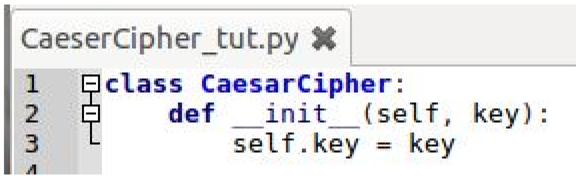
Caesar Cipher - Python

In this tutorial we will write an object oriented version of the Caesar Cipher. The Caesar Cipher was used to encrypt simple text messages by shifting all the letters by a fixed amount (the key). I am assuming you know the mechanics of the cipher, but if you don't you can read about it here.

https://learncryptography.com/classical-encryption/caesar-cipher

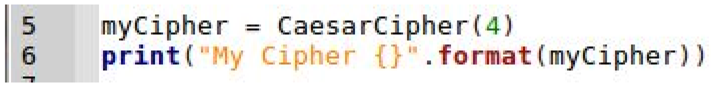
Create a Class

In Object Oriented programming, *classes* are used to associate methods with data. This allows the programmer to encapsulate concepts in reusable units of code. We will see how this works by diving straight into our cipher class. Start a new Python file and type the following.



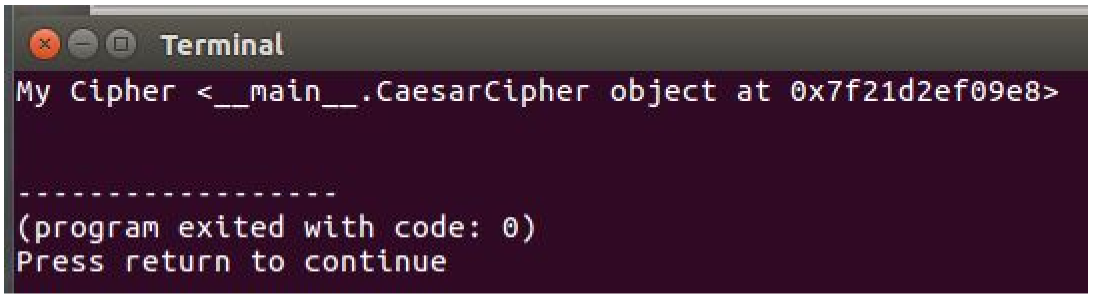
```
1  class CaesarCipher:
2      def __init__(self, key):
3          self.key = key
```

This gives us a class called **CaesarCipher**, and we are going to give that class a single bit of data to play with; the key. The function that starts on line 2 is a special function known as a *constructor*. It will be called when we create an *instance* of our class. Let us do this now.



```
5  myCipher = CaesarCipher(4)
6  print("My Cipher {}".format(myCipher))
```

Line 5 creates an instance of our cipher, and sets the **key** to 4. We do not have to pass in **self**, this is done for us by Python. Line 6 is just to print the object we have created. Run the code and the output will look something like this:

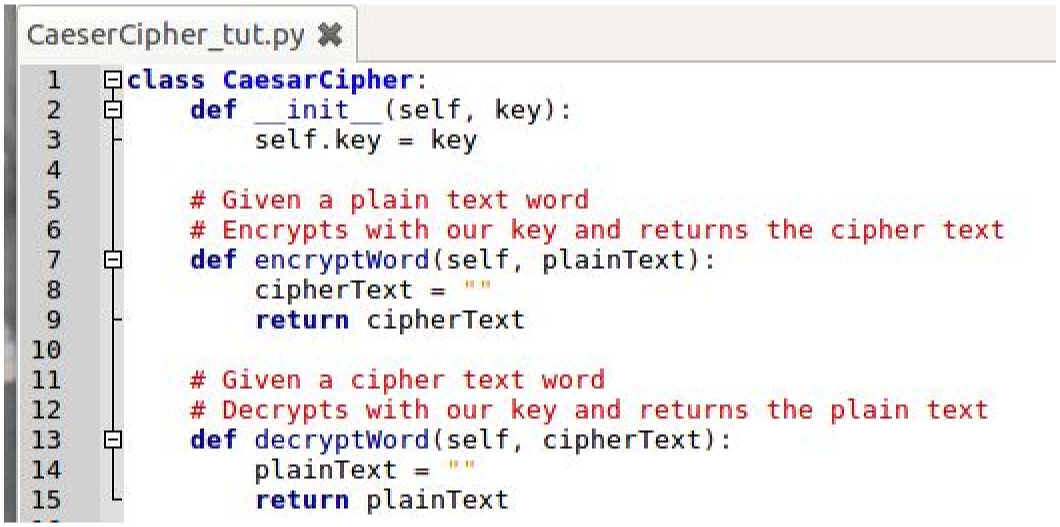


```
My Cipher <__main__.CaesarCipher object at 0x7f21d2ef09e8>


------------------
(program exited with code: 0)
Press return to continue
```

Eww...well we don't need to print the cipher object anyway.

Add Methods to Class

For our class to be useful, we should now add functions that can be called. I will be adding 2 functions to start with, **decryptWord** and **encryptWord**. Take note of the indentation, the methods are shown to be inside the class. The functions will look like this (lines 5 to 15):



```
 1  class CaesarCipher:
 2      def __init__(self, key):
 3          self.key = key
 4
 5      # Given a plain text word
 6      # Encrypts with our key and returns the cipher text
 7      def encryptWord(self, plainText):
 8          cipherText = ""
 9          return cipherText
10
11      # Given a cipher text word
12      # Decrypts with our key and returns the plain text
13      def decryptWord(self, cipherText):
14          plainText = ""
15          return plainText
```

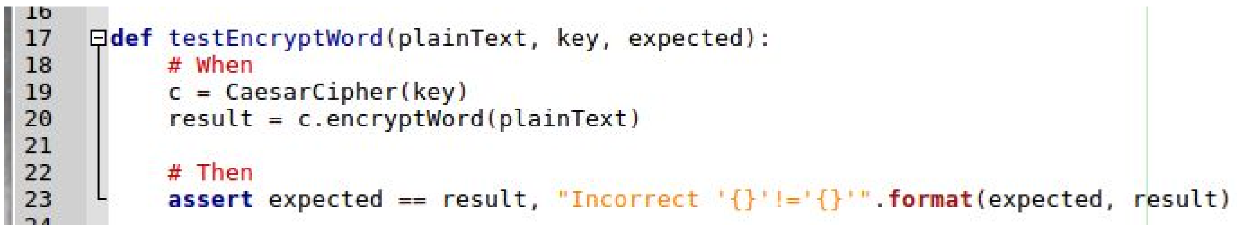
Note that I have added some comments to the functions, this will help programmers that read the code to understand what is going on. They are ignored by Python during execution though so you can leave them out if you want to.

Both of our class methods require the first parameter to be **self**, later you will see how this is used to refer to the self.key in the encrypt and decrypt functions.

Write a Test

It should be fairly obvious that our functions do not yet work, they simply return a blank string no matter what the input is. But let us write a test anyway. We can then develop the encryption and decryption until the tests pass!

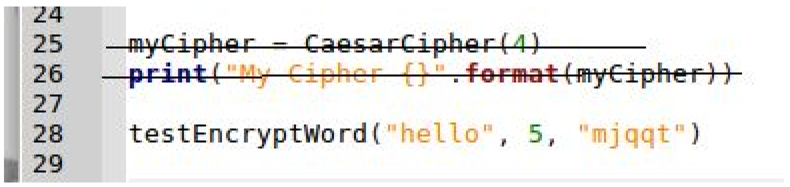
I will write a function for testing the encryption of a word, it will use the **assert** keyword to ensure that the result is correct. This will give us a chance to see how assertions are handled when they fail. Write the following function below the Caesar Cipher class (note indentation; it will be outside of the class).



```python
def testEncryptWord(plainText, key, expected):
    # When
    c = CaesarCipher(key)
    result = c.encryptWord(plainText)

    # Then
    assert expected == result, "Incorrect '{}'!='{}'".format(expected, result)
```

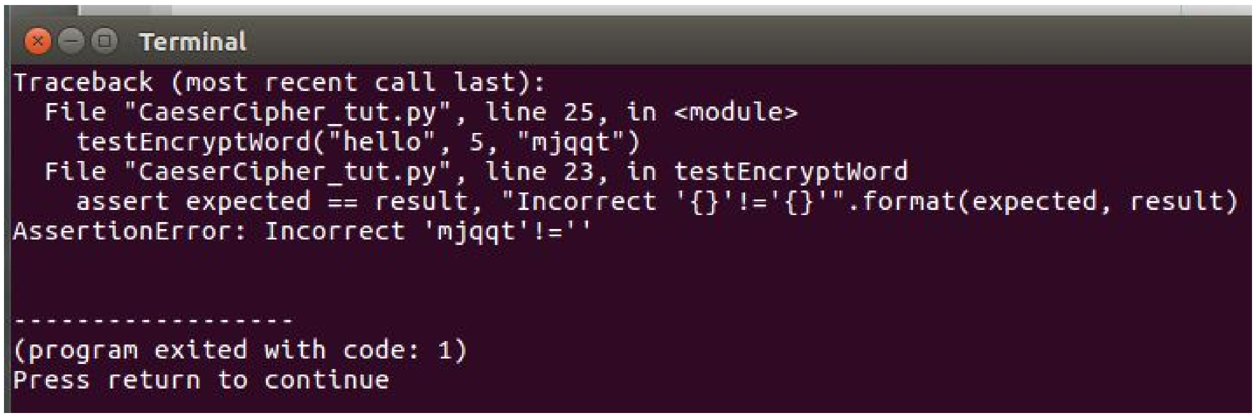
The assert line checks that the result matches the expected one, it prints an error message using the format function on string. The curly brackets {} are used as placeholders for the parameters in the format function.

Let us see this test function in action, delete the two lines (shown here as 25 and 26) we wrote earlier and add the line shown here as line 28



```
24
25  ~~myCipher = CaesarCipher(4)~~
26  ~~print("My Cipher {}".format(myCipher))~~
27
28  testEncryptWord("hello", 5, "mjqqt")
29
```

Run this and the test should fail, the terminal print will look something like this.



```
Traceback (most recent call last):
  File "CaeserCipher_tut.py", line 25, in <module>
    testEncryptWord("hello", 5, "mjqqt")
  File "CaeserCipher_tut.py", line 23, in testEncryptWord
    assert expected == result, "Incorrect '{}'!='{}'".format(expected, result)
AssertionError: Incorrect 'mjqqt'!=''


------------------
(program exited with code: 1)
Press return to continue
```

The part after the **AssertionError:** is the message we composed in our code, this lets you see what the problem is.

Do Yourself

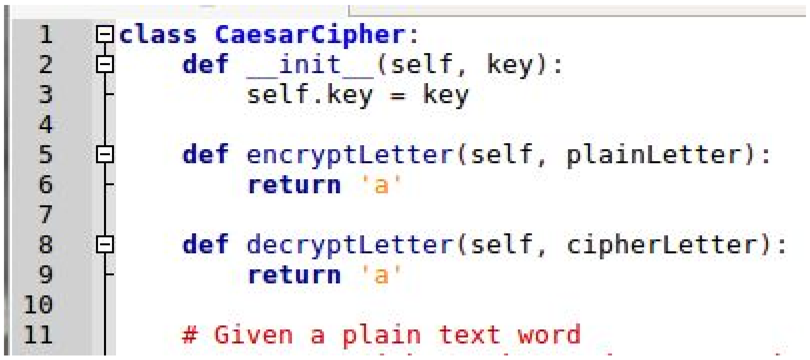
Write a decrypt word function using the following test data

- Cipher Text: mehbs
- Key: 15
- Expected Plain Text: world

Write Encrypt/Decrypt Letter Functions

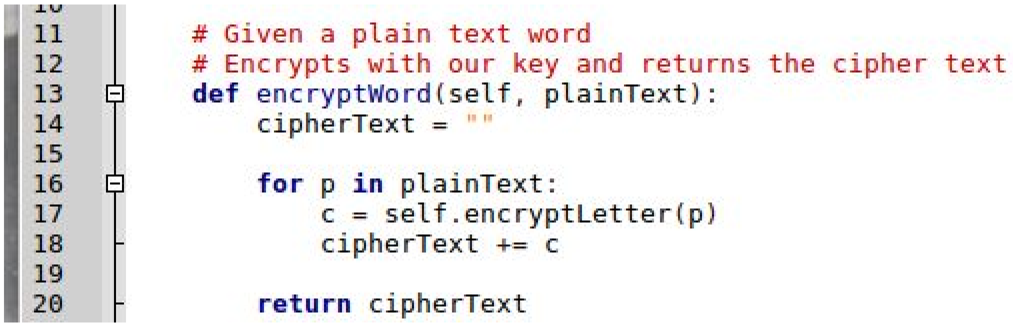
Our encryption algorithm can work on each character in the word separately, so I am going to add functions for **encryptLetter** and **decryptLetter**. We can then loop through the letters in our words in the **encryptWord** and **decryptWord** functions and build our text that way.

Add the following functions to our class. I have put them above our decryptWord and encryptWord functions.



```python
class CaesarCipher:
    def __init__(self, key):
        self.key = key

    def encryptLetter(self, plainLetter):
        return 'a'

    def decryptLetter(self, cipherLetter):
        return 'a'

    # Given a plain text word
```

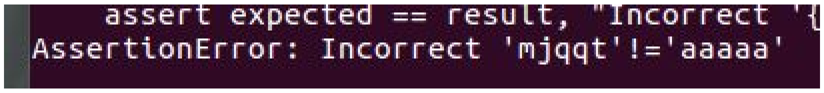
I will show you how to use the encryptLetter function in our encryptWord function.



```python
    # Given a plain text word
    # Encrypts with our key and returns the cipher text
    def encryptWord(self, plainText):
        cipherText = ""

        for p in plainText:
            c = self.encryptLetter(p)
            cipherText += c

        return cipherText
```

I am using a simple for loop to iterate through the letters of our plainText, then using the neat += operator to add the encrypted letter to our cipher text.

Rerun the tests and the failure should look different.



```
    assert expected == result, "Incorrect '{
AssertionError: Incorrect 'mjqqt'!='aaaaa'
```

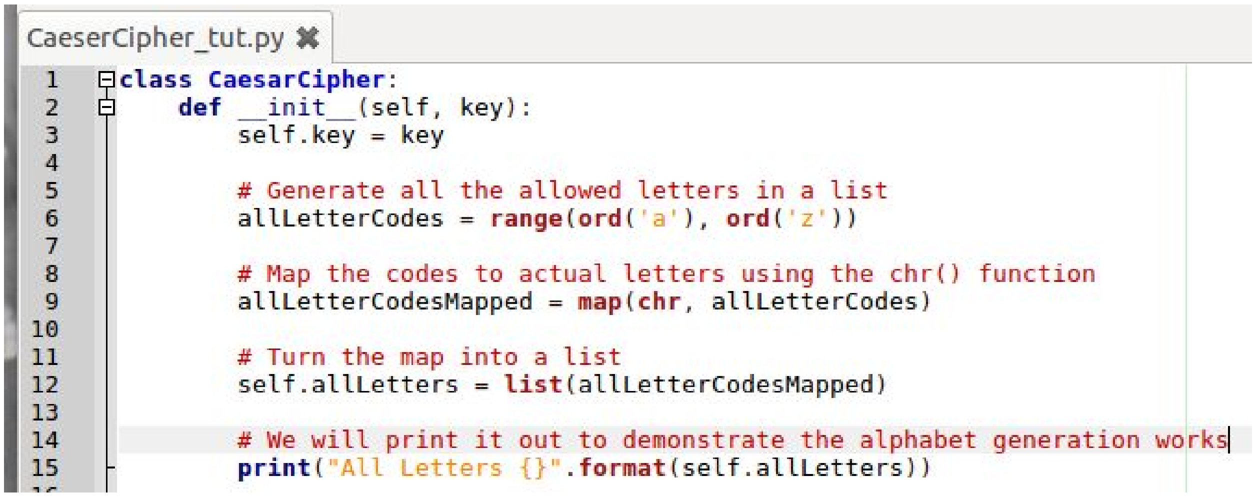
The incorrect value is now the right length, but the characters are all ‘a’ which is obviously no good.

Do Yourself

Fill in the decrypt word function in a similar fashion. Rerun the test from earlier (comment out the encrypt test call so it doesn't interfere). You should find the decrypted text is the right length, but the wrong contents.

Generate an Alphabet

Our algorithm requires an alphabet of letters in a list, we will use the range and map functions in Python to do this very succinctly. Add the following code to our constructor.



CaeserCipher_tut.py

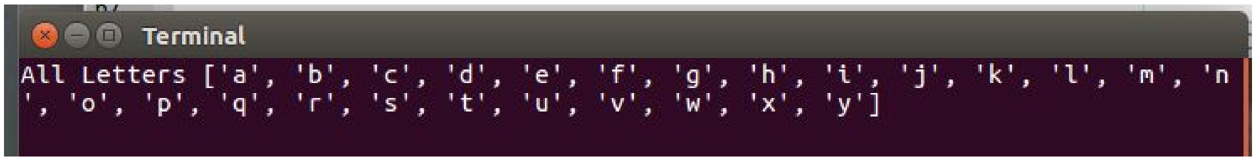
```
1   class CaesarCipher:
2       def __init__(self, key):
3           self.key = key
4
5           # Generate all the allowed letters in a list
6           allLetterCodes = range(ord('a'), ord('z'))
7
8           # Map the codes to actual letters using the chr() function
9           allLetterCodesMapped = map(chr, allLetterCodes)
10
11          # Turn the map into a list
12          self.allLetters = list(allLetterCodesMapped)
13
14          # We will print it out to demonstrate the alphabet generation works
15          print("All Letters {}".format(self.allLetters))
```

There is a lot of cool stuff here, so let's break it down.

Line 6: The **ord()** function is used to get the numerical representation of a character. Here we are going from lowercase a to z and generating a **range()**.

Line 9: The **map()** function is shorthand for a loop, it loops round the **allLetterCodes** list and applies the **chr()** function. This syntax is much tighter than using a for loop.

Line 12: The **list()** function is used to take the map and generate a list. I have then printed the **allLetters** variable (which is put onto the object using **self**) so that you can see it working properly.



Terminal

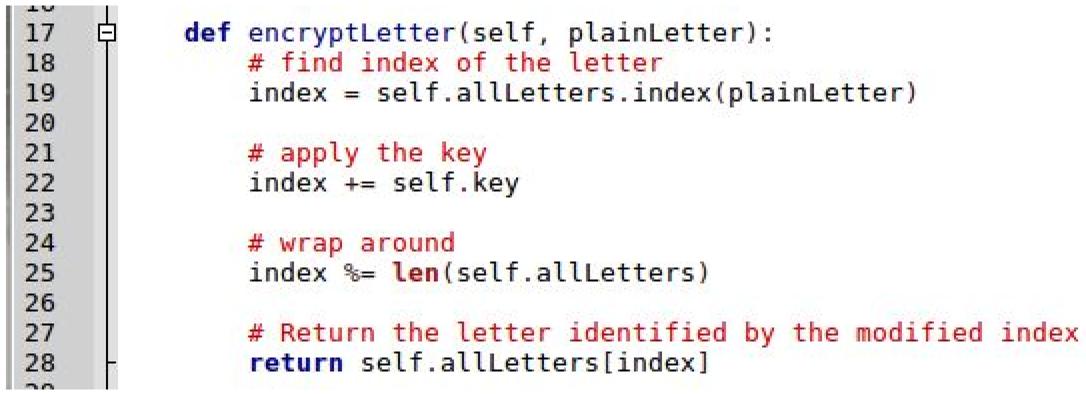
```
All Letters ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n
', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y']
```

Implement Encrypt Letter Correctly

We will now write the actual meat of the algorithm, the letter shifting. My method for this will be to create an array containing all the letters in the alphabet.
When given a letter to encrypt our program will:

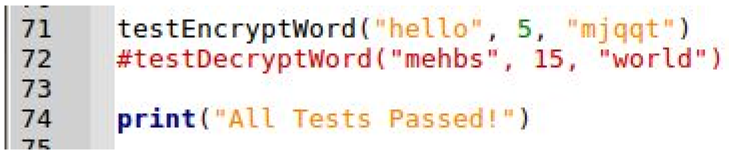
- Find the index of that letter in our alphabet
- Shift the index using the key
- If the index ends up larger than the alphabet we will use modulo to wrap it.
- Return the letter found at the new index.

The code looks like this

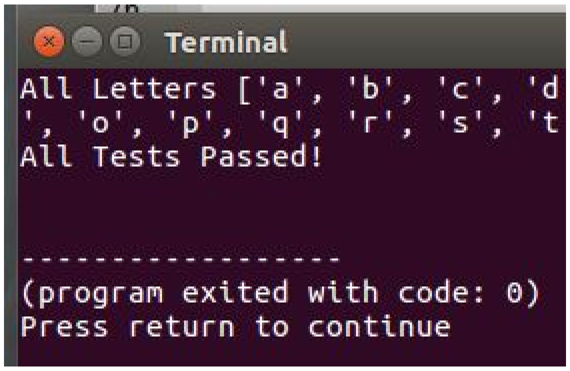


```python
    def encryptLetter(self, plainLetter):
        # find index of the letter
        index = self.allLetters.index(plainLetter)

        # apply the key
        index += self.key

        # wrap around
        index %= len(self.allLetters)

        # Return the letter identified by the modified index
        return self.allLetters[index]
```

Run your encryptWord test from earlier and you should find it passes. If the assertion passes then nothing is printed, so you will just see the All Letters print from earlier and the usual (program exited….) so you may wish to add a print line to the end of the code as shown, I have commented out the decrypt test (that you wrote earlier) so that it doesn't spoil the glorious moment of successful encryption.

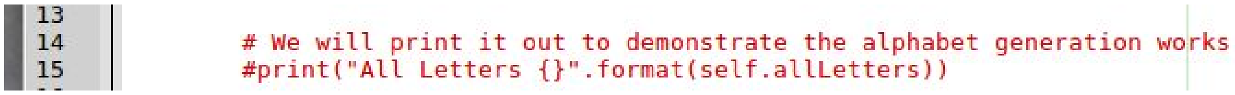


```python
testEncryptWord("hello", 5, "mjqqt")
#testDecryptWord("mehbs", 15, "world")

print("All Tests Passed!")
```



```
All Letters ['a', 'b', 'c', 'd
', 'o', 'p', 'q', 'r', 's', 't
All Tests Passed!


------------------
(program exited with code: 0)
Press return to continue
```

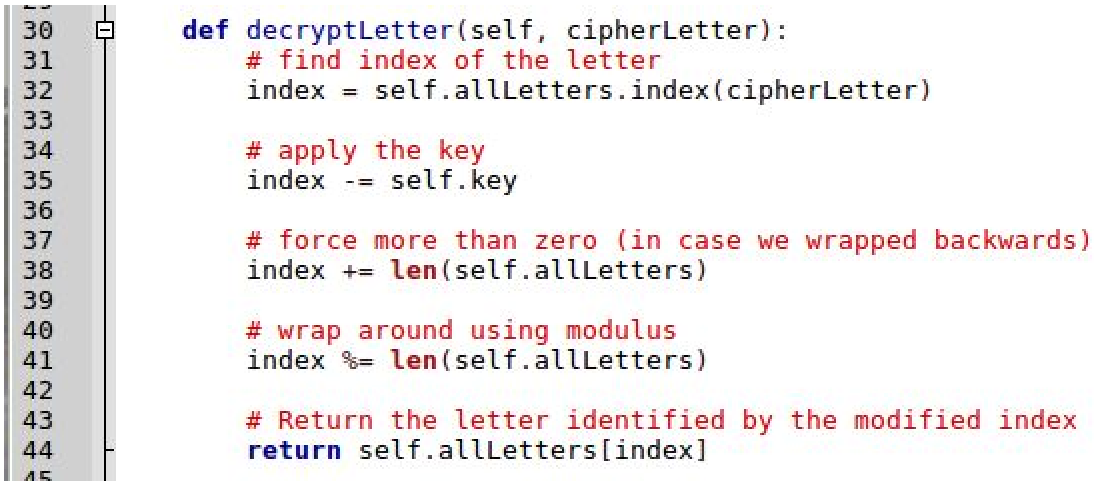
You may wish to comment out the alphabet print once you have satisfied yourself it is working.



```
        # We will print it out to demonstrate the alphabet generation works
        #print("All Letters {}".format(self.allLetters))
```

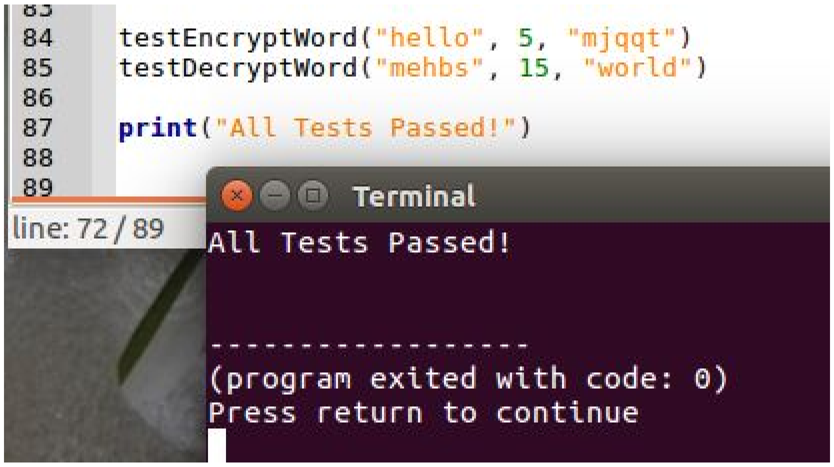
Implement Decrypt Letter Correctly

We will go through essentially the reverse process in decrypt letter. The code for that function will look like this.



```
    def decryptLetter(self, cipherLetter):
        # find index of the letter
        index = self.allLetters.index(cipherLetter)

        # apply the key
        index -= self.key

        # force more than zero (in case we wrapped backwards)
        index += len(self.allLetters)

        # wrap around using modulus
        index %= len(self.allLetters)

        # Return the letter identified by the modified index
        return self.allLetters[index]
```

Let your decrypt test function run and you should now find that all tests pass!



```
testEncryptWord("hello", 5, "mjqqt")
testDecryptWord("mehbs", 15, "world")

print("All Tests Passed!")
```

```
All Tests Passed!


------------------
(program exited with code: 0)
Press return to continue
```

Congratulations! You have built a Caesar Cipher and done a bit of testing.

Further Work

Now that your cipher works, try adding some user interaction using the input function. There are three things to capture

- If the user wishes to encrypt or decrypt
- The text to process
- The key

The program can then just print out the answer.