediction Step - estimates how far the object traveled during the time interval
    F = F_matrix(delta_t)
    Q = Q_matrix(delta_t, acceleration_variance)

    x_prime = F * x

```

```

P_prime = F * P * F.T() + Q

# Measurement Update Step - updates belief based on lidar measurement
y = m.Matrix([[lidar_measurements[i + 1]]]) - H * x_prime
S = H * P_prime * H.T() + R
K = P_prime * H.T() * S.inverse()
x = x_prime + K * y
P = (I - K * H) * P_prime

# Store distance and velocity belief and current time
x_result.append(x[0][0])
v_result.append(3600.0/1000 * x[1][0])
time_result.append(lidar_time[i+1])

result = pd.DataFrame(
    {'time': time_result,
     'distance': x_result,
     'velocity': v_result
})

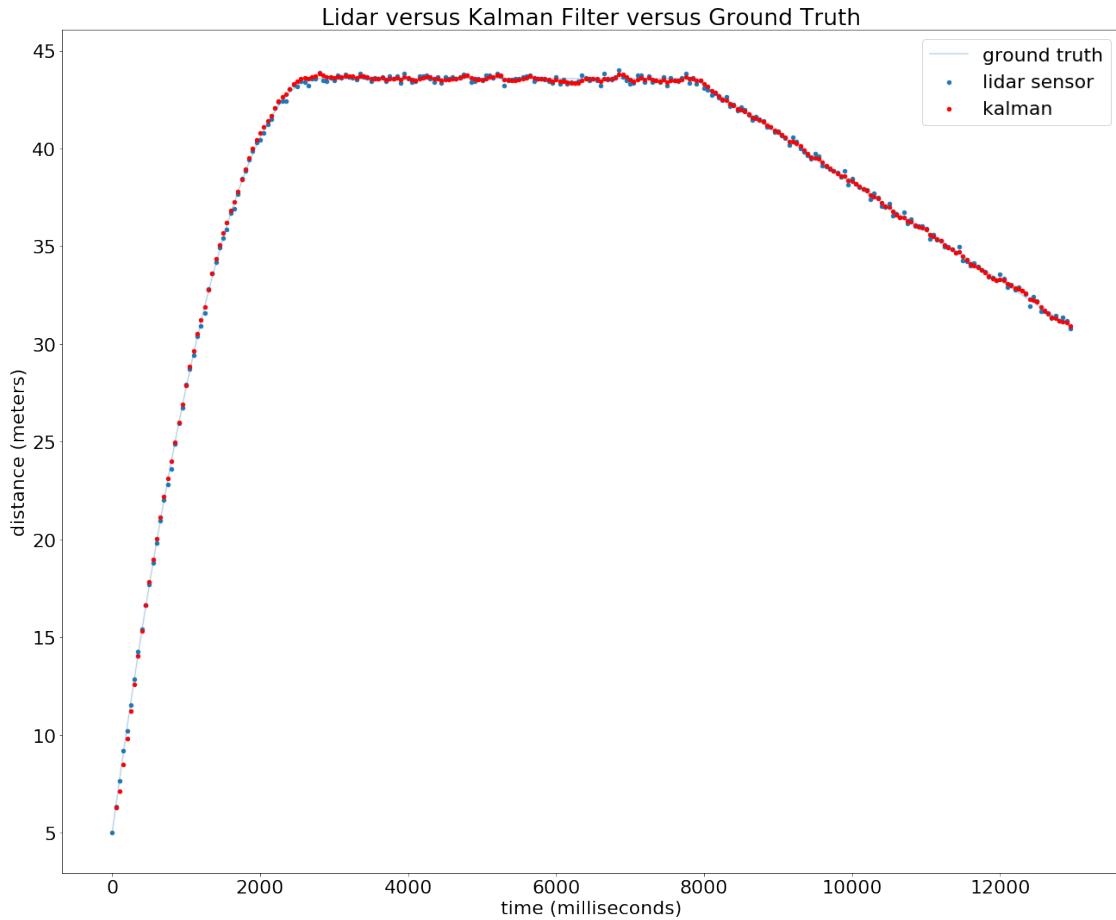
```

3.0.2 Visualize the Results

The following code cell outputs a visualization of the Kalman filter. The chart contains ground truth, the lidar measurements, and the Kalman filter belief. Notice that the Kalman filter tends to smooth out the information obtained from the lidar measurement.

It turns out that using multiple sensors like radar and lidar at the same time, will give even better results. Using more than one type of sensor at once is called sensor fusion, which you will learn about in the Self-Driving Car Engineer Nanodegree

```
In [10]: ax6 = data_lidar.plot(kind='line', x='time', y='distance', label='ground truth', figsi
ax7 = data_lidar.plot(kind='scatter', x='time', y='lidar', label='lidar sensor', ax=ax6)
ax8 = result.plot(kind='scatter', x='time', y='distance', label='kalman', ax=ax7, c
ax8.set(xlabel='time (milliseconds)', ylabel='distance (meters)')
plt.show()
```

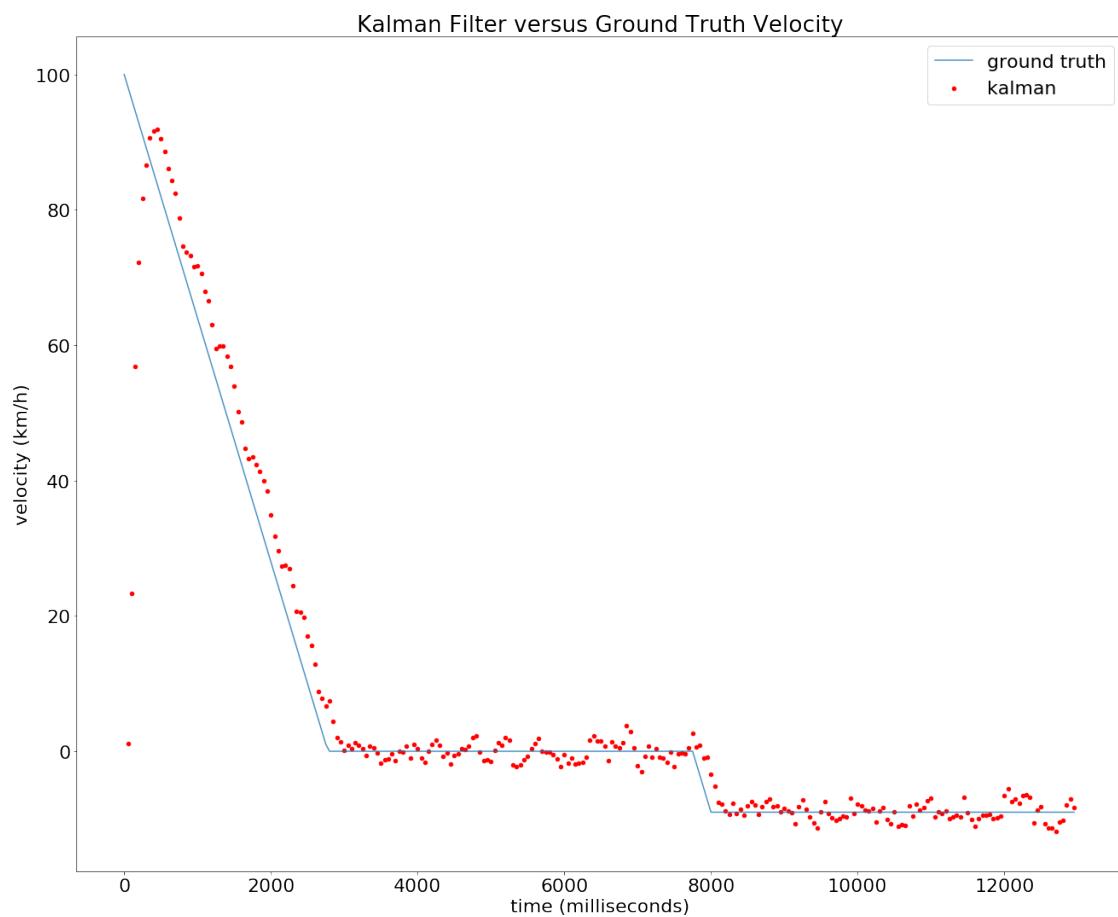


3.0.3 Visualize the Velocity

One of the most interesting benefits of Kalman filters is that they can give you insights into variables that you cannot directly measured. Although lidar does not directly give velocity information, the Kalman filter can infer velocity from the lidar measurements.

This visualization shows the Kalman filter velocity estimation versus the ground truth. The motion model used in this Kalman filter is relatively simple; it assumes velocity is constant and that acceleration a random noise. You can see that this motion model might be too simplistic because the Kalman filter has trouble predicting velocity as the object decelerates.

```
In [11]: ax1 = data_groundtruth.plot(kind='line', x='time', y='velocity', label='ground truth',
    ax2 = result.plot(kind='scatter', x = 'time', y = 'velocity', label='kalman', ax=ax1,
    ax2.set(xlabel='time (milliseconds)', ylabel='velocity (km/h)')
    plt.show()
```



In []:

Intro_to_dictionaries

March 28, 2020

1 Introduction to Dictionaries

In this notebook you will learn about Python Dictionaries.

1.1 Dictionaries associate keys with values

```
In [1]: ticket = {  
    "date" : "2018-12-28",  
    "priority" : "high",  
    "description" : """Vehicle did not slow down despite  
    SLOW  
    SCHOOL  
    ZONE"""}  
}
```

The `ticket` you see above is a Python dictionary. A dictionary is an **unordered** data structure. With a list, data is accessed by **position** (e.g. `my_list[0]`), but dictionaries are unordered. Data in a dictionary is accessed via it's **key** instead of it's position.

In the `ticket` example above, there are three **keys**. You can see what they are by running the code below

```
In [2]: print(ticket.keys())  
  
dict_keys(['date', 'priority', 'description'])
```

Each **key** in a dictionary is associated with a **value**. The code below retrieves the **value** associated with the **key** `description`.

```
In [3]: print(ticket['description'])  
  
Vehicle did not slow down despite  
    SLOW  
    SCHOOL  
    ZONE
```

1.2 Dictionaries are *mutable* (they can be modified)

Dictionaries are **mutable**, which means that they can be *mutated* (modified). Being mutable means:

1. Elements can be **added** to a dictionary.
2. Elements can be **removed** from a dictionary.
3. Elements can be **modified** in a dictionary.

The code below demonstrates how to add, remove, and modify elements in a dictionary.

```
In [4]: # Let's start with an empty dictionary. Eventually this will store  
# English to spanish translations...
```

```
eng_to_spa = {} # this creates an empty dictionary  
print(eng_to_spa)
```

```
{}
```

```
In [5]: # 1. Adding elements to a dictionary.  
#  
#     Elements can be added to a dictionary as follows:
```

```
eng_to_spa['blue'] = 'Azul'  
eng_to_spa['gren'] = 'verde'  
  
print(eng_to_spa)
```

```
{'blue': 'Azul', 'gren': 'verde'}
```

Oops! I misspelled "green". Let's remove that element from the dictionary...

```
In [6]: # 2. Removing elements from a dictionary  
#  
#     Elements can be removed from a dictionary using the del keyword  
  
del eng_to_spa['gren']  
  
print(eng_to_spa)
```

```
{'blue': 'Azul'}
```

It looks like "azul" is capitalized... let's change that!

```
In [7]: # 3. Modifying elements in a dictionary.  
#  
#     Modifying the value associated with a key works just  
#     like adding a new value...
```

```

eng_to_spa['blue'] = 'azul'

print(eng_to_spa)

{'blue': 'azul'}

```

1.2.1 TODO - Complete eng_to_spa

In the code cell below, add elements for a few additional colors so that eng_to_spa reflects the following:

English	Spanish
blue	azul
green	verde
pink	rosa
orange	naranja
gray	gris
brown	marron

```

In [8]: eng_to_spa['blue'] = 'azul'
        eng_to_spa['green'] = 'verde'
        eng_to_spa['pink'] = 'rosa'
        eng_to_spa['orange'] = 'naranja'
        eng_to_spa['gray'] = 'gris'
        eng_to_spa['brown'] = 'marron'

# YOUR CODE HERE - complete the eng_to_spa
#   dictionary so it contains all the information
#   shown in the table above.

# Testing code below
assert(eng_to_spa['blue'] == 'azul')
assert(eng_to_spa['gray'] == 'gris')
assert(eng_to_spa['orange'] == 'naranja')
assert(eng_to_spa['pink'] == 'rosa')
print("Your english to spanish dictionary is looking good!")

```

Your english to spanish dictionary is looking good!

1.3 Dictionaries are *unordered*

What does it mean for a dictionary to be **unordered**? Consider the following two tickets.

```

In [9]: dictionary_ticket_1 = {
            "date" : "2018-12-31",

```

```

    "priority" : "high",
    "description": "Vehicle made unexpected stop."
}

dictionary_ticket_2 = {
    "priority" : "high",
    "description": "Vehicle made unexpected stop.",
    "date"      : "2018-12-31"
}

# these dictionaries contain the same information, but the
# ordering looks different. Does Python consider them identical?

print("Does dictionary_ticket_1 == dictionary_ticket_2?")
print(dictionary_ticket_1 == dictionary_ticket_2)

Does dictionary_ticket_1 == dictionary_ticket_2?
True

```

1.3.1 Looping through a dictionary

Despite not having a well-defined ordering of its elements, you can still loop through the **keys** of a dictionary...

```
In [10]: # demonstration of dictionary looping
          # demo 1 - keys only

        for key in dictionary_ticket_1:
            print(key)

date
priority
description
```

```
In [11]: # demonstration of dictionary looping
          # demo 2 - keys and values

        for key in dictionary_ticket_1:
            value = dictionary_ticket_1[key]
            print(key, ':', value)

date : 2018-12-31
priority : high
description : Vehicle made unexpected stop.
```

1.4 TODO - Exercise: "Reverse" a dictionary

The code below re-defines the `eng_to_spa` dictionary. Your job is to write a function called `reverse_dictionary` which takes, for example, an english-to-spanish dictionary and returns a spanish-to-english dictionary.

```
In [1]: def reverse_dictionary(dictionary):
    new_d = {}
    for key in dictionary:
        new_d[dictionary[key]] = key

    # TODO - your code here
    return new_d

eng_to_spa = {
    "red" : "rojo",
    "blue" : "azul",
    "green" : "verde",
    "black" : "negro",
    "white" : "blanco",
    "yellow" : "amarillo",
    "orange" : "naranja",
    "pink" : "rosa",
    "purple" : "morado",
    "gray" : "gris"
}

# TESTING CODE

spa_to_eng = reverse_dictionary(eng_to_spa)

assert(len(spa_to_eng) == len(eng_to_spa))
assert(spa_to_eng['rojo'] == 'red')
assert(spa_to_eng['azul'] == 'blue')
assert(spa_to_eng['verde'] == 'green')

print("Nice work! Your reverse_dictionary function is correct.")
```

Nice work! Your `reverse_dictionary` function is correct.

1.5 Spanish to French

In this exercise you will take two dictionaries (one spanish-to-english and one french-to-english) and combine them to make a single spanish-to-french dictionary.

First, let's take a look at the two existing dictionaries.

```
In [2]: eng_to_spa = {
    "red" : "rojo",
```

```

        "blue"    : "azul",
        "green"   : "verde",
        "black"   : "negro",
        "white"   : "blanco",
        "yellow"  : "amarillo",
        "orange"  : "naranja",
        "pink"    : "rosa",
        "purple"  : "morado",
        "gray"    : "gris"
    }

french_to_eng = {
    "bleu"    : "blue",
    "noir"   : "black",
    "vert"    : "green",
    "violet" : "purple",
    "gris"   : "gray",
    "rouge"  : "red",
    "orange" : "orange",
    "rose"   : "pink",
    "marron" : "brown",
    "jaune"  : "yellow",
    "blanc"  : "white",
}

```

```

print("there are    ", len(eng_to_spa), "colors in eng_to_spa")
print("but there are", len(french_to_eng), "colors in french_to_eng")

```

there are 10 colors in eng_to_spa
but there are 11 colors in french_to_eng

In [3]: # don't forget you have the "reverse_dictionary" function!

```

english_to_french = reverse_dictionary(french_to_eng)

for english_word in english_to_french:
    french_word = english_to_french[english_word]
    print("The french word for", english_word, "is", french_word)

```

The french word for blue is bleu
The french word for black is noir
The french word for green is vert
The french word for purple is violet
The french word for gray is gris
The french word for red is rouge
The french word for orange is orange
The french word for pink is rose

```
The french word for brown is marron  
The french word for yellow is jaune  
The french word for white is blanc
```

```
In [10]: def spanish_to_french(english_to_spanish, english_to_french):  
    """  
        Given an English to Spanish dictionary and an English  
        to French dictionary, returns a Spanish to French dictionary.  
    """  
    If any words appear in one dictionary but NOT the other,  
    they should not be included in the resulting dictionary.  
    """  
    s2f = {}  
    #  
    # TODO - your code here  
    #  
    spa_to_eng = reverse_dictionary(eng_to_spa)  
    for english_word in english_to_french:  
        if english_word in english_to_spanish:  
            french_word = english_to_french[english_word]  
            spanish_word = english_to_spanish[english_word]  
            s2f[spanish_word] = french_word  
    return s2f  
  
# TESTING CODE  
S2F = spanish_to_french(eng_to_spa, english_to_french)  
  
assert(S2F["rojo"] == "rouge")  
assert(S2F["morado"] == "violet")  
  
print("Nice work! Your spanish to french function works correctly!")
```

```
Nice work! Your spanish to french function works correctly!
```

1.5.1 My Solution

Make sure you've attempted to solve the problem above before continuing!

When I tested my first solution, I ran into an error. Try running my first attempt (below) to see what it was

```
In [9]: def spanish_to_french_1(english_to_spanish, english_to_french):  
    s2f = {}  
    for english_word in english_to_french:  
        french_word = english_to_french[english_word]  
        spanish_word = english_to_spanish[english_word]  
        s2f[spanish_word] = french_word  
    return s2f
```

```
S2F_1 = spanish_to_french_1(eng_to_spa, english_to_french)
```

```
-----
KeyError                                         Traceback (most recent call last)

<ipython-input-9-37fb66b1d546> in <module>()
    7     return s2f
    8
----> 9 S2F_1 = spanish_to_french_1(eng_to_spa, english_to_french)

<ipython-input-9-37fb66b1d546> in spanish_to_french_1(english_to_spanish, english_to_fre
    3     for english_word in english_to_french:
    4         french_word = english_to_french[english_word]
----> 5         spanish_word = english_to_spanish[english_word]
    6         s2f[spanish_word] = french_word
    7     return s2f

KeyError: 'brown'
```

I keep getting the following message:

```
KeyError: 'brown'
```

That's because "brown" isn't in my Spanish to English dictionary!

That's okay, I can just do a **membership check** on each key. Compare the code above to the code below.

```
In [ ]: def spanish_to_french_2(english_to_spanish, english_to_french):
    s2f = {}
    for english_word in english_to_french:
        if english_word in english_to_spanish:
            french_word = english_to_french[english_word]
            spanish_word = english_to_spanish[english_word]
            s2f[spanish_word] = french_word
    return s2f

S2F_2 = spanish_to_french_2(eng_to_spa, english_to_french)
```

1.5.2 A note on the `in` keyword (and "testing for membership")

Pay attention to the if statement in the code above. When we write

```
if X in Y
```

we are "testing for membership". When Y is a dictionary we are testing to see if X is a **key** in that dictionary. If it is, the test returns True. Otherwise it returns False.

```
In [13]: capitals = {  
    "spain" : "madrid",  
    "france" : "paris",  
    "china" : "beijing"  
}  
  
print("Is 'china' a key in this dictionary?")  
print("china" in capitals)  
print()  
  
print("Is 'paris' a key in this dictionary?")  
print("paris" in capitals)
```

```
Is 'china' a key in this dictionary?  
True
```

```
Is 'paris' a key in this dictionary?  
False
```

1.6 Dictionary Summary

Important things to remember about dictionaries!

1. **Dictionaries associate keys with values** The following dictionary has 2 keys ("abc" and "def") and 2 values (123 and 456)

```
d = {"abc":123, "def":456}
```

2. **Dictionaries are Unordered** Two dictionaries are identical if they have the same keys **and** those keys are associated with the same values. Order doesn't matter.

```
> d1 = {"abc":123, "def":456}  
> d2 = {"def":456, "abc":123}  
> d1 == d2  
True
```

3. **Dictionaries are "mutable"** This means they can be modified in various ways

3.1 adding new elements

```
> d = {}  
> d['k'] = 'v'  
> print(d)  
{'k': 'v'}
```

3.2 removing elements

```
> del d['k']
> print(d)
{}
```

3.3 changing elements

```
> d['key'] = 'value' # first need to add an element back in
> print(d)
{'key': 'value'}

> d['key'] = 'other value'
> print(d)
{'key' : 'other value'}
```

4. Looping through a dictionary When looping through a dictionary, you loop through the **keys** of that dictionary.

5. Membership Testing Python's `in` keyword can be used to test if something is a **key** in a dictionary.

Introduction to Dictionaries 2

Dictionaries can be especially useful when used to represent larger chunks of related data. The image below shows a more realistic "ticket" from the ticketing software known as Jira.

Take a look!

The screenshot shows a Jira ticket page with the following details:

Project: iSDCND / CARND-273
Title: This is a Test

Buttons: Edit, Comment, Assign, To Do, In Progress, Done, Attach, Download, More

Details:

- Type: Bug (selected)
- Status: DONE
- Priority: Medium
- Resolution: Done
- Labels: bug, python

People:

- Assignee: Unassigned
- Reporter: Andy Brown
- Votes: 0
- Watchers: 1 (Stop watching this issue)

Description: Testing 123

Dates:

- Created: 6 days ago
- Updated: 6 days ago
- Resolved: 6 days ago

Attachments: Drop files to attach, or browse.

In [1]: # Note how the structure of a ticket is captured in the dictionary

```
ticket = {
    "type" : "bug",
    "status": "done",
    "priority": "medium",
    "resolution": "done",
    "description" : "testing 123",
    "attachments" : [],
    "people": {
        "assignee" : None,
        "reporter" : {
            "name" : "Andy Brown",
            "image": "www.example_image_url.com"
        },
        "votes" : 0,
        "watchers" : [
            {
                "name": "Andy Brown",
                "image": "www.example_image_url.com"
            }
        ]
    },
    "dates" : {
        "created" : "6 days ago",
        "updated" : "6 days ago",
        "resolved": "6 days ago"
    }
}

# In this example, ticket is a dictionary with the following "keys":
print("The keys for this dictionary are...")
ticket.keys()
```

The keys for this dictionary are...

Out[1]: dict_keys(['type', 'status', 'priority', 'resolution', 'description', 'attachments', 'people', 'dates'])

In [2]: # notice how nicely the following code reads...

```
ticket['people']['reporter']['name']
```

Out[2]: 'Andy Brown'

In [3]: # Let's pull in a bunch of "dummy" data

```
from sample_data import big_tickets
import random

print("there are", len(big_tickets), "big tickets")
```

there are 250 big tickets

In [4]: # grab a random ticket and take a look at it

```
random_ticket = random.choice(bug_tickets)
random_ticket
```

Out[4]: {
 'type': 'bug',
 'status': 'done',
 'priority': 'medium',
 'resolution': 'done',
 'description': 'testing 123',
 'attachments': [],
 'people': {'assignee': None,
 'reporter': {'name': 'Kagure Kabue', 'image': 'www.example_image_url.com'},
 'votes': 0},
 'watchers': [{'name': 'Cezanne Camacho',
 'image': 'www.example_image_url.com'},
 {'name': 'Jonathan Sullivan', 'image': 'www.example_image_url.com'},
 {'name': 'Sebastian Thrun', 'image': 'www.example_image_url.com'}]},
 'dates': {'created': '6 days ago',
 'updated': '6 days ago',
 'resolved': '6 days ago'}}

TODO - Exercise - Find all tickets with 8 or more watchers

Lists and dictionaries can work together nicely. Notice how in the random ticket shown above the `people` field is a dictionary. That dictionary contains additional data including a `watchers` key whose associated value is a list.

In this exercise you will filter a list of 250 tickets (which are each a dictionary) down to a smaller list of only the most popular tickets (as defined by number of watchers).

```
In [10]: def get_popular_tickets(tickets):
    """
        From a list of tickets, fetch all the tickets with 8 or
        more "watchers".
    """
    popular_tickets = []
    #
    # TODO - your code here
    #
    for ticket in tickets:
        if len(ticket['people']['watchers']) >= 8:
            popular_tickets.append(ticket)

    return popular_tickets

popular = get_popular_tickets(big_tickets)

# TESTING CODE
assert( len(popular) > 0 ) # must be at least one popular ticket

for ticket in popular:
    assert( len(ticket['people']['watchers']) >= 8 )

print("Nice job! It looks like your function is working.")
```

Nice job! It looks like your function is working.

Intro to Sets

March 28, 2020

1 Introduction to Sets

A set in Python is a collection of **unique** and **immutable** (unchangeable) objects.

To get a feel for how a set works, take a look at the following code:

In [1]: # start with an empty set

```
my_set = set()  
  
print(my_set)  
  
set()
```

In [2]: # add a few elements

```
my_set.add('a')  
my_set.add('b')  
my_set.add('c')  
my_set.add(1)  
my_set.add(2)  
my_set.add(3)  
  
print(my_set)
```

{1, 2, 3, 'b', 'a', 'c'}

In [3]: # like a dictionary, a set is UNORDERED.

We can still loop through a set though.

```
for element in my_set:  
    print(element)
```

1
2
3
b

```
a  
c
```

```
In [4]: # let's see how many elements are in this set...
```

```
print("there are", len(my_set), "elements in my_set")
```

```
there are 6 elements in my_set
```

```
In [5]: # can we make the set bigger by adding more "copies"  
# of existing elements?
```

```
my_set.add("a")  
my_set.add("a")  
my_set.add("a")  
my_set.add("a")  
my_set.add("a")  
my_set.add("a")  
my_set.add("a")  
my_set.add("a")
```

```
print("there are", len(my_set), "elements in my_set")
```

```
there are 6 elements in my_set
```

```
In [6]: # there are still only 6 elements...
```

```
#  
# that's because sets only care about UNIQUE elements.  
# They do not allow for multiple "copies"
```

```
print(my_set)
```

```
{1, 2, 3, 'b', 'a', 'c'}
```

```
In [7]: # and they haven't changed. What if we remove "a"
```

```
my_set.remove("a")  
print("there are", len(my_set), "elements in my_set")  
print(my_set)
```

```
there are 5 elements in my_set
```

```
{1, 2, 3, 'b', 'c'}
```

lists_timing_and_performance

March 29, 2020

```
In [1]: import time
        import random
        from matplotlib import pyplot as plt
```

1 Lists, Timing, and Performance

In this notebook we're going to explore the performance of lists. Specifically, we're going to see how the time it takes to perform a **membership check** on a list is affected by various properties of the list. A "membership check" is what you do when you write code like:

```
my_list = [1,2,3]
if 3 in my_list:
    # we just checked my_list for membership
    # of the element 3
```

Let's explore the following two questions in code:

1. When an element IS in a list, does the location of that element (near the beginning vs near the end) impact the time it takes to perform a membership check?
2. When an element IS NOT in a list, does the size of the list impact the time it takes to perform a membership check?

What we find will motivate a more in depth discussion about the tradeoffs between lists and other data structures.

```
In [2]: # before we continue, make sure you understand
        # what we mean by "testing for membership" in a list.
```

```
L = [1,2,3]
2 in L
```

```
Out[2]: True
```

```
In [3]: L = [1,2,3]
4 in L
```

```
Out[3]: False
```

1.1 Question 1

Does position in list impact the time it takes to perform a membership test?

First we're going to need to figure out how to do these timings...

```
In [4]: # let's make a small list to begin with
L = list(range(10))
print(L)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [6]: # now let's time how long it takes to check for
# membership of the very first element (0)...

element = 0

start_time = time.clock()

element in L      # this line of code looks weird
                  # but it's valid python!

end_time = time.clock()

# make sure you understand why the following code makes sense.
duration_in_seconds = end_time - start_time
milliseconds = duration_in_seconds * 1000

print(milliseconds, "milliseconds to find 0 in list")

0.04999999999988347 milliseconds to find 0 in list
```

```
In [7]: # Run the above code a few times. You should notice that the time
# can change quite a bit. What we actually want is to conduct
# this experiment a bunch of times and find the average.
```

```
L = list(range(1000)) # use a bigger list
element = 500          # look in the middle of the list
num_trials = 1000       # perform experiment many times

start = time.clock()
for _ in range(num_trials):
    element in L
end = time.clock()
secs = end-start
millis = secs * 1000
millis_per_check = millis/num_trials
print("on average, it took", millis_per_check, "ms per membership test")
```

```
on average, it took 0.008938000000000113 ms per membership test
```

```
In [8]: # That's a useful bit of code!
# Let's generalize it and turn it into a reusable function
```

```
def avg_millis_to_check_el_in_list(element, target_list, N=20):
    start = time.clock()
    for _ in range(N):
        element in target_list
    end = time.clock()
    return (end-start)*1000 / N

avg_millis = avg_millis_to_check_el_in_list(500, list(range(1000)))
print("on average, it took", avg_millis, "ms per membership test")
```

```
on average, it took 0.00805000000003887 ms per membership test
```

1.1.1 Note - Take your time!

Make sure you take your time going through the next few cells. Try to really read through the code **before** you run it and try to make a prediction about what will happen...

```
In [9]: # now we can compare time to lookup low numbers (near the
# beginning of the list) vs higher numbers (near the end).
```

```
# Let's use a really big list this time
list_size = 1000000
L = list(range(list_size))

# Now make three separate timings...
T_beginning = avg_millis_to_check_el_in_list(1000, L)
T_middle     = avg_millis_to_check_el_in_list(500000, L)
T_end        = avg_millis_to_check_el_in_list(999999, L)

print("T_beginning: ", T_beginning)
print("T_middle:     ", T_middle)
print("T_end:        ", T_end)

T_beginning:  0.015500000000001624
T_middle:      7.996500000000006
T_end:         16.1915
```

Interesting! There's clearly a relationship. Let's see if we can dig deeper though. A [scatter plot](#) showing average time vs position in list might help us out here.

```
In [10]: # Making a scatter plot of position in list (X-axis)
          # vs. average time to find element (y-axis)

list_size = 100000
L = list(range(list_size))

# check between start and end in increments of 10000. This will
# be our X axis too!
positions = list(range(0, list_size, 10000))

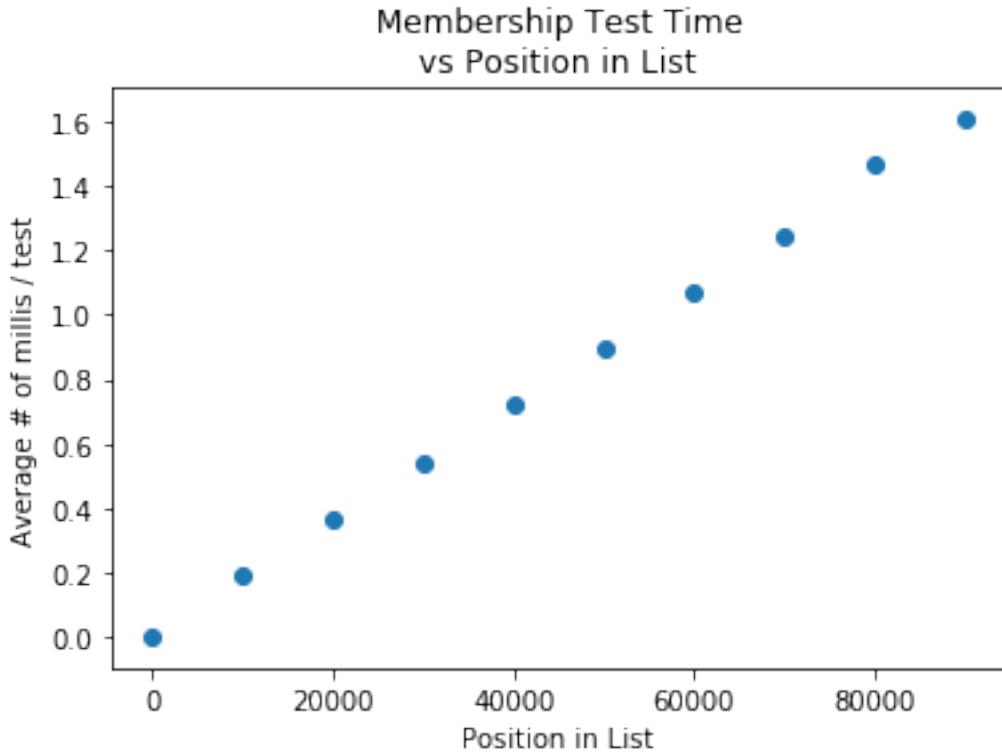
# use list comprehension to generate Y-axis data!
millis = [avg_millis_to_check_el_in_list(pos, L) for pos in positions]

# first, let's look at the raw data
print("positions checked:", positions)
print("average millis:    ", millis)
```

```
positions checked: [0, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000]
average millis:    [0.00029999999999752447, 0.1943499999999987, 0.36665000000000017, 0.53790000000
```

```
In [11]: # now let's make the scatter plot!
```

```
X = positions
Y = millis
plt.scatter(X, Y)
plt.title("Membership Test Time\nvs Position in List")
plt.xlabel("Position in List")
plt.ylabel("Average # of millis / test")
plt.show()
```



1.1.2 Answer to Question 1

Yes! Where an element is in a list **definitely** impacts how long it takes to discover that the element exists in the list!

Elements near the beginning of a list are found very quickly. Elements near the end of the list take longer.

1.2 Question 2

Does the size of a list impact the time it takes to test for membership of elements when they are NOT in the list?

In [12]: # Let's jump right into writing a function

```
def avg_millis_to_test_for_non_existent_el(list_size, num_trials=20):
    # 1. prepare list and nonexistent element
    L = list(range(list_size))
    element = -1

    # 2. start the timer
    start = time.clock()
```

```

# 3. repeat membership test num_trials times
for _ in range(num_trials):
    element in L

# 4. stop the timer
end = time.clock()

# 5. do the math and return the result
millis_per_test = (end-start) * 1000 / num_trials
return millis_per_test

```

In [13]: # Let's use this function on lists of different sizes

```

small   = 10000
medium  = 100000
large   = 1000000

T_small  = avg_millis_to_test_for_non_existent_el(small)
T_medium = avg_millis_to_test_for_non_existent_el(medium)
T_large  = avg_millis_to_test_for_non_existent_el(large)

print("T_small: ", T_small)
print("T_medium: ", T_medium)
print("T_large: ", T_large)

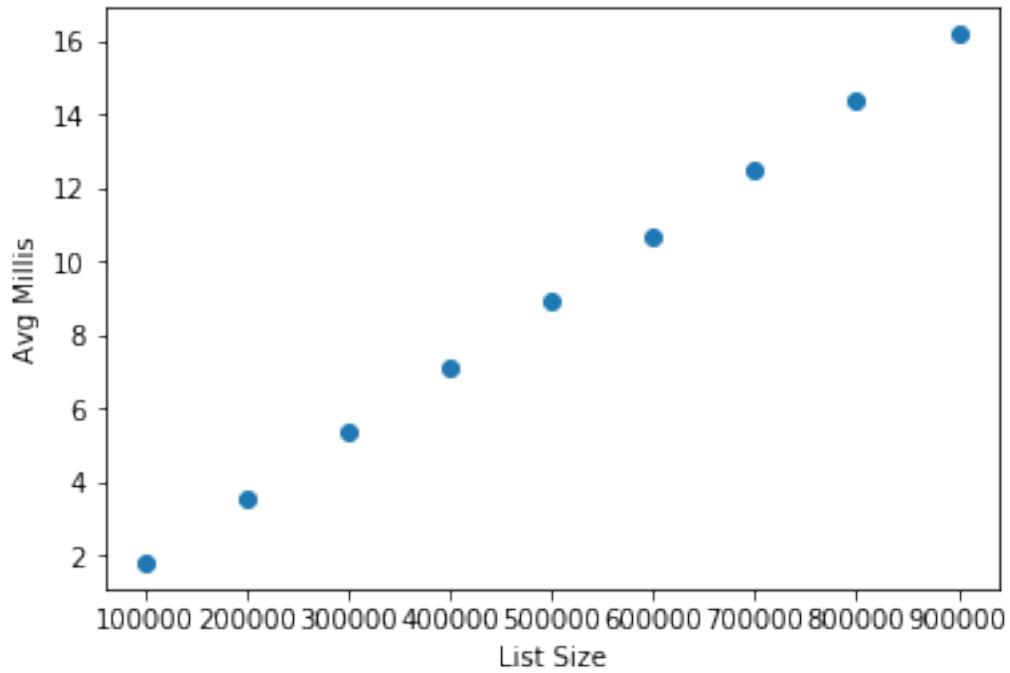
T_small:  0.176299999999959
T_medium: 1.5359500000000192
T_large:  15.604249999999984

```

Once again, we see that there IS a relationship...

Let's make another scatter plot.

In [14]: sizes = list(range(100000, 1000000, 100000))
times = [avg_millis_to_test_for_non_existent_el(s) for s in sizes]
plt.scatter(sizes, times)
plt.xlabel("List Size")
plt.ylabel("Avg Millis")
plt.show()



1.2.1 Answer to Question 2

Yes! When checking for membership of an element in a list, it takes longer to figure out an element **doesn't** exist in that list when the list is big.

1.2.2 Next Steps

Either continue on or keep exploring more about lists and timing. If you do keep exploring and you find anything interesting be sure to share it in Student Hub!

In []: # *TODO (optional)* - keep exploring!

performance_of_sets_and_dictionaries

March 29, 2020

1 Performance of Sets and Dictionaries

In [1]: # Run this cell first!

```
import time
import random
from matplotlib import pyplot as plt
```

1.0.1 Feel the *slowness*

In the previous notebook you *saw* the slowness of lists. As a list gets bigger it takes longer and longer to perform membership tests.

But you can *feel* the slowness too. Compare how long it takes to run the next two cells.

In [2]: # SMALL list membership tests

```
small_list = list(range(10)) # ten element list of integers
nonexistent_element = -1
num_trials = 5000

start = time.clock()

# do lots of membership tests
for _ in range(num_trials):
    nonexistent_element in small_list

end = time.clock()
millis = (end-start) * 1000
print("Execution complete! That took", millis, "milliseconds")
```

Execution complete! That took 1.4529999999999266 milliseconds

In [3]: # BIG list membership tests

```
big_list = list(range(100000)) # 100K element list of integers
nonexistent_element = -1
```

other_data_structures

March 29, 2020

1 Other Data Structures [optional]

The purpose of this notebook is to show you some of the many other data structures you can use without going into too much detail. You can learn more by reading [documentation from Python's collections library](#).

1.1 1. Tuples

The only standard library data structure that we haven't discussed. The tuple is an immutable (unchangeable) sequence of Python objects.

The tuple is very similar to a list. You can read more about it in the [Python tuple documentation](#)

In [1]: # tuples are created with (parentheses)

```
my_tuple = (1,2,3)
print(my_tuple)
print(type(my_tuple))
```

```
(1, 2, 3)
<class 'tuple'>
```

In [2]: # elements can be accessed just like they are with lists.

```
print( my_tuple[0] )
print( my_tuple[1] )
print( my_tuple[2] )
```

```
1
2
3
```

In [3]: # there are some things you can't do with tuples
due to them being immutable.

```
my_tuple[1] = 4
```

```
-----  
TypeError Traceback (most recent call last)  
<ipython-input-3-c5caf46120c9> in <module>()  
      2 # due to them being immutable.  
      3  
----> 4 my_tuple[1] = 4
```

```
TypeError: 'tuple' object does not support item assignment
```

```
In [4]: # but there are also some things you CAN do with tuples  
# that you can't do with lists...
```

```
t1 = ('a', 'b', 'c')  
t2 = (1, 2, 3)  
  
set_of_tuples = set()  
  
set_of_tuples.add(t1)  
set_of_tuples.add(t2)  
  
print(set_of_tuples)  
{('a', 'b', 'c'), (1, 2, 3)}
```

```
In [5]: L1 = ['a', 'b', 'c']  
L2 = [1, 2, 3]
```

```
set_of_lists = set()  
  
set_of_lists.add(L1)  
set_of_lists.add(L2)  
  
print(set_of_lists)
```

```
-----  
TypeError Traceback (most recent call last)  
<ipython-input-5-4b4d5a49a68e> in <module>()  
      4 set_of_lists = set()  
      5  
----> 6 set_of_lists.add(L1)
```

```
7 set_of_lists.add(L2)
8
```

```
TypeError: unhashable type: 'list'
```

1.2 2. Namedtuple

Very similar to a tuple except the fields can be named as well! I use namedtuples when I want to use object.property notation but don't want to define a full class.

```
In [6]: # named tuple's need to be imported from the collections library
         from collections import namedtuple

         # here we define Point as a new type of thing.
         # It has properties x and y.
         Point = namedtuple("Point", ["x", "y"])

         # here we actually instantiate a point
         p1 = Point(5, -3)

         print(p1)

Point(x=5, y=-3)
```

```
In [7]: # there are two ways to access the fields in a point...
```

```
# ... by position
print( p1[0] )
print( p1[1] )
```

```
5
-3
```

```
In [8]: # ... or by name
```

```
print( p1.x )
print( p1.y )
```

```
5
-3
```

1.3 3. Counter

Often we want to count how many times something occurs. The code below demonstrates how to use a Counter to count the number of occurrences of various characters in a string.

```
In [9]: from collections import Counter

string = "the quick brown fox jumped over the lazy dog"

character_counter = Counter()
for character in string:
    character_counter[character] += 1

character_counter.most_common()
```

```
Out[9]: [(' ', 8),
          ('e', 4),
          ('o', 4),
          ('t', 2),
          ('h', 2),
          ('u', 2),
          ('r', 2),
          ('d', 2),
          ('q', 1),
          ('i', 1),
          ('c', 1),
          ('k', 1),
          ('b', 1),
          ('w', 1),
          ('n', 1),
          ('f', 1),
          ('x', 1),
          ('j', 1),
          ('m', 1),
          ('p', 1),
          ('v', 1),
          ('l', 1),
          ('a', 1),
          ('z', 1),
          ('y', 1),
          ('g', 1)]
```

It looks like this string had 8 spaces, 4 e's, 4 o's, etc...

```
In [10]: # something that's nice about counters is that they don't throw
# an error if you try to access a key that isn't there. Instead
# they return 0.

# how many capital A's are in the string above?

print(character_counter["A"])
```

0

```
In [11]: # but how many lowercase a's?
```

```
print(character_counter["a"])
```

```
1
```

1.4 4. defaultdict

A default dict is best explained by example. Let's go back to the "three boxes of tickets" example from earlier.

```
In [12]: TICKET_BOXES = {
    "low"      : [],
    "medium"   : [],
    "high"     : []
}

unfiled_tickets = [
{
    "priority"  : "high",
    "description": "slammed on brakes"
},
{
    "priority"  : "low",
    "description": "windshield chipped"
},
{
    "priority"  : "low",
    "description": "failed to use turn signal"
},
{
    "priority"  : "medium",
    "description": "did not come to complete stop at stop sign"
}
]

def file_ticket(ticket):
    priority = ticket['priority']
    TICKET_BOXES[priority].append(ticket)

for ticket in unfiled_tickets:
    file_ticket(ticket)

print(TICKET_BOXES)
```

```
In [13]: # so far so good! But what if we try to file a ticket  
# with a priority "highest" (as we saw in Jira)?
```

```
new_ticket = {  
    "priority" : "highest",  
    "description": "vehicle crashed!"  
}  
  
file_ticket(new_ticket)
```

```
-----  
KeyError Traceback (most recent call last)
```

```
<ipython-input-13-0d6e7dcf92da> in <module>()  
    7 }  
    8  
----> 9 file_ticket(new_ticket)  
  
<ipython-input-12-44880f72c04b> in file_ticket(ticket)  
    27 def file_ticket(ticket):  
    28     priority = ticket['priority']  
---> 29     TICKET_BOXES[priority].append(ticket)  
    30  
    31 for ticket in unfiled_tickets:  
  
KeyError: 'highest'
```

```
In [14]: # as expected, we get a key error... one way to fix this  
# is as follows
```

```
def file_ticket_fixed(ticket):  
    priority = ticket['priority']  
  
    # new code  
    if priority not in TICKET_BOXES:  
        TICKET_BOXES[priority] = []  
  
    TICKET_BOXES[priority].append(ticket)  
  
file_ticket_fixed(new_ticket)  
print(TICKET_BOXES)  
  
{'low': [{'priority': 'low', 'description': 'windshield chipped'}, {'priority': 'low', 'descript
```

```
In [15]: # OR we can use a "defaultdict"
from collections import defaultdict

TICKET_BOXES = defaultdict(list) # notice the argument of list...

def file_ticket(ticket):
    priority = ticket['priority']
    TICKET_BOXES[priority].append(ticket)

for ticket in unfiled_tickets:
    file_ticket(ticket)

file_ticket(new_ticket)

print(TICKET_BOXES)

defaultdict(<class 'list'>, {'high': [{'priority': 'high', 'description': 'slammed on brakes'}]},
```

When you try to access a key that doesn't exist, defaultdict adds that key to the dictionary and associates a **default** value with it (in this case a list).

If you want to learn more you can read the [documentation on defaultdict](#)

1.5 5. Other data structures from collections

```
In [16]: from collections import deque, OrderedDict
```

```
In [17]: d = deque([4,5,6])
print(d)
```

```
deque([4, 5, 6])
```

```
In [18]: d.append(7)
print(d)
```

```
deque([4, 5, 6, 7])
```

```
In [19]: d.appendleft(3)
print(d)
```

```
deque([3, 4, 5, 6, 7])
```

```
In [20]: last = d.pop()
print("last element was", last)
print("now d is", d)
```

```
last element was 7
now d is deque([3, 4, 5, 6])
```

```
In [21]: first = d.popleft()
         print("first element was", first)
         print("now d is", d)
```

```
first element was 3
now d is deque([4, 5, 6])
```

```
In [22]: # # # # #
```

```
In [23]: od = OrderedDict()
```

```
In [24]: od['a'] = 1
         od['b'] = 2
         od['c'] = 3
```

```
In [25]: # as the name implies, an OrderedDict is a dictionary that
         # keeps track of the order in which elements were added.
```

```
print(od)

OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

```
In [ ]:
```

daysBetweenDates

March 27, 2020

1 Days Between Dates

This lesson will focus on one problem: calculating the number of days between two dates.

This workspace is yours to use in whatever way is helpful. You might want to keep it open in a second tab as you go through the videos.

```
In [27]: def isLeapYear(year):
    # from Wikipedia pseudocode
    if year % 400 == 0:
        return True
    elif year % 100 == 0:
        return False
    elif year % 4 == 0:
        return True
    else:
        return False

def daysInMonth(year, month):
    if month == 1 or month == 3 or month == 5 or month == 7 \
        or month == 8 or month == 10 or month == 12:
        return 31
    else:
        if month == 2:
            if isLeapYear(year):
                return 29
            else:
                return 28
        else:
            return 30

def nextDay(year, month, day):
    ...
    Original code
    if day < 30:
        return year, month, day + 1
    else:
        if month < 12:
```

```

        return year, month + 1, 1
    else:
        return year + 1, 1, 1
    ...

    if day < daysInMonth(year, month):
        return year, month, day + 1
    else:
        if month < 12:
            return year, month + 1, 1
        else:
            return year + 1, 1, 1

def dateIsBefore(year1, month1, day1, year2, month2, day2):
    """
    Returns True if year1-month1-day1 is before year2-month2-day2.
    Otherwise, returns False.
    """

    if year1 < year2:
        return True
    if year1 == year2:
        if month1 < month2:
            return True
        if month1 == month2:
            return day1 < day2
    return False

def daysBetweenDates(year1, month1, day1, year2, month2, day2):
    """
    Calculates the number of days between two dates.
    """

    # TODO - by the end of this lesson you will have
    # completed this function. You do not need to complete
    # it yet though!

    days = 0
    while dateIsBefore(year1, month1, day1, year2, month2, day2):
        year1, month1, day1 = nextDay(year1, month1, day1)
        days += 1

    return days

def testDaysBetweenDates():

    # test same day
    assert(daysBetweenDates(2017, 12, 30,
                           2017, 12, 30) == 0)

```

```
# test adjacent days
assert(daysBetweenDates(2017, 12, 30,
                        2017, 12, 31) == 1)

# test new year
assert(daysBetweenDates(2017, 12, 30,
                        2018, 1, 1) == 2)

# test full year difference
assert(daysBetweenDates(2012, 6, 29,
                        2013, 6, 29) == 365)

print("Congratulations! Your daysBetweenDates")
print("function is working correctly!")

testDaysBetweenDates()

print('Testing ended.')
```

```
Congratulations! Your daysBetweenDates
function is working correctly!
Testing ended.
```

In []:

Plotting Position vs Time

March 29, 2020

1 Plotting Position vs Time

In this notebook you will plot a position vs time graph of the data you just saw.

First, I will demonstrate such a plot by following these steps:

1. Importing `pyplot`, Python's most popular plotting library.
2. Storing data to be plotted in variables named `X` and `Y`
3. Creating a scatter plot of this data using `pyplot's scatter()` function.
4. Adding a line connecting two data points using `pyplot's plot()` function.
5. Adding axis labels and a title to the graph.

In [1]: # Step 1.

```
# we import pyplot as plt so we can refer to the pyplot
# succinctly. This is a standard convention for this library.
```

```
from matplotlib import pyplot as plt
```

Initially, I only told you the mileage at 2:00 and 3:00. The data looked like this.

Time	Odometer (miles)
2:00	30
3:00	80

I'd like to make a scatter plot of this data and I want my **horizontal** axis to show time and my **vertical** axis to show mileage.

In this notebook (and those that follow), we are going to use a capital `X` to store horizontal axis data and a capital `Y` to store vertical axis data. In this case:

In [2]: # Step 2.

```
# get the data into variables called X and Y. This naming pattern
# is a convention. You could use any variables you like.
```

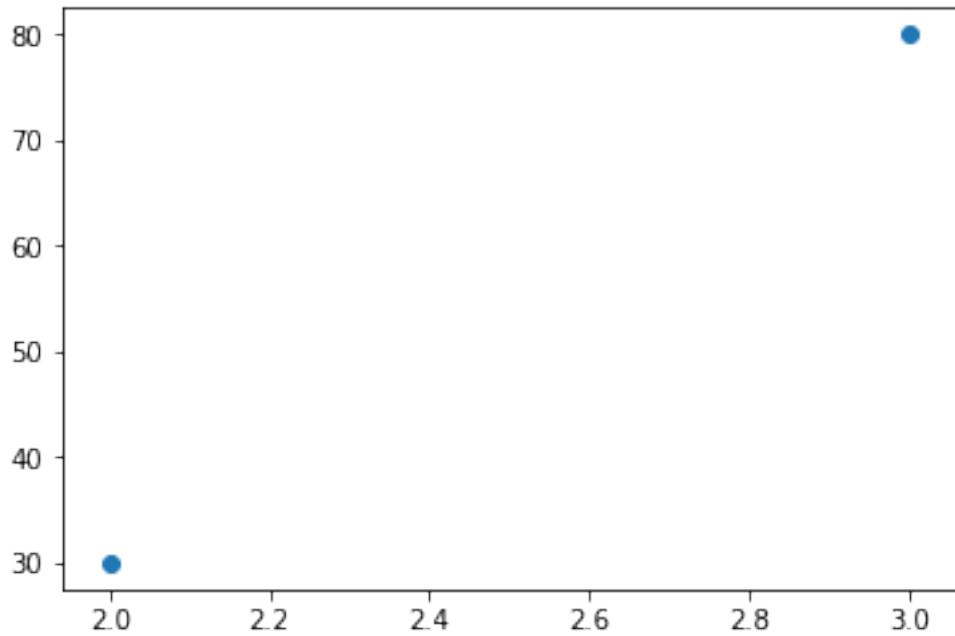
```
X = [2,3]
Y = [30,80]
```

In [3]: # Step 3.

```
# create a scatter plot using plt.scatter. Note that you NEED
```

```
# to call plt.show() to actually see the plot. Forgetting to
# call plt.show() is a common source of problems for people
# new to this library

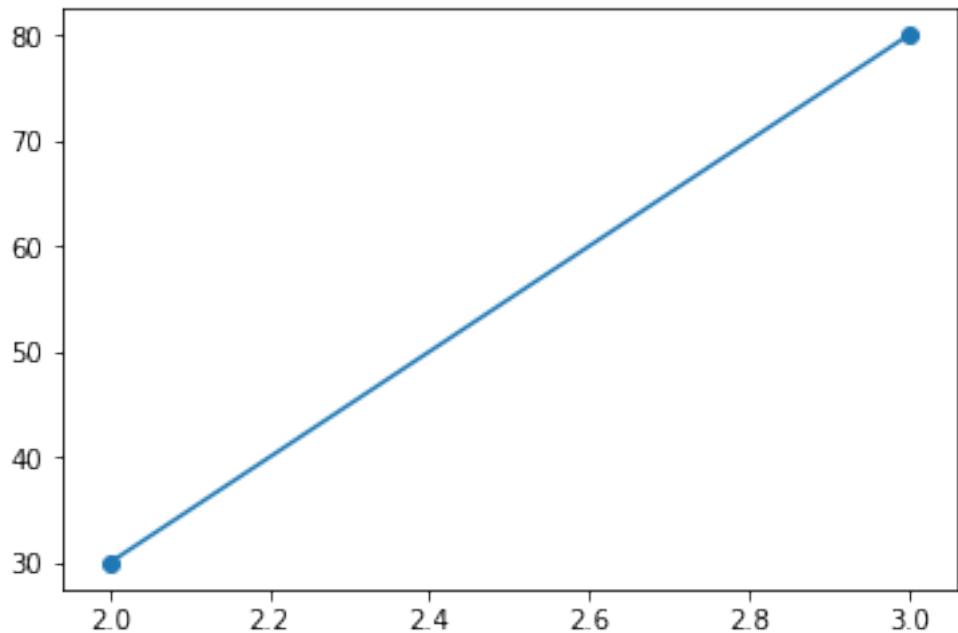
plt.scatter(X,Y)
plt.show()
```



This isn't a very exciting scatter plot since it only has two data points. Let's add a line connecting these data points as well.

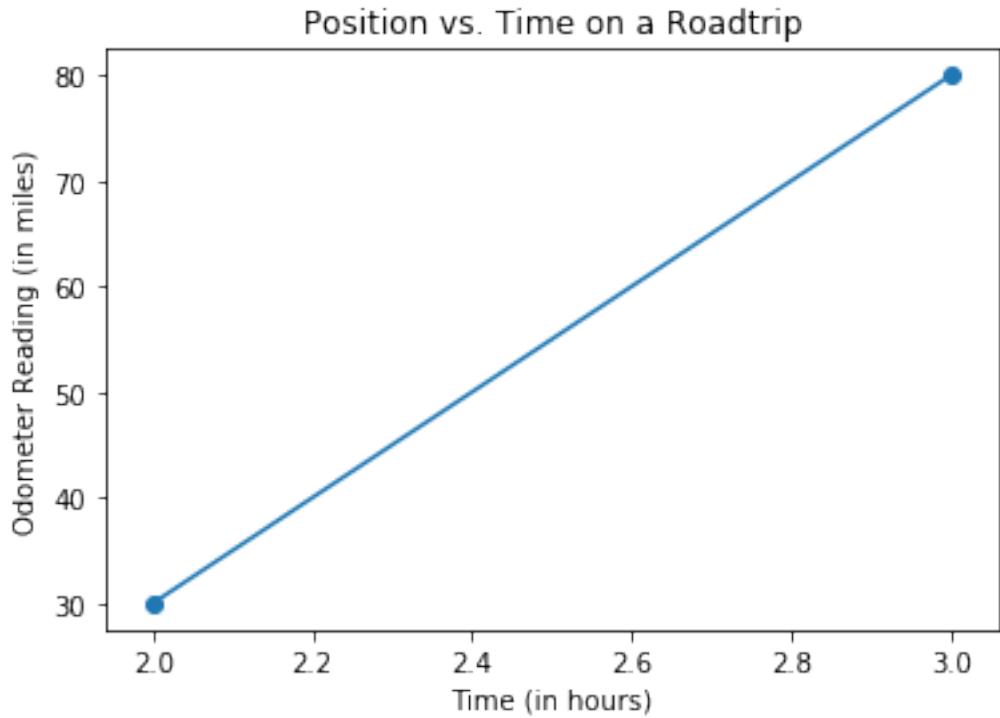
```
In [4]: # Step 4.
# add lines connecting adjacent points

plt.scatter(X,Y)
plt.plot(X,Y)
plt.show()
```



Let's add a title and labels to the X and Y axes

```
In [5]: plt.scatter(X,Y)
plt.plot(X,Y)
plt.title("Position vs. Time on a Roadtrip")
plt.xlabel("Time (in hours)")
plt.ylabel("Odometer Reading (in miles)")
plt.show()
```



1.1 Twenty minute resolution

When looking at the odometer every 20 minutes, the data looks like this:

Time	Odometer (miles)
2:00	30
2:20	40
2:40	68
3:00	80

But a better way to think about it for plotting is like this (note the difference in how time is represented):

Time	Odometer (miles)
2.000	30
2.333	40
2.667	68
3.000	80

Understanding the Derivative

March 29, 2020

1 Understanding the Derivative

You just saw these three statements.

1. Velocity is the instantaneous rate of change of **position**
2. Velocity is the slope of the tangent line of **position**
3. Velocity is the derivative of **position**

But there's another, more formal (and mathematical) definition of the derivative that you're going to explore in this notebook as you build an intuitive understanding for what a derivative is.

1.1 BEFORE YOU CONTINUE

This notebook is a long one and it really requires focus and attention to be useful. Before you continue, make sure that:

1. You have **at least 30 minutes** of time to spend here.
 2. You have the mental energy to read through math and some (occasionally) complex code.
-

1.2 Formal definition of the derivative

The **derivative of $f(t)$ with respect to t** is the function $\dot{f}(t)$ ("f dot of t") and is defined as

$$\dot{f}(t) = \lim_{\Delta t \rightarrow 0} \frac{f(t + \Delta t) - f(t)}{\Delta t}$$

You should read this equation as follows:

"F dot of t is equal to the limit as delta t goes to zero of F of t plus delta t minus F of t all over delta t"

1.3 Outline

In this notebook we are going to unpack this definition by walking through a series of activities that will end with us defining a python function called `approximate_derivative`. This function will look very similar to the math shown above.

A rough outline of how we'll get there:

1. **Discrete vs Continuous Motion** - A quick reminder of the difference between **discrete** and **continuous** motion and some practice defining continuous functions in code.

2. **Plotting continuous functions** - This is where you'll see `plot_continuous_function` which is a function that takes **another function** as an input.
 3. **Finding derivatives "by hand"** - Here you'll find the **velocity** of an object *at a particular time* by zooming in on its **position vs time** graph and finding the slope.
 4. **Finding derivatives algorithmically** - Here you'll use a function to reproduce the steps you just did "by hand".
 5. **OPTIONAL: Finding the full derivative** - In steps 3 and 4 you actually found the derivative of a function *at a particular time*, here you'll see how you can get the derivative of a function for **all times** at once. Be warned - the code gets a little weird here.
-

1.4 1 - Discrete vs Continuous Motion

The data we deal with in a self driving car comes to us discretely. That is, it only comes to us at certain timestamps. For example, we might get a position measurement at timestamp $t=352.396$ and the next position measurement at timestamp $t=352.411$. But what happened in between those two measurements? Did the vehicle **not have a position** at, for example, $t=352.400$?

Of course not!

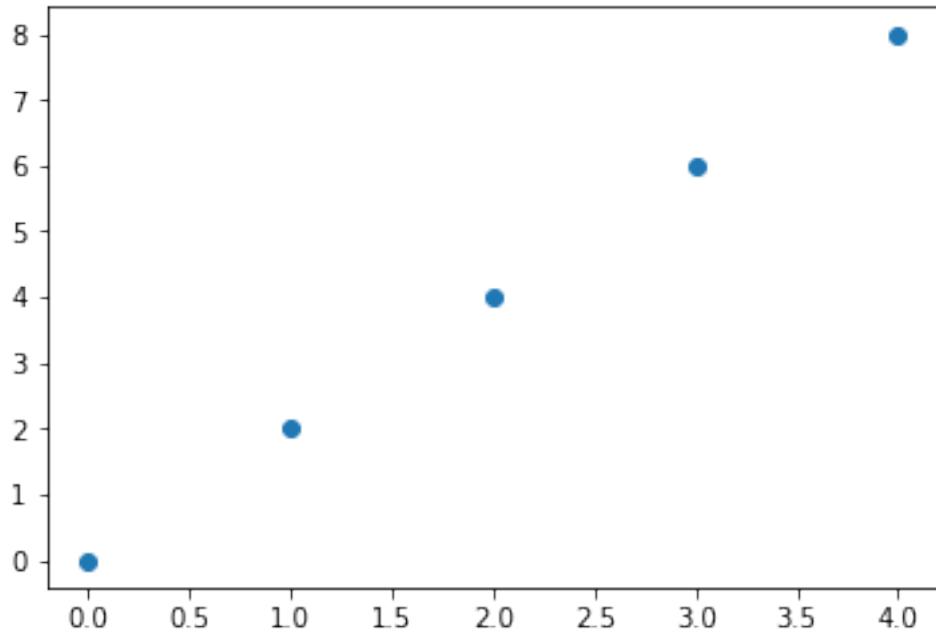
Even though the position data we measure comes to us **discretely**, we know that the actual motion of the vehicle is **continuous**.

Let's say I start moving forwards from $x=0$ at $t=0$ with a speed of $2m/s$. At $t=1$, x will be 2 and at $t=4$, x will be 8. I can plot my position at 1 second intervals as follows:

```
In [1]: from matplotlib import pyplot as plt
%matplotlib inline

t = [0,1,2,3,4]
x = [0,2,4,6,8]

plt.scatter(t,x)
plt.show()
```



This graph above is a **discrete** picture of motion. And this graph came from two Python **lists**...
 But what about the underlying **continuous** motion? We can represent this motion with a function f like this:

$$f(t) = 2t$$

How can we represent that in code?

A list won't do! We need to define (surprise, surprise) a function!

```
In [2]: def position(time):
    return 2*time

    print("at t =", 0, "position is", position(0))
    print("at t =", 1, "position is", position(1))
    print("at t =", 2, "position is", position(2))
    print("at t =", 3, "position is", position(3))
    print("at t =", 4, "position is", position(4))

at t = 0 position is 0
at t = 1 position is 2
at t = 2 position is 4
at t = 3 position is 6
at t = 4 position is 8
```

That looks right (and it matches our data from above). Plus it can be used to get the position of the vehicle in between "sensor measurements!"

```
In [3]: print("at t =", 2.2351, "position is", position(2.2351))  
at t = 2.2351 position is 4.4702
```

This `position(time)` function is a continuous function of time. When you see $f(t)$ in the formal definition of the derivative you should think of something like this.

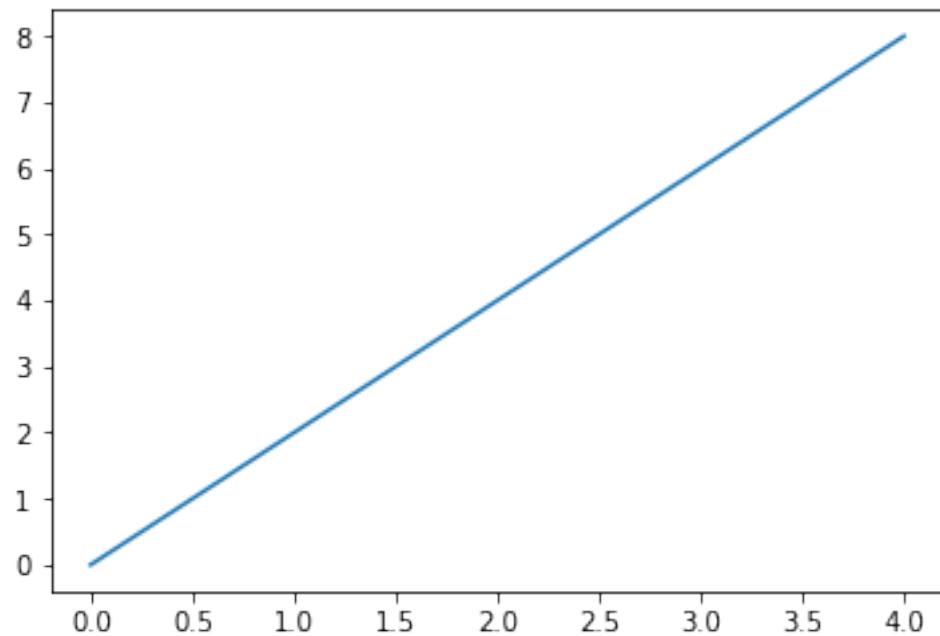
1.5 2 - Plotting Continuous Functions

Now that we have a continuous function, how do we plot it??

We're going to use `numpy` and a function called `linspace` to help us out. First let me demonstrate plotting our position function for times between 0 and 4.

```
In [4]: # Demonstration of continuous plotting
```

```
import numpy as np  
  
t = np.linspace(0, 4)  
x = position(t)  
  
plt.plot(t, x)  
plt.show()
```



EXERCISE - create and plot a continuous function of time Write a function, `position_b(time)` that represents the following motion:

$$f(t) = -4.9t^2 + 30t$$

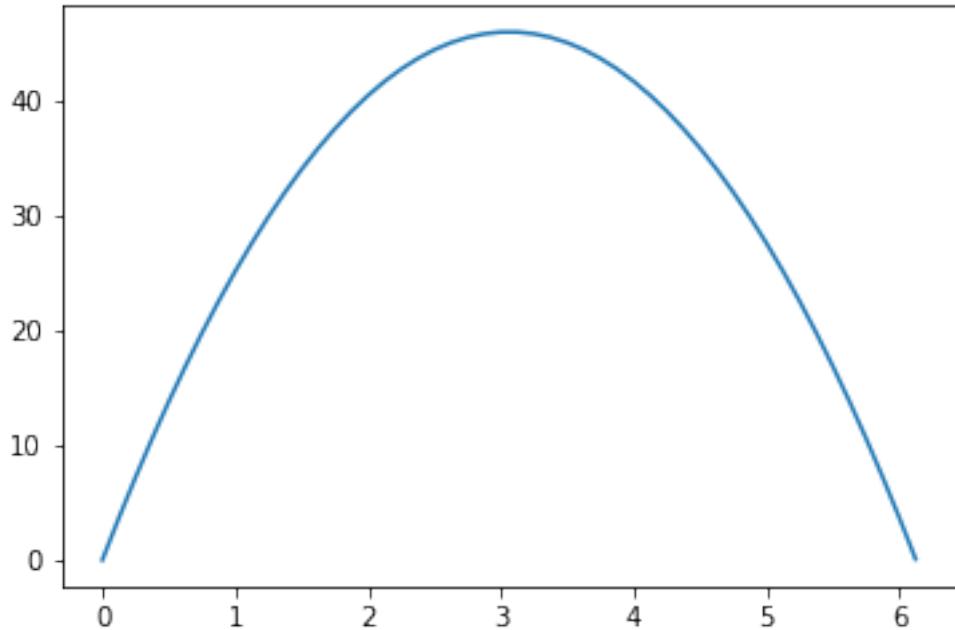
then plot the function from $t = 0$ to $t = 6.12$

```
In [6]: # EXERCISE
def position_b(time):
    # todo
    return - 4.9 * np.square(time) + 30 * time

t = np.linspace(0, 6.12)
z = position_b(t)

plt.plot(t, z)
plt.show()

# don't forget to plot this function from t=0 to t=6.12
# Solution is below.
```



```
In [ ]: #
```

```
#
```

```
#
```

```

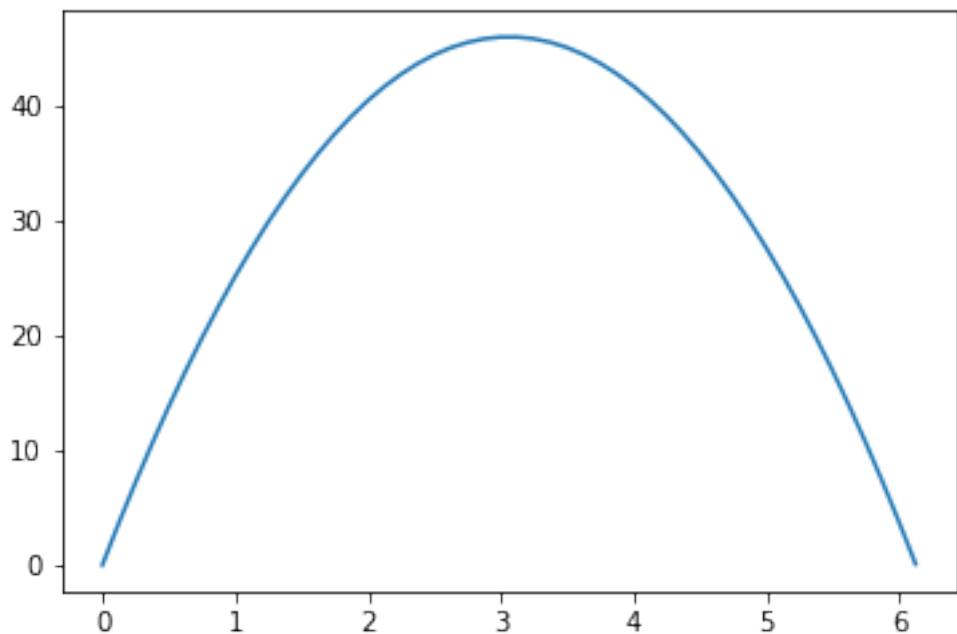
# Spoiler alert! Solution below!

#
#
#
In [7]: def position_b(time):
           return -4.9 * time ** 2 + 30 * time

t = np.linspace(0, 6.12)
z = position_b(t)

plt.plot(t, z)
plt.show()

```



Fun fact (maybe)

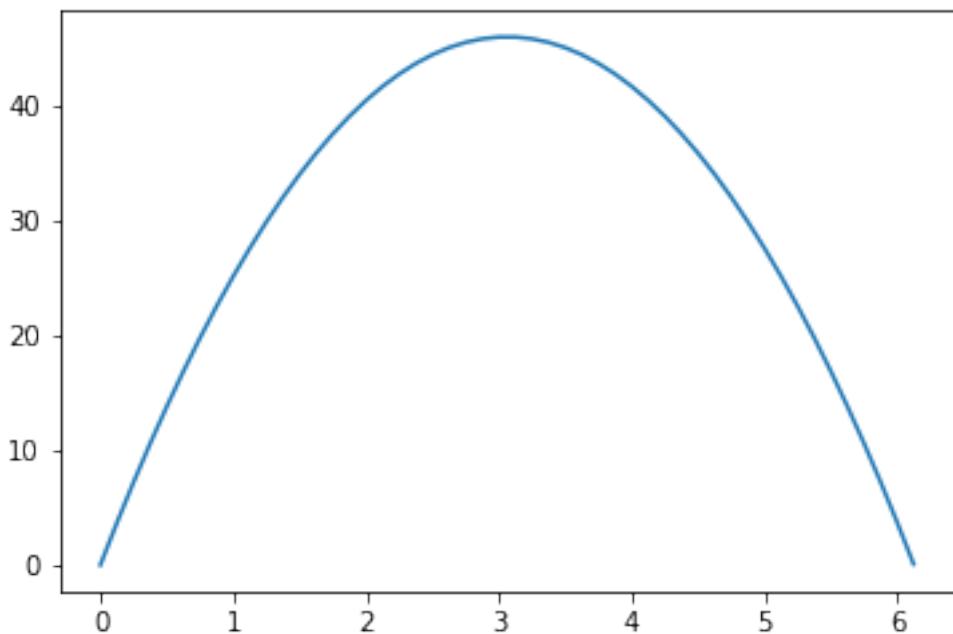
There's a reason I used the variable `z` in my plotting code. `z` is typically used to represent distance above the ground and the function you just plotted actually represents the height of a ball thrown upwards with an initial velocity of 30m/s . As you can see the ball reaches its maximum height about 3 seconds after being thrown.

1.5.1 2.1 - Generalize our plotting code

I don't want to have to keep copy and pasting plotting code so I'm just going to write a function...

```
In [8]: def plot_continuous_function(function, t_min, t_max):
    t = np.linspace(t_min, t_max)
    x = function(t)
    plt.plot(t,x)
```

```
In [9]: plot_continuous_function(position_b, 0, 6.12)
plt.show()
```



Take a look at `plot_continuous_function`.

Notice anything weird about it?

This function actually *takes another function as input*. This is a perfectly valid thing to do in Python, but I know the first time I saw code like this I found it pretty hard to wrap my head around what was going on.

Just wait until a bit later in this notebook when you'll see a function that actually `returns` another function!

For now, let me show you other ways you can use `plot_continuous_function`.

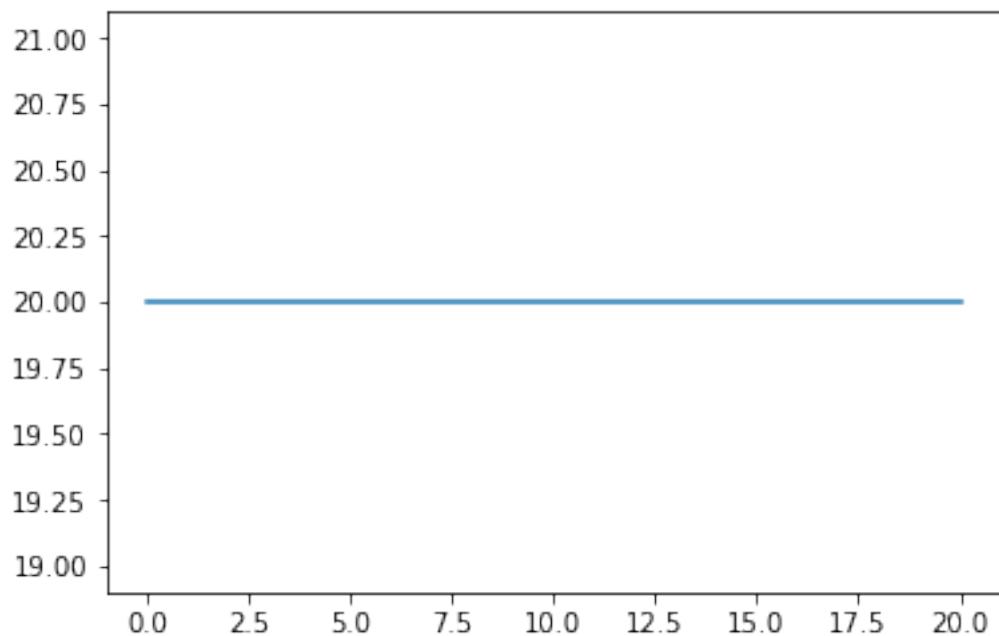
```
In [10]: def constant_position_motion(time):
    position = 20
    return position + 0*time

def constant_velocity_motion(time):
    velocity = 10
    return velocity * time

def constant_acceleration_motion(time):
    acceleration = 9.8
```

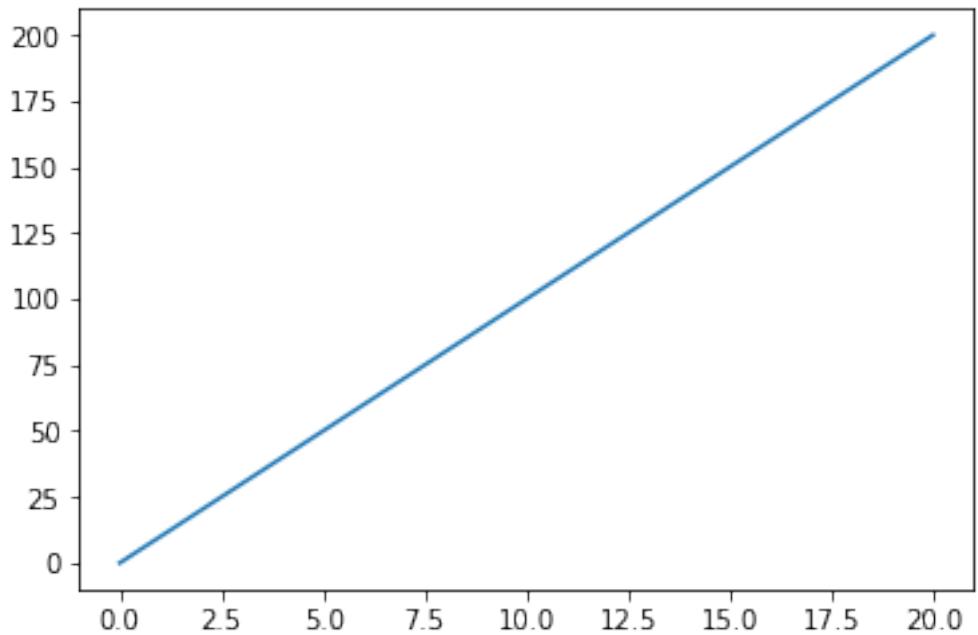
```
    return acceleration / 2 * time ** 2

plot_continuous_function(constant_position_motion, 0, 20)
plt.show()
```

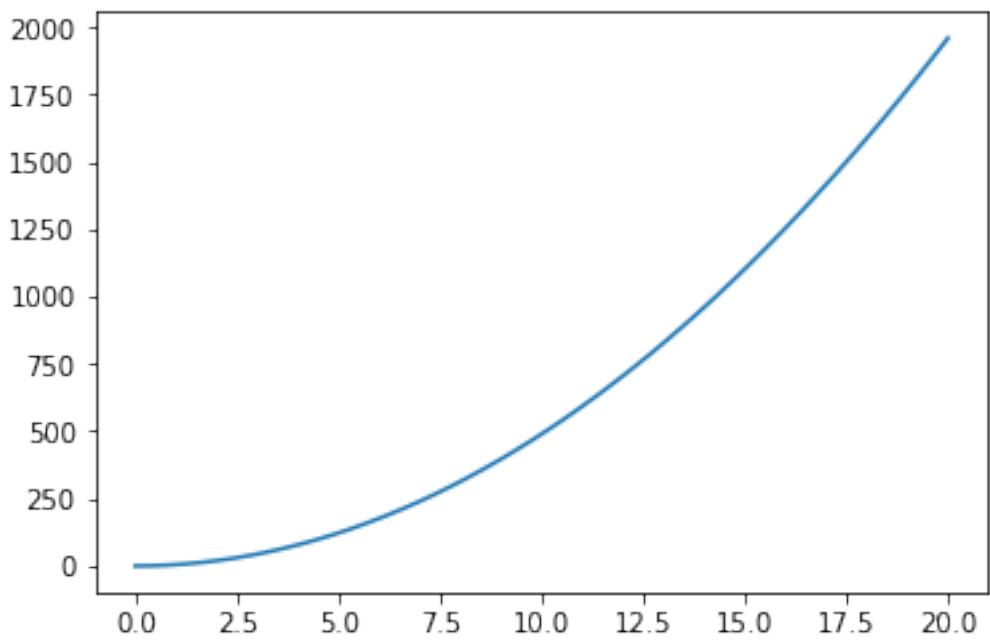


```
In [11]: # position vs time
# with constant VELOCITY motion

plot_continuous_function(constant_velocity_motion, 0, 20)
plt.show()
```



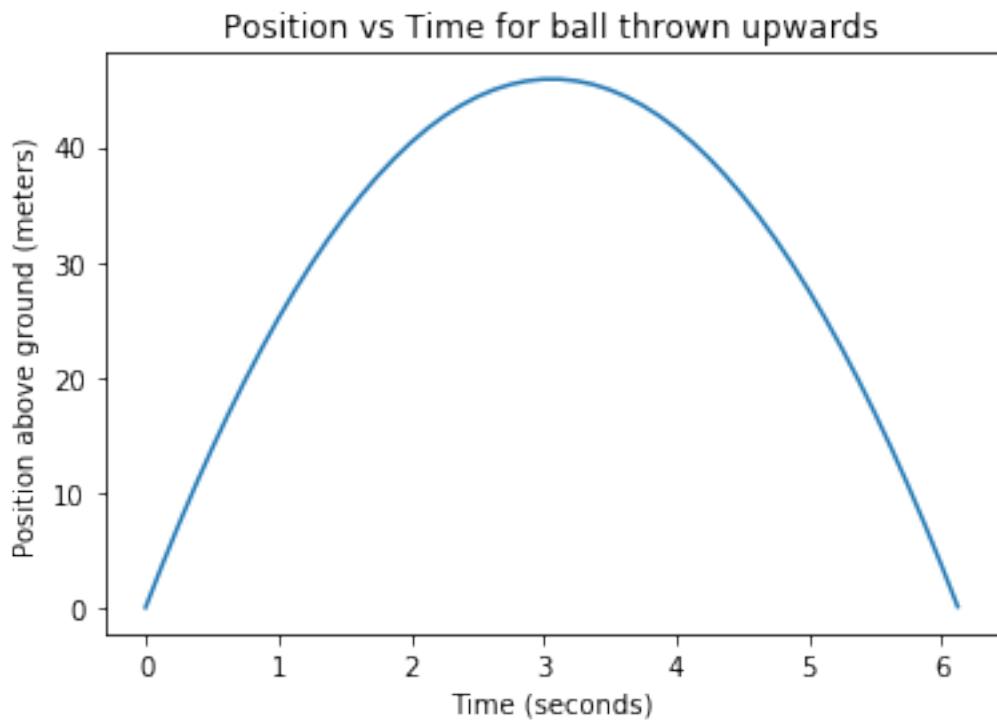
```
In [12]: # position vs time  
# with constant ACCELERATION motion  
  
plot_continuous_function(constant_acceleration_motion, 0, 20)  
plt.show()
```



1.6 3 - Find derivative "by hand" at a specific point

Let's go back to the ball-thrown-in-air example from before and see if we can find the **velocity** of the ball at various times. Remember, the graph looked like this:

```
In [13]: plt.title("Position vs Time for ball thrown upwards")
plt.ylabel("Position above ground (meters)")
plt.xlabel("Time (seconds)")
plot_continuous_function(position_b,0,6.12)
plt.show()
```



Now I would like to know the **velocity** of the ball at $t=2$ seconds.

GOAL - Find the velocity of the ball at $t=2$ seconds

And remember, **velocity is the derivative of position**, which means **velocity is the slope of the tangent line of position**

Well we have the position vs time graph... now we just need to find the slope of the tangent line to that graph AT $t=2$.

One way to do that is to just zoom in on the graph until it starts to look straight. I can do that by changing the `t_min` and `t_max` that I pass into `plot_continuous_function`.

Speed from Position Data

March 29, 2020

1 Speed from Position Data

In this Notebook you'll work with data just like the data you'll be using in the final project for this course. That data comes from CSVs that looks like this:

timestamp	displacement	yaw_rate	acceleration
0.0	0	0.0	0.0
0.25	0.0	0.0	19.6
0.5	1.225	0.0	19.6
0.75	3.675	0.0	19.6
1.0	7.35	0.0	19.6
1.25	12.25	0.0	0.0
1.5	17.15	-2.82901631903	0.0
1.75	22.05	-2.82901631903	0.0
2.0	26.95	-2.82901631903	0.0
2.25	31.85	-2.82901631903	0.0

```
In [1]: from helpers import process_data
        from matplotlib import pyplot as plt

In [2]: PARALLEL_PARK_DATA = process_data("parallel_park.pickle")

In [3]: # This is what the first few entries in the parallel
        # park data look like.

        PARALLEL_PARK_DATA[:5]

Out[3]: [(0.0, 0, 0.0, 0.0),
          (0.0625, 0.0, 0.0, 1.9600000000000002),
          (0.125, -0.007656250000000007, 0.0, 1.9600000000000002),
          (0.1875, -0.022968750000000003, -0.0, 1.9600000000000002),
          (0.25, -0.045937500000000006, -0.0, 1.9600000000000002)]

In [4]: # In this exercise we'll be differentiating (taking the
        # derivative of) displacement data. This will require
        # using only the first two columns of this data.
        timestamps = [row[0] for row in PARALLEL_PARK_DATA]
```

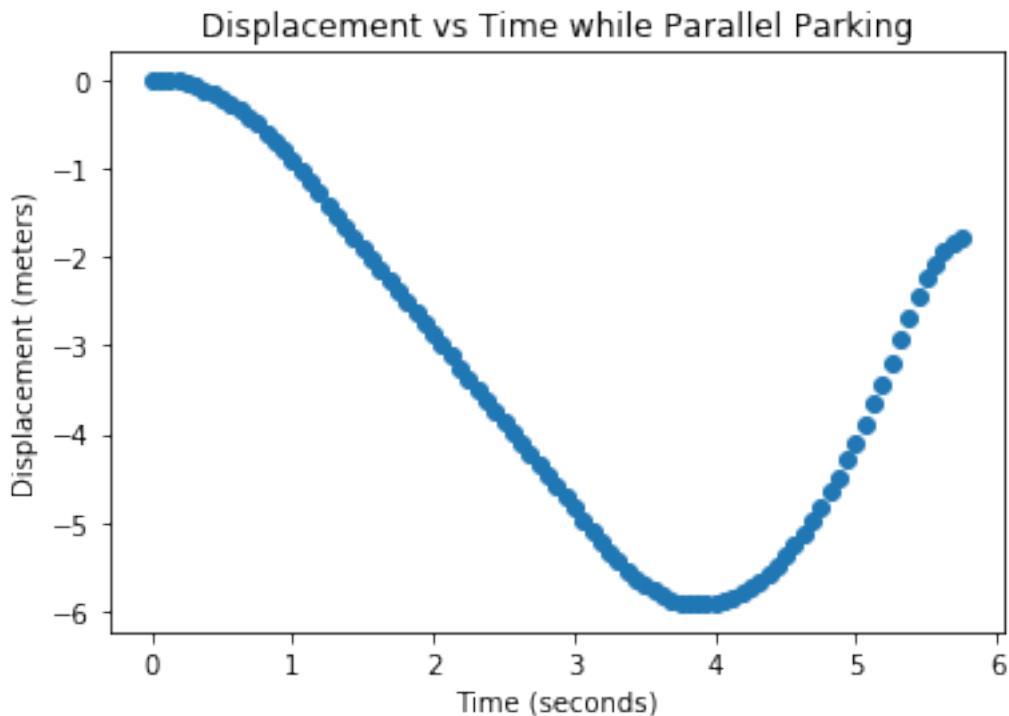
```

displacements = [row[1] for row in PARALLEL_PARK_DATA]

# You'll use these data in the next lesson on integration
# You can ignore them for now.
yaw_rates      = [row[2] for row in PARALLEL_PARK_DATA]
accelerations = [row[8] for row in PARALLEL_PARK_DATA]

In [5]: plt.title("Displacement vs Time while Parallel Parking")
plt.xlabel("Time (seconds)")
plt.ylabel("Displacement (meters)")
plt.scatter(timestamps, displacements)
plt.show()

```



In the graph above, you can see displacement vs time data for a car as it parallel parks. Note that backwards motion winds back the odometer and reduces displacement (this isn't actually how odometers work on modern cars. Sorry Ferris Bueller)

Note how for approximately 4 seconds the motion is backwards and then for the last two the car goes forwards.

Let's look at some data somewhere in the middle of this trajectory

```

In [6]: print(timestamps[20:22])
        print(displacements[20:22])

[1.25, 1.3125]
[-1.4087500000000004, -1.5312500000000004]

```

So you can see that at $t = 1.25$ the car has displacement $x = -1.40875$ and at $t = 1.3125$ the car has displacement $x = -1.53125$

This means we could calculate the speed / slope as follows:

$$\text{slope} = \frac{\text{vertical change}}{\text{horizontal change}} = \frac{\Delta x}{\Delta t}$$

and for the numbers I just mentioned this would mean:

$$\frac{\Delta x}{\Delta t} = \frac{-1.53125 - -1.40875}{1.3125 - 1.25} = \frac{-0.1225 \text{ meters}}{0.0625 \text{ seconds}} = -1.96 \frac{\text{m}}{\text{s}}$$

So I can say the following:

Between $t = 1.25$ and $t = 1.3125$ the vehicle had an **average speed of -1.96 meters per second**

I could make this same calculation in code as follows

```
In [7]: delta_x = displacements[21] - displacements[20]
        delta_t = timestamps[21] - timestamps[20]
        slope   = delta_x / delta_t

        print(slope)
```

-1.96

Earlier in this lesson you worked with truly continuous functions. In that situation you could make Δt as small as you wanted!

But now we have real data, which means the size of Δt is dictated by how frequently we made measurements of displacement. In this case it looks like subsequent measurements are separated by

$$\Delta t = 0.0625 \text{ seconds}$$

In the `get_derivative_from_data` function below, I demonstrate how to "take a derivative" of real data. Read through this code and understand how it works: in the next notebook you'll be asked to reproduce this code yourself.

```
In [8]: def get_derivative_from_data(position_data, time_data):
    """
    Calculates a list of speeds from position_data and
    time_data.

    Arguments:
        position_data - a list of values corresponding to
                        vehicle position

        time_data      - a list of values (equal in length to
                        position_data) which give timestamps for each
```

```

position measurement

Returns:
    speeds      - a list of values (which is shorter
                  by ONE than the input lists) of speeds.

"""
# 1. Check to make sure the input lists have same length
if len(position_data) != len(time_data):
    raise(ValueError, "Data sets must have same length")

# 2. Prepare empty list of speeds
speeds = []

# 3. Get first values for position and time
previous_position = position_data[0]
previous_time     = time_data[0]

# 4. Begin loop through all data EXCEPT first entry
for i in range(1, len(position_data)):

    # 5. get position and time data for this timestamp
    position = position_data[i]
    time     = time_data[i]

    # 6. Calculate delta_x and delta_t
    delta_x = position - previous_position
    delta_t = time - previous_time

    # 7. Speed is slope. Calculate it and append to list
    speed = delta_x / delta_t
    speeds.append(speed)

    # 8. Update values for next iteration of the loop.
    previous_position = position
    previous_time     = time

return speeds

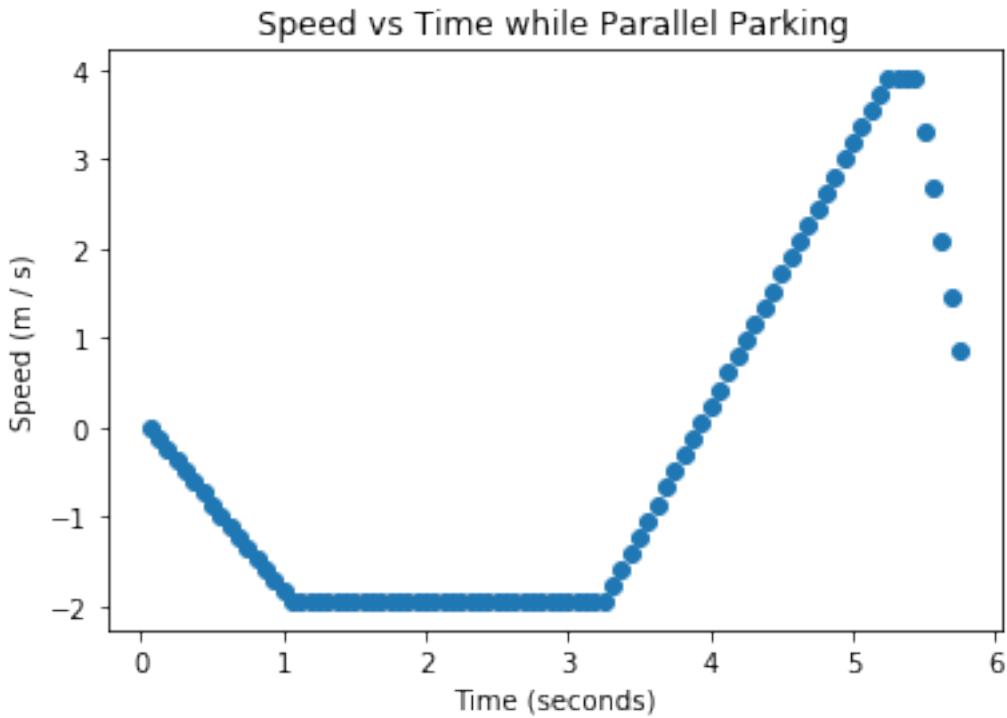
# 9. Call this function with appropriate arguments
speeds = get_derivative_from_data(displacements, timestamps)

# 10. Prepare labels for a plot
plt.title("Speed vs Time while Parallel Parking")
plt.xlabel("Time (seconds)")
plt.ylabel("Speed (m / s)")

# 11. Make the plot! Note the slicing of timestamps!
plt.scatter(timestamps[1:], speeds)

```

```
plt.show()
```



Now that you've read through the code and seen how it's used (and what the resulting plot looks like), I want to discuss the numbered sections of the code.

1. The time and position data need to have equal lengths, since each position measurement is meant to correspond to one of those timestamps.
2. The speeds list will eventually be returned at the end of the function.
3. The use of the word "previous" in these variable names will be clearer in step 8. But basically we need to have TWO positions if we're ever going to calculate a delta X. This is where we grab the first position in the position_data list.
4. Note that we loop from `range(1, len(position_data))`, which means that the first value for `i` will be 1 and **not** 0. That's because we already grabbed element 0 in step 3.
5. Get the data for this `i`.
6. Calculate the change in position and time.
7. Find the slope (which is the speed) and append it to the speeds list.
8. This sets the values of `previous_position` and `previous_time` so that they are correct for the *next* iteration of this loop.
9. Here we call the function with the `displacements` and `timesteps` data that we used before.

10. Self-explanatory
11. This part is interesting. Note that we only plot `timesteps[1 :]`. This means "every element in `timesteps` except the first one". Remember how in step 4 we looped through every element except the first one? That means that our `speeds` array ends up being 1 element shorter than our original data.

1.1 What to Remember

You don't need to memorize any of this. The important thing to remember is this:

When you're working with real time-series data, you calculate the "derivative" by finding the slope between adjacent data points.

You'll be implementing this on your own in the next notebook. Feel free to come back here if you need help, but try your best to get it on your own.

Implement an Accelerometer

March 29, 2020

1 Implement an Accelerometer

In this notebook you will define your own `get_derivative_from_data` function and use it to differentiate position data ONCE to get velocity information and then again to get acceleration information.

In part 1 I will demonstrate what this process looks like and then in part 2 you'll implement the function yourself.

1.1 Part 1 - Reminder and Demonstration

In [1]: # run this cell for required imports

```
from helpers import process_data
from helpers import get_derivative_from_data as solution_derivative
from matplotlib import pyplot as plt
```

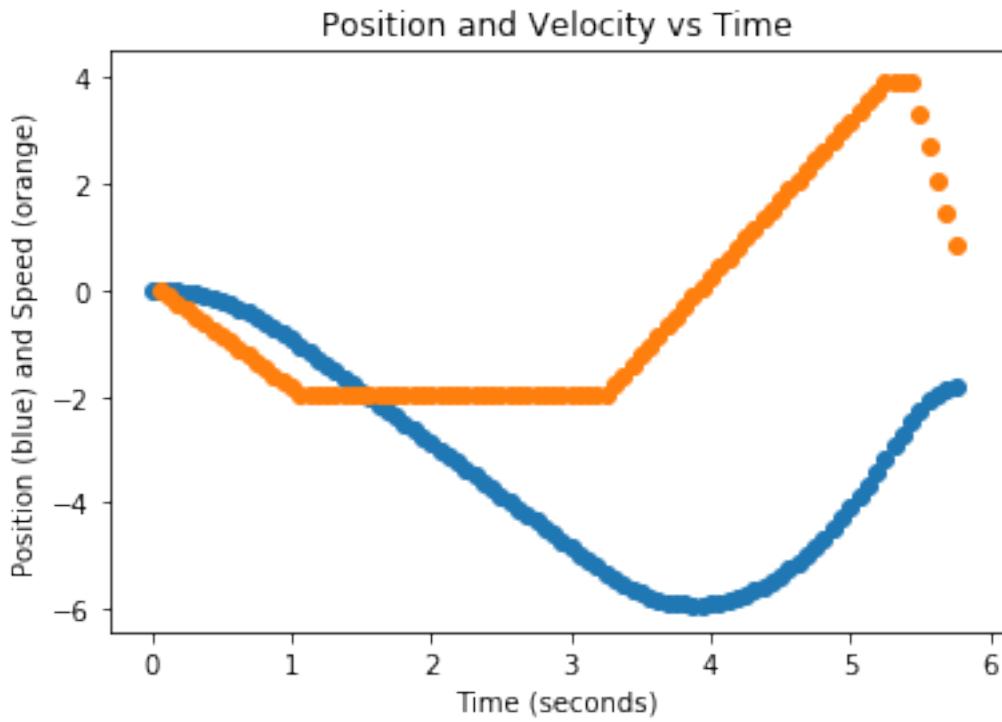
In [2]: # load the parallel park data

```
PARALLEL_PARK_DATA = process_data("parallel_park.pickle")
```

```
# get the relevant columns
timestamps = [row[0] for row in PARALLEL_PARK_DATA]
displacements = [row[1] for row in PARALLEL_PARK_DATA]
```

```
# calculate first derivative
speeds = solution_derivative(displacements, timestamps)
```

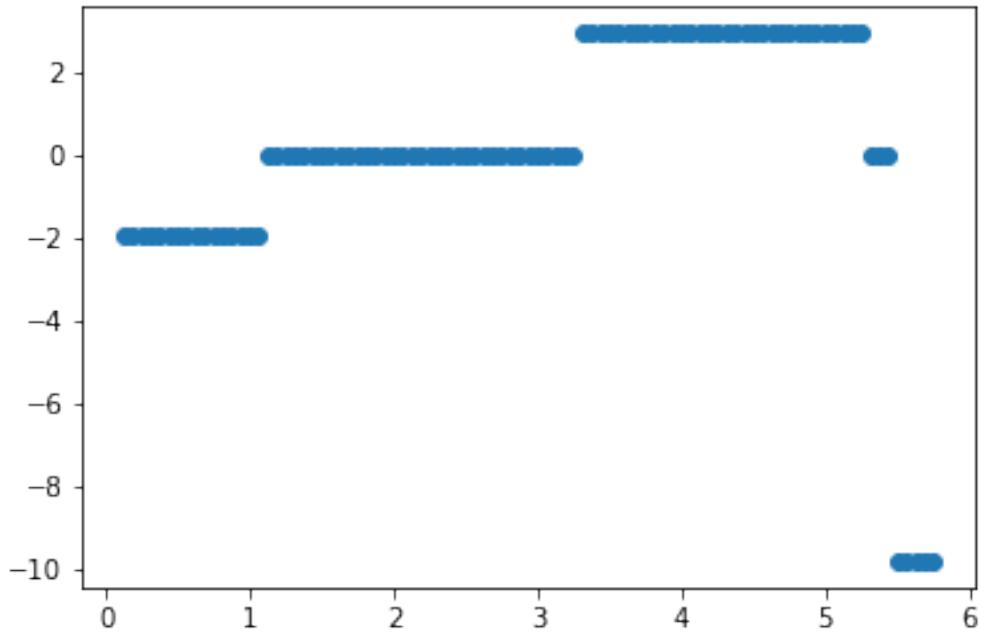
```
# plot
plt.title("Position and Velocity vs Time")
plt.xlabel("Time (seconds)")
plt.ylabel("Position (blue) and Speed (orange)")
plt.scatter(timestamps, displacements)
plt.scatter(timestamps[1:], speeds)
plt.show()
```



But you just saw that acceleration is the derivative of velocity... which means we can use the same derivative function to calculate acceleration!

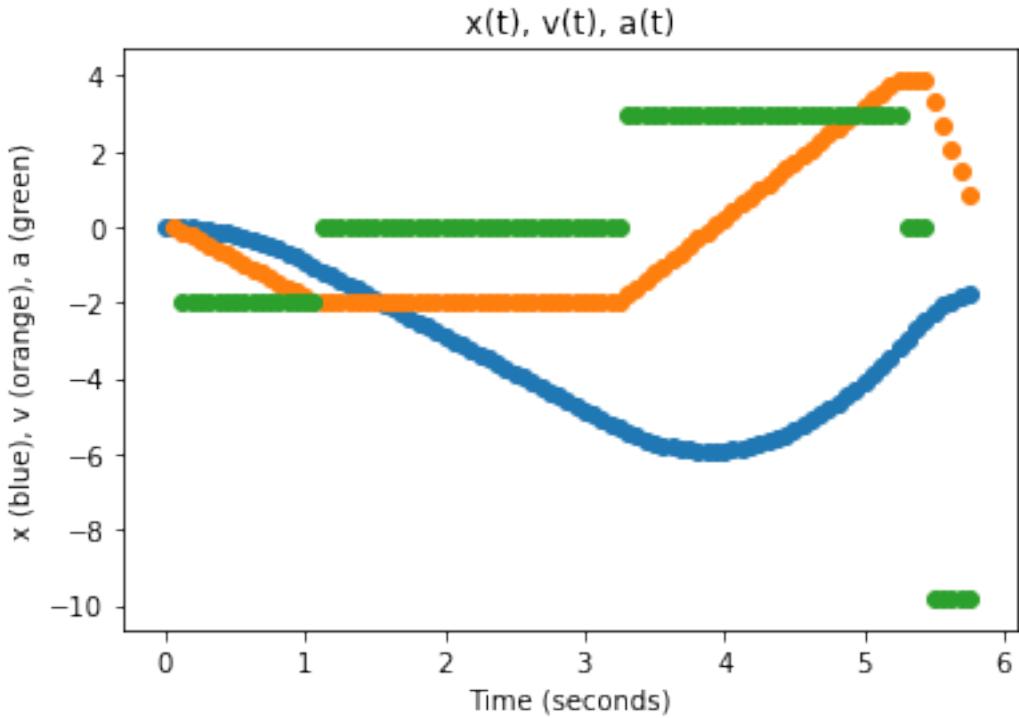
```
In [3]: # calculate SECOND derivative
accelerations = solution_derivative(speeds, timestamps[1:])

# plot (note the slicing of timestamps from 2 --> end)
plt.scatter(timestamps[2:], accelerations)
plt.show()
```



As you can see, this parallel park motion consisted of four segments with different (but constant) acceleration. We can plot all three quantities at once like this:

```
In [4]: plt.title("x(t), v(t), a(t)")  
plt.xlabel("Time (seconds)")  
plt.ylabel("x (blue), v (orange), a (green)")  
plt.scatter(timestamps, displacements)  
plt.scatter(timestamps[1:], speeds)  
plt.scatter(timestamps[2:], accelerations)  
plt.show()
```



1.2 Part 2 - Implement it yourself!

```
In [5]: def get_derivative_from_data(position_data, time_data):
    # TODO - try your best to implement this code yourself!
    #           if you get really stuck feel free to go back
    #           to the previous notebook for a hint.

    assert len(position_data) == len(time_data)

    previous_position = position_data[0]
    previous_time = time_data[0]
    speed = []

    for i in range(1, len(position_data)):
        position = position_data[i]
        time = time_data[i]
        delta_z = position - previous_position
        delta_t = time - previous_time
        slope = delta_z / delta_t
        speed.append(slope)
        previous_position = position
        previous_time = time
```

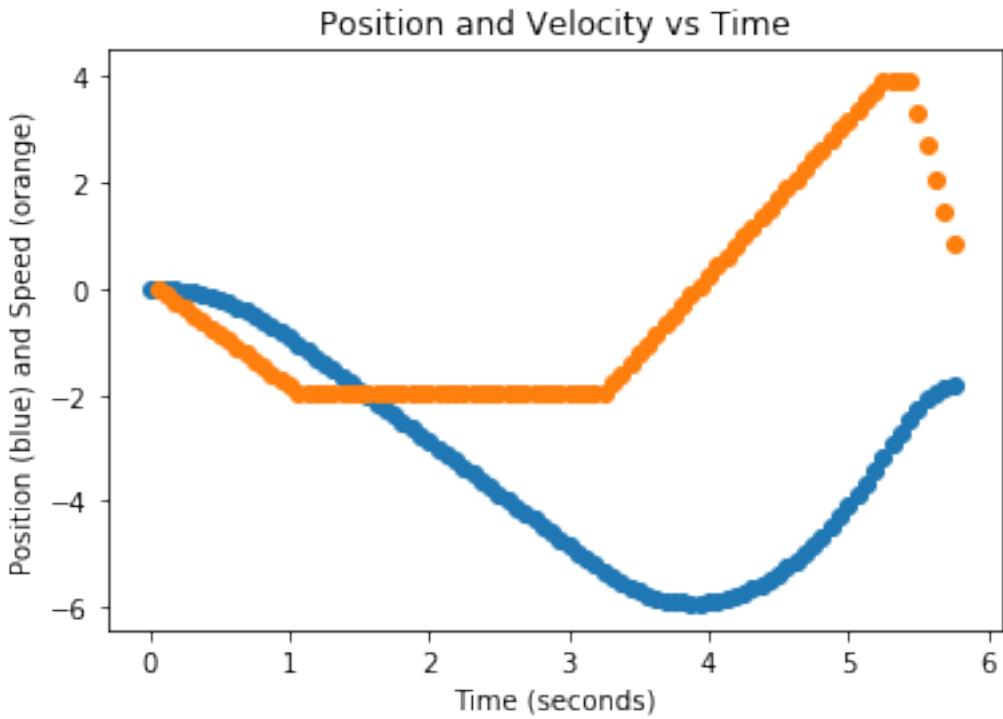
```

    return speed

In [6]: # Testing part 1 - visual testing of first derivative
        # compare this output to the corresponding graph above.
speeds = get_derivative_from_data(displacements, timestamps)

plt.title("Position and Velocity vs Time")
plt.xlabel("Time (seconds)")
plt.ylabel("Position (blue) and Speed (orange)")
plt.scatter(timestamps, displacements)
plt.scatter(timestamps[1:], speeds)
plt.show()

```



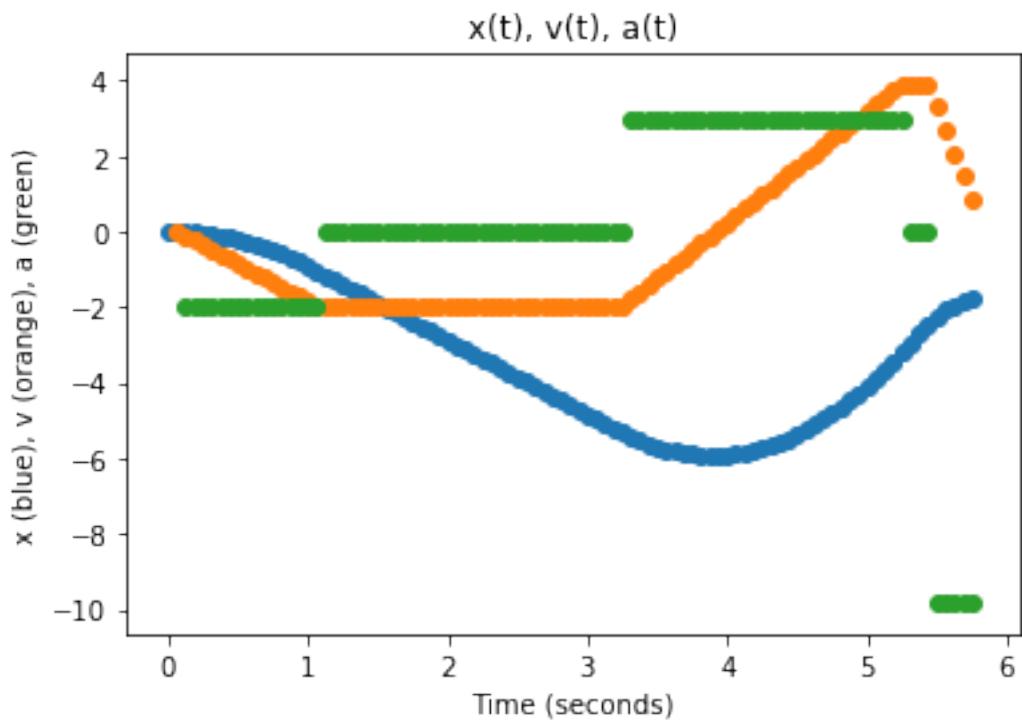
```

In [7]: # Testing part 2 - visual testing of second derivative
        # compare this output to the corresponding graph above.
speeds = get_derivative_from_data(displacements, timestamps)
accelerations = get_derivative_from_data(speeds, timestamps[1:])

plt.title("x(t), v(t), a(t)")
plt.xlabel("Time (seconds)")
plt.ylabel("x (blue), v (orange), a (green)")
plt.scatter(timestamps, displacements)
plt.scatter(timestamps[1:], speeds)

```

```
plt.scatter(timestamps[2:], accelerations)
plt.show()
```



In []:

Plotting Elevator Acceleration

March 29, 2020

1 Plotting Elevator Acceleration

I found an app for my iPhone called [Sensor Kinetics](#) that gives you direct access to the phone's IMU data, including data from the accelerometers, rate gyros, and magnetometer.

I put the phone on the floor of the Udacity elevator and pressed "Begin collecting data".

Then I pressed the elevator button for the third floor and let the app collect data as the elevator moved up to the third floor.

I want to show you a plot of that data.

In [1]: # imports and getting the data from the CSV

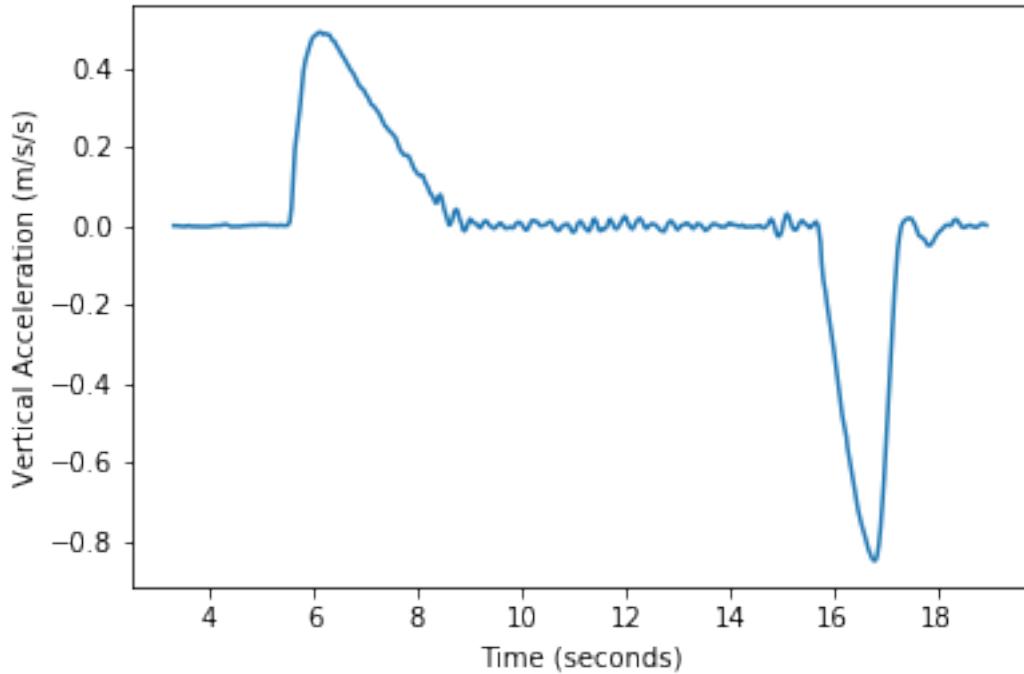
```
from matplotlib import pyplot as plt
import numpy as np
data = np.genfromtxt("elevator-lac.csv", delimiter=",")[100:570]
```

In [2]: # unpack that data

```
t, a_x, a_y, a_z = data.T
```

In [3]: # make the graph

```
plt.ylabel("Vertical Acceleration (m/s/s)")
plt.xlabel("Time (seconds)")
plt.plot(t,a_z+0.12) # the "+0.12" is to account for bias in the data
plt.show()
```



This is real acceleration data from my phone's accelerometer, and this data tells a story. Before you move on, I just want you to think about a few questions:

1.0.1 Reflection Questions

1. What's going on in that first hump between $t=6$ and $t=9$? What part of the elevator's motion does it correspond to?
2. What's going on in the mostly-flat section between $t=9$ and $t=16$?
3. Finally, why is that last bump between $t=16$ and 18 there? What part of the motion does that correspond to?

1.0.2 By the end of this lesson...

You'll be able to:

1. Use data like this to figure out how **fast** the elevator was moving at any time.
2. Use data like this to figure out how **far** this elevator moved in this trip.

Approximating the Integral

March 29, 2020

1 Approximating the Integral

This notebook is a playground to explore this idea of chopping up a function into rectangles to approximate its integral.

After this (in the next notebook) you will actually integrate the elevator accelerometer data you saw before.

1.1 Part 1 - Visualizing Rectangles

```
In [1]: from matplotlib import pyplot as plt
        import numpy as np
        import warnings
        warnings.filterwarnings('ignore')

In [2]: def show_approximate_integral(f, t_min, t_max, N):
        t = np.linspace(t_min, t_max)
        plt.plot(t, f(t))

        delta_t = (t_max - t_min) / N

        print("Approximating integral for delta_t =",delta_t, "seconds")
        box_t = np.linspace(t_min, t_max, N, endpoint=False)
        box_f_of_t = f(box_t)
        plt.bar(box_t, box_f_of_t,
                width=delta_t,
                alpha=0.5,
                facecolor="orange",
                align="edge",
                edgecolor="gray")
        plt.show()

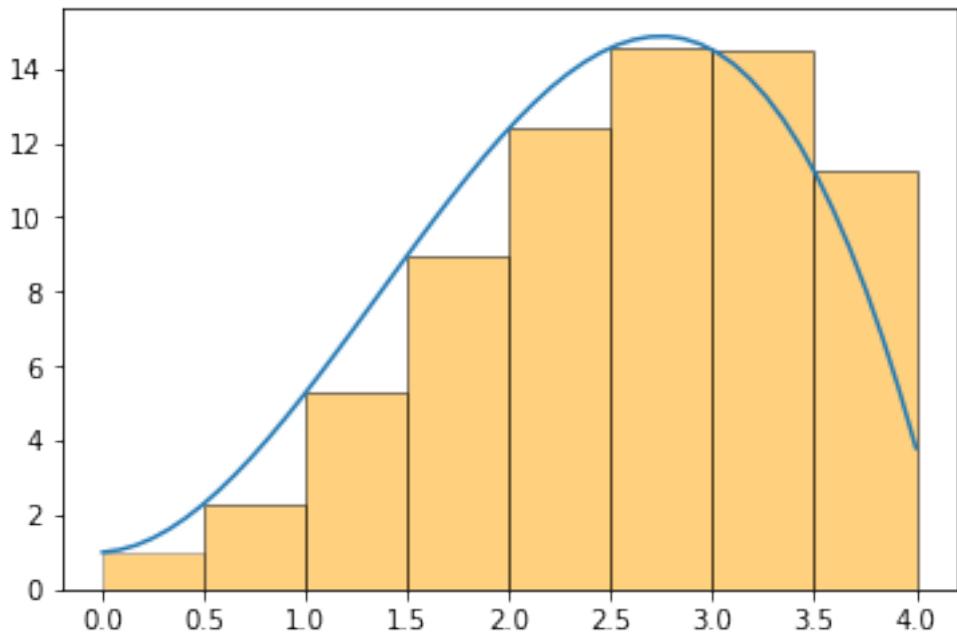
In [3]: def f1(t):
        return -1.3 * t**3 + 5.3 * t ** 2 + 0.3 * t + 1

In [10]: # TODO - increase N from 2 to 4 to 8 etc... and run
          #         this cell each time. Notice how the bars
          #         get closer and closer to approximating
```

```
#           the true area under the curve.
```

```
N = 8  
show_approximate_integral(f1, 0, 4, N)
```

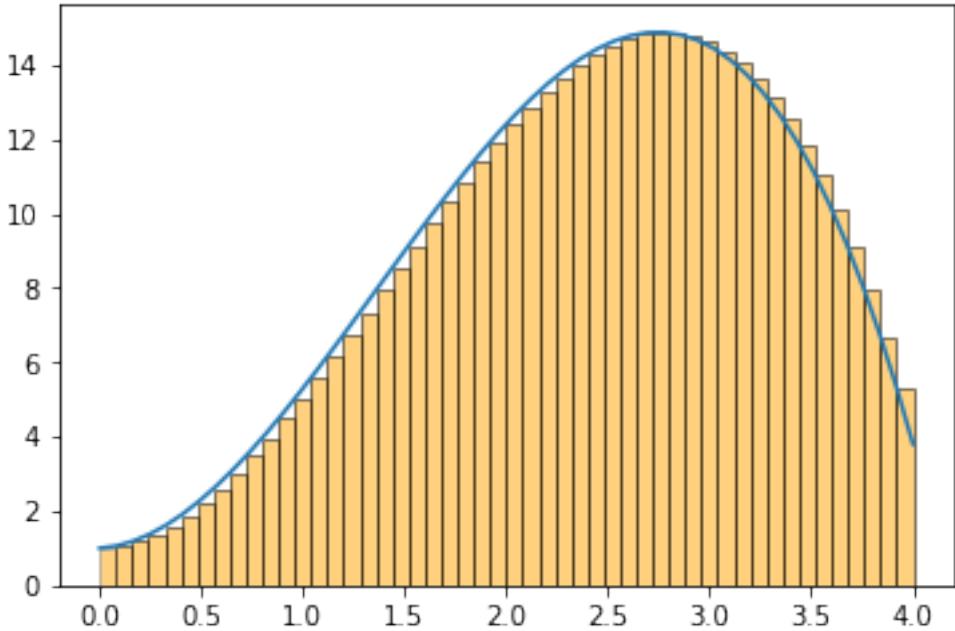
Approximating integral for delta_t = 0.5 seconds



In [11]: # When N is big, the approximation is PRETTY
close to reality.

```
N = 50  
show_approximate_integral(f1, 0, 4, N)
```

Approximating integral for delta_t = 0.08 seconds



1.2 Part 2 - Approximating Integrals

In this section, you will solve some integration "homework problems".

These are problems that you would see in a typical calculus textbook (and would be expected to solve **exactly** using clever integration techniques)

First, let's take a look at the function you'll be using to perform these approximations!

```
In [12]: def integral(f, t1, t2, dt=0.1):
    # area begins at 0.0
    area = 0.0

    # t starts at the lower bound of integration
    t = t1

    # integration continues until we reach upper bound
    while t < t2:

        # calculate the TINY bit of area associated with
        # this particular rectangle and add to total
        dA = f(t) * dt
        area += dA
        t += dt
    return area
```

I'll work through the first example for you. ##### Homework 1 - Example
Compute the following integral:

$$\int_2^4 t^2 dt$$

EXPECTED ANSWER: 18.66

In [13]: # solution step 1: define the function to be integrated

```
def f1(t):
    return t**2
```

In [14]: # solution step 2: try to solve it...
integral(f1, 2, 4)

Out[14]: 18.07000000000001

that's pretty close, but I'd like more accuracy. Let's decrease dt from the default value of 0.1...

In [15]: integral(f1, 2, 4, 0.01)

Out[15]: 18.766699999999705

In [16]: integral(f1, 2, 4, 0.001)

Out[16]: 18.67666699999851

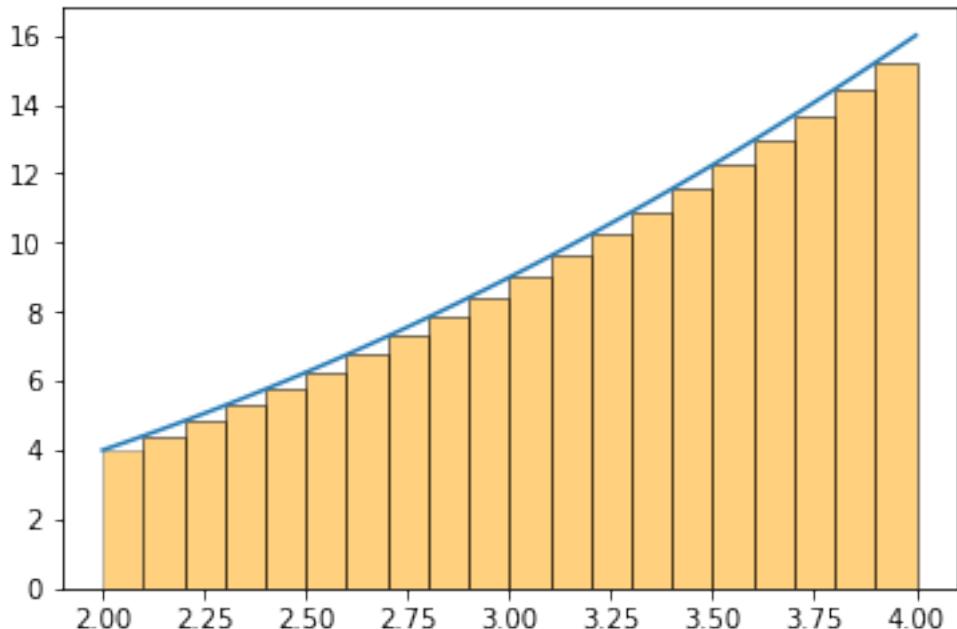
In [17]: integral(f1, 2, 4, 0.0001)

Out[17]: 18.666066670028115

Nice! We should probably use this value for dt in future calculations. Before we continue, let's just visualize this integral.

In [18]: show_approximate_integral(f1, 2, 4, 20)

Approximating integral for delta_t = 0.1 seconds



Homework 2 Compute the following integral

$$\int_{-2}^2 3t^3 - 4tdt$$

```
In [24]: # Your code here
def f2(t):
    return 3 * t ** 3 - 4 * t

integral(f2, -2, 2, 0.0001)
```

Out[24]: -6.485876072326313e-12

Homework 3 (this one can be tricky) Compute the following integral

$$\int_3^7 \frac{1}{\sqrt{2\pi \times 0.2}} e^{-\frac{(t-5)^2}{2 \times 0.2}} dt$$

```
In [30]: # Your code here
def f3(t):
    return (1.0 / np.sqrt(2 * np.pi * 0.2)) * np.exp(-(t - 5)**2 / (2 * 0.2))

integral(f3, 3, 7, 0.0001)

Out[30]: 0.99999225983512119
```

SOLUTIONS

```
In [ ]: # Solution 2
def f2(t):
    return 3 * t**3 - 4*t

integral(f2, -2, 2, 0.0001)
```

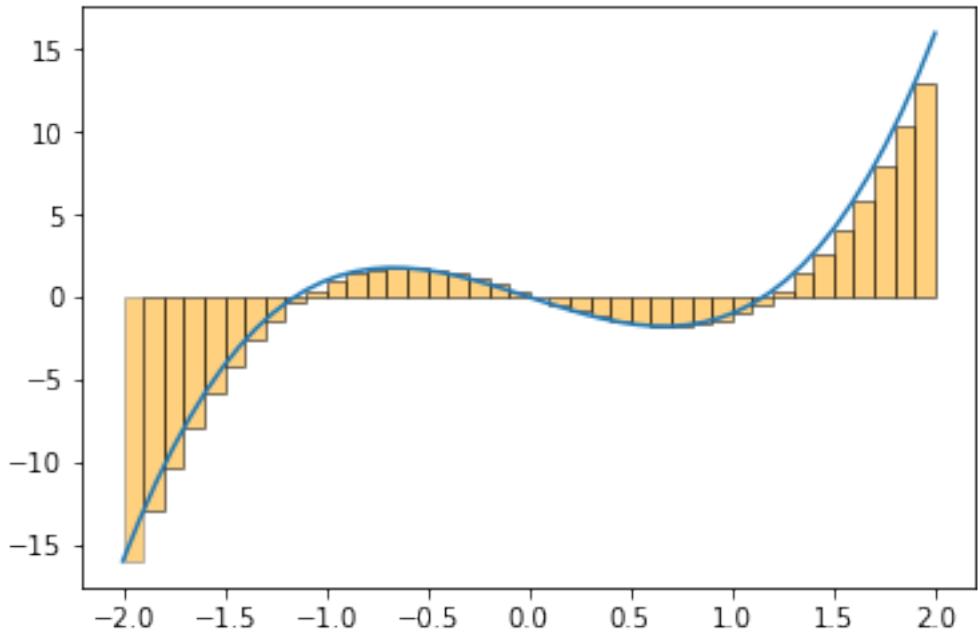
This number ends with "e-12", which means $\times 10^{-12}$

That means that this integral is 0.0000000000648 (basically zero). This shouldn't be surprising since we integrated symmetrically across an odd function (all the exponents on the t's were odd).

This means that for any positive contribution on one side of zero, there's a negative contribution to the total area on the other side of zero.

```
In [31]: show_approximate_integral(f2, -2, 2, 40)
```

Approximating integral for delta_t = 0.1 seconds



In []: # Solution 3

```
from math import sqrt, pi

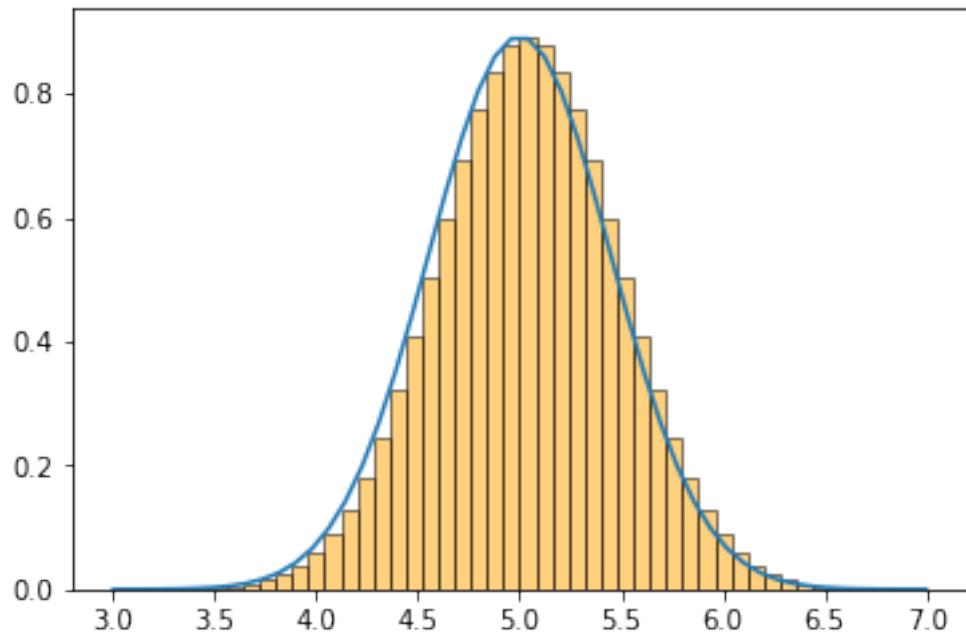
def f3(t):
    coeff    = 1.0 / sqrt(2 * pi * 0.2)
    exponent = -(t-5)**2 / (2*0.2)
    return coeff * np.exp(exponent)

integral(f3, 3, 7, 0.001)
```

That's pretty close to 1! That's because the function I just had you integrate was a Gaussian probability distribution.

In [32]: show_approximate_integral(f3, 3, 7, 50)

Approximating integral for delta_t = 0.08 seconds



In []:

Integrating Accelerometer Data

March 29, 2020

1 Integrating Accelerometer Data

In the last lesson, I gave you code for a `get_derivative_from_data` function and then later asked you to implement it yourself. We'll be doing something similar for `get_integral_from_data` here.

1.1 Part 1 - Refamiliarize $x(t) \rightarrow v(t) \rightarrow a(t)$

First, refamiliarize yourself with what you did last lesson

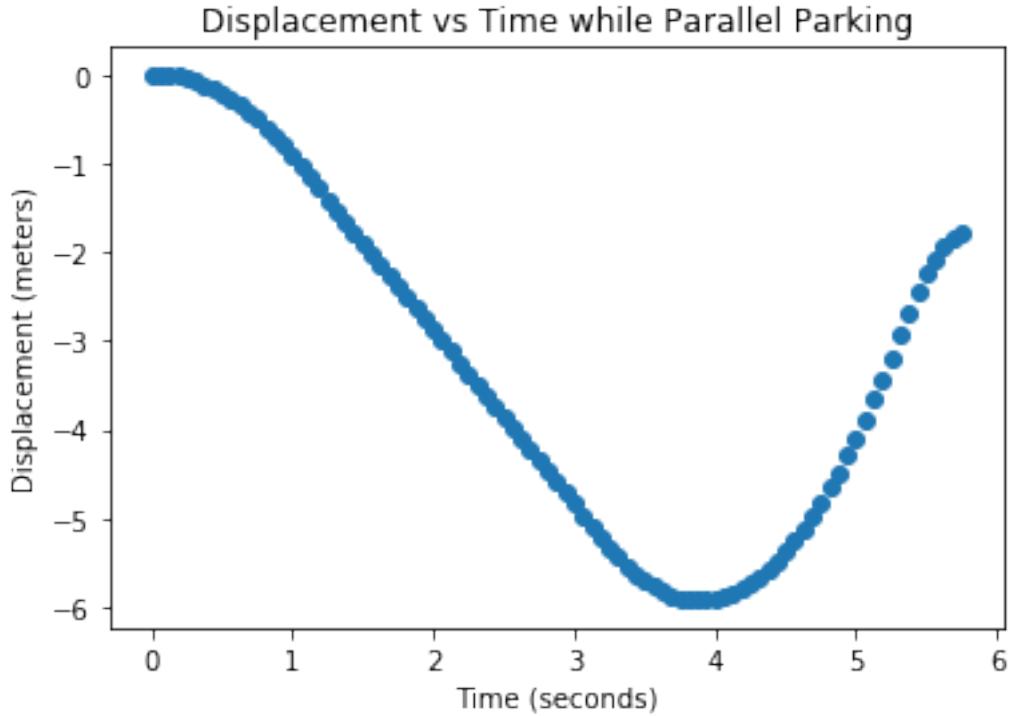
```
In [1]: from helpers import process_data, get_derivative_from_data
from matplotlib import pyplot as plt
```

```
PARALLEL_PARK_DATA = process_data("parallel_park.pickle")

TIMESTAMPS      = [row[0] for row in PARALLEL_PARK_DATA]
DISPLACEMENTS   = [row[1] for row in PARALLEL_PARK_DATA]
YAW_RATES       = [row[2] for row in PARALLEL_PARK_DATA]
ACCELERATIONS   = [row[3] for row in PARALLEL_PARK_DATA]
```

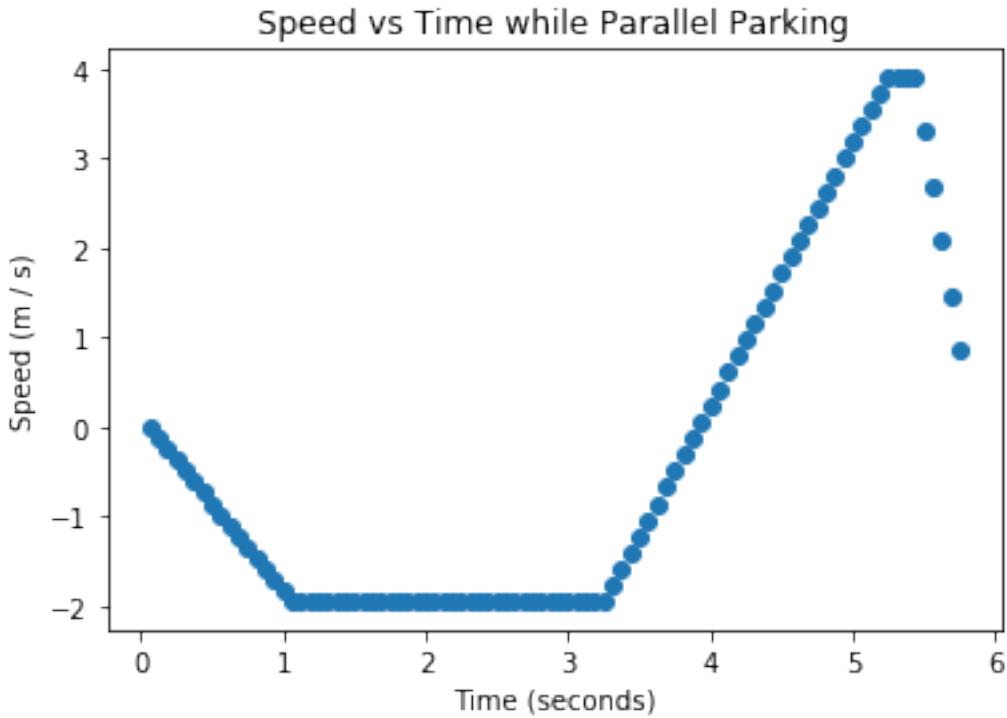
```
In [2]: # You saw this plot of displacement versus time.
```

```
plt.title("Displacement vs Time while Parallel Parking")
plt.xlabel("Time (seconds)")
plt.ylabel("Displacement (meters)")
plt.scatter(TIMESTAMPS, DISPLACEMENTS)
plt.show()
```



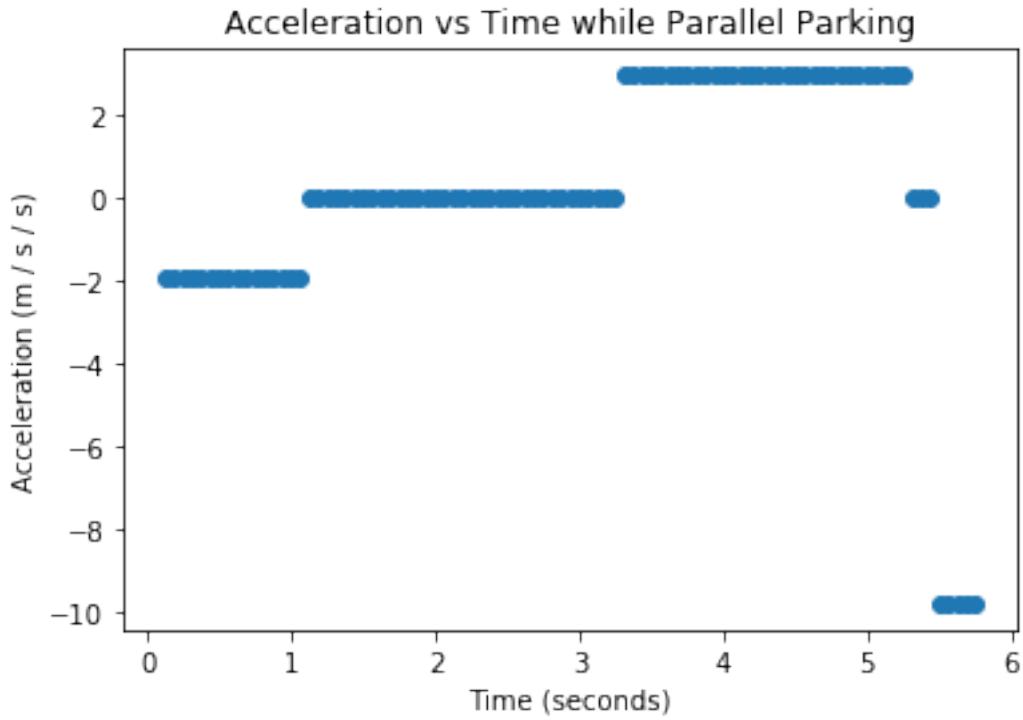
```
In [3]: # and you saw how you could differentiate this data  
# to get velocity vs time
```

```
speeds = get_derivative_from_data(DISPLACEMENTS, TIMESTAMPS)  
  
plt.title("Speed vs Time while Parallel Parking")  
plt.xlabel("Time (seconds)")  
plt.ylabel("Speed (m / s)")  
plt.scatter(TIMESTAMPS[1:], speeds)  
plt.show()
```



```
In [4]: # AND you saw how you could differentiate velocity data  
# to get acceleration data...
```

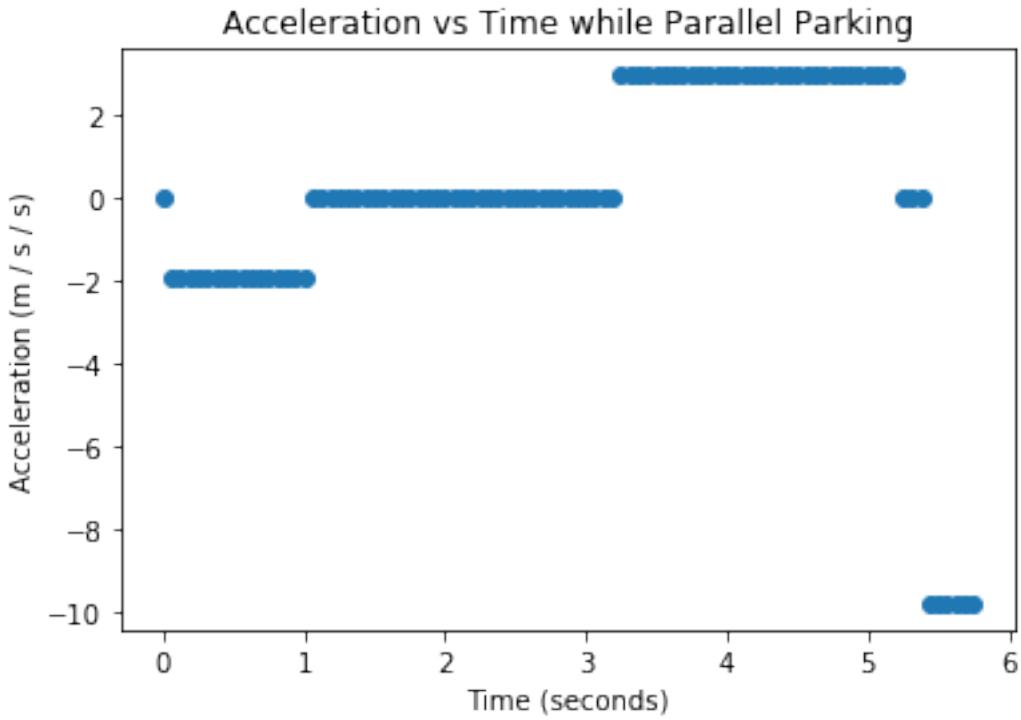
```
accels = get_derivative_from_data(speeds, TIMESTAMPS[1:])  
  
plt.title("Acceleration vs Time while Parallel Parking")  
plt.xlabel("Time (seconds)")  
plt.ylabel("Acceleration (m / s / s)")  
plt.scatter(TIMESTAMPS[2:], accels)  
plt.show()
```



1.2 Part 2 - The Other Way: $a(t) \rightarrow v(t) \rightarrow x(t)$

Now we're going to use the integral to go from acceleration data back to position data. First, let's plot the raw accelerometer data and compare it to the graph immediately above this cell.

```
In [5]: plt.title("Acceleration vs Time while Parallel Parking")
plt.xlabel("Time (seconds)")
plt.ylabel("Acceleration (m / s / s)")
plt.scatter(TIMESTAMPS, ACCELERATIONS)
plt.show()
```



As you can see they look pretty much identical... there is some missing data though that got lost in the differentiation step right near the beginning of this time window.

Now I'm going to show you a `get_integral_from_data` function. Read through the code and try to understand it because in a later notebook you will be asked to implement it yourself (without looking back here if possible).

```
In [6]: def get_integral_from_data(acceleration_data, times):
    # 1. We will need to keep track of the total accumulated speed
    accumulated_speed = 0.0

    # 2. The next lines should look familiar from the derivative code
    last_time = times[0]
    speeds = []

    # 3. Once again, we lose some data because we have to start
    #      at i=1 instead of i=0.
    for i in range(1, len(times)):

        # 4. Get the numbers for this index i
        acceleration = acceleration_data[i]
        time = times[i]

        # 5. Calculate delta t
        delta_t = time - last_time
```

```

# 6. This is an important step! This is where we approximate
#     the area under the curve using a rectangle w/ width of
#     delta_t.
delta_v = acceleration * delta_t

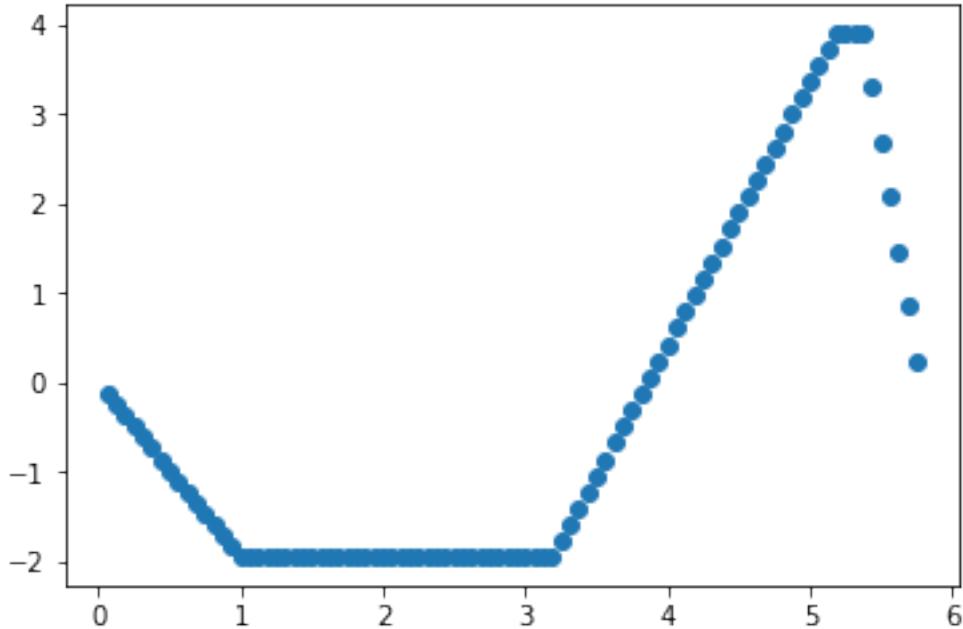
# 7. The actual speed now is whatever the speed was before
#     plus the new change in speed.
accumulated_speed += delta_v

# 8. append to speeds and update last_time
speeds.append(accumulated_speed)
last_time = time
return speeds

# 9. Now we use the function we just defined
integrated_speeds = get_integral_from_data(ACCELERATIONS, TIMESTAMPS)

# 10. Plot
plt.scatter(TIMESTAMPS[1:], integrated_speeds)
plt.show()

```



Does that graph look familiar? Scroll up and compare this to the graph that came from **differentiating** position vs time. How similar does it look to the graph we JUST made by **integrating** acceleration vs time?

Code walkthrough 1 - We're going to be summing up the area of lots of little rectangles. Each of those little rectangles will contribute to the total accumulated area (which represents speed when integrating acceleration data).

2 - 5 - These should look familiar. You saw similar code in the `get_derivative_from_data` function

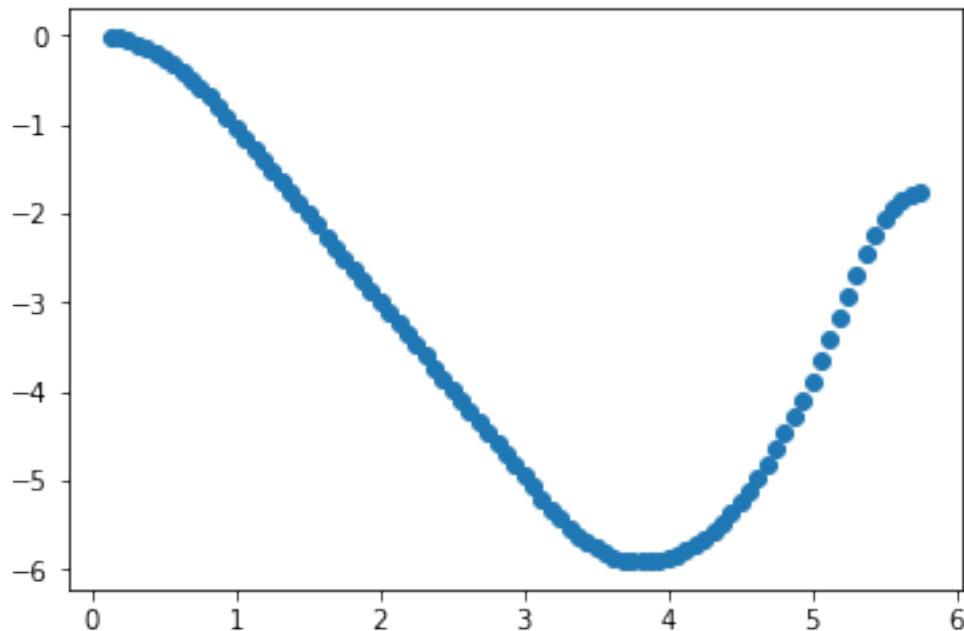
6 - This Δv is the little bit of area in whatever rectangle we are calculating in this iteration of the loop.

7 - We add this Δv to the total accumulated velocity.

8 - 10 - This should look familiar

```
In [7]: # Integrate AGAIN! Let's see what happens when we integrate  
# again to get displacement data...
```

```
integrated_displacements = get_integral_from_data(integrated_speeds,  
                                                TIMESTAMPS[1:])  
plt.scatter(TIMESTAMPS[2:], integrated_displacements)  
plt.show()
```



1.3 What to Remember

Once again, don't try to memorize this code! The key thing to remember is this:

An integral accumulates change by calculating the area of lots of little rectangles and summing them up.

Integrating Rate Gyro Data

March 29, 2020

1 Integrating Rate Gyro Data

The **yaw rate** of a vehicle can be measured by a **rate gyro**.

The yaw rate gives the rate of change of the vehicle's heading in radians per second and since a vehicle's heading is usually given by the greek letter θ (theta), yaw **rate** is given by $\dot{\theta}$ (theta dot).

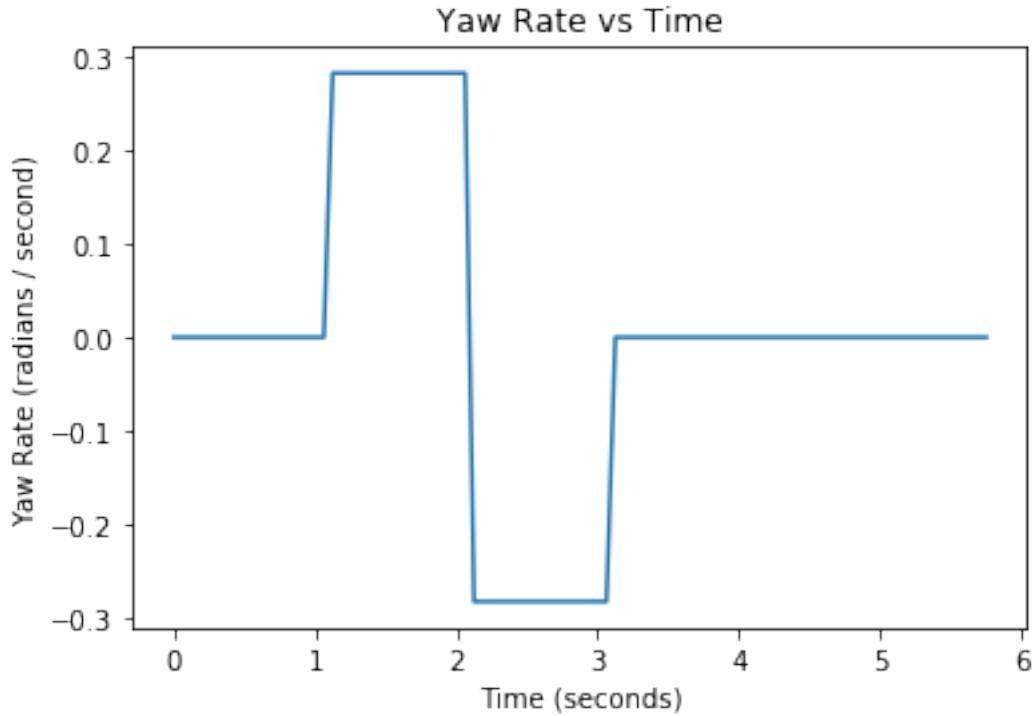
Integrating the yaw rate gives total change in heading.

```
In [1]: from helpers import process_data, get_derivative_from_data
        from matplotlib import pyplot as plt

PARALLEL_PARK_DATA = process_data("parallel_park.pickle")

TIMESTAMPS      = [row[0] for row in PARALLEL_PARK_DATA]
DISPLACEMENTS   = [row[1] for row in PARALLEL_PARK_DATA]
YAW_RATES       = [row[2] for row in PARALLEL_PARK_DATA]
ACCELERATIONS   = [row[3] for row in PARALLEL_PARK_DATA]

In [2]: plt.title("Yaw Rate vs Time")
        plt.xlabel("Time (seconds)")
        plt.ylabel("Yaw Rate (radians / second)")
        plt.plot(TIMESTAMPS, YAW_RATES)
        plt.show()
```



Here's what I make of this data

From $t=0$ to $t=1$: The yaw rate is zero so the wheels are straight (or the car isn't moving). This is when the car is backing up straight.

From $t=1$ to $t=2$: This is where the driver cuts the steering wheel hard to the right and keeps backing up. Since the yaw rate is non-zero, this means the vehicle is turning.

From $t=2$ to $t=3$: This is where the driver cuts the wheel back to the left to straighten out.

After $t=3$: Here the vehicle isn't turning so it's probably just adjusting its position within the spot by driving forward and/or backward slowly.

1.0.1 Your job

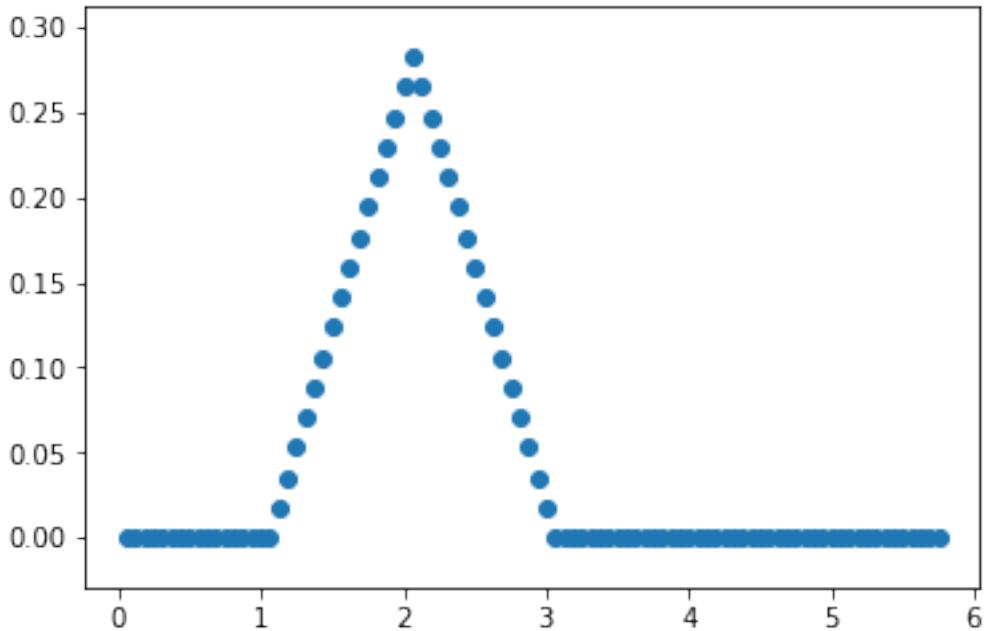
In this notebook you will write the `get_integral_from_data` function yourself and then use that function to keep track of a vehicle's heading as it drives.

First, take a look at what the integrated rate gyro data should look like when you get your function working correctly

```
In [3]: from helpers import get_integral_from_data as solution_integral

thetas = solution_integral(YAW_RATES, TIMESTAMPS)

plt.scatter(TIMESTAMPS[1:], thetas)
plt.show()
```



As you can see, the vehicle's heading is initially $\theta = 0$ radians. From $t = 1$ to $t = 2$ the heading increases to a maximum of about 0.28 radians (which is about 16 degrees).

```
In [6]: def get_integral_from_data(data, times):
    # TODO - write integration code!
    assert len(data) == len(times)

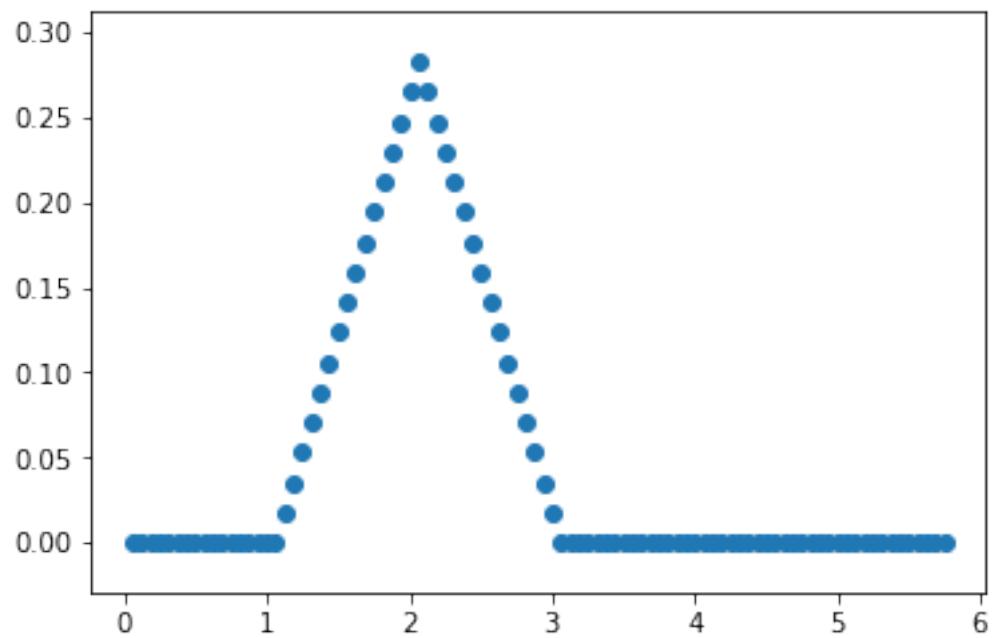
    accumulated_area = 0
    integrated_data = []
    previous_time = times[0]
    for i in range(1, len(data)):
        datum = data[i]
        time = times[i]
        DELTA_T = time - previous_time
        area = datum * DELTA_T
        accumulated_area += area
        integrated_data.append(accumulated_area)
        previous_time = time

    return integrated_data

In [7]: # Visual Testing - Compare the result of your
       # integration code to the plot above

thetas = get_integral_from_data(YAW_RATES, TIMESTAMPS)
```

```
plt.scatter(TIMESTAMPS[1:], thetas)  
plt.show()
```



In []:

Accumulating Errors

March 29, 2020

1 Accumulating Errors

Accelerometers are often **biased**. That means that even when acceleration is zero, they measure some non-zero value. Bias can be reduced through careful calibration (and buying better sensors) but it's hard to remove entirely.

In this notebook you'll briefly explore how bias tends to accumulate.

1.1 Part 1 - What is Bias?

A sensor is biased when it consistently reports a number that is too high or too low. Remember the elevator acceleration data?

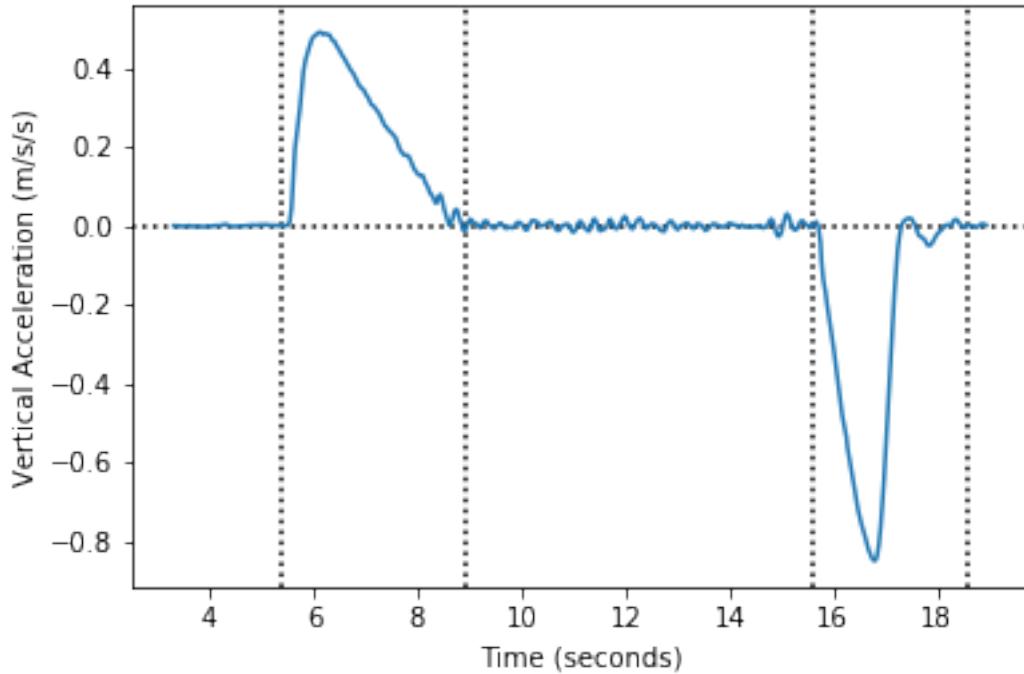
The code below is similar to what you used to plot that data.

```
In [2]: from matplotlib import pyplot as plt
        import numpy as np
        %matplotlib inline

# get elevator data
data = np.genfromtxt("elevator-lac.csv", delimiter=",")[:100:570]

# unpack that data
t, a_x, a_y, a_z = data.T
plt.ylabel("Vertical Acceleration (m/s/s)")
plt.xlabel("Time (seconds)")
region_delimiters = [5.4, 8.9, 15.6, 18.6]
for x_val in region_delimiters:
    plt.axvline(x=x_val, color="black", linestyle='dotted')

plt.axhline(y=0, color='black', linestyle='dotted')
plt.plot(t, a_z + 0.12)
plt.show()
```



Note how the flat parts of the graph (0 acceleration) do NOT line up with the dotted $a = 0$ line. But they should!

This offset is due to **bias** in the accelerometer.

If we know there's a bias we can correct it manually.

TODO - Correct the bias > Change the second to last line of code in the code cell above to the following > python > plt.plot(t, a_z+0.12) >> and then re-run the cell.

1.2 Part 2 - Visualize the Effect of Bias on discrete data

1.2.1 Rewriting get_integral_from_data with bias

In the code cell below you will see how I've rewritten this function to include an **optional** third parameter called **bias**. The default value for this parameter is zero.

When this parameter is non-zero, it simulates (or corrects) an offset in the data.

```
In [3]: def get_integral_from_data(acceleration_data, times, bias=0.0):
    """
    Numerically integrates data AND artificially introduces
    bias to that data.

    Note that the bias parameter can also be used to offset
    a biased sensor.
    """
    accumulated_speed = 0.0
    last_time = times[0]
    speeds = []
```

```

for i in range(1, len(times)):

    # THIS is where the bias is introduced. No matter what the
    # real acceleration is, this biased accelerometer adds
    # some bias to the reported value.
    acceleration = acceleration_data[i] + bias

    time = times[i]
    delta_t = time - last_time
    delta_v = acceleration * delta_t
    accumulated_speed += delta_v
    speeds.append(accumulated_speed)
    last_time = time
return speeds

```

TODO - plot velocity and acceleration with and without bias

```

In [7]: # TODO
#     0. read through the code below to get a sense for what it does.
#     1. run this cell with OFFSET = 0.0
#     2. look at the graph. Note how drastic the error is with the
#        integrated speed!
#     3. change OFFSET to OFFSET = 0.12 and re-run the cell

OFFSET = 0.12

plt.title("The Effect of Bias on Integrated Data")
plt.xlabel("Time (seconds)")
plt.ylabel("Acceleration (m/s/s) AND Speed (m/s)")

# get the elevator speeds by integrating the acceleration data
elevator_speeds = get_integral_from_data(a_z, t, OFFSET)

# plot acceleration data
plt.plot(t, a_z+OFFSET)

# plot INFERRED (from integration) speed data
plt.plot(t[1:], elevator_speeds)

# add a legend to the plot
plt.legend(["Acceleration", "Speed"])

# vertical lines
region_delimiters = [5.4, 8.9, 15.6, 18.6]
for x_val in region_delimiters:
    plt.axvline(x=x_val, color="black", linestyle='dotted')

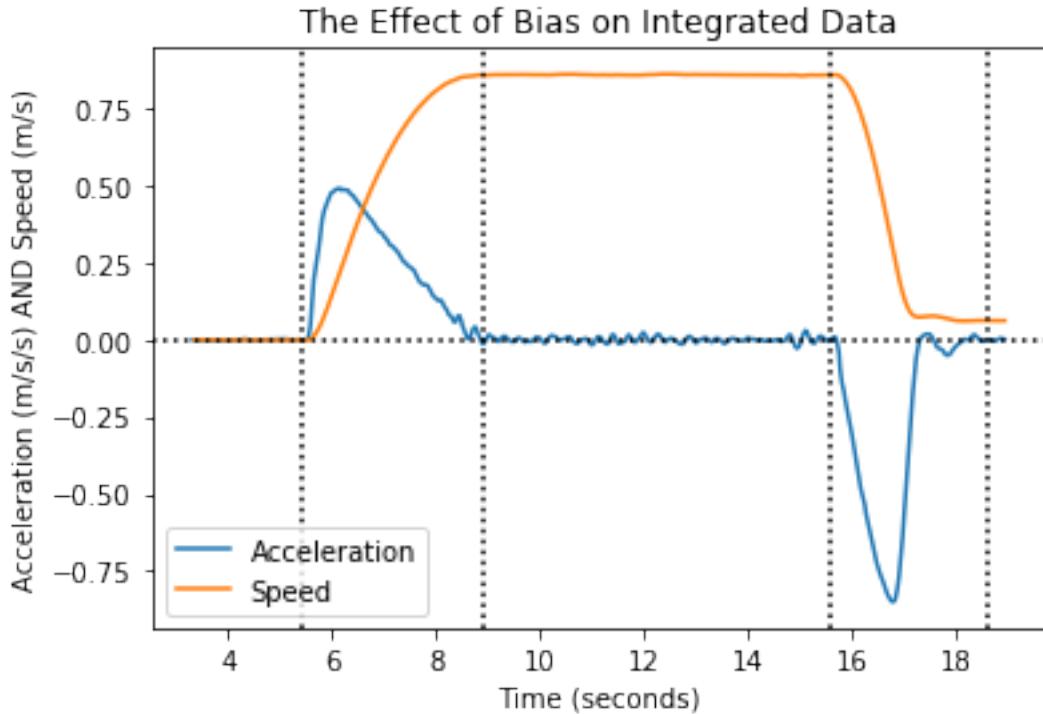
# a=0 reference line

```

```

plt.axhline(y=0, color='black', linestyle='dotted')
plt.show()

```



When `OFFSET = 0.0`, the speed graph looks absurd! It says that the speed of the elevator at the end of the journey is -1.8 m/s (clearly it should be 0.0).

And note that the effect of bias gets worse (for the speed graph) as time goes on!

When `OFFSET = 0.12` this graph looks much more reasonable (though still not perfect).

We should be more suspicious of integrated data when the integrations happen over a long period of time.

1.3 Part 3 - The Effect of Repeated Integration

You saw that bias has a bigger impact on integrated data when we give that data time to accumulate. What happens when we integrate **twice**?

```

In [8]: # This is just a helper function for integrating twice.
def double_integral(data, times, bias=0.0):
    print(bias)
    speeds = get_integral_from_data(data, times, bias)
    displacements = get_integral_from_data(speeds, times[1:])
    return displacements

```

TODO - Observe effect of bias on double integration

1. Run the cell below (`OFFSET = 0.12`). This shows the elevator ascending to a height of about 8 meters.
2. Change `OFFSET` to 0.0 and rerun. Look at how bad this inferred position is! It looks like we actually went DOWN.

In [11]: `OFFSET = 0.12`

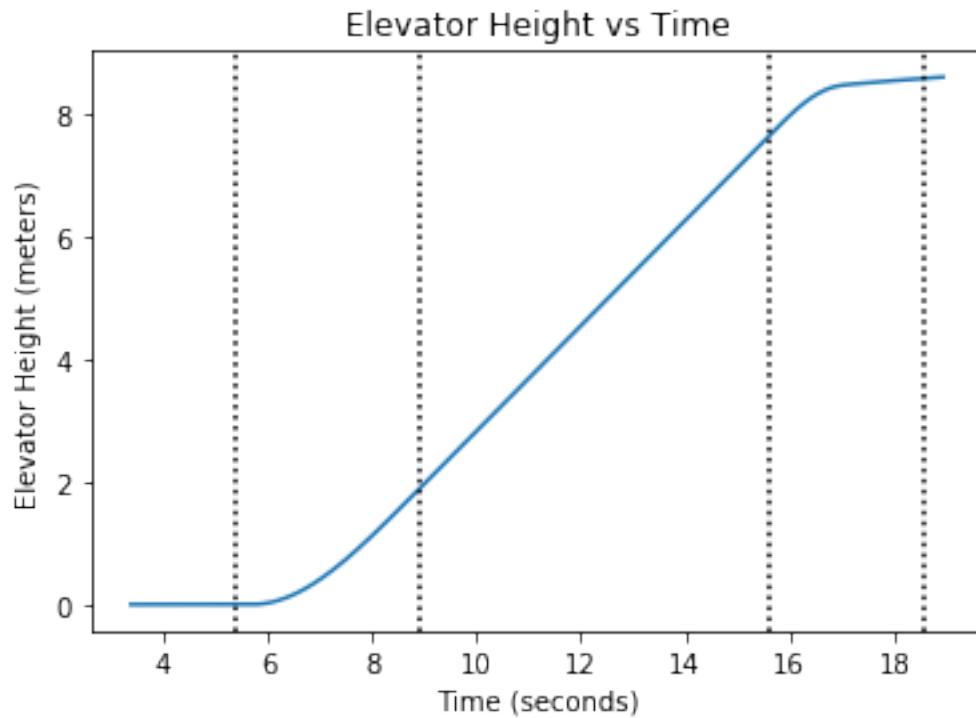
```
plt.title("Elevator Height vs Time")
plt.xlabel("Time (seconds)")
plt.ylabel("Elevator Height (meters)")

displacements = double_integral(a_z, t, OFFSET)
plt.plot(t[2:], displacements)

# vertical lines
region_delimiters = [5.4, 8.9, 15.6, 18.6]
for x_val in region_delimiters:
    plt.axvline(x=x_val, color="black", linestyle='dotted')

plt.show()
```

`0.12`



1.4 Part 4 - Parallel Parking

Same walkthrough, different data. Note that this time the accelerometer is "perfect" (because the data here is faked) and we will be using the `bias` parameter to *introduce* bias (instead of *correcting* it).

```
In [12]: from helpers import process_data

PARALLEL_PARK_DATA = process_data("parallel_park.pickle")

TIMESTAMPS      = [row[0] for row in PARALLEL_PARK_DATA]
DISPLACEMENTS   = [row[1] for row in PARALLEL_PARK_DATA]
YAW_RATES       = [row[2] for row in PARALLEL_PARK_DATA]
ACCELERATIONS   = [row[3] for row in PARALLEL_PARK_DATA]
```

Observing the Effects of bias Run the code cell below. You'll see that when bias is really small (like 0.01), the reported acceleration (blue dots) agrees very well with the actual acceleration (orange dots).

When bias is increased, things get bad fast!

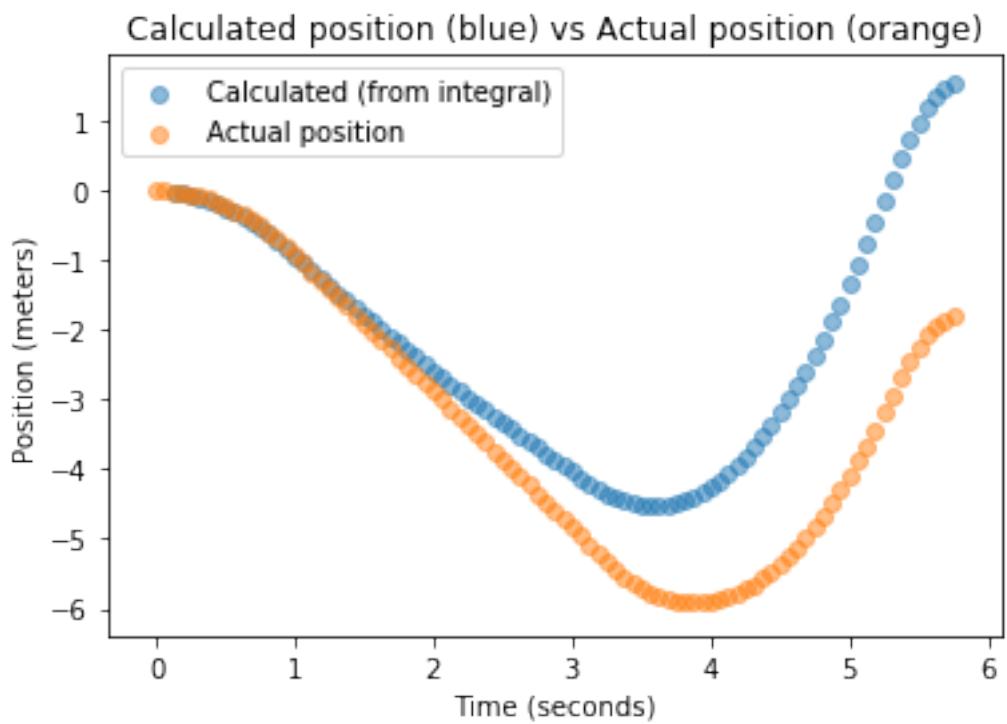
```
In [19]: # Try running this cell with BIAS set to 0.01, 0.05, 0.1, 0.2, 0.4
          # and so on. Even though these are all small numbers the effect of
          # bias tends to accumulate as the sensor runs for longer and longer.

BIAS = 0.2

integrated_displacements = double_integral(ACCELERATIONS, TIMESTAMPS, BIAS)

plt.title("Calculated position (blue) vs Actual position (orange)")
plt.xlabel("Time (seconds)")
plt.ylabel("Position (meters)")
plt.scatter(TIMESTAMPS[2:], integrated_displacements, alpha=0.5)
plt.scatter(TIMESTAMPS, DISPLACEMENTS, alpha=0.5)
plt.legend(["Calculated (from integral)", "Actual position"])
plt.show()
```

0.2



In []:

right angle robot

March 29, 2020

1 Right Angle Robot

Before we jump into trigonometry, I want to familiarize you with the Vehicle class you'll be using in this lesson (and get you thinking about motion in general).

In this notebook you will complete a Vehicle class by filling out two methods: `drive_forward` and `turn_right`.

Note that this version of a Vehicle class can ONLY face in one of 4 directions: (E)ast, (N)orth, (W)east, or (S)outh. The vehicle's current direction is stored in its `heading` property.

When you've implemented the two methods below you can run the testing cells at the bottom of the Notebook to ensure everything is behaving as expected.

1.0.1 TODO - Implement `drive_forward` and `turn_right`

Solution code is provided in the next notebook.

In [4]: `from matplotlib import pyplot as plt`

```
class Vehicle:
    def __init__(self):
        """
        Creates new vehicle at (0, 0) with a heading pointed East.
        """
        self.x      = 0 # meters
        self.y      = 0
        self.heading = "E" # Can be "N", "S", "E", or "W"
        self.history = []

    # TODO-1 - Implement this function
    def drive_forward(self, displacement):
        """
        Updates x and y coordinates of vehicle based on
        heading and appends previous (x,y) position to
        history.
        """

        # this line appends the current (x,y) coordinates
        # to the vehicle's history. Useful for plotting
```

```

# the vehicle's trajectory. You shouldn't need to
# change this line.
self.history.append((self.x, self.y))

# vehicle currently pointing east...
if self.heading == "E":
    self.x += displacement

# north
elif self.heading == "N":
    self.y += displacement

# west
elif self.heading == "W":
    self.x += - displacement

# south
else:
    self.y += - displacement

def turn(self, direction):
    if direction == "L":
        self.turn_left()
    elif direction == "R":
        self.turn_right()
    else:
        print("Error. Direction must be 'L' or 'R'")
        return

def turn_left(self):
    """
    Updates heading (for a left turn) based on current heading
    """
    next_heading = {
        "N" : "W",
        "W" : "S",
        "S" : "E",
        "E" : "N",
    }
    self.heading = next_heading[self.heading]

# TODO-2 - implement this function
def turn_right(self):
    next_heading = {
        "N" : "E",
        "W" : "N",
        "S" : "W",
    }

```

```

        "E" : "S"
    }
    self.heading = next_heading[self.heading]

def show_trajectory(self):
    """
    Creates a scatter plot of vehicle's trajectory.
    """
    X = [p[0] for p in self.history]
    Y = [p[1] for p in self.history]

    X.append(self.x)
    Y.append(self.y)

    plt.scatter(X,Y)
    plt.plot(X,Y)
    plt.show()

```

In [5]: # TESTING CODE 1

```

# instantiate vehicle
v = Vehicle()

# drive in spirals of decreasing size
v.drive_forward(8)
v.turn("L")

v.drive_forward(5)
v.turn("L")

v.drive_forward(5)
v.turn("L")

v.drive_forward(4)
v.turn("L")

v.drive_forward(4)
v.turn("L")

v.drive_forward(3)
v.turn("L")

v.drive_forward(3)
v.turn("L")

v.drive_forward(2)
v.turn("L")

```

```

v.drive_forward(2)
v.turn("L")

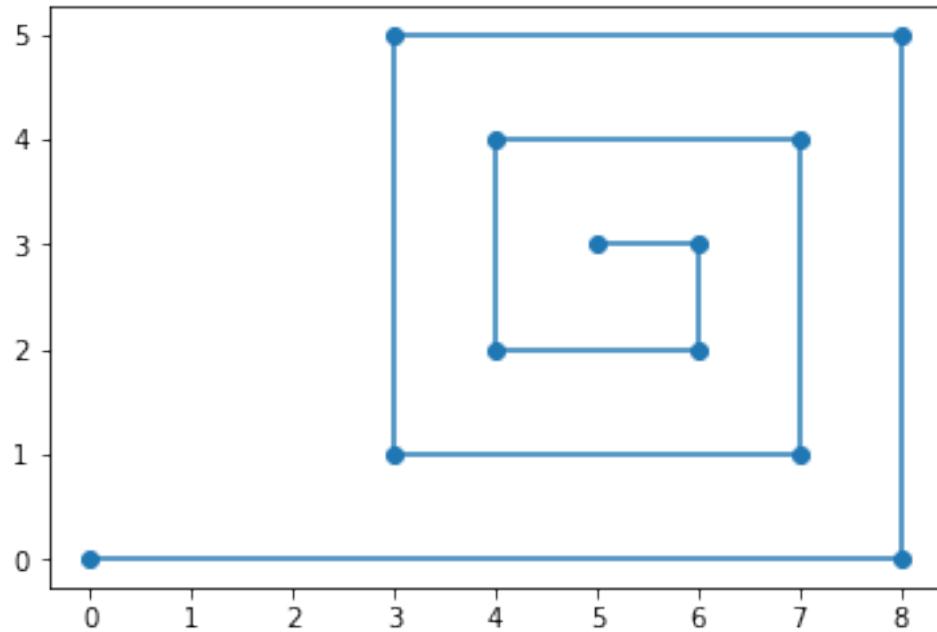
v.drive_forward(1)
v.turn("L")

v.drive_forward(1)

# show the trajectory. It should look like a spiral
v.show_trajectory()

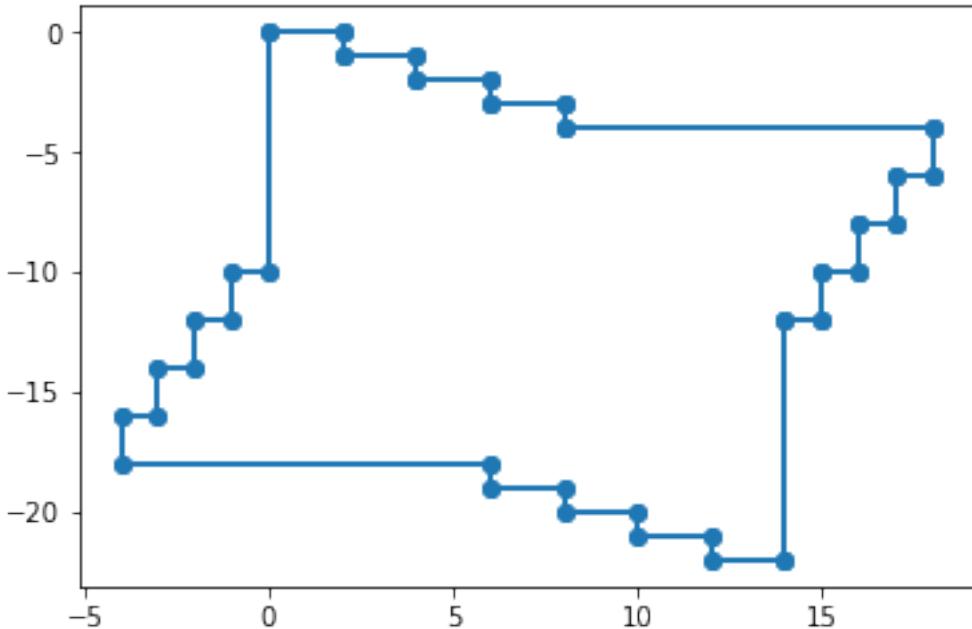
# TESTING
assert(v.x == 5)
assert(v.y == 3)
first_5 = [
    (0, 0),
    (8, 0),
    (8, 5),
    (3, 5),
    (3, 1)
]
assert(first_5 == v.history[:5])
print("Nice job! Your vehicle is behaving as expected!")

```



Nice job! Your vehicle is behaving as expected!

```
In [6]: # TESTING CODE Part 2
def test_zig_zag():
    v = Vehicle()
    for i in range(10):
        for _ in range(4):
            v.drive_forward(2)
            v.turn("R")
            v.drive_forward(1)
            v.turn("L")
        v.drive_forward(10)
        v.turn("R")
    first_six = [
        (0,0),
        (2,0),
        (2,-1),
        (4,-1),
        (4,-2),
        (6,-2)
    ]
    v.show_trajectory()
    assert(v.x == 14)
    assert(v.y == -22)
    assert(v.history[:6] == first_six)
    print("Nice job! Your vehicle passed the zig zag test.")
test_zig_zag()
```



Nice job! Your vehicle passed the zig zag test.

1.1 What's Next?

We want to be able to keep track of vehicle trajectory for ANY heading, not just the four compass directions.

Keeping Track of x and y (solution)

March 29, 2020

1 Keeping Track of x and y (solution)

This notebook contains solution code for the previous exercise.

```
In [1]: import numpy as np
        from math import pi
        from matplotlib import pyplot as plt

        # these 2 lines just hide some warning messages.
        import warnings
        warnings.filterwarnings('ignore')

        class Vehicle:
            def __init__(self):
                self.x      = 0.0 # meters
                self.y      = 0.0
                self.heading = 0.0 # radians
                self.history = []

            def drive_forward(self, displacement):
                """
                Updates x and y coordinates of vehicle based on
                heading and appends previous (x,y) position to
                history.
                """
                delta_x = displacement * np.cos(self.heading)
                delta_y = displacement * np.sin(self.heading)

                new_x = self.x + delta_x
                new_y = self.y + delta_y

                self.history.append((self.x, self.y))

                self.x = new_x
                self.y = new_y

            def set_heading(self, heading_in_degrees):
```

```

"""
Set's the current heading (in radians) to a new value
based on heading_in_degrees. Vehicle heading is always
between -pi and pi.
"""

assert(-180 <= heading_in_degrees <= 180)
rads = (heading_in_degrees * pi / 180) % (2*pi)
self.heading = rads

def turn(self, degrees):
    rads = (degrees * pi / 180)
    new_head = self.heading + rads % (2*pi)
    self.heading = new_head

def show_trajectory(self):
    """
Creates a scatter plot of vehicle's trajectory.
    """

    # get the x and y coordinates from vehicle's history
    X = [p[0] for p in self.history]
    Y = [p[1] for p in self.history]

    # don't forget to add the CURRENT x and y
    X.append(self.x)
    Y.append(self.y)

    # create scatter AND plot (to connect the dots)
    plt.scatter(X,Y)
    plt.plot(X,Y)

    plt.title("Vehicle (x, y) Trajectory")
    plt.xlabel("X Position")
    plt.ylabel("Y Position")
    plt.axes().set_aspect('equal', 'datalim')
    plt.show()

```

```
In [2]: # Use this testing code to check your code for correctness.
from testing import test_drive_forward, test_set_heading

test_set_heading(Vehicle)
test_drive_forward(Vehicle)
```

Your set_heading function looks good!
 Congratulations! Your vehicle's drive_forward method works

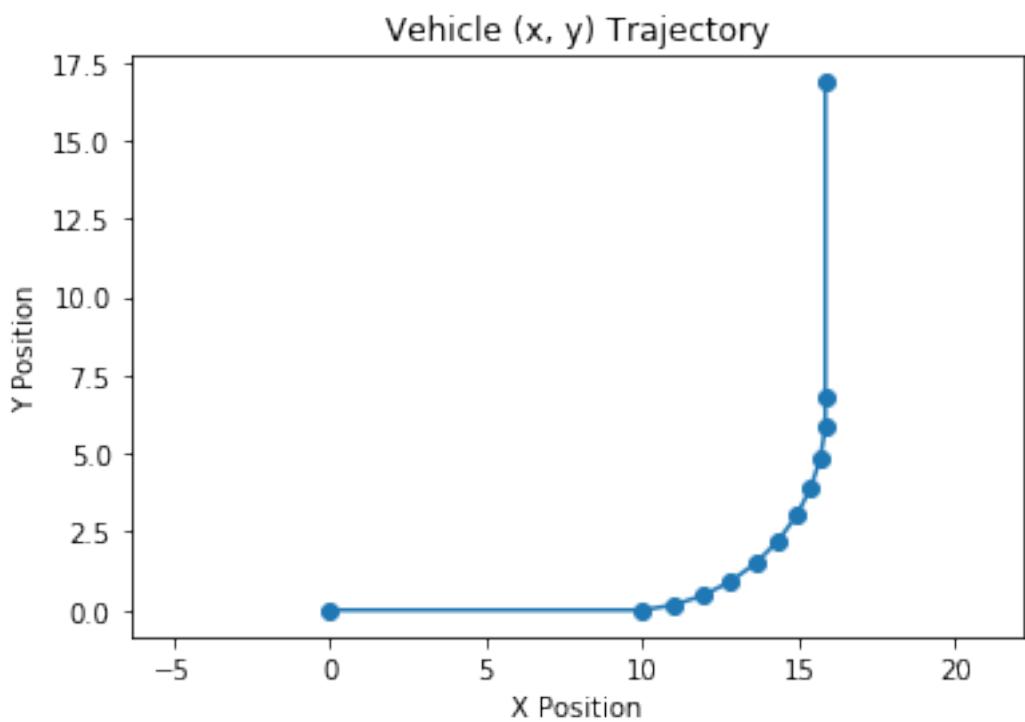
```
In [3]: # instantiate vehicle
v = Vehicle()
```

```
# drive forward 10 meters
v.drive_forward(10)

# turn left in 10 increments of 9 degrees each.
for _ in range(10):
    v.turn(9.0)
    v.drive_forward(1)

v.drive_forward(10)

v.show_trajectory()
```



In []:

Raw Input Data

The data you'll be working with has been preprocessed from CSVs that looks like this:

timestamp	displacement	yaw_rate	acceleration
0.0	0	0.0	0.0
0.25	0.0	0.0	19.6
0.5	1.225	0.0	19.6
0.75	3.675	0.0	19.6
1.0	7.35	0.0	19.6
1.25	12.25	0.0	0.0
1.5	17.15	-2.82901631903	0.0
1.75	22.05	-2.82901631903	0.0
2.0	26.95	-2.82901631903	0.0
2.25	31.85	-2.82901631903	0.0
2.5	36.75	-2.82901631903	0.0
2.75	41.65	-2.82901631903	0.0
3.0	46.55	-2.82901631903	0.0
3.25	51.45	-2.82901631903	0.0
3.5	56.35	-2.82901631903	0.0

This data is currently saved in a file called `trajectory_example.pickle`. It can be loaded using a helper function we've provided (demonstrated below):

```
In [28]: from helpers import process_data
%matplotlib inline

data_list = process_data("trajectory_example.pickle")

for entry in data_list:
    print(entry)

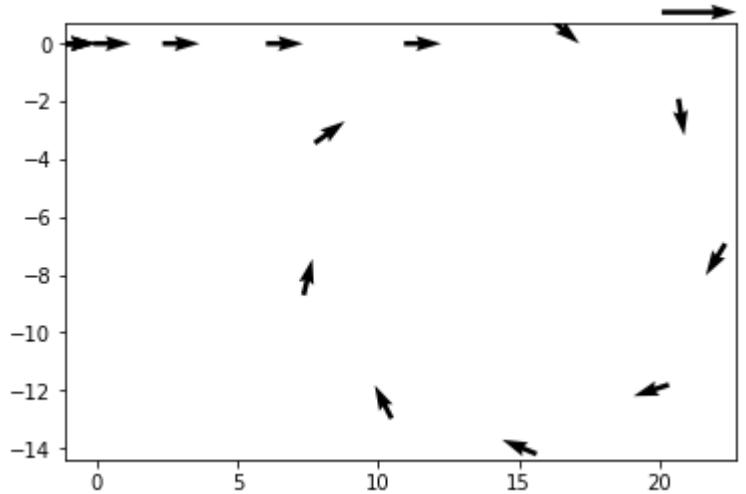
(0.0, 0, 0.0, 0.0)
(0.25, 0.0, 0.0, 19.600000000000001)
(0.5, 1.2250000000000001, 0.0, 19.600000000000001)
(0.75, 3.675000000000003, 0.0, 19.600000000000001)
(1.0, 7.350000000000005, 0.0, 19.600000000000001)
(1.25, 12.25, 0.0, 0.0)
(1.5, 17.14999999999999, -2.8290163190291664, 0.0)
(1.75, 22.04999999999997, -2.8290163190291664, 0.0)
(2.0, 26.94999999999996, -2.8290163190291664, 0.0)
(2.25, 31.84999999999994, -2.8290163190291664, 0.0)
(2.5, 36.74999999999993, -2.8290163190291664, 0.0)
(2.75, 41.64999999999991, -2.8290163190291664, 0.0)
(3.0, 46.54999999999999, -2.8290163190291664, 0.0)
(3.25, 51.44999999999989, -2.8290163190291664, 0.0)
(3.5, 56.34999999999987, -2.8290163190291664, 0.0)
```

as you can see, each entry in `data_list` contains four fields. Those fields correspond to `timestamp` (seconds), `displacement` (meters), `yaw_rate` (rads / sec), and `acceleration` (m/s/s).

The Point of this Project!

Data tells a story but you have to know how to find it!

Contained in the data above is all the information you need to reconstruct a fairly complex vehicle trajectory. After processing **this** exact data, it's possible to generate this plot of the vehicle's X and Y position:



as you can see, this vehicle first accelerates forwards and then turns right until it almost completes a full circle turn.

Data Explained

timestamp - Timestamps are all measured in seconds. The time between successive timestamps (Δt) will always be the same *within* a trajectory's data set (but not *between* data sets).

displacement - Displacement data from the odometer is in meters and gives the **total** distance traveled up to this point.

yaw_rate - Yaw rate is measured in radians per second with the convention that positive yaw corresponds to *counter-clockwise* rotation.

acceleration - Acceleration is measured in $\frac{m/s}{s}$ and is always **in the direction of motion of the vehicle** (forward).

NOTE - you may not need to use all of this data when reconstructing vehicle trajectories.

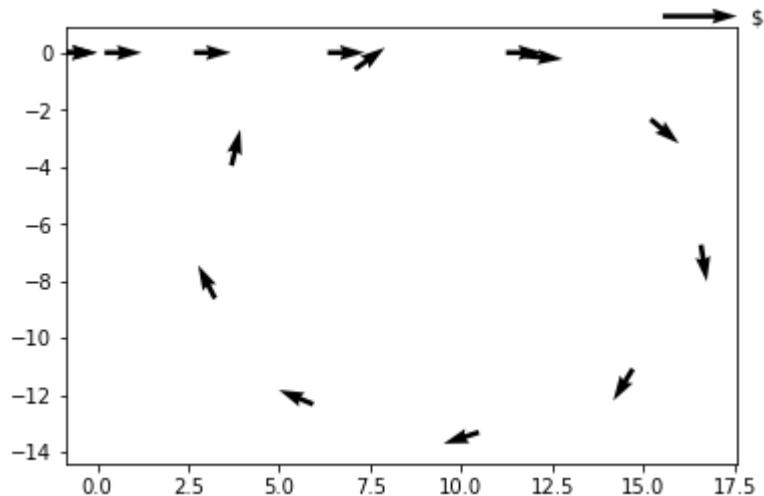
Your Job

Your job is to complete the following functions, all of which take a processed `data_list` (with N entries, each Δt apart) as input:

- `get_speeds` - returns a length N list where entry i contains the speed (m/s) of the vehicle at $t = i \times \Delta t$
- `get_headings` - returns a length N list where entry i contains the heading (radians, $0 \leq \theta < 2\pi$) of the vehicle at $t = i \times \Delta t$
- `get_x_y` - returns a length N list where entry i contains an (x, y) tuple corresponding to the x and y coordinates (meters) of the vehicle at $t = i \times \Delta t$
- `show_x_y` - generates an x vs. y scatter plot of vehicle positions.

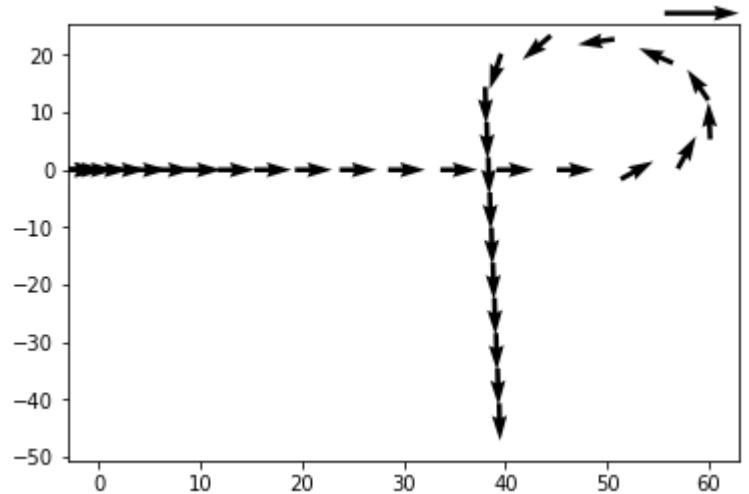
```
In [29]: # I've provided a solution file called solution.py
# You are STRONGLY encouraged to NOT look at the code
# until after you have solved this yourself.
#
# You SHOULD, however, feel free to USE the solution
# functions to help you understand what your code should
# be doing. For example...
from helpers import process_data
import solution

data_list = process_data("trajectory_example.pickle")
solution.show_x_y(data_list)
```

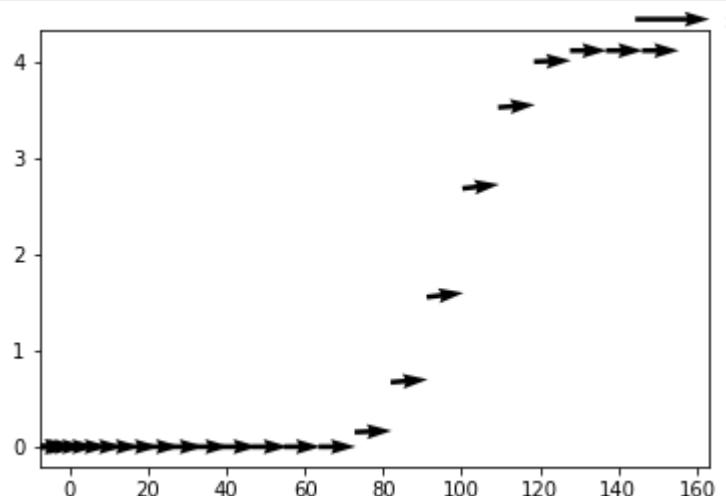


```
In [30]: # What about the other trajectories?
```

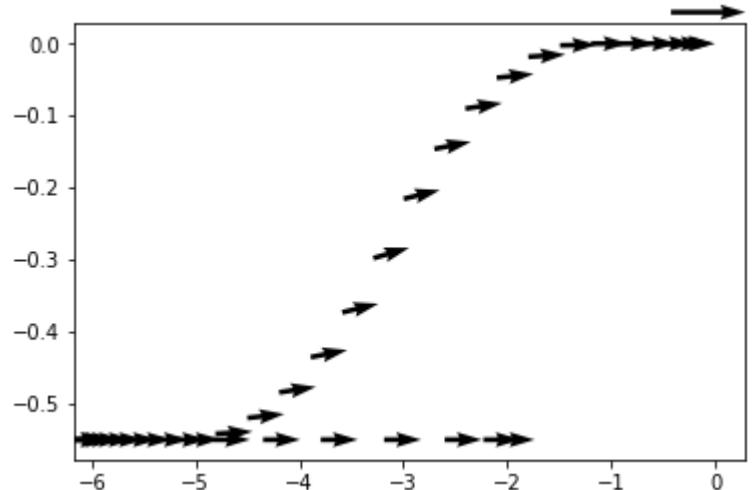
```
three_quarter_turn_data = process_data("trajectory_1.pickle")
solution.show_x_y(three_quarter_turn_data, increment=10)
```



```
In [31]: merge_data = process_data('trajectory_2.pickle')
solution.show_x_y(merge_data,increment=10)
```



```
In [32]: parallel_park = process_data("trajectory_3.pickle")
solution.show_x_y(parallel_park,increment=5)
```



How do you make those cool arrows?!

I did a Google search for "python plot grid of arrows" and the second result led me to some [demonstration code](#) (https://matplotlib.org/examples/pylab_examples/quiver_demo.html) that was really helpful.

Testing Correctness

Testing code is provided at the bottom of this notebook. Note that only `get_speeds` , `get_x_y` , and `get_headings` are tested automatically. You will have to "test" your `show_x_y` function by manually comparing your plots to the expected plots.

Initial Vehicle State

The vehicle always begins with all state variables equal to zero. This means `x` , `y` , `theta` (heading), `speed` , `yaw_rate` , and `acceleration` are 0 at $t=0$.

Your Code!

Complete the functions in the cell below. I recommend completing them in the order shown. Use the cells at the end of the notebook to test as you go.

```
In [33]: from matplotlib import pyplot as plt
from math import pi, sin, cos
import numpy as np

def get_speeds(data_list):
    last_time = 0.0
    last_disp = 0.0
    speeds = []
    speeds.append(0.0)

    for entry in data_list[1:]:
        ts, disp, yaw, acc = entry

        dx = disp - last_disp
        dt = ts - last_time
        if dt < 0.0001:
            print("error! dt is too small")
            speeds.append(0.0)
            continue
        v = dx / dt

        speeds.append(v)

        last_time = ts
        last_disp = disp
    return speeds

def get_headings(data_list):
    last_time = 0.0
    theta = 0.0
    thetas = []
    thetas.append(0.0)

    for entry in data_list[1:]:
        time, disp, yaw_rate, acc = entry
        dt = time - last_time
        d_theta = dt * yaw_rate
        theta += d_theta
        theta %= 2*pi
        thetas.append(theta)
        last_time = time

    return thetas

def get_x_y(data_list):
    speeds = get_speeds(data_list)
    thetas = get_headings(data_list)
    x = 0.0
    y = 0.0
    last_time = 0
    XY = [(x, y)]
    for i in range(1, len(data_list)):
        speed = speeds[i]
        theta = thetas[i]
        entry = data_list[i]
        time, disp, yaw_rate, acc = entry
```

```

        dt = time - last_time
        D = speed * dt
        dx = D * cos(theta)
        dy = D * sin(theta)
        x += dx
        y += dy
        XY.append((x, y))
        last_time = time
    return XY

def show_x_y(data_list, increment = 1):
    XY = get_x_y(data_list)
    headings = get_headings(data_list)
    X = [d[0] for d in XY]
    Y = [d[1] for d in XY]
    h_x = np.cos(headings)
    h_y = np.sin(headings)
    Q = plt.quiver(X[::increment], Y[::increment],
                    h_x[::increment], h_y[::increment],
                    units = 'x', pivot = 'tip')
    qk = plt.quiverkey(Q, 0.9, 0.9, 2, r'$1 \frac{m}{s}$',
                        labelpos = 'E', coordinates = 'figure')
    plt.show()

```

Testing

Test your functions by running the cells below.

In [34]: `from testing import test_get_speeds, test_get_x_y, test_get_headings`

`test_get_speeds(get_speeds)`

PASSED test of `get_speeds` function!

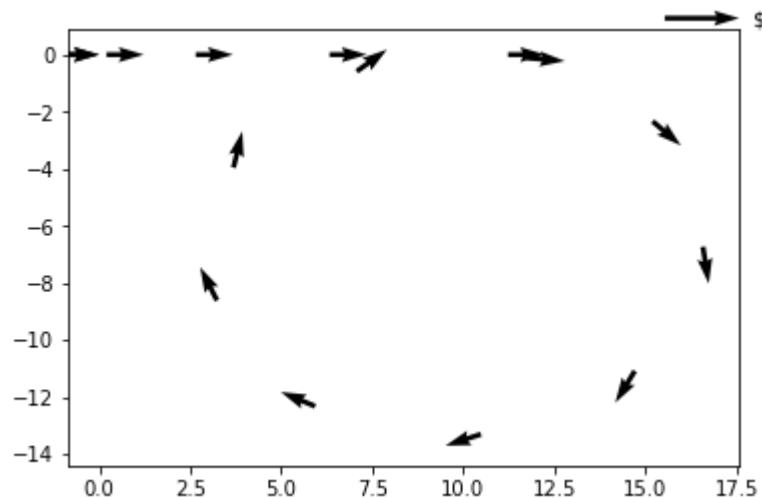
In [35]: `test_get_headings(get_headings)`

PASSED test of `get_headings` function!

In [36]: `test_get_x_y(get_x_y)`

PASSED test of `get_x_y` function!

In [37]: `show_x_y(data_list)`



In []:

In []:

Images as Numerical Data

March 30, 2020

1 Images as Grids of Pixels

1.0.1 Import resources

```
In [1]: import numpy as np
        import matplotlib.image as mpimg # for reading in images

        import matplotlib.pyplot as plt
        import cv2 # computer vision library

        %matplotlib inline
```

1.0.2 Read in and display the image

```
In [2]: # Read in the image
        image = mpimg.imread('images/waymo_car.jpg')

        # Print out the image dimensions
        print('Image dimensions:', image.shape)

        # Change from color to grayscale
        gray_image = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

        plt.imshow(gray_image, cmap='gray')

Image dimensions: (427, 640, 3)

Out[2]: <matplotlib.image.AxesImage at 0x7ff7fb36a358>
```



```
In [3]: # Prints specific grayscale pixel values
# What is the pixel value at x = 400 and y = 300 (on the body of the car)?
x = 400
y = 300
print(gray_image[y,x])
```

159

```
In [4]: # Finds the maximum and minimum grayscale values in this image
```

```
max_val = np.amax(gray_image)
min_val = np.amin(gray_image)

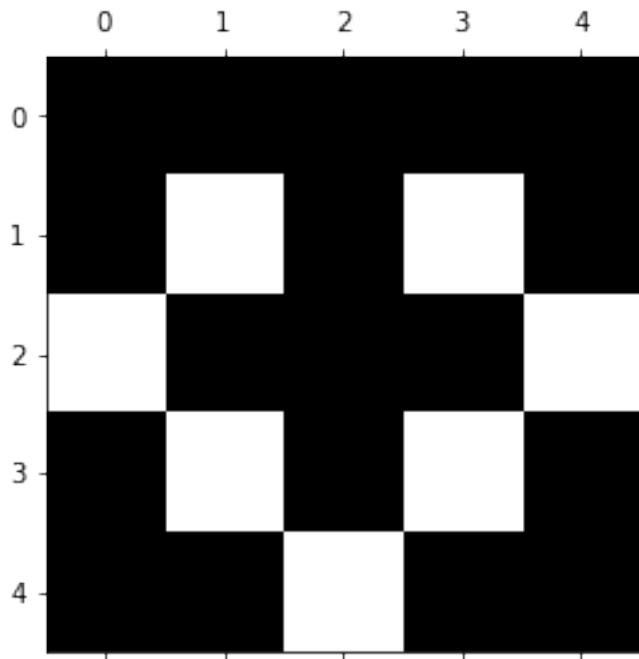
print('Max: ', max_val)
print('Min: ', min_val)
```

```
Max: 255
Min: 2
```

```
In [6]: # Create a 5x5 image using just grayscale, numerical values
tiny_image = np.array([[0, 0, 0, 0, 0],
                      [0, 255, 0, 255, 0],
```

```
[255, 0, 0, 0, 255],  
[0, 255, 0, 255, 0],  
[0, 0, 255, 0, 0]])  
  
# To show the pixel grid, use matshow  
plt.matshow(tiny_image, cmap='gray')  
  
## TODO: See if you can draw a tiny smiley face or something else!  
## You can change the values in the array above to do this
```

In [6]: <matplotlib.image.AxesImage at 0x7ff7c21e32b0>



In []:

Visualizing RGB Channels

March 30, 2020

1 RGB colorspace

1.0.1 Import resources

```
In [1]: import matplotlib.pyplot as plt  
        import matplotlib.image as mpimg  
  
        %matplotlib inline
```

1.0.2 Read in an image

```
In [2]: # Read in the image  
        image = mpimg.imread('images/wa_state_highway.jpg')  
  
        plt.imshow(image)  
  
Out[2]: <matplotlib.image.AxesImage at 0x7fd22e757d68>
```



1.0.3 RGB channels

Visualize the levels of each color channel. Pay close attention to the traffic signs!

In [4]: # Isolate RGB channels

```
r = image[:, :, 0]
g = image[:, :, 1]
b = image[:, :, 2]

# Visualize the individual color channels
f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,10))
ax1.set_title('R channel')
ax1.imshow(r, cmap='gray')
ax2.set_title('G channel')
ax2.imshow(g, cmap='gray')
ax3.set_title('B channel')
ax3.imshow(b, cmap='gray')

## Which area has the lowest value for red? What about for blue?
```

Out[4]: <matplotlib.image.AxesImage at 0x7fd22d514400>



In []:

Pre-processing Examples

March 30, 2020

1 Cropping and Resizing

1.0.1 Import resources

```
In [1]: import matplotlib.pyplot as plt
        import matplotlib.image as mpimg

        import numpy as np
        import cv2

        %matplotlib inline
```

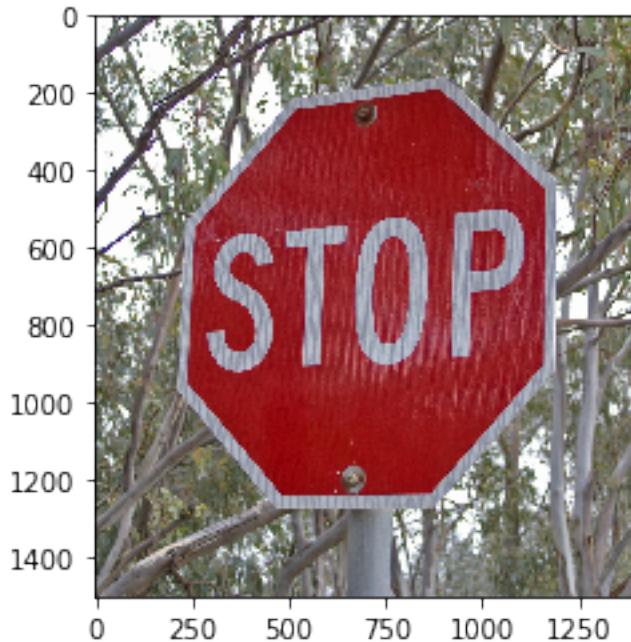
1.0.2 Read in the first image of a stop sign

```
In [2]: # Read in the image
        stop1 = mpimg.imread('images/stop_sign.jpg')

        print('Image shape: ', stop1.shape)
        plt.imshow(stop1)

Image shape:  (1500, 1389, 3)
```

```
Out[2]: <matplotlib.image.AxesImage at 0x7f69c5415080>
```



1.0.3 Read in the second image

```
In [3]: # Read in the image
stop2 = mpimg.imread('images/stop_sign2.jpg')

print('Image shape: ', stop2.shape)
plt.imshow(stop2)

Image shape:  (640, 960, 3)
```

```
Out[3]: <matplotlib.image.AxesImage at 0x7f698fb38588>
```



1.1 Crop this image so that it resembles the first image

In [4]: # To crop an image, you can use image slicing
which is just slicing off a portion of the image array

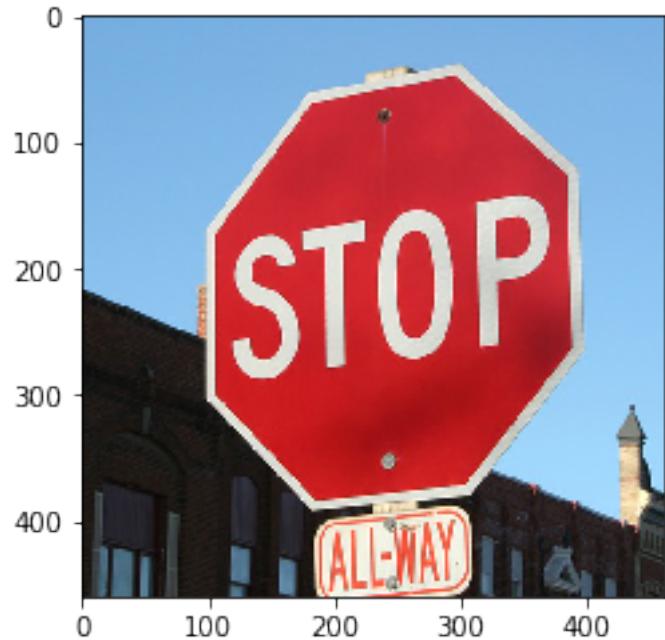
```
# Make a copy of the image to manipulate  
image_crop = np.copy(stop2)
```

```
# Define how many pixels to slice off the sides of the original image  
row_crop = 90  
col_crop = 250
```

```
# Using image slicing, subtract the row_crop from top/bottom and col_crop from left/right  
image_crop = stop2[row_crop:-row_crop, col_crop:-col_crop, :]
```

```
plt.imshow(image_crop)
```

Out [4]: <matplotlib.image.AxesImage at 0x7f698fb1bd30>



1.2 Resize the cropped image to be the same as the first

Recall that the shape of the first image is (1500, 1389, 3).

```
In [5]: # Use OpenCV's resize function
        standardized_im = cv2.resize(image_crop, (1389, 1500))

        print('Image shape: ', standardized_im.shape)

        # Plot the two images side by side
        f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
        ax1.set_title('Stop sign 1')
        ax1.imshow(stop1)
        ax2.set_title('Standardized stop sign 2')
        ax2.imshow(standardized_im)

Image shape: (1500, 1389, 3)
```

```
Out[5]: <matplotlib.image.AxesImage at 0x7f698fa36908>
```



1.3 Compare these images

Now you should be able to compare these images pixel by pixel! We'll add up the red channel values in each image, and they should be fairly close, which means we can use this similarity to characterize these images.

For comparison, we'll also show what happens when you perform this comparison using the original `stop_sign2.jpg`.

```
In [6]: # Sum all the red channel values and compare
red_sum1 = np.sum(stop1[:, :, 0])
red_sum2 = np.sum(standardized_im[:, :, 0])

print('Sum of all red pixel values in the first stop sign image: ', red_sum1)
print('Sum of red pixel values in the second, standardized image: ', red_sum2)

red_sum_orig = np.sum(stop2[:, :, 0])

print('\nFor comparison, the sum of red pixels in the non-standardized image: ', red_sum)
```

Sum of all red pixel values in the first stop sign image: 294214944
 Sum of red pixel values in the second, standardized image: 300109548

For comparison, the sum of red pixels in the non-standardized image: 67011798

```
In [ ]: ## Note: you have been given two other images:
# `yield.jpg` and `walk.jpg`
# You can look at these images and see what kind of RGB values might distinguish them
```

Green Screen Car

March 30, 2020

1 Color Masking, Green Screen

1.0.1 Import resources

```
In [1]: import matplotlib.pyplot as plt
        import matplotlib.image as mpimg

        import numpy as np
        import cv2

%matplotlib inline
```

1.0.2 Read in and display the image

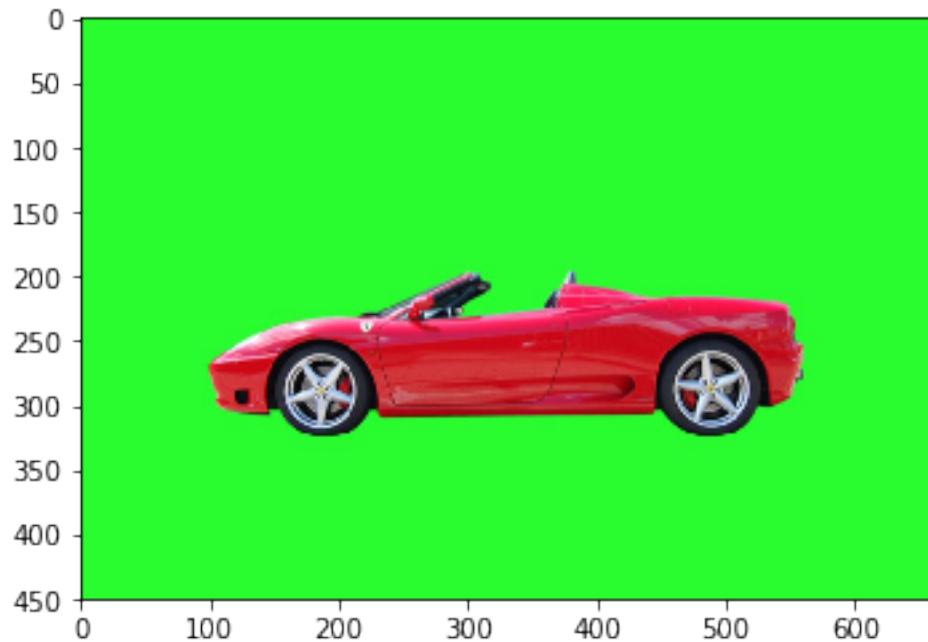
```
In [2]: # Read in the image
        image = mpimg.imread('images/car_green_screen.jpg')

        # Print out the image dimensions (height, width, and depth (color))
        print('Image dimensions:', image.shape)

        # Display the image
        plt.imshow(image)
```

Image dimensions: (450, 660, 3)

Out[2]: <matplotlib.image.AxesImage at 0x7f2e16b68320>



1.0.3 Define the color threshold

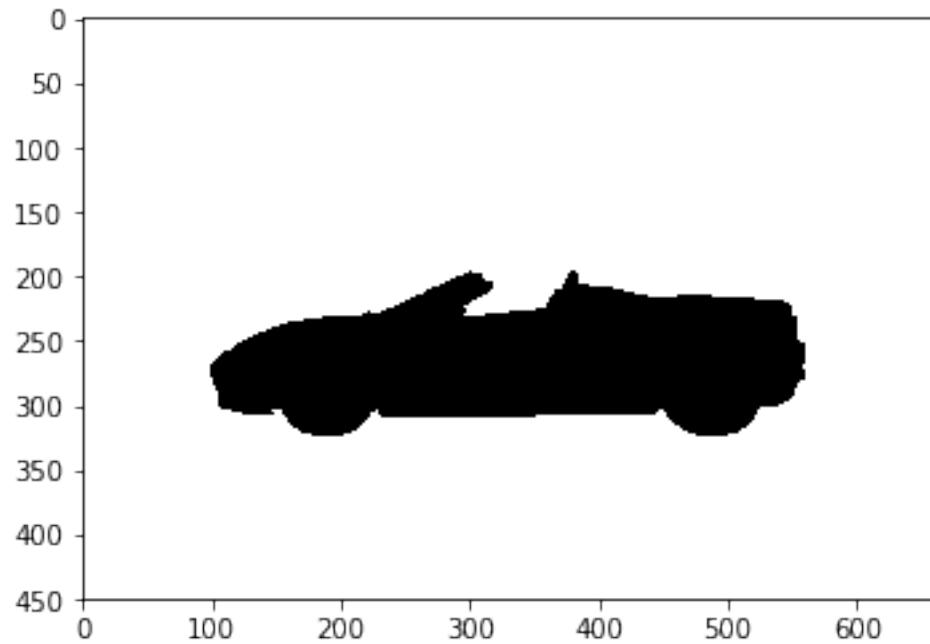
```
In [3]: # Define our color selection boundaries in RGB values
lower_green = np.array([0,180,0])
upper_green = np.array([100,255,100])
```

1.0.4 Create a mask

```
In [4]: # Define the masked area
mask = cv2.inRange(image, lower_green, upper_green)

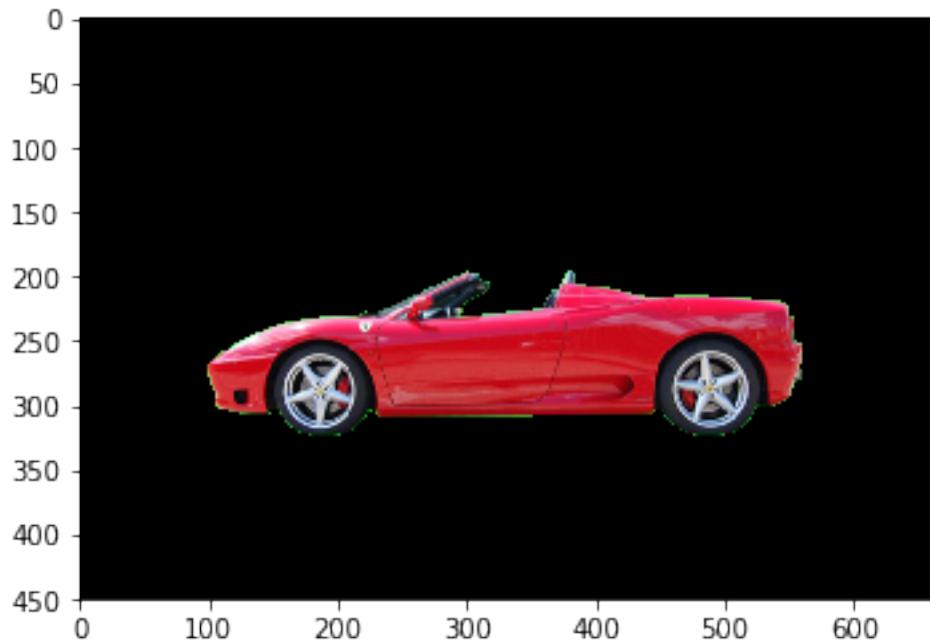
# Vizualize the mask
plt.imshow(mask, cmap='gray')
```

Out[4]: <matplotlib.image.AxesImage at 0x7f2ddf286dd8>



```
In [5]: # Mask the image to let the car show through  
masked_image = np.copy(image)  
  
masked_image[mask != 0] = [0, 0, 0]  
  
# Display it!  
plt.imshow(masked_image)
```

Out[5]: <matplotlib.image.AxesImage at 0x7f2dd2002b0>



1.1 TODO: Mask and add a background image

```
In [13]: # Load in a background image, and convert it to RGB
background_image = mpimg.imread('images/sky.jpg')

## TODO: Crop it or resize the background to be the right size (450x660)
# Hint: Make sure the dimensions are in the correct order!
print(background_image.shape)
standardized_bgi = cv2.resize(background_image, (660, 450))
#plt.imshow(standardized_bgi)
assert standardized_bgi.shape == image.shape

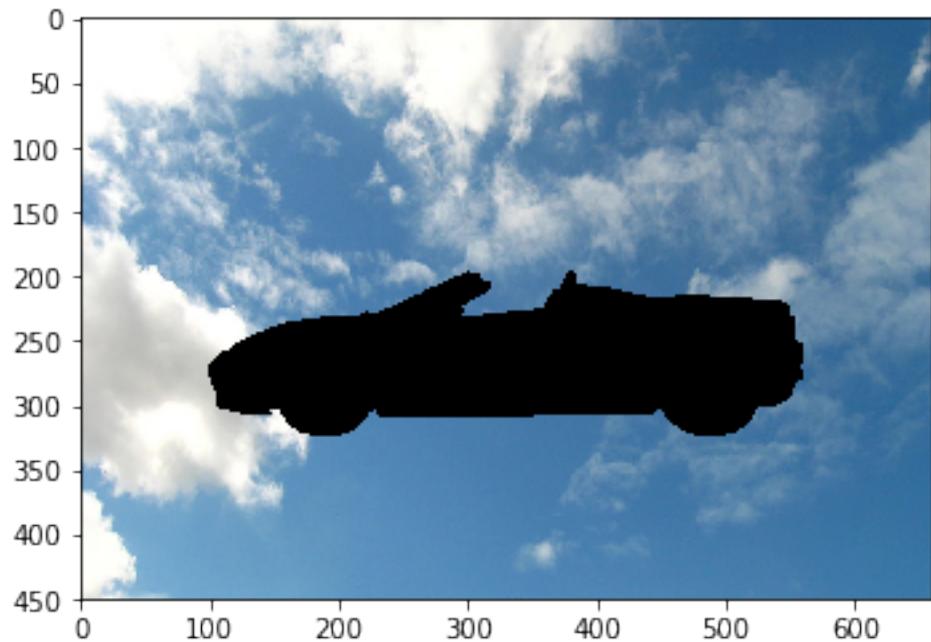
## TODO: Mask the cropped background so that the car area is blocked
# Hint: mask the opposite area of the previous image
masked_background = np.copy(standardized_bgi)

masked_background[mask == 0] = [0, 0, 0]

## TODO: Display the background and make sure
plt.imshow(masked_background)
```

(575, 1024, 3)

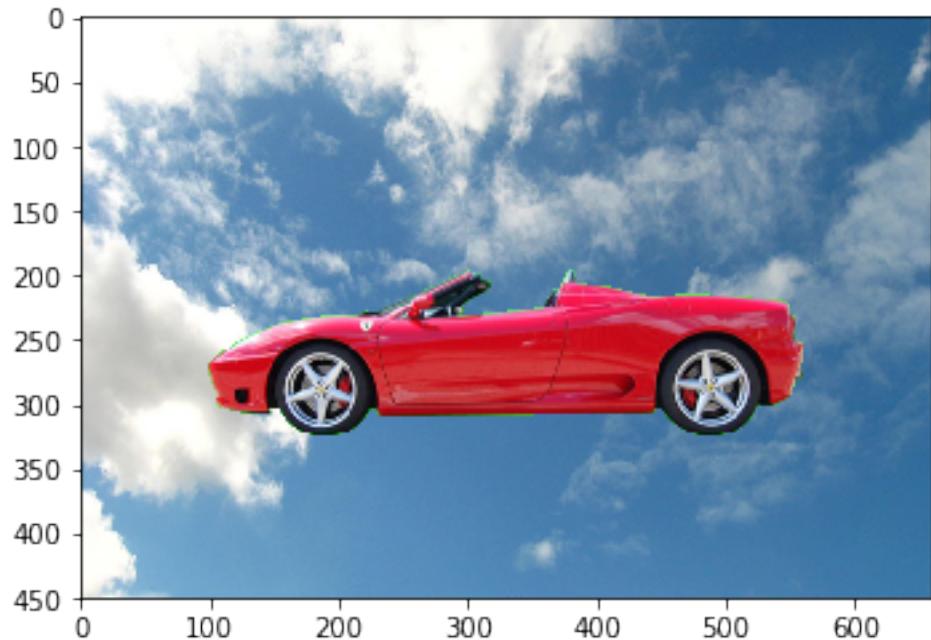
Out[13]: <matplotlib.image.AxesImage at 0x7f2ddd744cf8>



1.1.1 TODO: Create a complete image

```
In [14]: ## TODO: Add the two images together to create a complete image!
complete_image = masked_image + masked_background
plt.imshow(complete_image)
```

```
Out[14]: <matplotlib.image.AxesImage at 0x7f2ddd72be48>
```



In []:

HSV color conversion

March 30, 2020

1 HSV colorspace

1.0.1 Import resources

```
In [1]: import matplotlib.pyplot as plt
        import matplotlib.image as mpimg

        import numpy as np
        import cv2

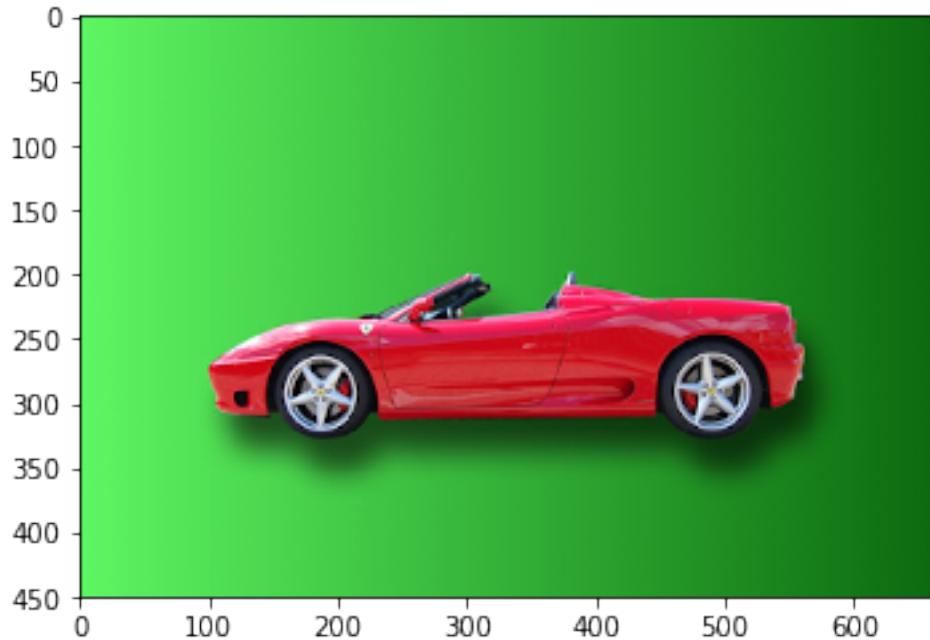
%matplotlib inline
```

1.0.2 Read in RGB image

```
In [11]: # Read in the image
          image = mpimg.imread('images/car_green_screen2.jpg')

          plt.imshow(image)
          print(image[50, 10, :])
```

[92 244 97]



1.0.3 RGB threshold

Visualize the green threshold you defined in the previous, consistent green color case.

```
In [3]: # Define our color selection boundaries in RGB values
lower_green = np.array([0,180,0])
upper_green = np.array([100,255,100])

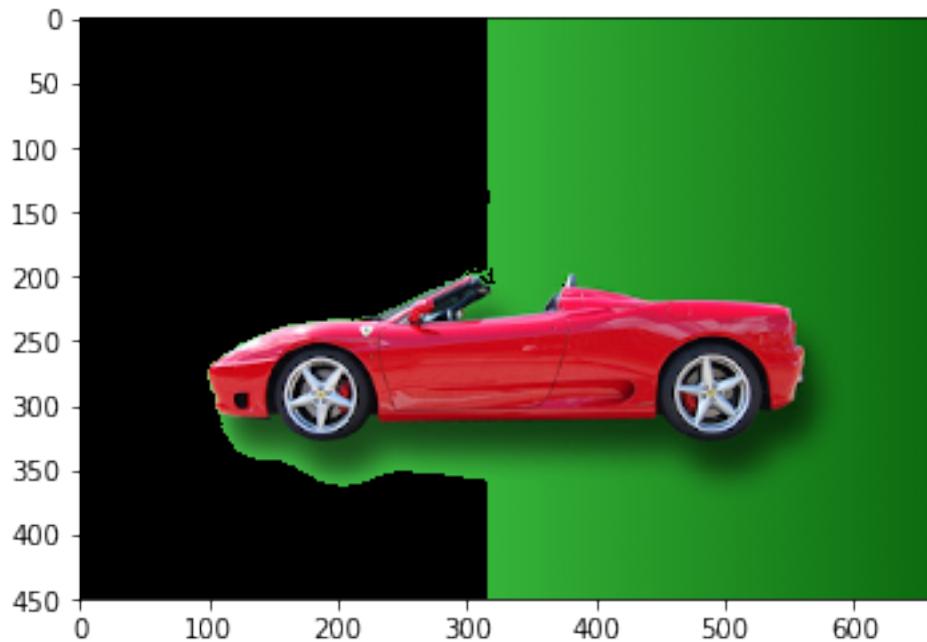
# Define the masked area
mask = cv2.inRange(image, lower_green, upper_green)

# Mask the image to let the car show through
masked_image = np.copy(image)

masked_image[mask != 0] = [0, 0, 0]

# Display it!
plt.imshow(masked_image)
```

Out[3]: <matplotlib.image.AxesImage at 0x7f76857dd2e8>



1.0.4 Convert to HSV

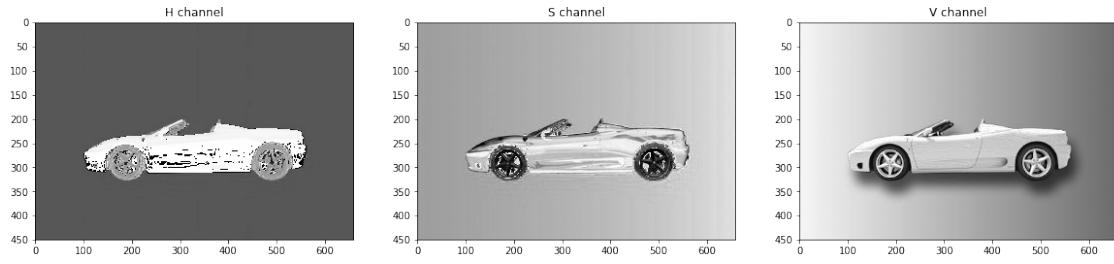
```
In [12]: # Convert to HSV
    hsv = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
    print(hsv[50, 10, :])

    # HSV channels
    h = hsv[:, :, 0]
    s = hsv[:, :, 1]
    v = hsv[:, :, 2]

    # Visualize the individual color channels
    f, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20,10))
    ax1.set_title('H channel')
    ax1.imshow(h, cmap='gray')
    ax2.set_title('S channel')
    ax2.imshow(s, cmap='gray')
    ax3.set_title('V channel')
    ax3.imshow(v, cmap='gray')

[ 61 159 244]
```

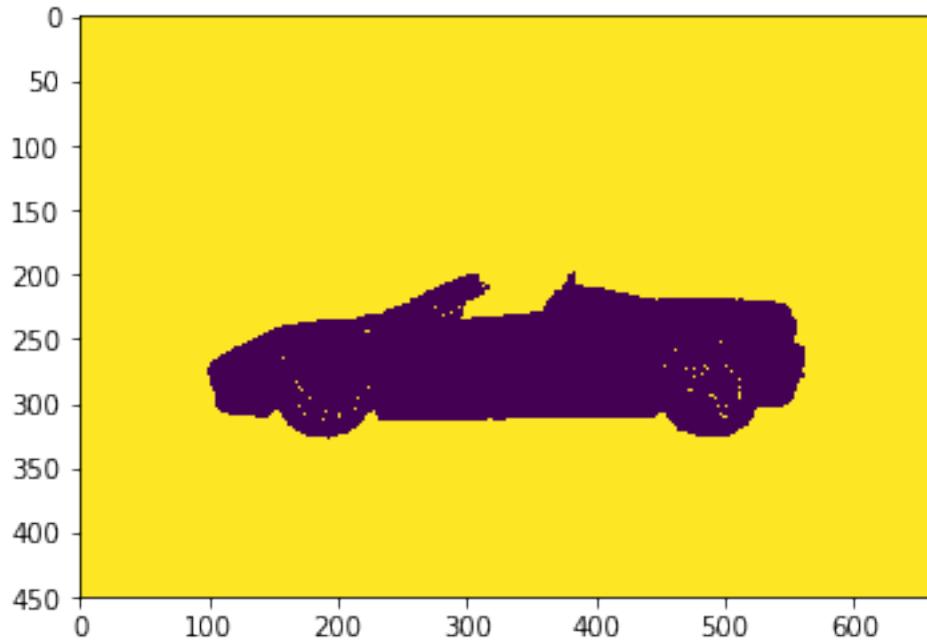
```
Out[12]: <matplotlib.image.AxesImage at 0x7f76854e9278>
```



1.0.5 TODO: Mask the green area using HSV color space

```
In [31]: ## TODO: Define the color selection boundaries in HSV values
#print(hsv[50,50,0]) --> ~60
lower_green = np.array([50, 0, 0])
upper_green = np.array([70, 255, 255])
## TODO: Define the masked area and mask the image
# Don't forget to make a copy of the original image to manipulate
masked_hsv = np.copy(hsv)
mask_hsv = cv2.inRange(masked_hsv, lower_green, upper_green)
plt.imshow(mask_hsv)
#masked_hsv[mask_hsv !=]
```

Out[31]: <matplotlib.image.AxesImage at 0x7f767f4484e0>



CS 312 Lecture 26

Debugging Techniques

Testing a program against a well-chosen set of input tests gives the programmer confidence that the program is correct. During the testing process, the programmer observes input-output relationships, that is, the output that the program produces for each input test case. If the program produces the expected output and obeys the specification for each test case, then the program is successfully tested.

But if the output for one of the test cases is not the one expected, then the program is incorrect -- it contains **errors** (or **defects**, or "bugs"). In such situations, testing only reveals the presence of errors, but doesn't tell us what the errors are, or how the code needs to be fixed. In other words, testing reveals the **effects** (or **symptoms**) of errors, not the **cause** of errors. The programmer must then go through a debugging process, to identify the causes and fix the errors.

Bug Prevention and Defensive Programming

Surprisingly, the debugging process may take significantly more time than writing the code in the first place. A large amount (if not most) of the development of a piece of software goes into debugging and maintaining the code, rather than writing it.

Therefore, the best thing to do is to **avoid the bug when you write the program in the first place!** It is important to sit and think before you code: decide exactly what needs to be achieved, how you plan to accomplish that, design the high-level algorithm cleanly, convince yourself it is correct, decide what are the concrete data structures you plan to use, and what are the invariants you plan to maintain. All the effort spent in designing and thinking about the code before you write it will pay off later. The benefits are twofold. First, having a clean design will reduce the probability of defects in your program. Second, even if a bug shows up during testing, a clean design with clear invariants will make it much easier to track down and fix the bug.

It may be very tempting to write the program as fast as possible, leaving little or no time to think about it before. The programmer will be happy to see the program done in a short amount. But it's likely he will get frustrated shortly afterwards: without good thinking, the program will be complex and unclear, so maintenance and bug fixing will become an endless process.

Once the programmer starts coding, he should use **defensive programming**. This is similar to defensive driving, which means driving under worst-case scenarios (e.g., other drivers violating traffic laws, unexpected events or obstacles, etc.). Similarly, defensive programming means developing code such that it works correctly under the worst-case scenarios from its environment. For instance, when writing a function, one should assume worst-case inputs to that function, i.e., inputs that are too large, too small, or inputs that violate some property, condition, or invariant; the code should deal with these cases, even if the programmer doesn't expect them to happen under normal circumstances.

Remember, the goal is not to become an expert at fixing bugs, but rather to get better at writing robust, (mostly) error-free programs in the first place. As a matter of attitude, programmers should not feel proud when they fix bugs, but rather embarrassed that their code had bugs. If there is a bug in the program, it is only because the programmer made mistakes.

Classes of Defects

Even after careful thought and defensive programming, a program may still have defects. Generally speaking, there are several kinds of errors one may run into:

- **Syntax or type errors.** These are always caught by the compiler, and reported via error messages. Typically, an error message clearly indicates the cause of error; for instance, the line number, the incorrect piece of code, and an explanation. Such messages usually give enough information about where the problem is and what needs to be done. In addition, editors with syntax highlighting can give good indication about such errors even before compiling the program.

- **Typos and other simple errors** that have passed undetected by the type-checker or the other checks in the compiler. Once these are identified, they can easily be fixed. Here are a few examples: missing parentheses, for instance writing `x + y * z` instead of `(x + y) * z`; typos, for instance case `t` of `... | x :: t1 => contains(x, t)`; passing parameters in incorrect order; or using the wrong element order in tuples.
- **Implementation errors.** It may be the case that logic in the high-level algorithm of a program is correct, but some low-level, concrete data structures are being manipulated incorrectly, breaking some internal representation invariants. For instance, a program that maintains a sorted list as the underlying data structure may break the sorting invariant. Building separate ADTs to model each data abstraction can help in such cases: it can separate the logic in the algorithm from the manipulation of concrete structures; in this way, the problem is being isolated in the ADT. Calls to `repOK()` can further point out what parts of the ADT cause the error.
- **Logical errors.** If the algorithm is logically flawed, the programmer must re-think the algorithm. Fixing such problems is more difficult, especially if the program fails on just a few corner cases. One has to closely examine the algorithm, and try to come up with an argument why the algorithm works. Trying to construct such an argument of correctness will probably reveal the problem. A clean design can help a lot figuring out and fixing such errors. In fact, in cases where the algorithm is too difficult to understand, it may be a good idea to redo the algorithm from scratch and aim for a cleaner formulation.

Difficulties

The debugging process usually consists of the following: examine the error symptoms, identify the cause, and finally fix the error. This process may be quite difficult and require a large amount of work, because of the following reasons:

- *The symptoms may not give clear indications about the cause.* In particular, the cause and the symptom may be remote, either in space (i.e., in the program code), or in time (i.e., during the execution of the program), or both. Defensive programming can help reduce the distance between the cause and the effect of an error.
- *Symptoms may be difficult to reproduce.* Replay is needed to better understand the problem. Being able to reproduce the same program execution is a standard obstacle in debugging concurrent programs. An error may show up only in one particular interleaving of statements from the parallel threads, and it may be almost impossible to reproduce that same, exact interleaving.
- *Errors may be correlated.* Therefore, symptoms may change during debugging, after fixing some of the errors. The new symptoms need to be re-examined. The good part is that the same error may have multiple symptoms; in that case, fixing the error will eliminate all of them.
- *Fixing an error may introduce new errors.* Statistics indicate that in many cases fixing a bug introduces a new one! This is the result of trying to do quick hacks to fix the error, without understanding the overall design and the invariants that the program is supposed to maintain. Once again, a clean design and careful thinking can avoid many of these cases.

Debugging strategies

Although there is no precise procedure for fixing all bugs, there are a number of useful strategies that can reduce the debugging effort. A significant part (if not all) of this process is spent localizing the error, that is, figuring out the cause from its symptoms. Below are several useful strategies to help with this. Keep in mind that different techniques are better suited in different cases; there is no clear best method. It is good to have knowledge and experience with all of these approaches. Sometimes, a combination of one or more of these approaches will lead you to the error.

- **Incremental and bottom-up program development.** One of the most effective ways to localize errors is to develop the program incrementally, and test it often, after adding each piece of code. It is highly likely that if there is an error, it occurs in the last piece of code that you wrote. With incremental program development, the last portion of code is small; the search for bugs is therefore limited to small code fragments. An added benefit is that small code increments will likely lead to few errors, so the

programmer is not overwhelmed with long lists of errors.

Bottom-up development maximizes the benefits of incremental development. With bottom-up development, once a piece of code has been successfully tested, its behavior won't change when more code is incrementally added later. Existing code doesn't rely on the new parts being added, so if an error occurs, it must be in the newly added code (unless the old parts weren't tested well enough).

- **Instrument program to log information.** Typically, print statements are inserted. Although the printed information is effective in some cases, it can also become difficult to inspect when the volume of logged information becomes huge. In those cases, automated scripts may be needed to sift through the data and report the relevant parts in a more compact format. Visualization tools can also help understanding the printed data. For instance, to debug a program that manipulates graphs, it may be useful to use a graph visualization tool (such as ATT's `graphviz`) and print information in the appropriate format (.dot files for graphviz).
- **Instrument program with assertions.** Assertions check if the program indeed maintains the properties or invariants that your code relies on. Because the program stops as soon as an assertion fails, it's likely that the point where the program stops is much closer to the cause, and is a good indicator of what the problem is. An example of assertion checking is the `repOK()` function that verifies if the representation invariant holds at function boundaries. Note that checking invariants or conditions is the basis of defensive programming. The difference is that the number of checks is usually increased during debugging for those parts of the program that are suspected to contain errors.
- **Use debuggers.** If a debugger is available, it can replace the manual instrumentation using print statements or assertions. Setting breakpoints in the program, stepping into and over functions, watching program expressions, and inspecting the memory contents at selected points during the execution will give all the needed run-time information without generating large, hard-to-read log files.
- **Backtracking.** One option is to start from the point where the problem occurred and go back through the code to see how that might have happened.
- **Binary search.** The backtracking approach will fail if the error is far from the symptom. A better approach is to explore the code using a divide-and-conquer approach, to quickly pin down the bug. For example, starting from a large piece of code, place a check halfway through the code. If the error doesn't show up at that point, it means the bug occurs in the second half; otherwise, it is in the first half. Thus, the code that needs inspection has been reduced to half. Repeating the process a few times will quickly lead to the actual problem.
- **Problem simplification.** A similar approach is to gradually eliminate portions of the code that are not relevant to the bug. For instance, if a function `fun f() = (g(); h(); k())` yields an error, try eliminating the calls to g, h, and k successively (by commenting them out), to determine which is the erroneous one. Then simplify the code in the body of buggy function, and so on. Continuing this process, the code gets simpler and simpler. The bug will eventually become evident. A similar technique can be applied to simplify data rather than code. If the size of the input data is too large, repeatedly cut parts of it and check if the bug is still present. When the data set is small enough, the cause may be easier to understand.
- **A scientific method: form hypotheses.** A related approach is as follows: inspect the test case results; form a hypothesis that is consistent with the observed data; and then design and run a simple test to refute the hypothesis. If the hypothesis has been refuted, derive another hypothesis and continue the process. In some sense, this is also a simplification process: it reduces the number of possible hypotheses at each step. But unlike the above simplification techniques, which are mostly mechanical, this process is driven by active thinking about an explanation. A good approach is to try to come with the simplest hypotheses and the simplest corresponding test cases.

Consider, for example, a function `palindrome(s:string):bool`, and suppose that `palindrome("able was I ere I saw elba")` returns an incorrect value of false. Here are several possible hypotheses for this failure. Maybe `palindrome` fails for inputs with spaces (try " "); maybe it fails for programs with upper case letters (try "I"); maybe it fails for inputs of odd length greater than one (try "ere"), and so on. Forming and testing these hypotheses one after another can lead the

programmer to the source of the problem.

- **Bug clustering.** If a large number of errors are being reported, it is useful to group them into classes of related bugs (or similar bugs), and examine only one bug from each class. The intuition is that bugs from each class have the same cause (or a similar cause). Therefore, fixing a bug will automatically fix all the other bugs from the same class (or will make it obvious how to fix them).
- **Error-detection tools.** Such tools can help programmers quickly identify violations of certain classes of errors. For instance, tools that check safety properties can verify that file accesses in a program obey the open-read/write-close file sequence; that the code correctly manipulates locks; or that the program always accesses valid memory. Such tools are either dynamic (they instrument the program to find errors at run-time), or use static analysis (look for errors at compile-time). For instance, **Purify** is a popular dynamic tool that instruments programs to identify memory errors, such as invalid accesses or memory leaks. Examples of static tools include **ESC Java** and **Spec#**, which use theorem proving approaches to check more general user specifications (pre and post-conditions, or invariants); or tools from a recent company **Coverity** that use dataflow analysis to detect violations of safety properties. Such tools can dramatically increase productivity, but checking is restricted to a particular domain or class of properties. There is also an associated learning curve, although that is usually low. Currently, there are relatively few such tools and this is more an (active) area of research.

A number of other strategies can be viewed as a matter of attitude about where to expect the errors:

- *The bug may not be where you expect it.* If a large amount of time has unsuccessfully been spent inspecting a particular piece of code, the error may not be there. Keep an open mind and start questioning the other parts of the program.
- *Ask yourself where the bug is not.* Sometimes, looking at the problem upside-down gives a different perspective. Often, trying to prove that the absence of a bug in a certain place actually reveals the bug in that place.
- *Explain to yourself or to somebody else why you believe there is no bug.* Trying to articulate the problem can lead to the discovery of the bug.
- *Inspect input data, test harness.* The test case or the test harness itself may be broken. One has to check these carefully, and make sure that the bug is in the actual program.
- *Make sure you have the right source code.* One must ensure that the source code being debugged corresponds to the actual program being run, and that the correct libraries are linked. This usually requires a few simple checks; using makefiles and make programs (e.g., the Compilation Manager and .cm files) can reduce this to just typing a single command.
- *Take a break.* If too much time is spent on a bug, the programmer becomes tired and debugging may become counterproductive. Take a break, clear your mind; after some rest, try to think about the problem from a different perspective.

All of the above are techniques for localizing errors. Once they have been identified, errors need to be corrected. In some cases, this is trivial (e.g., for typos and simple errors). Some other times, it may be fairly straightforward, but the change must ensure maintaining certain invariants. The programmer must think well about how the fix is going to affect the rest of the code, and make sure no additional problems are created by fixing the error. Of course, proper documentation of these invariants is needed. Finally, bugs that represent conceptual errors in an algorithm are the most difficult to fix. The programmer must re-think and fix the logic of the algorithm.

Standardizing the Data

March 31, 2020

0.1 # Day and Night Image Classifier

The day/night image dataset consists of 200 RGB color images in two categories: day and night. There are equal numbers of each example: 100 day images and 100 night images.

We'd like to build a classifier that can accurately label these images as day or night, and that relies on finding distinguishing features between the two types of images!

Note: All images come from the [AMOS dataset](#) (Archive of Many Outdoor Scenes).

0.1.1 Import resources

Before you get started on the project code, import the libraries and resources that you'll need.

```
In [1]: import cv2 # computer vision library
         import helpers

         import numpy as np
         import matplotlib.pyplot as plt
         import matplotlib.image as mpimg

         %matplotlib inline
```

0.2 Training and Testing Data

The 200 day/night images are separated into training and testing datasets.

- 60% of these images are training images, for you to use as you create a classifier.
- 40% are test images, which will be used to test the accuracy of your classifier.

First, we set some variables to keep track of some where our images are stored:

```
image_dir_training: the directory where our training image data is stored
image_dir_test: the directory where our test image data is stored
```

```
In [2]: # Image data directories
         image_dir_training = "day_night_images/training/"
         image_dir_test = "day_night_images/test/"
```

0.3 Load the datasets

These first few lines of code will load the training day/night images and store all of them in a variable, IMAGE_LIST. This list contains the images and their associated label ("day" or "night").

For example, the first image-label pair in IMAGE_LIST can be accessed by index: IMAGE_LIST[0] [:].

```
In [38]: # Using the load_dataset function in helpers.py
# Load training data
IMAGE_LIST = helpers.load_dataset(image_dir_training)
```

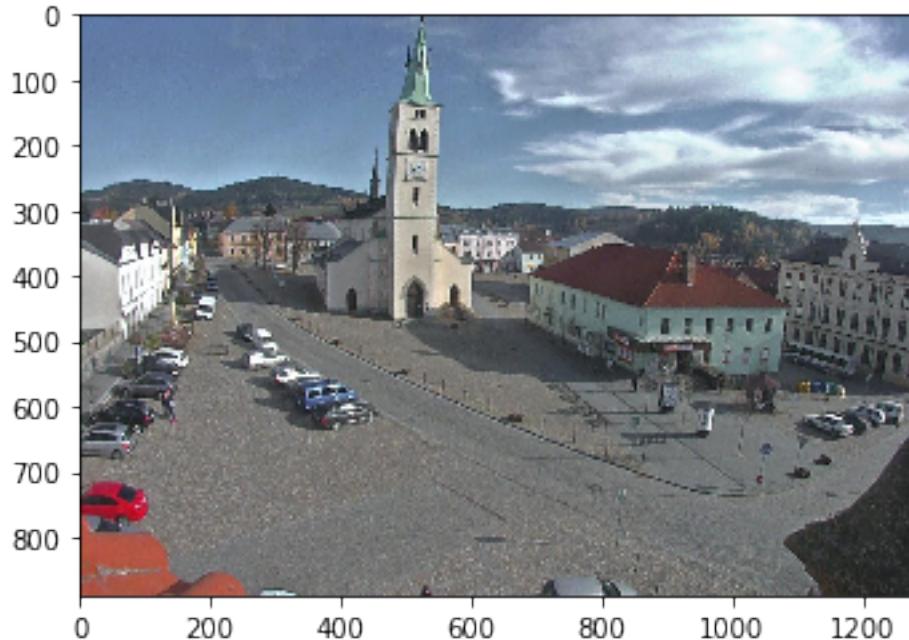
1 1. Visualize the input images

```
In [40]: # Print out 1. The shape of the image and 2. The image's label
# Select an image and its label by list index
image_index = 7
selected_image = IMAGE_LIST[image_index][0]
selected_label = IMAGE_LIST[image_index][1]

# Display image and data about it
plt.imshow(selected_image)
print("Shape: "+str(selected_image.shape))
print("Label: " + str(selected_label))
```

Shape: (889, 1280, 3)

Label: day



2 2. Pre-process the Data

After loading in each image, you have to standardize the input and output!

Solution code You are encouraged to try to complete this code on your own, but if you are struggling or want to make sure your code is correct, there will be solution code in **the next Notebook: Average Brightness Feature Extraction** in the `helpers.py` file. You can jump ahead and look into that python file to see complete `standardize_input` and `encode` function code. For this day and night challenge, you can often jump one notebook ahead to see the solution code for a previous notebook!

2.0.1 Input

It's important to make all your images the same size so that they can be sent through the same pipeline of classification steps! Every input image should be in the same format, of the same size, and so on.

TODO: Standardize the input images

- Resize each image to the desired input size: 600x1100px (hxw).

```
In [58]: # This function should take in an RGB image and return a new, standardized version
def standardize_input(image):

    ## TODO: Resize image so that all "standard" images are the same size 600x1100 (hxw)
    standard_im = cv2.resize(image, (1100, 600))

    return standard_im
```

2.0.2 TODO: Standardize the output

With each loaded image, you also need to specify the expected output. For this, use binary numerical values 0/1 = night/day.

```
In [46]: # Examples:
# encode("day") should return: 1
# encode("night") should return: 0

def encode(label):

    ## TODO: complete the code to produce a numerical label
```

```
if(label == 'day'):
    return 1
else:
    return 0
```

2.1 Construct a STANDARDIZED_LIST of input images and output labels.

This function takes in a list of image-label pairs and outputs a **standardized** list of resized images and numerical labels.

This uses the functions you defined above to standardize the input and output, so those functions must be complete for this standardization to work!

```
In [59]: def standardize(image_list):

    # Empty image data array
    standard_list = []

    # Iterate through all the image-label pairs
    for item in image_list:
        image = item[0]
        label = item[1]

        # Standardize the image
        standardized_im = standardize_input(image)

        # Create a numerical label
        binary_label = encode(label)

        # Append the image, and it's one hot encoded label to the full, processed list
        standard_list.append((standardized_im, binary_label))

    return standard_list

# Standardize all training images
STANDARDIZED_LIST = standardize(IMAGE_LIST)
```

2.2 Visualize the standardized data

Display a standardized image from STANDARDIZED_LIST.

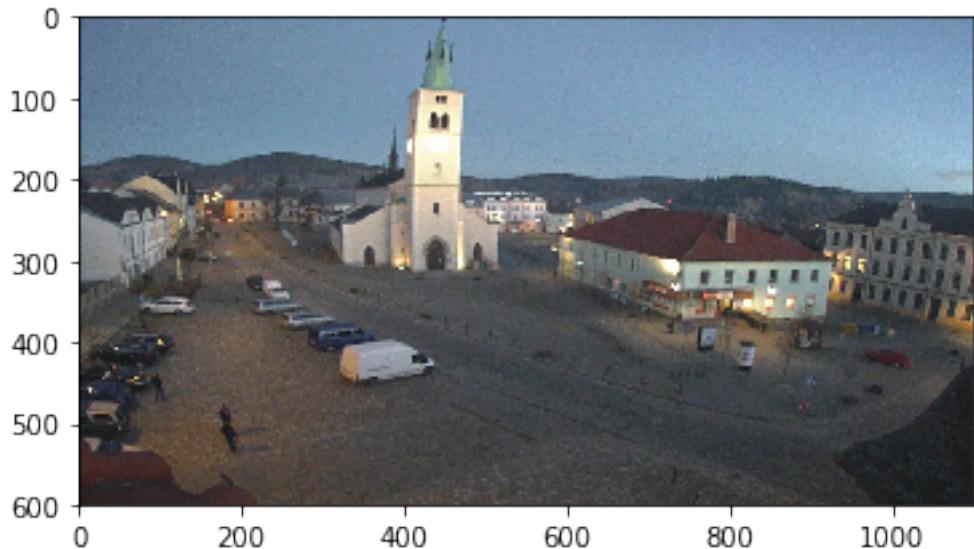
```
In [60]: # Display a standardized image and its label

    # Select an image by index
    image_num = 100
    selected_image = STANDARDIZED_LIST[image_num][0]
    selected_label = STANDARDIZED_LIST[image_num][1]

    # Display image and data about it
```

```
## TODO: Make sure the images have numerical labels and are of the same size
plt.imshow(selected_image)
print("Shape: "+str(selected_image.shape))
print("Label [1 = day, 0 = night]: " + str(selected_label))

Shape: (600, 1100, 3)
Label [1 = day, 0 = night]: 1
```



In []:

Average Brightness

March 31, 2020

0.1 # Day and Night Image Classifier

The day/night image dataset consists of 200 RGB color images in two categories: day and night. There are equal numbers of each example: 100 day images and 100 night images.

We'd like to build a classifier that can accurately label these images as day or night, and that relies on finding distinguishing features between the two types of images!

Note: All images come from the [AMOS dataset](#) (Archive of Many Outdoor Scenes).

0.1.1 Import resources

Before you get started on the project code, import the libraries and resources that you'll need.

```
In [1]: import cv2 # computer vision library
         import helpers

         import numpy as np
         import matplotlib.pyplot as plt
         import matplotlib.image as mpimg

         %matplotlib inline
```

0.2 Training and Testing Data

The 200 day/night images are separated into training and testing datasets.

- 60% of these images are training images, for you to use as you create a classifier.
- 40% are test images, which will be used to test the accuracy of your classifier.

First, we set some variables to keep track of some where our images are stored:

```
image_dir_training: the directory where our training image data is stored
image_dir_test: the directory where our test image data is stored
```

```
In [2]: # Image data directories
         image_dir_training = "day_night_images/training/"
         image_dir_test = "day_night_images/test/"
```

0.3 Load the datasets

These first few lines of code will load the training day/night images and store all of them in a variable, IMAGE_LIST. This list contains the images and their associated label ("day" or "night").

For example, the first image-label pair in IMAGE_LIST can be accessed by index: IMAGE_LIST[0][:] .

```
In [3]: # Using the load_dataset function in helpers.py
# Load training data
IMAGE_LIST = helpers.load_dataset(image_dir_training)
```

0.4 Construct a STANDARDIZED_LIST of input images and output labels.

This function takes in a list of image-label pairs and outputs a **standardized** list of resized images and numerical labels.

```
In [4]: # Standardize all training images
STANDARDIZED_LIST = helpers.standardize(IMAGE_LIST)
```

0.5 Visualize the standardized data

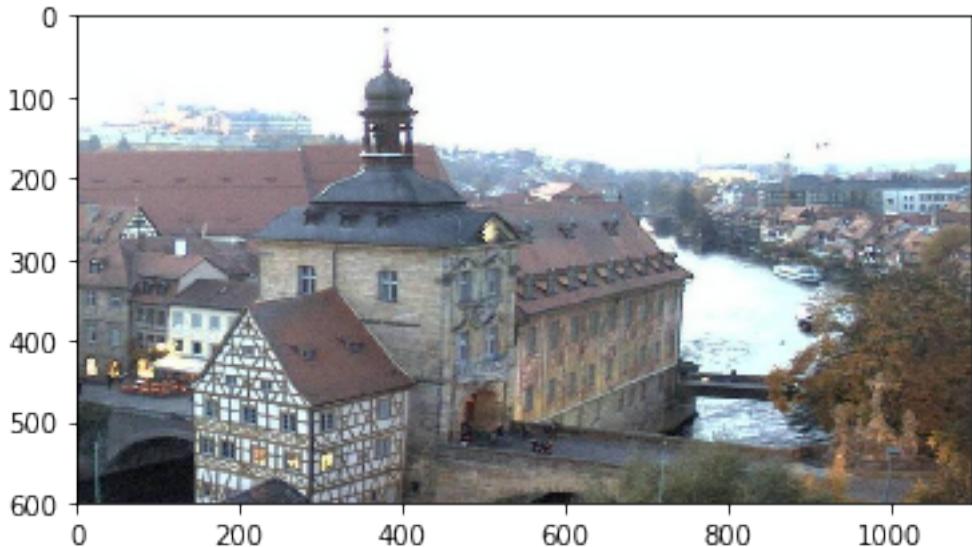
Display a standardized image from STANDARDIZED_LIST.

```
In [5]: # Display a standardized image and its label

# Select an image by index
image_num = 0
selected_image = STANDARDIZED_LIST[image_num][0]
selected_label = STANDARDIZED_LIST[image_num][1]

# Display image and data about it
plt.imshow(selected_image)
print("Shape: "+str(selected_image.shape))
print("Label [1 = day, 0 = night]: " + str(selected_label))

Shape: (600, 1100, 3)
Label [1 = day, 0 = night]: 1
```



1 Feature Extraction

Create a feature that represents the brightness in an image. We'll be extracting the **average brightness** using HSV colorspace. Specifically, we'll use the V channel (a measure of brightness), add up the pixel values in the V channel, then divide that sum by the area of the image to get the average Value of the image.

1.1 RGB to HSV conversion

Below, a test image is converted from RGB to HSV colorspace and each component is displayed in an image.

```
In [6]: # Convert and image to HSV colorspace
        # Visualize the individual color channels

image_num = 0
test_im = STANDARDIZED_LIST[image_num][0]
test_label = STANDARDIZED_LIST[image_num][1]

# Convert to HSV
hsv = cv2.cvtColor(test_im, cv2.COLOR_RGB2HSV)

# Print image label
print('Label: ' + str(test_label))

# HSV channels
h = hsv[:, :, 0]
s = hsv[:, :, 1]
```

```

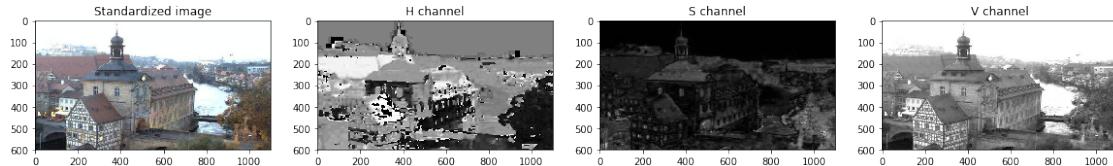
v = hsv[:, :, 2]

# Plot the original image and the three channels
f, (ax1, ax2, ax3, ax4) = plt.subplots(1, 4, figsize=(20,10))
ax1.set_title('Standardized image')
ax1.imshow(test_im)
ax2.set_title('H channel')
ax2.imshow(h, cmap='gray')
ax3.set_title('S channel')
ax3.imshow(s, cmap='gray')
ax4.set_title('V channel')
ax4.imshow(v, cmap='gray')

```

Label: 1

Out [6]: <matplotlib.image.AxesImage at 0x7f8cea166cf8>



1.1.1 Find the average brightness using the V channel

This function takes in a **standardized** RGB image and returns a feature (a single value) that represent the average level of brightness in the image. We'll use this value to classify the image as day or night.

```

In [7]: # Find the average Value or brightness of an image
def avg_brightness(rgb_image):

    # Convert image to HSV
    hsv = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2HSV)

    # Add up all the pixel values in the V channel
    sum_brightness = np.sum(hsv[:, :, 2])

    ## TODO: Calculate the average brightness using the area of the image
    # and the sum calculated above
    total_pixels = 600 * 1100
    avg = sum_brightness / total_pixels

    return avg

```

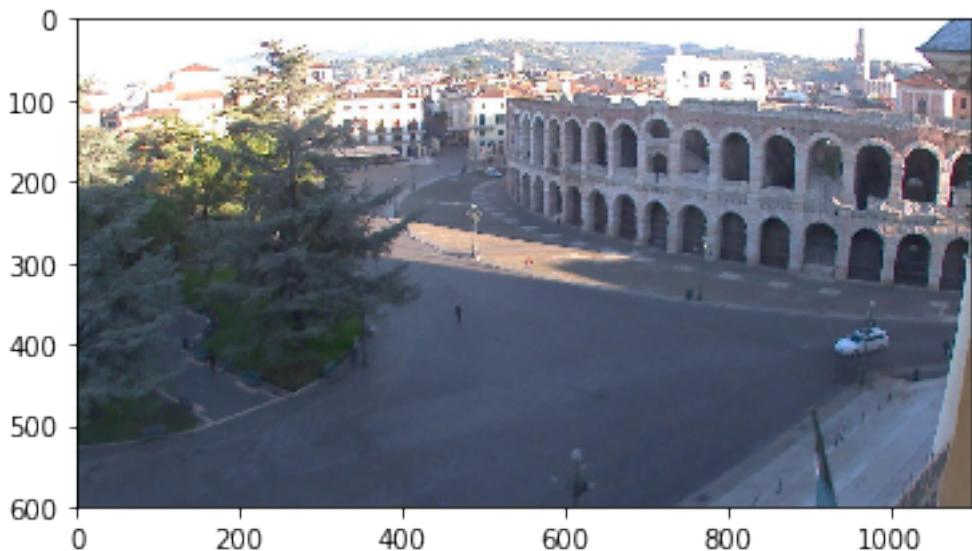
```
In [15]: # Testing average brightness levels
# Look at a number of different day and night images and think about
# what average brightness value separates the two types of images

# As an example, a "night" image is loaded in and its avg brightness is displayed
image_num = 28
test_im = STANDARDIZED_LIST[image_num][0]

avg = avg_brightness(test_im)
print('Avg brightness: ' + str(avg))
plt.imshow(test_im)
```

Avg brightness: 129.793778788

Out[15]: <matplotlib.image.AxesImage at 0x7f8ce9cbd2e8>



In []:

Finding Edges and Custom Kernels

March 31, 2020

1 Creating a Filter, Edge Detection

1.0.1 Import resources and display image

```
In [1]: import matplotlib.pyplot as plt
        import matplotlib.image as mpimg

        import cv2
        import numpy as np

%matplotlib inline

# Read in the image
image = mpimg.imread('images/curved_lane.jpg')

plt.imshow(image)

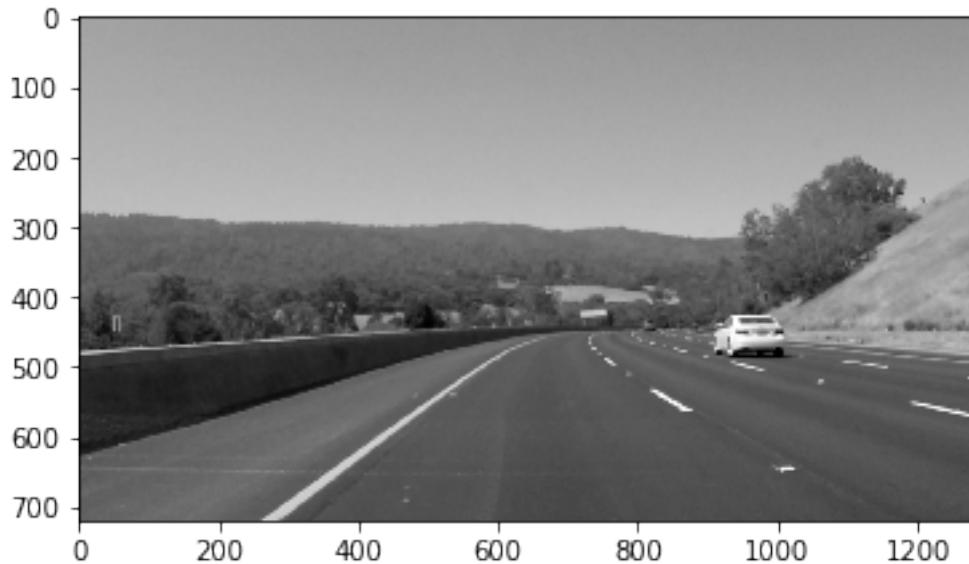
Out[1]: <matplotlib.image.AxesImage at 0x7fb127bebf60>
```



1.0.2 Convert the image to grayscale

```
In [2]: # Convert to grayscale for filtering  
gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)  
  
plt.imshow(gray, cmap='gray')
```

Out[2]: <matplotlib.image.AxesImage at 0x7fb0f2319048>



1.0.3 TODO: Create a custom kernel

Below, you've been given one common type of edge detection filter: a Sobel operator.

The Sobel filter is very commonly used in edge detection and in finding patterns in intensity in an image. Applying a Sobel filter to an image is a way of **taking (an approximation) of the derivative of the image** in the x or y direction, separately. The operators look as follows.

It's up to you to create a Sobel x operator and apply it to the given image.

For a challenge, see if you can put the image through a series of filters: first one that blurs the image (takes an average of pixels), and then one that detects the edges.

```
In [10]: # Create a custom kernel  
  
# 3x3 array for edge detection  
sobel_y = np.array([[ -1, -2, -1],  
                   [ 0, 0, 0],  
                   [ 1, 2, 1]])  
  
## TODO: Create and apply a Sobel x operator  
sobel_x = np.array([[-1, 0, 1],
```

```

[-2, 0, 2],
[-1, 0, 1]])

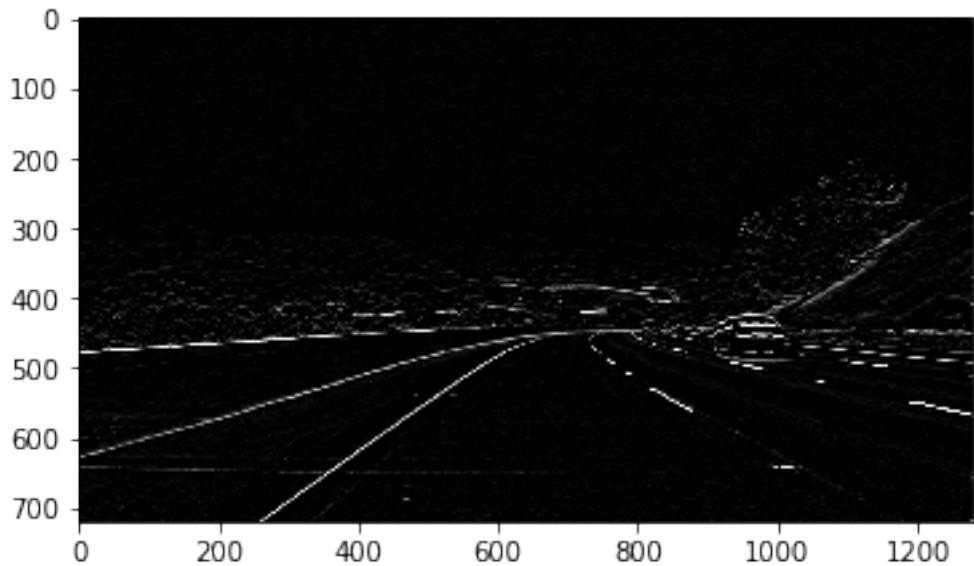
sobel_edges = np.array([[0, -1, 0],
                      [-1, 4, -1],
                      [0, -1, 0]])

# Filter the image using filter2D, which has inputs: (grayscale image, bit-depth, kernel)
filtered_image = cv2.filter2D(gray, -1, sobel_y)

plt.imshow(filtered_image, cmap='gray')

```

Out[10]: <matplotlib.image.AxesImage at 0x7fb0ee52a1d0>



1.0.4 Test out other filters!

You're encouraged to create other kinds of filters and apply them to see what happens! As an **optional exercise**, try the following:

- * Create a filter with decimal value weights.
- * Create a 5x5 filter
- * Apply your filters to the other images in the `images` directory.

In []:

Feature Vectors

March 31, 2020

1 Histograms of Color

Let's go back to our classification task: classifying images as day or night. You've looked at using raw pixel values to construct a brightness feature and now we'll look at histograms of pixel intensity (color histograms) as **feature vectors**.

Features can be arrays of useful values. Even the filtered images you've been creating are considered feature extracted images. Feature vectors are 1D arrays (or lists) of values, and they can be used when a single value is just not enough to classify an image.

In this notebook you'll see how to create a common type of feature vector: a **histogram**. A histogram is a graphical display of data that shows bars of different heights. Each bar groups data (in this case, pixel values) into ranges and the height of each bar indicates the number of times the data falls into that range. So, a taller bar show that more data falls in that specific range.

Let's see what an HSV color histogram looks like.

1.0.1 Import resources

```
In [1]: import cv2 # computer vision library  
  
import numpy as np  
import matplotlib.pyplot as plt  
import matplotlib.image as mpimg  
  
%matplotlib inline
```

1.0.2 Read in and standardize day and night images

We'll be analyzing two images: one day and one night from our training set of data; of the same scene. These have yet to be standardized, so they are resized to be the same.

```
In [2]: # Read in a day and a night image  
# These are directly extracted by name -- do not change  
day_image = mpimg.imread("day_night_images/training/day/20151102_074952.jpg")  
night_image = mpimg.imread("day_night_images/training/night/20151102_175445.jpg")  
  
# Make these images the same size  
width = 1100  
height = 600
```

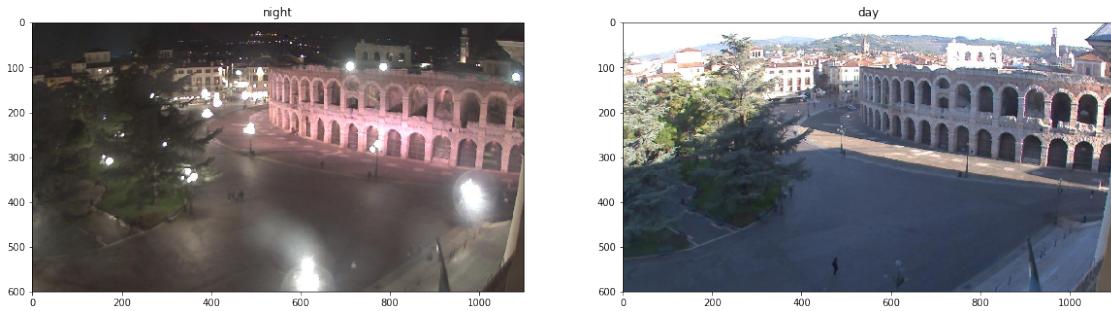
```

night_image = cv2.resize(night_image, (width, height))
day_image = cv2.resize(day_image, (width, height))

# Visualize both images
f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))
ax1.set_title('night')
ax1.imshow(night_image)
ax2.set_title('day')
ax2.imshow(day_image)

```

Out [2]: <matplotlib.image.AxesImage at 0x7f7574a0c358>



1.0.3 Create HSV histograms

First, convert these images to HSV colorspace. Then use numpy's [histogram function](#) to bin the color values into ranges. Bins are ranges of values like 0-15 for dark intensities or 200-255 for bright values.

With np.histogram(), you don't have to specify the number of bins or the range, but here I've arbitrarily chosen 32 bins and specified range=(0, 256) in order to get orderly bin sizes. np.histogram() returns a tuple of two arrays. In this case, for example, h_hist[0] contains the counts in each of the bins and h_hist[1] contains the bin edges (so it is one element longer than h_hist[0]).

To plot these results, we can compute the bin centers from the bin edges. Each of the histograms in this case have the same bins, so I'll just use the rhist bin edges: you can define the number of bins.

```

In [3]: def hsv_histograms(rgb_image):
    # Convert to HSV
    hsv = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2HSV)

    # Create color channel histograms
    h_hist = np.histogram(hsv[:, :, 0], bins=32, range=(0, 180))
    s_hist = np.histogram(hsv[:, :, 1], bins=32, range=(0, 256))
    v_hist = np.histogram(hsv[:, :, 2], bins=32, range=(0, 256))

    # Generating bin centers

```

```

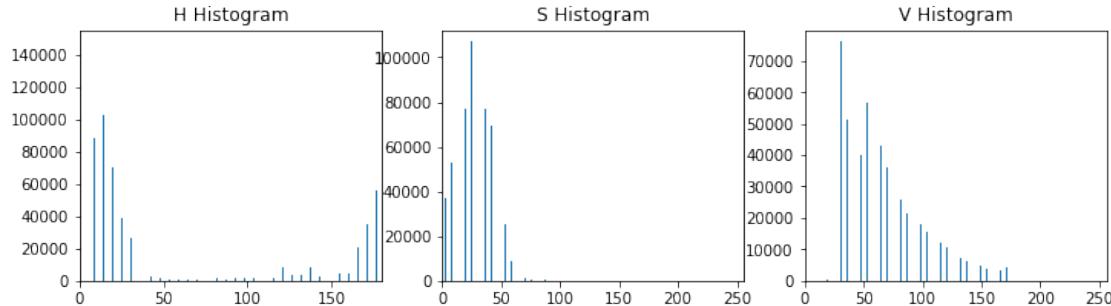
bin_edges = h_hist[1]
bin_centers = (bin_edges[1:] + bin_edges[0:len(bin_edges)-1])/2

# Plot a figure with all three histograms
fig = plt.figure(figsize=(12,3))
plt.subplot(131)
plt.bar(bin_centers, h_hist[0])
plt.xlim(0, 180)
plt.title('H Histogram')
plt.subplot(132)
plt.bar(bin_centers, s_hist[0])
plt.xlim(0, 256)
plt.title('S Histogram')
plt.subplot(133)
plt.bar(bin_centers, v_hist[0])
plt.xlim(0, 256)
plt.title('V Histogram')

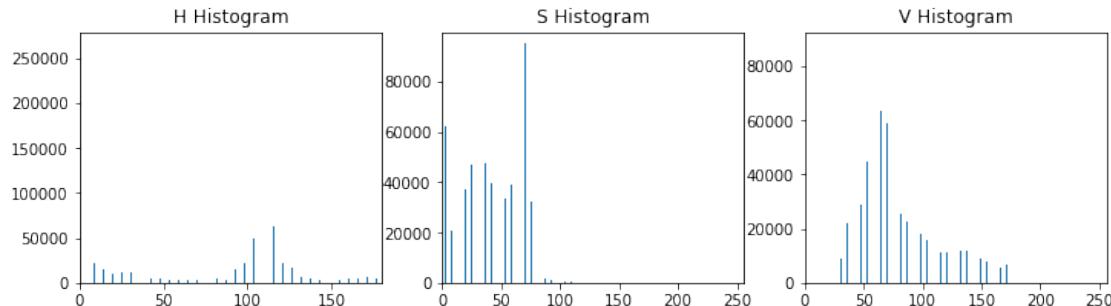
return h_hist, s_hist, v_hist

```

In [4]: # Call the function for "night"
`night_h_hist, night_s_hist, night_v_hist = hsv_histograms(night_image)`



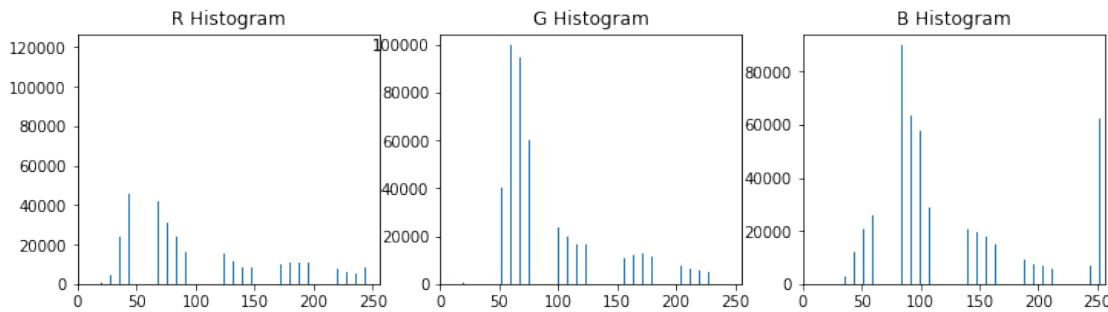
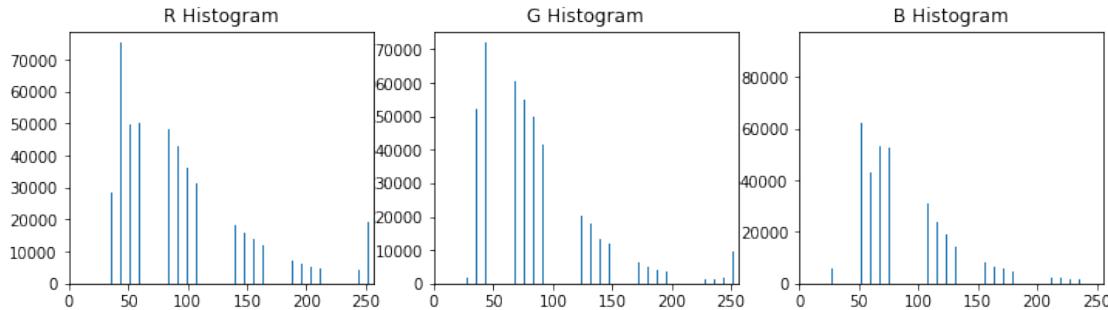
In [5]: # Call the function for "day"
`day_h_hist, day_s_hist, day_v_hist = hsv_histograms(day_image)`



1.0.4 Look at the differences

```
In [6]: # Which bin do most V values fall in?  
# Does the Hue channel look helpful?  
# What patterns can you see that might distinguish these two images?  
  
# Out of 32 bins, if the most common bin is in the middle or high up, then it's likely a  
fullest_vbin_day = np.argmax(day_v_hist[0])  
fullest_vbin_night = np.argmax(night_v_hist[0])  
  
print('Fullest Value bin for day: ', fullest_vbin_day)  
print('Fullest Value bin for night: ', fullest_vbin_night)  
  
Fullest Value bin for day: 10  
Fullest Value bin for night: 5
```

```
In [14]: ## TODO: Create and look at RGB histograms  
# Practice what you've learned and look at RGB color histograms of these same images  
def rgb_histograms(rgb_image):  
    r_hist = np.histogram(rgb_image[:, :, 0], bins = 32, range = (0, 256))  
    g_hist = np.histogram(rgb_image[:, :, 1], bins = 32, range = (0, 256))  
    b_hist = np.histogram(rgb_image[:, :, 2], bins = 32, range = (0, 256))  
  
    bin_edges = r_hist[1]  
    bin_centers = (bin_edges[1:] + bin_edges[0:len(bin_edges)-1])/2  
  
    # Plot a figure with all three histograms  
    fig = plt.figure(figsize=(12,3))  
    plt.subplot(131)  
    plt.bar(bin_centers, r_hist[0])  
    plt.xlim(0, 256)  
    plt.title('R Histogram')  
    plt.subplot(132)  
    plt.bar(bin_centers, g_hist[0])  
    plt.xlim(0, 256)  
    plt.title('G Histogram')  
    plt.subplot(133)  
    plt.bar(bin_centers, b_hist[0])  
    plt.xlim(0, 256)  
    plt.title('B Histogram')  
  
    return r_hist, g_hist, b_hist  
  
night_r_hist, night_g_hist, night_b_hist = rgb_histograms(night_image)  
day_r_hist, day_g_hist, day_b_hist = rgb_histograms(day_image)
```



1.0.5 Summations to create Feature Vectors

To keep spatial information, it is also a useful technique to sum up pixel values along the columns or rows of an image. This allows you to see spikes in various color values over space.

Let's look at the night image as an example; it's mostly dark but has a lot of little bright spots from artificial lights. I'll look at the Value component in the image, sum up those pixel values along the columns using `np.sum()`, and I'll plot that summation.

1.0.6 Sum the V component of the day image and compare the two

```
In [8]: # Convert the night image to HSV colorspace
hsv_night = cv2.cvtColor(night_image, cv2.COLOR_RGB2HSV)

# Isolate the V component
v = hsv_night[:, :, 2]

# Sum the V component over all columns (axis = 0)
v_sum = np.sum(v[:, :], axis=0)

f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 10))

ax1.set_title('Value sum over columns')
```

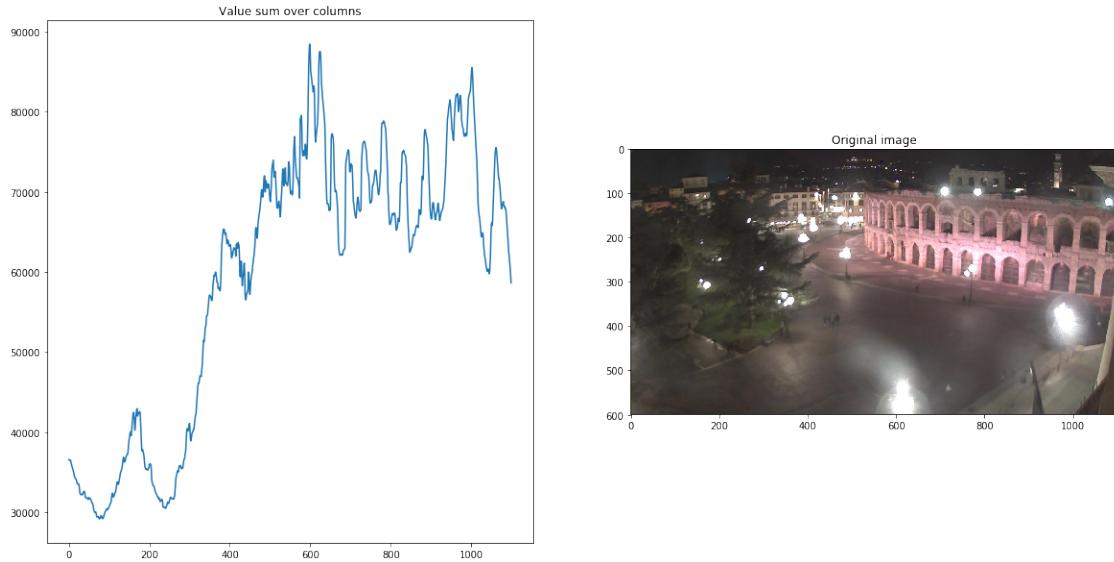
```

ax1.plot(v_sum)

ax2.set_title('Original image')
ax2.imshow(night_image, cmap='gray')

```

Out[8]: <matplotlib.image.AxesImage at 0x7f7572c4fa58>



```

In [10]: # Convert the night image to HSV colorspace
          hsv_day = cv2.cvtColor(day_image, cv2.COLOR_RGB2HSV)

          # Isolate the V component
          v = hsv_day[:, :, 2]

          # Sum the V component over all columns (axis = 0)
          v_sum = np.sum(v[:, :], axis=0)

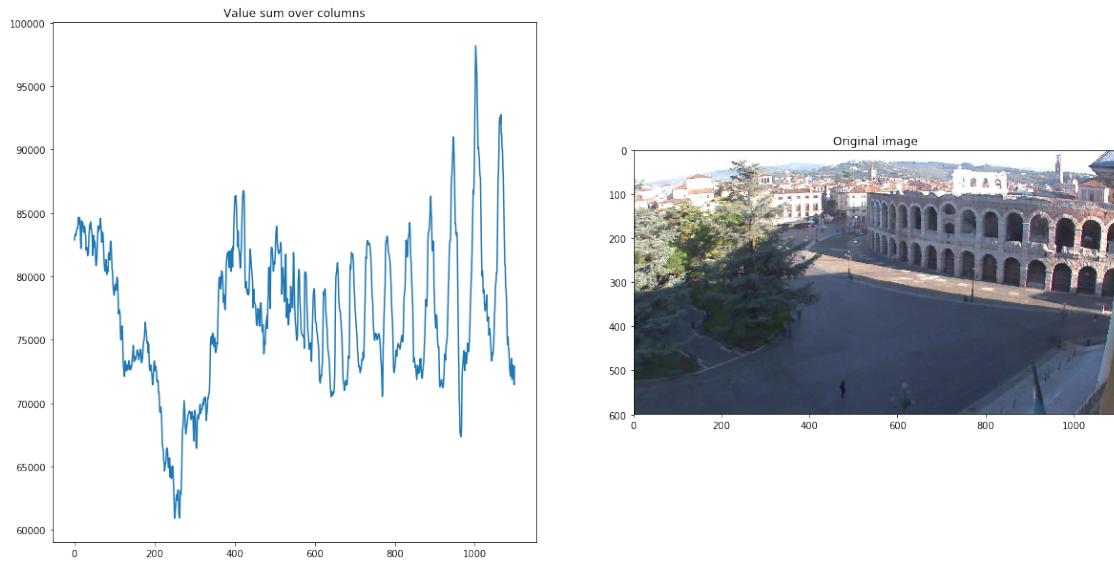
          f, (ax1, ax2) = plt.subplots(1, 2, figsize=(20,10))

          ax1.set_title('Value sum over columns')
          ax1.plot(v_sum)

          ax2.set_title('Original image')
          ax2.imshow(day_image, cmap='gray')

```

Out[10]: <matplotlib.image.AxesImage at 0x7f7572a6ae80>



In []:

Classification

March 31, 2020

0.1 # Day and Night Image Classifier

The day/night image dataset consists of 200 RGB color images in two categories: day and night. There are equal numbers of each example: 100 day images and 100 night images.

We'd like to build a classifier that can accurately label these images as day or night, and that relies on finding distinguishing features between the two types of images!

Note: All images come from the [AMOS dataset](#) (Archive of Many Outdoor Scenes).

0.1.1 Import resources

Before you get started on the project code, import the libraries and resources that you'll need.

```
In [1]: import cv2 # computer vision library
        import helpers

        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.image as mpimg

        %matplotlib inline
```

0.2 Training and Testing Data

The 200 day/night images are separated into training and testing datasets.

- 60% of these images are training images, for you to use as you create a classifier.
- 40% are test images, which will be used to test the accuracy of your classifier.

First, we set some variables to keep track of some where our images are stored:

```
image_dir_training: the directory where our training image data is stored
image_dir_test: the directory where our test image data is stored
```

```
In [2]: # Image data directories
        image_dir_training = "day_night_images/training/"
        image_dir_test = "day_night_images/test/"
```

0.3 Load the datasets

These first few lines of code will load the training day/night images and store all of them in a variable, IMAGE_LIST. This list contains the images and their associated label ("day" or "night").

For example, the first image-label pair in IMAGE_LIST can be accessed by index: IMAGE_LIST[0][:] .

```
In [3]: # Using the load_dataset function in helpers.py
# Load training data
IMAGE_LIST = helpers.load_dataset(image_dir_training)
```

0.4 Construct a STANDARDIZED_LIST of input images and output labels.

This function takes in a list of image-label pairs and outputs a **standardized** list of resized images and numerical labels.

```
In [4]: # Standardize all training images
STANDARDIZED_LIST = helpers.standardize(IMAGE_LIST)
```

0.5 Visualize the standardized data

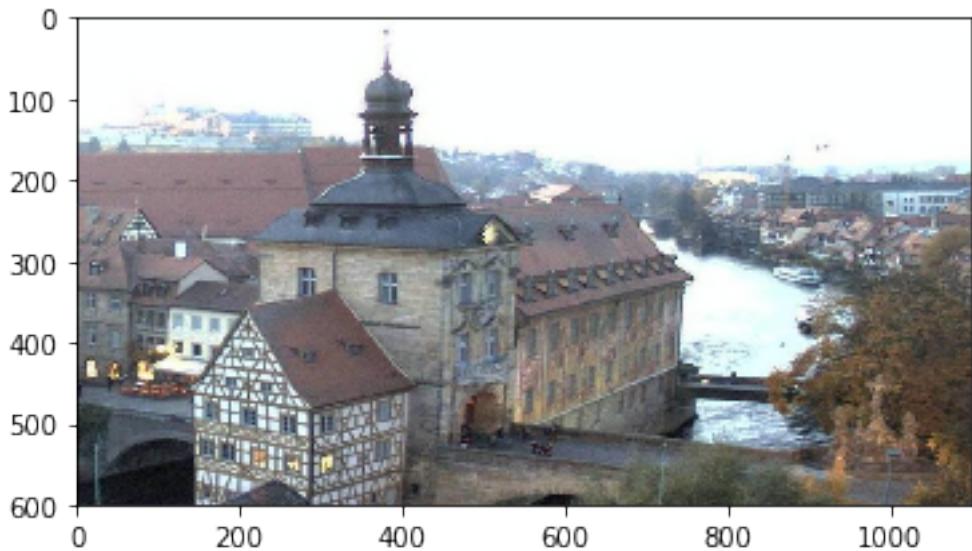
Display a standardized image from STANDARDIZED_LIST.

```
In [5]: # Display a standardized image and its label

# Select an image by index
image_num = 0
selected_image = STANDARDIZED_LIST[image_num][0]
selected_label = STANDARDIZED_LIST[image_num][1]

# Display image and data about it
plt.imshow(selected_image)
print("Shape: "+str(selected_image.shape))
print("Label [1 = day, 0 = night]: " + str(selected_label))

Shape: (600, 1100, 3)
Label [1 = day, 0 = night]: 1
```



1 Feature Extraction

Create a feature that represents the brightness in an image. We'll be extracting the **average brightness** using HSV colorspace. Specifically, we'll use the V channel (a measure of brightness), add up the pixel values in the V channel, then divide that sum by the area of the image to get the average Value of the image.

1.0.1 Find the average brightness using the V channel

This function takes in a **standardized** RGB image and returns a feature (a single value) that represent the average level of brightness in the image. We'll use this value to classify the image as day or night.

```
In [6]: # Find the average Value or brightness of an image
def avg_brightness(rgb_image):
    # Convert image to HSV
    hsv = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2HSV)

    # Add up all the pixel values in the V channel
    sum_brightness = np.sum(hsv[:, :, 2])
    area = 600*1100.0  # pixels

    # find the avg
    avg = sum_brightness/area

return avg
```

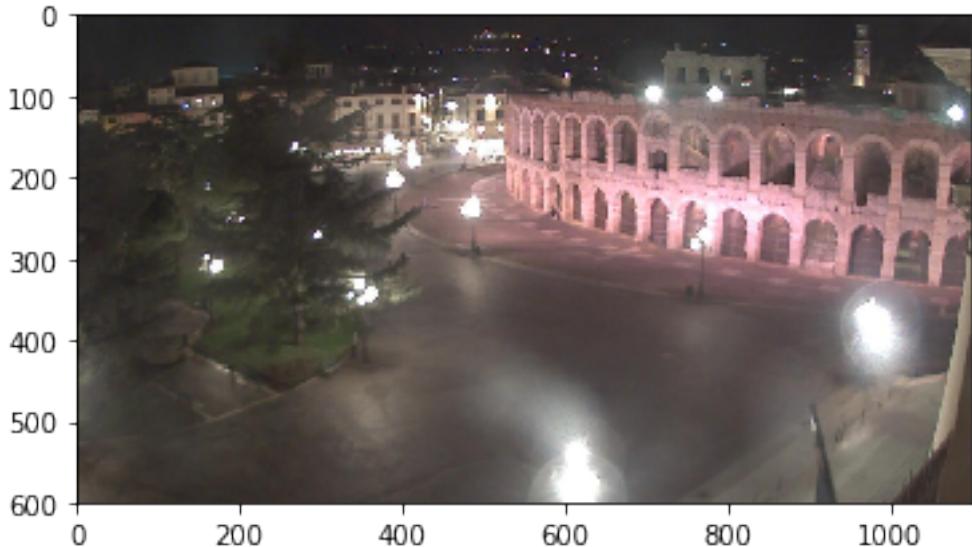
```
In [7]: # Testing average brightness levels
# Look at a number of different day and night images and think about
# what average brightness value separates the two types of images

# As an example, a "night" image is loaded in and its avg brightness is displayed
image_num = 190
test_im = STANDARDIZED_LIST[image_num][0]

avg = avg_brightness(test_im)
print('Avg brightness: ' + str(avg))
plt.imshow(test_im)
```

Avg brightness: 99.2990121212

Out[7]: <matplotlib.image.AxesImage at 0x7f18759e2b38>



2 Classification and Visualizing Error

In this section, we'll turn our average brightness feature into a classifier that takes in a standardized image and returns a `predicted_label` for that image. This `estimate_label` function should return a value: 0 or 1 (night or day, respectively).

2.0.1 TODO: Build a complete classifier

Set a threshold that you think will separate the day and night images by average brightness.

```
In [12]: # This function should take in RGB image input
def estimate_label(rgb_image):

    ## TODO: extract average brightness feature from an RGB image
    # Use the avg brightness feature to predict a label (0, 1)
    predicted_label = 0

    ## TODO: set the value of a threshold that will separate day and night images
    threshold = 100

    ## TODO: Return the predicted_label (0 or 1) based on whether the avg is
    # above or below the threshold
    if(avg_brightness(rgb_image) > threshold):
        predicted_label = 1

    return predicted_label
```

```
In [40]: ## Test out your code by calling the above function and seeing
# how some of your training data is classified
```

```
image_num = 183
test_im = STANDARDIZED_LIST[image_num][0]

prediction = estimate_label(test_im)

print('Prediction is [0: Night, 1: Day] --> ' + str(prediction))
plt.imshow(test_im)
```

Prediction is [0: Night, 1: Day] --> 0

```
Out[40]: <matplotlib.image.AxesImage at 0x7f1871307828>
```



In []:

Accuracy and Misclassification

March 31, 2020

0.1 # Day and Night Image Classifier

The day/night image dataset consists of 200 RGB color images in two categories: day and night. There are equal numbers of each example: 100 day images and 100 night images.

We'd like to build a classifier that can accurately label these images as day or night, and that relies on finding distinguishing features between the two types of images!

Note: All images come from the [AMOS dataset](#) (Archive of Many Outdoor Scenes).

0.1.1 Import resources

Before you get started on the project code, import the libraries and resources that you'll need.

```
In [1]: import cv2 # computer vision library
         import helpers

         import numpy as np
         import matplotlib.pyplot as plt
         import matplotlib.image as mpimg

         %matplotlib inline
```

0.2 Training and Testing Data

The 200 day/night images are separated into training and testing datasets.

- 60% of these images are training images, for you to use as you create a classifier.
- 40% are test images, which will be used to test the accuracy of your classifier.

First, we set some variables to keep track of some where our images are stored:

```
image_dir_training: the directory where our training image data is stored
image_dir_test: the directory where our test image data is stored
```

```
In [2]: # Image data directories
         image_dir_training = "day_night_images/training/"
         image_dir_test = "day_night_images/test/"
```

0.3 Load the datasets

These first few lines of code will load the training day/night images and store all of them in a variable, IMAGE_LIST. This list contains the images and their associated label ("day" or "night").

For example, the first image-label pair in IMAGE_LIST can be accessed by index: IMAGE_LIST[0][:] .

```
In [3]: # Using the load_dataset function in helpers.py
    # Load training data
    IMAGE_LIST = helpers.load_dataset(image_dir_training)
```

0.4 Construct a STANDARDIZED_LIST of input images and output labels.

This function takes in a list of image-label pairs and outputs a **standardized** list of resized images and numerical labels.

```
In [4]: # Standardize all training images
    STANDARDIZED_LIST = helpers.standardize(IMAGE_LIST)
```

0.5 Visualize the standardized data

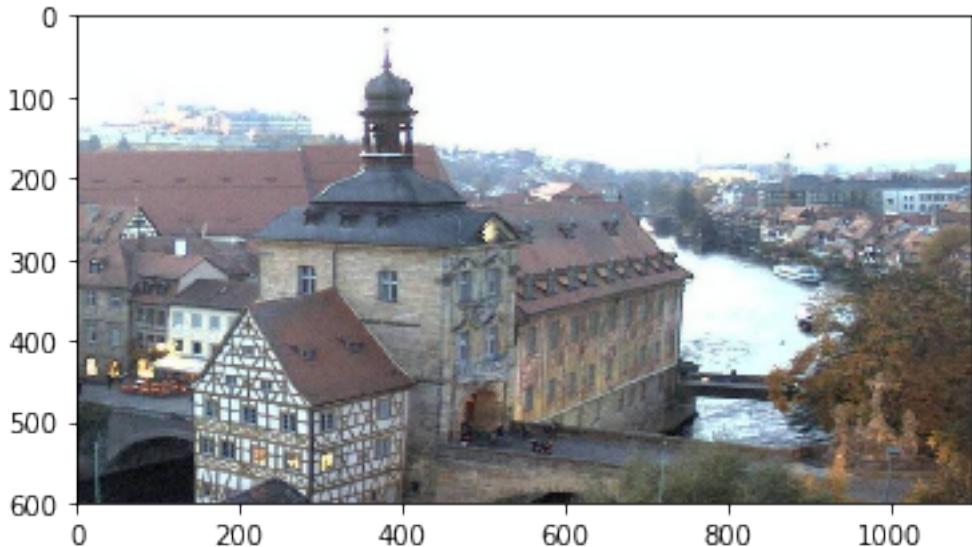
Display a standardized image from STANDARDIZED_LIST.

```
In [5]: # Display a standardized image and its label

    # Select an image by index
    image_num = 0
    selected_image = STANDARDIZED_LIST[image_num][0]
    selected_label = STANDARDIZED_LIST[image_num][1]

    # Display image and data about it
    plt.imshow(selected_image)
    print("Shape: "+str(selected_image.shape))
    print("Label [1 = day, 0 = night]: " + str(selected_label))

Shape: (600, 1100, 3)
Label [1 = day, 0 = night]: 1
```



1 Feature Extraction

Create a feature that represents the brightness in an image. We'll be extracting the **average brightness** using HSV colorspace. Specifically, we'll use the V channel (a measure of brightness), add up the pixel values in the V channel, then divide that sum by the area of the image to get the average Value of the image.

1.0.1 Find the average brightness using the V channel

This function takes in a **standardized** RGB image and returns a feature (a single value) that represent the average level of brightness in the image. We'll use this value to classify the image as day or night.

```
In [6]: # Find the average Value or brightness of an image
def avg_brightness(rgb_image):
    # Convert image to HSV
    hsv = cv2.cvtColor(rgb_image, cv2.COLOR_RGB2HSV)

    # Add up all the pixel values in the V channel
    sum_brightness = np.sum(hsv[:, :, 2])
    area = 600*1100.0  # pixels

    # find the avg
    avg = sum_brightness/area

return avg
```

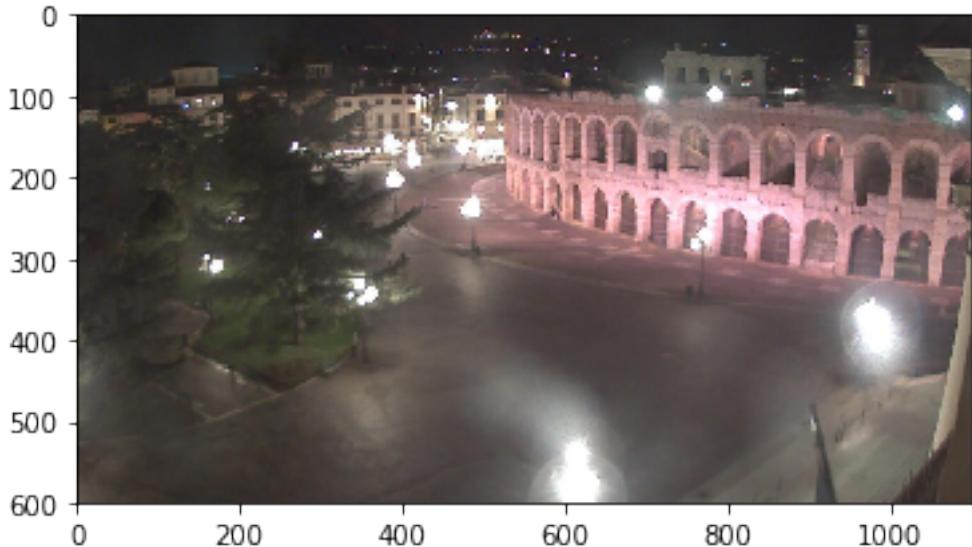
```
In [7]: # Testing average brightness levels
# Look at a number of different day and night images and think about
# what average brightness value separates the two types of images

# As an example, a "night" image is loaded in and its avg brightness is displayed
image_num = 190
test_im = STANDARDIZED_LIST[image_num][0]

avg = avg_brightness(test_im)
print('Avg brightness: ' + str(avg))
plt.imshow(test_im)
```

Avg brightness: 99.2990121212

Out[7]: <matplotlib.image.AxesImage at 0x7fb384e7a080>



2 Classification and Visualizing Error

In this section, we'll turn our average brightness feature into a classifier that takes in a standardized image and returns a `predicted_label` for that image. This `estimate_label` function should return a value: 0 or 1 (night or day, respectively).

2.0.1 TODO: Build a complete classifier

Complete this code so that it returns an estimated class label given an input RGB image.

```
In [8]: # This function should take in RGB image input
def estimate_label(rgb_image):

    # TO-DO: Extract average brightness feature from an RGB image

    avg = avg_brightness(rgb_image)

    # Use the avg brightness feature to predict a label (0, 1)
    predicted_label = 0
    # TO-DO: Try out different threshold values to see what works best!
    threshold = 100
    if(avg > threshold):
        # if the average brightness is above the threshold value, we classify it as "day"
        predicted_label = 1
    # else, the predicted_label can stay 0 (it is predicted to be "night")

    return predicted_label
```

2.1 Testing the classifier

Here is where we test your classification algorithm using our test set of data that we set aside at the beginning of the notebook!

Since we are using a pretty simple brightness feature, we may not expect this classifier to be 100% accurate. We'll aim for around 75-85% accuracy using this one feature.

2.1.1 Test dataset

Below, we load in the test dataset, standardize it using the `standardize` function you defined above, and then `shuffle` it; this ensures that order will not play a role in testing accuracy.

```
In [9]: import random

# Using the load_dataset function in helpers.py
# Load test data
TEST_IMAGE_LIST = helpers.load_dataset(image_dir_test)

# Standardize the test data
STANDARDIZED_TEST_LIST = helpers.standardize(TEST_IMAGE_LIST)

# Shuffle the standardized test data
random.shuffle(STANDARDIZED_TEST_LIST)
```

2.2 Determine the Accuracy

Compare the output of your classification algorithm (a.k.a. your "model") with the true labels and determine the accuracy.

This code stores all the misclassified images, their predicted labels, and their true labels, in a list called `misclassified`.

```
In [10]: # Constructs a list of misclassified images given a list of test images and their labels
def get_misclassified_images(test_images):
    # Track misclassified images by placing them into a list
    misclassified_images_labels = []

    # Iterate through all the test images
    # Classify each image and compare to the true label
    for image in test_images:

        # Get true data
        im = image[0]
        true_label = image[1]

        # Get predicted label from your classifier
        predicted_label = estimate_label(im)

        # Compare true and predicted labels
        if(predicted_label != true_label):
            # If these labels are not equal, the image has been misclassified
            misclassified_images_labels.append((im, predicted_label, true_label))

    # Return the list of misclassified [image, predicted_label, true_label] values
    return misclassified_images_labels
```

```
In [11]: # Find all misclassified images in a given test set
MISCLASSIFIED = get_misclassified_images(STANDARDIZED_TEST_LIST)

# Accuracy calculations
total = len(STANDARDIZED_TEST_LIST)
num_correct = total - len(MISCLASSIFIED)
accuracy = num_correct/total

print('Accuracy: ' + str(accuracy))
print("Number of misclassified images = " + str(len(MISCLASSIFIED)) +' out of '+ str(total))
```

Accuracy: 0.925
Number of misclassified images = 12 out of 160

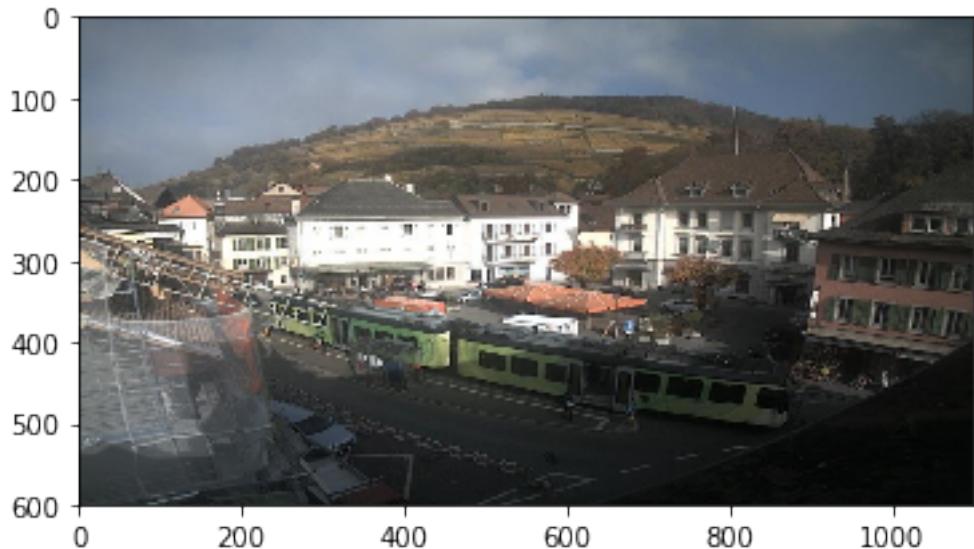
TO-DO: Visualize the misclassified images

Visualize some of the images you classified wrong (in the MISCLASSIFIED list) and note any qualities that make them difficult to classify. This will help you identify any weaknesses in your classification algorithm.

```
In [16]: # Visualize misclassified example(s)
num = 10
test_mis_im = MISCLASSIFIED[num][0]
```

```
plt.imshow(test_mis_im)
## TODO: Display an image in the `MISCLASSIFIED` list
## TODO: Print out its predicted label -
## to see what the image *was* incorrectly classified as
```

Out[16]: <matplotlib.image.AxesImage at 0x7fb384aa59e8>



(Question): After visualizing these misclassifications, what weaknesses do you think your classification algorithm has?

Answer: Write your answer, here.

3 5. Improve your algorithm!

- (Optional) Tweak your threshold so that accuracy is better.
-

3.1 (Optional) Add another feature that tackles a weakness you identified!

In []: