Start Coding with Python

My Solution To MIT 6.0001 Introduction to Computer Science and Programming in Python Problem Sets

by Jiarui Michelle Xie

10th Grade, 2024

Start Coding with Python

My Solution To MIT 6.0001 Introduction to Computer Science and Programming in Python Problem Sets

Published by Sisters of Impact Press
Hattiesburg, Mississippi 39402

Author: Jiarui Michelle Xie
Title: Start Coding with Python: My Solution To MIT 6.0001 Introduction to Computer Science and Programming in Python Problem Sets
Description: Hattiesburg, Mississippi: Python Learning Notes, 2024

Printed in the United States of America

Contents

Chapter 1

Introducing python

1.1 Introduction

What is python?

Python is a high-level, general-purpose programming language. It is commonly used for developing websites and soft ware as well as data analysis. Created by Guido van Rossum and first released in 1991, Python has since become one of the most widely used programming languages. Its clear and concise syntax allows users to express ideas and concepts in fewer lines of code than languages like C++ or Java, making it an ideal choice for both beginners and experienced programmers. There are two main types of functions in python: built-in functions and user-defined functions. There are 68 built-in functions like: str(), int(), float() etc, and infinitely many user-defined funtions.
Also, python has played a pivotal role in our lives, often without us noticing. For example, many games like The Sims 4 and Battlefield 2 used python. Also, in the AI world, python is a dominant force. Many of the frameworks and libraries essential for developing AI applications, such as TensorFlow and PyTorch, are Python-based.

1.2 Inputs

When using the input() function in python, we are asking the user to type in something. Whatever the user enters as an input, the input function converts it to a string (string is letters/words). Within the parenthesis, we usually ask the user what they should input.
For example:
input("How old are you?")
Make sure that when you type in the question, add quotations around it! Or else it will not appear in the terminal.
Most of the time, we name the input as a variable.
For example:
Question = input("How old are you?")
This is for convenience. Later on in the program, instead of having to type out the input and the

question every time, we can just use the variable “Question”.

1.3 Float, Integer, String

Float and integers are used to represent numbers. Float returns a floating-point number for a provided number. In other words, it returns it in decimal form. Integers only return integer numbers. We express these in the form of float() and int(). String is a collection of alphabets, words or other characters. Python has a built-in string class named str().Normally when we ask the user to input a number that has to be either a float or integer, we combine the two steps into one (but don’t forget to close all of the parenthesis when you are done!)
Example:
Question = int(input(“How old are you?”)) *notice how I used two parenthesis at the end. Also, if the user enters something that is not an integer, the program will stop and tell you that there is something wrong.
Example:
Question = float(input(“How old are you?”))

1.4 For/While loops

In Python, loops are used to repeatedly execute a block of code as long as a certain condition is true or for a specified number of times. There are two main types of loops in Python: for loop and while loop.
For loops
The for loop in Python is used to iterate over a sequence (such as a string) and execute a block of code for each item in the sequence. Generally, you will use for loops when you know the number of iterations you need to do (e.g. 500 trials). Using for loops will decrease the risk of writing the same loop infinite number of times.

```
#Python Program Example 1:
for i in range (5):
        print("I love programming!")
```

The code should print out:

```
I love programming!
I love programming!
I love programming!
I love programming!
I love programming!
```

While loops
The while loop is used to repeatedly execute a block of code as long as a specified condition is true. If you are waiting for the user to enter an input correctly or are waiting for a randomly generated

value to exceed a certain amount, you'll want to use a while loop. Problems that can be described using "until" should use while loops.

```
#Python Program Example 1:
num = 1
while num<=5:
        print(num, "John")
        num = num+1
```

In the example above, it should print out:

```
1 John
2 John
3 John
4 John
5 John
```

Also, in "Python Program Example 1", we have updated "num" to be "num + 1" in the while loop. This can be a little confusing, but basically when running this program, it will go down each function once. However when it bumps into loops it will run that specific loop over and over again until it reaches the requirement. Using the example, we see that the number starts off as 1 and the while loop will run until the number is less than or equal to five. Thus, the program printed out 1,2,3,4,5 and then stopped. However if you don't update the num, then it will print out "1. John" infinitely many times.

Chapter 2

Problem 1: House Hunting

Welcome to the first problem set in MIT's 6.0001 Introduction to Computer Science and Programming in Python! This MIT course consists of 5 problem sets in all. To access all of the information, please go to "https://ocw.mit.edu/courses/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/download/". With that being said, here's a little story to start you off...

2.1 Emma's story

In the city of Crestwood, Emma celebrated her graduation from the local university. With her degrees in hand, Emma began searching for jobs. She sent out countless resumes, attended job fairs, and faced a roller coaster of interviews. After a few months of perseverance, Emma landed a position as a graphic designer. Thrilled with her employment, she decided to embark on a new adventure – saving up to buy a house. Emma imagined a cozy place where she could build a future filled with happy memories. After searching for houses, she finally found her dream house. Everything was perfect. There was a nice fireplace, a gorgeous candelier, and even a jacuzzi in the backyard. Like I said, everything was perfect- that is, everything but the price. Emma's dream house costs 1 million dollars and her income wouldn't even be enough to pay for the down payment of the house. However, she were determined to purchase her dream house, no matter how long it would take to pay. But here comes the problem: how long would it take for Emma to save enough money to make the down payment of the house?

2.2 Just graduated!

> Big Results Require Big Ambition

Well, Emma's annual salary is called annual_salary and the cost of her dream house is called total_cost. Her goal right now is to save enough money to make the portion of the cost needed for

down payment (portion_down_payment), which is 25% of the total cost. The amount saved so far is 0 dollars (current_savings). Emma invests her current savings wisely and earns an annual_return of 4% (in other words, at the end of each month, she receives an additional fund: current_savings*annual_return/12 to put into her savings - the 12 is because the annual_return is an annual rate).

Earning from investment= current_savings*annual_return/12.

Emma decides to dedicate a certain amount of her salary each month to saving for the down payment (portion_saved, which should be in decimal form). At the end of each month, her savings will be increased by the return on her investment, plus a percentage of her monthly_salary (annual salary/12).
With the given information above, let's help Emma find out how long it would take her to save enough money to make the down payment!
These are the variables that will be needed throughout the code:

```
#code 1
current_savings = 0
num_month = 0
annual_return = 0.04
annual_salary = float(input("Enter your annual salary: "))
portion_saved = float(input("Enter the percent of salary to save,
                as a decimal: "))
total_cost = float(input("Enter the cost of your dream home: "))
portion_down_payment = 0.25*total_cost
monthly_salary = annual_salary/12
monthly_saving = monthly_salary*portion_saved
```

The num_month is what we are trying to find (aka the number of months it will take to make the down payment). Here, the monthly_saving is how much the user decides to save each month. The annual_salary, portion_saved, and total_cost are all what the user inputs.
With the basics down, let's get started with the actual code!
Our goal is to find out what is the minimum you have to save to reach the portion down payment. We shall use a while loop for this. e.g:

```
#code 2
while current_savings < portion_down_payment:
```

We must make the current savings less than the portion down payment in the while loop so that if the current savings is not enough, under the while loop it will run another month until the current savings is greater than or equal to the portion down payment. Our current savings begin with 0 dollars. However, under the while loop we need to update that value for every month to test if the saving is enough. e.g:

```
#code 3
current_savings = current_savings + monthly_saving
+ current_savings*annual_return/12
num_month = num_month + 1
```

This equation is taking the previous month's savings and adding it to the amount you save per month and then adding the earning from investment. This will result in the new and updated saving. Then we will update the number of months so it matches our new saving. Then we should print everything out.
Putting everything together, the finished code should look like this:

```
#code 4
current_savings = 0
num_month = 0
annual_return = 0.04
annual_salary = float(input("Enter your annual salary: "))
portion_saved = float(input("Enter the percent of salary to save,
                as a decimal: "))
total_cost = float(input("Enter the cost of your dream home: "))
portion_down_payment = 0.25*total_cost
monthly_salary = annual_salary/12
monthly_saving = monthly_salary*portion_saved
while current_savings < portion_down_payment:
        current_savings = current_savings + monthly_saving
                          + current_savings*annual_return/12
        num_month = num_month + 1
print("Number of months: ", num_month)
print("Current savings: ", current_savings)
```

Let's assume that Emma's annual salary is 120000, decides to save 10%, and her dream house is 1 million dollars. By using this code, we can help her solve her problem!

```
#result 1
Enter your annual salary: 120000
Enter the percent of salary to save, as a decimal: 0.1
Enter the cost of your dream home: 1000000
Number of months:  183
Current savings:  251569.61633503888
```

It turns out that she will need to save for 183 months.

2.3 Getting a raise

> Successful people are not gifted, they just work hard then succeed on purpose.

Emma has been very successful at her job and her boss has decided that she deserves a raise every 6 months! She now has a question: how long will it take for her to save up for the down payment, considering the fact that she has a raise?
Because this code is generally the same concept as the previous one, we should copy the previous

code and then add some new things:

```
#code 5
current_savings = 0
num_month = 0
annual_return = 0.04
annual_salary = float(input("Enter your annual salary: "))
portion_saved = float(input("Enter the percent of salary to save,
                as a decimal: "))
total_cost = float(input("Enter the cost of your dream home: "))
portion_down_payment = 0.25*total_cost
monthly_salary = annual_salary/12
monthly_saving = monthly_salary*portion_saved
```

We should first ask the user to input their raise, as a decimal (semi_annual_raise

```
#code 6
semi_annual_raise = float(input("Enter the semi_annual raise,
                as a decimal: "))
while current_savings < portion_down_payment:
    current_savings = current_savings + monthly_saving
                    + current_savings*annual_return/12
    num_month = num_month + 1
```

Remember: this is solving for the number of months without any raise, but now we must add an loop that can identify when the number of months reaches 6/when it is a multiple of 6 and during every 6 months there should be a raise. We should add an "if" loop.

```
#code 7
if num_month % 6==0:
    annual_salary = annual_salary*semi_annual_raise + annual_salary
    monthly_salary = annual_salary/12
    monthly_saving = monthly_salary*portion_saved
```

In the code above, the % symbol is called the modulo operator. It returns the remainder of dividing the left hand operand by right hand operand. Because we are trying to find multiples of 6, the remainder when the multiple divides 6 should be 0. Thus, if the remainder of the number of months divided by 6 is 0, it should continue with the code, else it won't run the rest of the code.
Now when the remainder is 0, it will update the annual_salary to the new salary with raise, which will lead to a new and updated monthly_salary and monthly_saving. Because in python the code will continue running from the top to the bottom until it reaches a stopping point, our code will go back to the "while" loop and run the code again, but now with the new current_savings (because we updated the monthly_saving in the "if" loop).
Also, keep in mind that it will not always run with the updated annual salary because once the number of months hits a number that is not a multiple of 6, it will not run the "if" loop, therefore the salary will stay the same(without the raise).

Assuming that Emma's raise is 10%, let's actually run the code!

```
#result 2
Enter your annual salary: 120000
Enter the percent of salary to save, as a decimal: 0.1
Enter the cost of your dream home: 1000000
portion_down_payment:  250000.0
Enter the semi_annual raise, as a decimal: 0.1
Number of months:  98
```

Perfect! Looks like it will only take Emma 98 months to pay for the down payment.

2.4 Setting up a Goal

> Setting goals is the first step in turning the invisible into the visible.

Emma has decided to be productive and wants to set up a goal. She wants to be able to afford the down payment in not 15 years, but in 3 years. Her question is: What is the best rate of savings to achieve her goal? Also, she would like to know if the annual salary she is making right now is possible to achieve that goal in 3 years.
Keep in mind that because hitting the exact rate is hard, the savings will be within 100 dollars of the required down payment.
Let's first copy the variables from the previous codes:

```
#code 8
current_savings = 0
annual_return = 0.04
num_month = 0
annual_salary = float(input("Enter your annual salary: "))
total_cost = float(input("Enter the cost of your dream home: "))
portion_down_payment = 0.25*total_cost
print("portion_down_payment: ", portion_down_payment)
monthly_salary = annual_salary/12
semi_annual_raise = 0.07
```

Next, we should first find out how much she needs to save in 3 years. We will define a variable and let's call it total_savings. We should be using the previous code, expect this time, we should set up a limit for the savings.

```
#code 9
    def total_savings(portion_saved, annual_salary):
        semi_annual_raise = 0.1
        num_month = 0
        monthly_salary = annual_salary/12
```

```
        monthly_saving = monthly_salary*portion_saved
        annual_return = 0.04
        current_savings = 0
        for i in range (36):
                current_savings = current_savings + monthly_saving +
                                  current_savings*annual_return/12
                num_month = num_month + 1
                if num_months%6 ==0:
                        annual_salary = annual_salary*semi_annual_raise
                                + annual_salary
                        monthly_salary = annual_salary/12
                        monthly_saving = monthly_salary*portion_saved
        return current_savings
```

This new function is basically the same as current_savings. The only difference is that this function will update the savings 36 times to reach the 3 year goal. Now that we have figured out how much she should save, let's figure out the best savings rate!
We should be using a method called the "bisection search". This is where we find a range where the solution lies, in this case the range will be between the lowest possible savings rate and the highest possible savings rate. Then we'll test the value in the middle of that range(which is just finding the average and this is where the "bisect" in "bisection" comes in because we are halving the range). Next we will determine if the solution is greater than or less than that value. Finally, we will change the bounds of the new range accordingly.
Starting with the code, we should first set up the range. Let's just call the lowest possible savings rate low_r and highest possible savings rate high_r. Of course, the lowest possible rate would be 0%, and highest would be 100%. But because we are using decimals, it should be 0 and 1. Then we will call how much money we need to save money_to_save and how many steps it will take to find the value steps. Both of these variables will start off with 0. Now we will set a while not loop. What we are looking for in this while loop is to put the money_to_save in a range because it will be extremely difficult to find the precise value. We want the range to be within 100 dollars of the down_payment.

```
    while not (portion_down_payment-100 < money_to_save
                                        < portion_down_payment+100):
```

Under this loop we will run the bisection search.

```
#code 10
#step1
    r = (low_r + high_r)/2
#step2
    r= int(r*10000)/10000
#step3
    money_to_save = total_savings(r, annual_salary)
```

In step 1 we are running the bisection search. In step two we are rounding the rate to the nearest

ten thousandths. Then for step 3 we are updating money_to_save. Next we will continue with the bisection search and further more break it down.

```
#code 11
    print( "money: ", money_to_save, "rate: ", r)
                if money_to_save>portion_down_payment:
                        high_r = r
                else:
                        low_r = r
                steps = steps + 1
```

We are printing the money_to_save and setting up an if loop. In the if loop, we are determining whether or not the money_to_save is greater than the portion_down_payment. If it is, we will adjust the high_r to r.
Now we will go back and add an if loop right under the definition of total_savings. This loop will tell us whether it is possible for Emma to pay the down payment in three years.

```
#code 12
    if total_savings(1, annual_salary)<portion_down_payment:
        print("It is not possible to pay the down payment
                in three years.")
    else:
        money_to_save = 0
        low_r = 0
        high_r = 1
        steps = 0
        while not (portion_down_payment-100 < money_to_save
                                            <portion_down_payment+100):
                r = (low_r + high_r)/2
                r= int(r*10000)/10000
                money_to_save = total_savings(r, annual_salary)
                print( "money: ", money_to_save, "rate: ", r)
                if money_to_save>portion_down_payment:
                        high_r = r
                else:
                        low_r = r
                steps = steps + 1
        print("Best Savings Rate: ", r)
        print("Steps in bisection search: ", steps)
```

Now we are finished! Let's run the code.

```
#result 3
Enter your annual salary: 120000
Enter the cost of your dream home: 1000000
portion_down_payment:  250000.0
money:  244141.6911874752 rate:  0.5
money:  366212.53678121296 rate:  0.75
```

```
money:  305177.11398434406 rate:  0.625
money:  274659.40258590964 rate:  0.5625
money:  259376.13271757375 rate:  0.5312
money:  251758.91195252456 rate:  0.5156
money:  247950.30156999987 rate:  0.5078
money:  249854.60676126232 rate:  0.5117
money:  250782.3451877746 rate:  0.5136
money:  250294.0618053996 rate:  0.5126
money:  250049.92011421212 rate:  0.5121
Best Savings Rate:  0.5121
Steps in bisection search:  11
```

We are finished! Now Emma will know how much to save and how long it will take.

Chapter 3

Hangman

3.1 Getting started

First we will implement a function called hangman that will allow the user to play hangman against the computer. The computer picks the word, and the player tries to guess letters in the word. We will put all of our code under this function so that at the end, all we have to do is print out this one function instead of many more.
First, we should go to "https://ocw.mit.edu/courses/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/download/", click on assignments, and download ps2.zip, which will contain the files hangman.py and words.txt. Hangman.py will give you the functions we will need and a brief explanation on what the function does. Let's define the functions:
secret_word is the word we are trying to guess. At the very top of the program, we will set the secret word to be "apple" to make the writing process easier. We will change it when we are done writing.

```
secret_word = 'apple'
```

letters_guessed is what letters the user guessed. Right under the secret word above, let's set the letters guessed to be: e, i, k, p, r, s.

```
letters_guessed = ['e', 'i', 'k', 'p', 'r', 's']
```

is_word_guessed takes in two parameters:secret_word and letters_guessed. This function returns a boolean - true if the letters in secret_word are in letters_guessed and false otherwise.
To write a code for this function, we will set up a new variable called all_letters_guessed.

```
def is_word_guessed(secret_word, letters_guessed):
    all_letters_guessed = all(letter in letters_guessed
                              for letter in secret_word)
    return all_letters_guessed
```

is_letter_guessed will tell us if the letter guessed is in the secret word.

```
def is_letter_guessed(letter_guessed, secret_word):
    return(letter_guessed in secret_word)
```

get_guessed_word takes in two parameters: secret_word and letters_guessed. This function will return a string that is comprised of letters and underscores, based on what letters from letters_guessed are in secret_word.
We will first set save_letter_guessed to be a blank space so that later on we can update this variable. Now let's set up a for loop.

```
  for letter in secret_word:
    if is_letter_guessed(letter, letters_guessed):
            save_letter_guessed = save_letter_guessed + letter
    else:
            save_letter_guessed = save_letter_guessed + "_ "
return save_letter_guessed
```

get_available_letters is a string that is comprised of lowercase English letters - all lowercase English letters that are not in letters_guessed.
We will also define a variable called available_letters and will set that to a blank space so we can update it later.

```
    available_letter = ""
    for letter in string.ascii_lowercase:
            if letter not in letters_guessed:
                    available_letter = available_letter + letter
    return available_letter
```

3.2 Set up for the Hangman game

Let's identify some of the requirements for the game:
The user starts off with 6 guesses. At the start of the game, let the user know how many letters the secret word contains and how many guesses s/he starts with.
1. Before each guess, you should display to the user: remind the user of how many guesses s/he has left after each guess, and all of the letters the user has not yet guessed.
2. Ask the user to supply one guess at a time
3. Immediately after each guess, the user should be told whether the letter is in the computer's word.
4. After each guess, you should also display to the user the computer's word, with guessed letters displayed and un-guessed letters replaced with an underscore and space(_)
5. The user also starts with 3 warnings.
6. If the user inputs anything besides an alphabet (symbols, numbers), tell the user that they can only input an alphabet.
    a. If the user has one or more warning left, the user should lose one warning. Tell the user

the number of remaining warnings.

b. If the user has no remaining warnings, they should lose one guess.

7. If the user inputs a letter that has already been guessed, print a message telling the user the letter has already been guessed before.

a. If the user has one or more warning left, the user should lose one warning. Tell the user the number of remaining warnings.

b. If the user has no warnings, they should lose one guess.

8. If the user inputs a letter that hasn't been guessed before and the letter is in the secret word, the user loses no guesses.

9. The game should end when the user constructs the full word or runs out of guesses.

10. If the player runs out of guesses before completing the word, tell them they lost and reveal the word to the user when the game ends.

11. If the user wins, print a congratulatory message and tell the user their score.

12. The total score is the number of guesses remaining once the user has guessed the secret word times the number of unique letters in the secret word.

Now that we know the requirements, let's get started!

So first off, under the function "hangman" we will first set the num_warnings to be 3, letters_guessed to be blank, and num_guesses to be 6. To find out how long the secret word is, we will use the "len" function.

```
num_warnings = 3
letters_guessed = ""
num_guesses = 6
num_letters = len(secret_word)
print("Welcome to the game Hangman!")
print("I am thinking of a word that is", num_letters,
      "letters long.")
print("You have 3 warnings left.")
print("You have 6 guesses left.")
print("Available letters: ", string.ascii_lowercase)
```

Now let's set up a while loop. While the number of guesses is greater than 0 and while not is_word_guessed, print the number of guesses, available letters, and also an input of the user's guessed letter. It should look like this:

```
while num_guesses > 0 and not is_word_guessed(secret_word,
                    letters_guessed):
    vowel ='aeiou'
    print("You have", num_guesses, "guesses left")
    print("Available letters: ", get_available_letters(letters_guessed))
    current_guess = input("Please guess a letter: ")
```

Then we will determine if the user's input is an alphabet or not because remember, in this game the user can only input an alphabet! We will be using the function str.isalpha. This function will check whether or not a character is an alphabet. This loop will only run if the input is not an alphabet. We will print "you can only input an alphabet". Also if the number of warnings has already reached 0,

then we will deducted one from the number of guesses as a punishment, if not then we will deducted from the number of warnings. Finally we will print everything out.

```
        if not str.isalpha(current_guess):
                print("You can only input an alphabet.")
                if num_warnings == 0:
                        num_guesses = num_guesses - 1
                else:
                        num_warnings = num_warnings - 1
                print("You have", num_warnings, "warnings left")
                print("Please guess a letter: ")
```

Next we will set up another if loop. If the current guess is an alphabet and if it is in the secret word, then we will run this loop. However, under this loop we need to write another if loop. If the number of warnings has reached 0, then we will, once again, deducted from the number of guesses. Else we will deducted one from the number of warnings.

```
if str.isalpha(current_guess) and is_letter_guessed(current_guess,
                                        letters_guessed):
        if num_warnings == 0:
                num_guesses = num_guesses - 1
        else:
                num_warnings = num_warnings - 1
        print("Oops! You've already guessed that letter.
            You now have", num_warnings, "warnings left.")
```

Then we will run ANOTHER if loop. If the current guess is an alphabet and if it is NOT in the letters guessed, then we will run this loop. However, just like the previous loop, we will have to run another if loop under this loop. If the current guess is in SECRET WORD, then we will print "good guess" and update the letters_guessed by adding the current guess to it. Else, print "oops, that letter is not in my word". Then come out of this if loop and update the letters_guessed just like we did previously.

```
if str.isalpha(current_guess) and not is_letter_guessed(
                    current_guess, letters_guessed):
    if is_letter_guessed(current_guess, secret_word):
            print("Good guess!")
            letters_guessed = letters_guessed + current_guess
            print(get_guessed_word(secret_word, letters_guessed))
    else:
            print("Oops! That letter is not in my word.")
    letters_guessed = letters_guessed + current_guess
```

Now let's come out of all of the if loops and make sure the indentation is aligned with the while loop. We have finished the main program will need an ending. If the user wins, we will print a congratulatory message and tell the user their score. The total scores is the number of guesses_remaining

once the user has guessed the secret_word times the number of unique letters in secret_word.

```
    guesses_remaining = num_guesses
        unique_letters_count = len(set(secret_word))
        total_score = guesses_remaining*unique_letters_count
        if is_word_guessed(secret_word, letters_guessed):
                print("Congratulations, you won!")
                print("Your total score for this game is: ",
                                                total_score)
        else:
                print("YOU LOST!!")
                print("The secret word was: ", secret_word)
```

The let's come out of all of the loops and print:

```
wordlist = load_words()
secret_word = choose_word(wordlist)
print(hangman(secret_word))
```

Here is the summed-up diagram of the code:

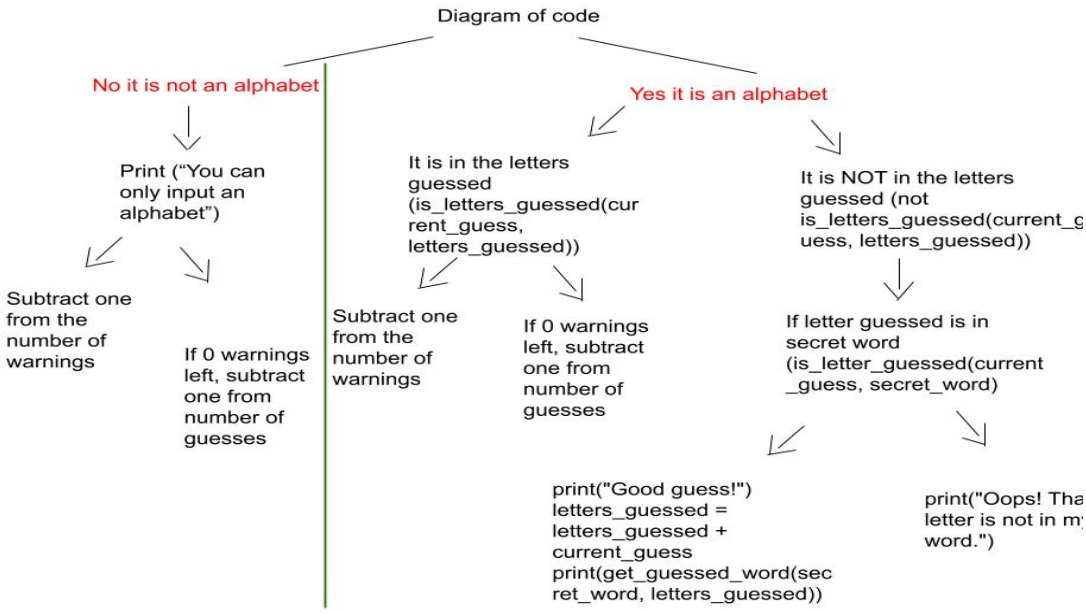

Figure 3.1: Diagram of hangman code

Although we are finished with the code, you can also add on a code for hints. Here's how.

3.3 Adding interface codes

We will need these functions: match_with_gaps(my_word, other_word), which takes in two parameters. My_word is an instance of a guessed word, in other words, it may have some _'s in place (such as 't_ _ t'). Other_word is just a normal English word. To make writing this code easier, let's set my_word to be b_ nana.This function should return True if the guessed letters of my_word match the corresponding letters of other_word. It should return False if the two words are not of the same length or if a guessed letter in my_word does not match the corresponding character in other_word. Remember that when a letter is guessed, your code reveals all the positions at which that letter occurs in the secret word. Therefore, the hidden letter (_) CANNOT BE one of the letters in the word that has already been revealed.

show_possible_matches(my_word) is the other function that we will need and it should print out all words in Wordlist (we have defined this at the beginning of the file) that matches my_word. This function will run if the current guess that the user inserts is

Now that we have these functions down, let's write them!
Starting off with match_with_gaps(my_word, other_word), we will need some helper functions. Because we are first going to compare the length of my word and other word, we will need to find the length of my word.

```
def get_my_word_len(my_word):
    length = 0
    for c in my_word:
            if c != " ":
                    length = length + 1
    return length
```

Next we will write a helper function to remove the spaces in my word so there won't be any mistakes.

```
def remove_space(my_word):
    my_word_new = ""
    for c in my_word:
            if c != " ":
                    my_word_new = my_word_new + c
    return my_word_new
print(remove_space(my_word))
```

Now let's actually start with match_with_gaps(my_word, other_word).

```
def match_with_gaps(my_word, other_word):
    my_word = remove_space(my_word)
    length_my_word = get_my_word_len(my_word)
    length_other_word = len(other_word)
    if length_my_word == length_other_word:
            for i in range (length_my_word):
                    if my_word[i].isalpha():
                            if my_word[i] != other_word[i]:
                                    return False
            return True
```

Then let's move on with the next function show_possible_matches(my_word).

```
def show_possible_matches(my_word):
    wordlist = load_words()
    for possible_word in wordlist:
            if match_with_gaps(my_word, possible_word):
                    print(possible_word)
    return
```

With these functions written, let's apply them into the hangman code. Right beneath the while loop but above the if not str.isalpha(current_guess), let's insert the show possible matches function.

```
if current_guess =="*":
    print("Possible word matches are: ")
    show_possible_matches(get_guessed_word(secret_word,
                                letters_guessed))
```

Congratulations! You have now finished the hangman code!

Chapter 4

Scrabble

4.1 Introduction

In this game, letters are dealt to players, who then construct one or more words using their letters. Each valid word earns the user points, based on the length of the word and the letters in that word. A player is dealt with a hand of HAND_SIZE letters of the alphabet, and the player arranges the hand into as many words as they want out of the letters, but using each letter at most once. In this case, a hand is in dictionary form and will look like this: hand = 'h': 1, 'e': 1, 'l':2, 'o':1. This hand will print out "hello". If you noticed, the numbers assigned to each letter in the hand represents how many times this letter appears. The letter "l" has a 2 assigned to it because "l" appears 2 times in the word "hello". Before we get started, let's first go to https://ocw.mit.edu/courses/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/download/ and download PS3.zip under the "assignments" tab. After downloading this zip file, you will see that there are four files: problem-set3.pdf, ps3.py, test_ps3.py and words.txt. The pdf file is the brief rundown of what we are doing in this problem set. The file ps3.py has a number of already-implemented functions that we are going to use when writing the solution. We will also fill in a number of modular functions in that file and then piece everything together to complete the game. We will be testing each function as we go by running the test_ps3.py to check our work. If we did it correctly, it will print a SUCCESS message, otherwise it will print a FAILURE message.

4.2 Word Scores

Now let's get started with the first part: word scores.
To calculate the word score, you will need the product of these two: the sum of the points for letters in the word and either [7*word_length - 3*(n-word_length)] or 1, whichever value is greater. To make things a little bit easier, we will write 3 separate functions. For all three of these functions, we need to input two parameters: word and num_letters. We can set word to be 'joy' and num_letters to be 8.
Part 1:

Because we are figuring out the sum of the points for letters in our word, we will be using SCRABBLE_LETTER_VALUE. This is in the ps3.py file you just downloaded, but I will paste it below just in case:
SCRABBLE_LETTER_VALUES = 'a': 1, 'b': 3, 'c': 3, 'd': 2, 'e': 1, 'f': 4, 'g': 2, 'h': 4, 'i': 1, 'j': 8, 'k': 5, 'l': 1, 'm': 3, 'n': 1, 'o': 1, 'p': 3, 'q': 10, 'r': 1, 's': 1, 't': 1, 'u': 1, 'v': 4, 'w': 4, 'x': 8, 'y': 4, 'z': 10
This tells us how many points each letter is worth.
First let's set sum_points to be 0. Then we can update this value by adding on the points earned from the rest of the letters in the word. We can use a for loop to achieve this goal.

```
for letter in word:
    sum_points = SCRABBLE_LETTER_VALUES.get(letter.lower(), 0)
                    + sum_points
```

The .get() function is a built-in method for dictionaries that allows you to retrieve the value associated with a given key. Letter.lower() is used to convert a string to lowercase, therefore we won't have any trouble dealing with uppercase letters. Then we will return sum_points. The finished function will look like this:

```
def part_1_score(word, num_letters):
sum_points = 0
for letter in word:
        sum_points = SCRABBLE_LETTER_VALUES.get(letter.lower(), 0)
                        + sum_points
    return sum_points
```

Part 2:
Remember that this problem already tells us the formula we need for the second part of get_word_score: 7*word_length - 3*(n - word_length). The only variable we need to define is word_length. We can do that by using "len()".

```
    def part_2_score(word, num_letters):
        word_length = len(word)
```

Then let's set "value" to equal to the formula mentioned above. If "value" is greater than 1, then we will keep it. Else, we will use 1.

```
if value>1:
                value
else:
                value = 1
return value
```

Now that we have both parts of get_word_score, let's implement it!

```
    def get_word_score(word, num_letters):
        score = part_1_score(word, num_letters)*part_2_score(word,
```

```
                                                num_letters)
        return score
```

We can check if our work is correct by running the test_ps3.py.

```
    Testing get_word_score...
1
1
9
22
39
49
49
19
28
SUCCESS: test_get_word_score()
```

Yes, it is correct!!

4.3 Dealing with hands

The program has deal_hand(num_letters), which generates a random hand(represented as a dictionary), and display_hand(hand), which displays the hand in a user-friendly way. These two are already written for us. We need to write update_hand(hand, word), which removes letters as the user plays the game. For example, the user is given a hand: j, m, q, o, y. If the user decides to play the word "joy", then this function will print out the remaining letters: m and q.
Let's now start writing our code for the function. We must first make a copy of the hand called current_hand because if we don't, when we run it in the terminal it will tell us that this function has mutated the original hand and is incorrect. Then, for each letter in the word, if that letter is in the current_hand and the value assigned to that letter is greater than 0, we will update the current_hand by subtracting 1 from the original current_hand. Keep in mind that the value assigned to each letter in the hand is how many times that letter appears in the hand. So if we have used the letter in our guessed word, we must subtract 1 from the value. The code should look like this:

```
    def update_hand(hand, word):
    current_hand = hand.copy()
    for letter in word:
        if letter.lower() in current_hand:
            if current_hand.get(letter.lower(), 0) > 0:
                current_hand[letter.lower()]=current_hand.get(
                                                    letter.lower(),0)-1
    return current_hand
```

Let's run it through the test_ps3.py program and see if it is correct.

```
Testing update_hand...
SUCCESS: test_update_hand()
```

Valid words

We now need to implement the is_valid_word function to see if the word is a valid word or not. Let's first set word to be word.lower(), which will get rid of all the problems regarding uppercase and lowercase letters. As long as the word is in word list and can_form_word(which we will define), it is a valid word.

Lets first begin with can_form_word. We will make a copy of the hand and name it current_hand. For each letter in the word, if that letter is not in the current_hand, or if the value associated with that letter is 0, then we will return false. Else, we will subtract 1 from the value associated with that letter.

```
def can_form_word(hand, word):
    current_hand = hand.copy()
    for letter in word:
        if letter not in current_hand or current_hand[letter] == 0:
                return False
        else:
            current_hand[letter] -= 1
    return True

def is_valid_word(word, hand, word_list):
    word = word.lower()
    if word in word_list and can_form_word(hand, word):
                    return True
            else:
                    return False
```

Let's run it through the test_ps3.py.

```
    Testing is_valid_word...
true
true
false
true
false
true
false
SUCCESS: test_is_valid_word()
```

Wildcards

Wildcards is a kind of placeholder represented by a single character, such as an wildcard, which can be interpreted as a number of literal characters or an empty string. In this game, wildcards can only replace vowels. Each hand dealt should initially contain a wildcard as one of the letters. For example, let the current hand be j, o, l, y, *. A word that the user could put would be "joy".

However, the user could also input “j*y”, which is correct because if the wildcard was “a”, it would form the word “jay”, which is a valid word. An invalid word using the wildcard would be “j*yl”, because no vowel taking the place of the wildcard would form a valid word. We don’t have to write any new functions, but we do need to modify the deal_hand function to support always giving one wildcard in each hand. We also need to modify the is_valid_word function to support wildcards. Let’s start off by modifying the deal_hand function.
If you look at the code closely, you will realize that we only need to modify the part of the code dealing with vowels.

```
hand={}
num_vowels = int(math.ceil(num_letters / 3))
for i in range(num_vowels):
    x = random.choice(VOWELS)
    hand[x] = hand.get(x, 0) + 1
```

Every time the code generates a new hand, everything changes except for the wildcard. If you look at the code, the hand right now is empty. However, we should start off by having a wildcard built into the hand.

```
hand={'*': 1}
```

The wildcard will take the place of one letter, therefore the number of times the for loop happens should decrease 1.

```
for i in range(num_vowels - 1):
```

That is all we have to modify for the deal_hand function. Now, let’s move on to the is_valid_word function.
We first need to set word to be word.lower(), again, to get rid of all problems dealing with uppercase letters. Then, if the wildcard appears in the word, we will use a for loop.

```
def is_valid_word(word, hand, word_list):
    word = word.lower()
    if '*' in word:
            for replacement_vowel in VOWELS:
```

The “replacement_vowel” is the same as “i”. Normally you would see the for loop as “for i in ...” But to keep things a little clearer, we will use replacement_vowel to represent one vowel being chosen randomly from VOWELS. We want to put the replacement_vowel in the place of the wildcard and test whether or not it will form a word. Let’s then define new_word to be word.lower. We will update that variable using .replace(), which is a string method that returns a new string with all occurrences of a specified characters replaced with another set of characters. In the parentheses, we will put what we want to replace and what we can replace it with. It should be the wildcard, and replacement_vowel. Now we can see whether or not this new word, with the replacement_vowel, is a valid word or not.

```
def is_valid_word(word, hand, word_list):
    word = word.lower()
    if '*' in word:
        for replacement_vowel in VOWELS:
            new_word = word.lower()
            new_word = word.replace('*', replacement_vowel)
            if new_word in word_list and can_form_word(hand, word):
                return True
        return False
```

This code only works if there is a wildcard in the word. If there isn't, then we will see if that word is a valid word or not. The finished code for this function should look like this:

```
def is_valid_word(word, hand, word_list):
    word = word.lower()
    if '*' in word:
        for replacement_vowel in VOWELS:
            new_word = word.lower()
            new_word = word.replace('*', replacement_vowel)
            if new_word in word_list and can_form_word(hand, word):
                return True
        return False
    else:
        if word in word_list and can_form_word(hand, word):
            return True
        else:
            return False
```

Let's run it through the test_ps3.py program and see if it returns SUCCESS or FAILURE.

```
    Testing wildcards...
false
29
22
29
SUCCESS: test_wildcard()
```

Yes, the code is correct!

4.4 Playing a hand

We have the majority of the helper functions written out, so we can implement the play_hand function. This function allows the user to play out a single hand, but we need one more helper function, calculate_handlen(hand), which calculates the hand length. Let's set handlen to be 0, so we can update it later. For each letter in the hand, we will add the value associated with the letter to handlen. That way, we will know how many letters there are in the hand.

```
def calculate_handlen(hand):
    handlen = 0
    for letter in hand:
            handlen = handlen + hand.get(letter.lower(), 0)
    return handlen
```

We can now write our play_hand function. If you look after the line "begin pseudocode", there are a bunch of lines that will help guide you in writing your function. This code should print out the current hand, allow the user to enter a word, or "!!" to end the hand early, and keep track of the total score. Remember that at the beginning of the code, we set num_letters to be 8. Let's set num_letters to be calculate_handlen(hand) and set total_score to be 0. We will also print out the current_hand. Like this:

```
num_letters = calculate_handlen(hand)
total_score = 0
print("Current hand: ", end= ' ')
display_hand(hand)
```

"end= ' '" allows display_hand(hand) and the words "Current hand =" to be on the same line. We will make a copy of the hand again and call it "current_hand". Then we will define input_word to ask the user to type in their word.

```
input_word = input("Enter word, or '! !' to indicate
                    that you are finished: ")
```

If the input_word is two exclamation marks, then we will break out of the loop. Otherwise (the input word is not two exclamation marks), if it is a valid word, tell the user how many points were earned and update the total_score. Otherwise (the word is not a valid word), print out a rejection message. We also need to return total_score as a result of the function. So let's start from the beginning. We will use an if loop. If the inputted word is not two exclamation marks, then we will first update the total score by adding the score of each word to it. We will show the user how many points the inputted word earned and what the total score is. Then we will update the current_hand to be update_hand and print it out to show the user what words are remaining in the hand. This loop should keep running until there are no letters left or if the user inputs two exclamation marks.

```
if input_word != '!!':
    if is_valid_word(input_word, current_hand, word_list):
            total_score = get_word_score(input_word, num_letters)
                                + total_score
            print(input_word, "earned", get_word_score(input_word,
            num_letters), "points. Total: ", total_score, "points.")
            current_hand = update_hand(current_hand, input_word)
            print("Current hand: ")
            display_hand(current_hand)
```

Let's finish the inner if loop. Using else, we will print out the message "That is not a valid word. Please choose another word." Then we will finish the outer if loop. So if the inputted word is two exclamation marks, then we will print out the total score. We have one more thing to do now because remember that this loop should keep running until there are no letters left or if the user inputs two exclamation marks. We acknowledged the fact that the letters might run out, therefore the code should break. So if current hand length has a value of 0, then it should print a message telling the user that there are no more letters. This is what the finished code should look like for this function:

```
def play_hand(hand, word_list):
    num_letters = calculate_handlen(hand)
    total_score = 0
    print("Current hand: ", end= ' ')
    display_hand(hand)
    current_hand = hand.copy()
    input_word = input("Enter word, or '! !' to indicate
                        that you are finished: ")
        if input_word != '!!':
            if is_valid_word(input_word, current_hand, word_list):
                    total_score = get_word_score(input_word,
                        num_letters) + total_score
                    print(input_word, "earned", get_word_score(
                    input_word, num_letters), "points. Total: ",
                        total_score, "points." )
                    current_hand = update_hand(current_hand,
                            input_word)
                    print("Current hand: ")
                    display_hand(current_hand)
            else:
                    print("That is not a valid word. Please choose
                another word.")
        else:
                print("Total score for this hand: ", total_score,
                            "points.")
            if calculate_handlen(current_hand) == 0:
                    print("Ran out of letters. Total score: ",
                        total_score, "points.")
    return total_score
```

If you run the code, you will find out that it will keep on running because we never broke out of the code. So we need to add something stop the code. Let's call this "something" stop_value. Set the stop_value to be 1. We want the loops to run while this stop_value is 1, otherwise, when we want to break out of the loops we can set the stop_value to be 0. So let's add this into our code.

```
def play_hand(hand, word_list):
    num_letters = calculate_handlen(hand)
    #print("p1 ", num_letters)
```

```
    total_score = 0
    print("Current hand: ", end= ' ')
    display_hand(hand)
    current_hand = hand.copy()
    #input_word = input("Enter word, or '! !' to indicate that
                    you are finished: ")
    stop_value=1
    while stop_value != 0:
            input_word = input("Enter word, or '! !' to indicate
                that you are finished: ")
            if input_word != '!!':
                    if is_valid_word(input_word, current_hand,
                                        word_list):
                            total_score = get_word_score(input_word,
                            num_letters) + total_score
                            print(input_word, "earned",
                            get_word_score(
                            input_word, num_letters), "points.
                            Total: ", total_score, "points." )
                            current_hand = update_hand(current_hand,
                            input_word)
                            print("Current hand: ")
                            display_hand(current_hand)
                    else:
                            print("That is not a valid word. Please
                        choose another word.")
            else:
                    print("Total score for this hand: ",
                        total_score, "points.")
                    stop_value = 0
            if calculate_handlen(current_hand) == 0:
                    print("Ran out of letters. Total score: ",
                            total_score, "points.")
                    stop_value = 0
    return total_score
```

Now that should be the finished code for the function play_hand. We have only one more main function to go before we complete this problem (with one helper function). In this final function, which we will call play_game, we will want to do the following: ask the user how many times he/she would want to play the game/hand. Then we will generate the hand and ask if they would like to substitute a letter. If replied with no, then we will print out the hand again to confirm and allow them to enter a word. If yes, then we will prompt them for their desired letter. This can only be done once during the game. Once the substitute option is used, the user should not be asked if they want to substitute letters in the future. We will continue to play unless the user runs out of letters or if they enter "!!". Then we will ask whether or not the user would like to replay this hand (if they say yes, then we will generate the same hand as before and allow them to play it again, taking the

highest score of the two). Keep in mind that this can only be done once during the game. Once the replay option is used, the user should not be asked if they want to replay future hands. Replaying the hand does not count as one of the total number of hands the user initially wanted to play. Note that if the user decides to replay a hand, they do not get to substitute a letter, they must use the letters they have.

Let's start with the code. We are going to be needing a helper function called substitute_hand It will take in two parameters: hand and letter. It's function is to allow the user to replace all copies of one letter in the hand with a new letter. The new letter should not be any letter already in the hand. Let's first write this helper function. First we will make a copy of the hand so it doesn't mutate the original hand. There are many ways to write this helper function, so the way that I will be showing you is based off of my person preference. The following would be my thought process: define a function to include all of the letters in the alphabet except for those already in the hand. Then we will choose any letter from the possible letters previously defined. Next we will tie the new letter to the value associated with the original letter we are substituting. This will get rid of all issues regarding how many times the original letter appears. Then we will set the the value associated with the original letter to be 0. Then we will return this new hand.

As mentioned, we have one helper functions to write for this substitute_hand function. Let's call this function get_available_letters, and it should take in one parameter which we will call letters_in_hand. Let's first define this parameter. It's goal is to return the letters in the hand. To write this function, we will set hand_letter to be an empty string so we can update it later. Then we will use the .keys() function, which is a built-in function in Python, to display a list of all the keys in the dictionary. For the letters in this list, we will go through each letter and add that letter to our hand_letter. Lastly, we will return this letter. This is what the function should look like:

```
def letters_in_hand(hand):
        hand_letter = ""
        for letter in hand.keys():
                for j in range(hand[letter]):
                        hand_letter = hand_letter + letter
        return hand_letter
```

Now that we have our parameter down for the function, we can actually write our function. Just like what we did at the beginning of our letters_in_hand function, we will set available_letter to be an empty string. Then, for each letter in the alphabet (we will use "string.ascii_lowercase" to help us out, which just returns the list of the lowercase letters in the alphabet), if the letter is not the one of the letters in the hand (which we have defined using the function letters_in_hand), then we will add this letter to available_letter. The function should look like this:

```
def get_available_letters(letters_in_hand):
          available_letter = ""
          for letter in string.ascii_lowercase:
               if letter not in letters_in_hand:
                 available_letter = available_letter + letter
          return available_letter
```

Let's get started with the substitute_hand function. I have given my thought process and the explanation of how I wrote the code above, so let's start writing the code.

```
def substitute_hand(hand, letter):
        new_hand = hand.copy()
        possible_letter = get_available_letters(
                              letters_in_hand(new_hand))
        substitute_letter = random.choice(possible_letter)
        new_hand[substitute_letter.lower()] = new_hand.get(
                                        letter.lower(),0)
        new_hand[letter.lower()] = 0
        return new_hand
```

We have finally finished doing the preparation for the final function that we will write for this game, so let's begin. I have already given an explanation for what we are going to do for this code above, so let's start writing!
First we will ask the user to input the total number of hands they would like to play, and then set replay_option to be 1 and total_score to be 0. This is how the code will start out:

```
def play_game(word_list):
        num_hands = int(input("Enter total number of hands: "))
        replay_option = 1
        total_score = 0
```

Next, we will set hand to be deal_hand(HAND_SIZE) so it'll be easier when we call for the deal_hand function because we don't have to type out as many letters. Then we will print out the current hand.

```
hand = deal_hand(HAND_SIZE)
print("Current hand: ", end= ' ')
display_hand(hand)
```

The end= ' ' is a built-in function in python that allows you to print something on the same line. So the words "Current hand: " and the actual hand will appear on the same line instead of separate lines. Next we will ask the user if they want to substitute a hand or not, and we will call that input ask_substitute. Then, if substitute_hand equals to "yes", then we will ask what letter they would like to replace and update the hand using the helper function substitute_hand.

```
ask_substitute = input("Would you like to substitute a letter?"]
if ask_substitute == 'yes':
        letter = input("Which letter would you like to substitute?")
         hand = substitute_hand(hand, letter)
```

Now we will calculate the score of the hand using the play_hand function. Remember, at the end of the play_hand function, we returned the score for that hand, which is why we can just use it to calculate the hand score.

```
        hand_score = play_hand(hand, word_list)
        total_score = total_score + hand_score
```

This sums it up for the if part of the loop. Let's now move on to the else part. Therefore, if they don't want to substitute a letter, then we will calculate the score for this hand and then the total score.

```
    else:
                        hand_score = play_hand(hand, word_list)
                        total_score = total_score + hand_score
```

Now don't forget that we have an option called replay_option that we previously set to be 1. If the replay_option is equal to 1, then we will first ask if the user would like to replay. If the answer is "yes", then first we will call replay_score to be equal to play_hand, because remember, the function play_hand returns the hand score. Then if replay_score is greater than hand_score, which is basically saying that if the score for the new hand is better than the score achieved previously, we will update the total score by subtracting the hand score and adding the replay score to it. Finally, to make sure that the user only uses this replay option once, we will set replay option to be 0. And lastly, we will print out a sentence telling the user their total score.

```
if replay_option == 1:
        replay = input("Would you like to replay the hand?")
        if replay == 'yes':
                replay_score = play_hand(hand, word_list)
                if replay_score > hand_score:
                        total_score = total_score + replay_score
                                - hand_score
                        replay_option = 0
         print("Total score over all hands: ", total_score)
```

Chapter 5

Recursion

Recursion is a programming concept where a function calls itself during its execution. In other words, a recursive function is one that solves a problem by breaking it down into smaller, similar sub-problems and calls itself to solve those sub-problems. Each recursive call contributes to solving the overall problem until the base case is reached. Here are just some terms to help you understand how recursion works:
Base case: This is what tells the function when to stop. The function will keep on going until it reaches the base case.
Recursive step: In this step, the problem is divided into smaller, more manageable sub-problems. The function calls itself to solve these sub-problems.
We will be calculating the factorial of a number as an example of recursion.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

# Example usage
result = factorial(5)
print("Factorial of 5:", result)
```

In this example, the parameter is n, and so when we implement this function we will replace the n with the number we want to find the factorial of. The base case is when n equals to 0. So if it does reach 0, the this function will end by returning a 1. Else, if it does not equal to 0, then we will run the recursive step, which calculates the factorial by multiplying n with the factorial of n-1.
It should return:

```
Factorial of 5: 120
```

However, let's breakdown the output:

```
factorial(5) = 5 * factorial(4)
             = 5 * (4 * factorial(3))
             = 5 * (4 * (3 * factorial(2)))
             = 5 * (4 * (3 * (2 * factorial(1))))
             = 5 * (4 * (3 * (2 * (1 * factorial(0)))))
             = 5 * (4 * (3 * (2 * (1 * 1))))
             = 120
```

Here we see that it solves the factorial of 5 all in one function. This is the benefit of using recursion - easy and quick. An important aspect of recursion would be sorting. Just like what it sounds, sorting is the process of arranging elements in a specific order, often in ascending or descending order, based on certain criteria. Sorting has many benefits, which includes: being able to organize the data faster, efficient searching, as well as data retrieval.

5.1 Code for Problem Set 4A

I am sorry I did not write the learning notes. But I did have my codes for Problem Set 4. In the code, I did not remove the instructions from the problem set.

```
    # Problem Set 4A
# Name: <your name here>
# Collaborators:
# Time Spent: x:xx

def get_permutations(sequence):
        '''
    Enumerate all permutations of a given string

    sequence (string): an arbitrary string to permute.
    Assume that it is a non-empty string.

    You MUST use recursion for this part. Non-recursive
    solutions will not be accepted.

    Returns: a list of all permutations of sequence

    Example:
    >>> get_permutations('abc')
    ['abc', 'acb', 'bac', 'bca', 'cab', 'cba']

    Note: depending on your implementation, you may return the
    permutations in a different order than what is listed here.
        '''
        if len(sequence) == 1:
                return sequence
```

```
        else:
                permutations = []
                sequence_0= sequence[0]
                sub_permutations = get_permutations(sequence[1:])
                for original_string in sub_permutations:
                        for j in range(len(original_string)+1):
                                result = original_string[:j]
                                + sequence_0
                                + original_string[j:]
                                permutations.append(result)
                return permutations



if __name__ == '__main__':
#    #EXAMPLE

        example_input = 'abc'
        print('Input:', example_input)
        print('Expected Output:', ['abc', 'acb', 'bac',
                                   'bca', 'cab', 'cba'])
        print('Actual Output:', get_permutations(example_input))

#    # Put three example test cases here (for your sanity,
#    limit your inputs to be three characters or fewer
#    as you will have n! permutations for a
#    sequence of length n)

   # pass #delete this line and replace with your code here



        s = 'aeiou'
        s_permutation = get_permutations(s)[3]
        print("s:", s)
        print("S_permutation:", s_permutation)
        cipher_dict = {}
        for i in range(5):
                cipher_dict[s[i]]=s_permutation[i]
                cipher_dict[s[i].upper()]=s_permutation[i].upper()

        print("cipher_dict = ", cipher_dict)
```

Here are the results when the code is implemented. This is first example for permutation *abc*.

```
Input: abc
Expected Output: ['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
Actual Output: ['abc', 'bac', 'bca', 'acb', 'cab', 'cba']
```

Here are the results of using permutation to make a cipher dictionary.

```
s: aeiou
S_permutation: eioau
cipher_direct{'a': 'e', 'A': 'E', 'e': 'i', 'E': 'I', 'i': 'o',
'I': 'O', 'o': 'a', 'O': 'A', 'u': 'u', 'U': 'U'}
```

5.2 Code for Problem Set 4B

```
    # Problem Set 4B
# Name: <your name here>
# Collaborators:
# Time Spent: x:xx

import string

### HELPER CODE ###
def load_words(file_name):
        '''
    file_name (string): the name of the file containing
    the list of words to load

    Returns: a list of valid words. Words are strings
    of lowercase letters.

    Depending on the size of the word list, this function may
    take a while to finish.
        '''
        print("Loading word list from file...")
    # inFile: file
        inFile = open(file_name, 'r')
    # wordlist: list of strings
        wordlist = []
        for line in inFile:
                wordlist.extend([word.lower() for word
                        in line.split(' ')])
        print("  ", len(wordlist), "words loaded.")
        return wordlist

def is_word(word_list, word):
        '''
```

```
    Determines if word is a valid word, ignoring
    capitalization and punctuation

    word_list (list): list of words in the dictionary.
    word (string): a possible word.

    Returns: True if word is in word_list, False otherwise

    Example:
    >>> is_word(word_list, 'bat') returns
    True
    >>> is_word(word_list, 'asdf') returns
    False
        '''
        word = word.lower()
        word = word.strip(" !@#$%^&*()-_+={}[]|\:;'<>?,./\"")
        return word in word_list

def get_story_string():
        """
    Returns: a story in encrypted text.
        """
        #f = open("story.txt", "r")
        #f = open("mystory.txt", "r")
        f = open("mystoryHidden.txt", "r")
        story = str(f.read())
        f.close()
        return story

### END HELPER CODE ###

WORDLIST_FILENAME = 'words.txt'

class Message(object):
        def __init__(self, text):
                '''
        Initializes a Message object

        text (string): the message's text

        a Message object has two attributes:
            self.message_text (string, determined by input text)
            self.valid_words (list, determined using helper
                                function load_words)
                '''
```

```
        #pass #delete this line and replace with your code here
        self.message_text = text
        self.valid_words = load_words(WORDLIST_FILENAME)

    def get_message_text(self):
        '''
    Used to safely access self.message_text outside of the class

    Returns: self.message_text
        '''
        #pass #delete this line and replace with your code here
        return self.message_text

    def get_valid_words(self):
        '''
    Used to safely access a copy of self.valid_words outside of
    the class.
    This helps you avoid accidentally mutating class attributes.

    Returns: a COPY of self.valid_words
        '''
        #pass #delete this line and replace with your code here
        return list(self.valid_words)

    def build_shift_dict(self, shift):
        '''
    Creates a dictionary that can be used to apply a cipher to
    a letter. The dictionary maps every uppercase and lowercase
    letter to a character shifted down the alphabet by the input
    shift. The dictionary should have 52 keys of all the uppercase
    letters and all the lowercase letters only.

    shift (integer): the amount by which to shift every letter
    of the alphabet. 0 <= shift < 26

    Returns: a dictionary mapping a letter (string) to
             another letter (string).
        '''
        #pass #delete this line and replace with your code here
        shift_dict = {}
        for char in 'abcdefghijklmnopqrstuvwxyz':
                shifted_char = chr((ord(char) - ord('a')
                        + shift) % 26 + ord('a'))
                shift_dict[char] = shifted_char
                shift_dict[char.upper()] = shifted_char.upper()
```

```
            return shift_dict


        def apply_shift(self, shift):
            '''
        Applies the Caesar Cipher to self.message_text with the
        input shift. Creates a new string that is self.message_text
        shifted down the alphabet by some number of characters
        determined by the input shift

        shift (integer): the shift with which to encrypt the message.
        0 <= shift < 26

        Returns: the message text (string) in which every character is
            shifted down the alphabet by the input shift
            '''
        #pass #delete this line and replace with your code here
            shift_dict = self.build_shift_dict(shift)
            #The following
            shifted_text=""
            for char in self.message_text:
                if char.isalpha():
                    shifted_text = shifted_text+shift_dict
                                                [char]
                else:
                    shifted_text = shifted_text+char
            #shifted_text = "".join(shift_dict[char] if
                                    char.isalpha()
                else char for char in
                                    self.message_text)
            return shifted_text

class PlaintextMessage(Message):
        def __init__(self, text, shift):
            '''
        Initializes a PlaintextMessage object

        text (string): the message's text
        shift (integer): the shift associated with this message

        A PlaintextMessage object inherits from Message and has
            five attributes:
            self.message_text (string, determined by input text)
            self.valid_words (list, determined using helper function
            load_words)
```

```
        self.shift (integer, determined by input shift)
        self.encryption_dict (dictionary, built using shift)
        self.message_text_encrypted (string, created using shift)

            '''
            pass #delete this line and replace with your code here
            super().__init__(text)
            self.shift = shift
            self.encryption_dict = self.build_shift_dict(shift)
            self.message_text_encrypted = self.apply_shift(shift)

    def get_shift(self):
            '''
    Used to safely access self.shift outside of the class

    Returns: self.shift
            '''
            #pass #delete this line and replace with your code here
            return self.shift

    def get_encryption_dict(self):
            '''
    Used to safely access a copy self.encryption_dict outside
                        of the class

    Returns: a COPY of self.encryption_dict
            '''
            #pass #delete this line and replace with your code here
            return dict(self.encryption_dict)

    def get_message_text_encrypted(self):
            '''
    Used to safely access self.message_text_encrypted
                outside of the class

    Returns: self.message_text_encrypted
            '''
            #pass #delete this line and replace with your code here
            return self.message_text_encrypted

    def change_shift(self, shift):
            '''
    Changes self.shift of the PlaintextMessage and updates other
    attributes determined by shift.
```

```
        shift (integer): the new shift that should be associated
                         with this message.
        0 <= shift < 26

        Returns: nothing
                '''
                #pass #delete this line and replace with your code here
                self.shift = shift
                self.encryption_dict = self.build_shift_dict(shift)
                self.message_text_encrypted = self.apply_shift(shift)


class CiphertextMessage(Message):
        def __init__(self, text):
                '''
        Initializes a CiphertextMessage object

        text (string): the message's text

        a CiphertextMessage object has two attributes:
            self.message_text (string, determined by input text)
            self.valid_words (list, determined using helper function
            load_words)
                '''
                #pass #delete this line and replace with your code here
                super().__init__(text)

        def decrypt_message(self):
                '''
        Decrypt self.message_text by trying every possible shift value
        and find the "best" one. We will define "best" as the shift
        that creates the maximum number of real words when we use
        apply_shift(shift) on the message text. If s is the original
        shift value used to encrypt the message, then we would expect
        26 - s to be the best shift value for decrypting it.

        Note: if multiple shifts are equally good such that they all
        create the maximum number of valid words, you may choose any
        of those shifts (and their corresponding decrypted messages)
        to return

        Returns: a tuple of the best shift value used to decrypt the
        message and the decrypted message text using that shift value
                '''
                #pass #delete this line and replace with your code here
```

```
                best_shift = 0
                max_valid_words = 0
                decrypted_message = ""
                for shift in range(26):
                        decrypted_text = self.apply_shift(26-shift)
                        valid_words_count = sum (word in self.
                    valid_words for word in decrypted_text.split())
                        #print(valid_words_count, max_valid_words)
                        if valid_words_count > max_valid_words:
                                max_valid_words = valid_words_count
                                best_shift = 26 - shift
                                decrypted_message = decrypted_text

                return best_shift, decrypted_message

if __name__ == '__main__':

#    #Example test case (PlaintextMessage)
        print("Example 1:")
        plaintext = PlaintextMessage('hello', 2)
        print('Expected Output: jgnnq')
        print('Actual Output:', plaintext.get_message_text_encrypted())
        print("End of Example 1.")

#    #Example test case (CiphertextMessage)
        print("Example 2:")
        ciphertext = CiphertextMessage('jgnnq')
        print('Expected Output:', (24, 'hello'))
        print('Actual Output:', ciphertext.decrypt_message())
        print("End of Example 2.")

    #TODO: WRITE YOUR TEST CASES HERE
        print("Example 3:")
    #TODO: best shift value and unencrypted story
        Object="try word"
        print(Message(Object).get_message_text())
        print(Message(Object).apply_shift(4))
        print(Message(Object).build_shift_dict(4))
        print("End of Example 3.")

        #pass #delete this line and replace with your code here



        #    #Example test case (PlaintextMessage)
```

```
print("Example 4:")
m = PlaintextMessage('Happy Chinese New Year! Be grateful!', 4)
print("original message: ", m.get_message_text())
print("encrypted message: ", m.get_message_text_encrypted())
encrypt_message = m.get_message_text_encrypted()
print("End of Example 4.")
```

Here are the results of the code implementation.

```
Example 1:
Loading word list from file...
   55901 words loaded.
Expected Output: jgnnq
Actual Output: jgnnq
End of Example 1.

Example 2:
Loading word list from file...
   55901 words loaded.
Expected Output: (24, 'hello')
Actual Output: (24, 'hello')
End of Example 2.

Example 3:
Loading word list from file...
   55901 words loaded.
try word
Loading word list from file...
   55901 words loaded.
xvc asvh
Loading word list from file...
   55901 words loaded.
{'a': 'e', 'A': 'E', 'b': 'f', 'B': 'F', 'c': 'g', 'C': 'G', 'd': 'h',
'D': 'H', 'e': 'i', 'E': 'I', 'f': 'j', 'F': 'J', 'g': 'k', 'G': 'K',
'h': 'l', 'H': 'L', 'i': 'm', 'I': 'M', 'j': 'n', 'J': 'N', 'k': 'o',
'K': 'O', 'l': 'p', 'L': 'P', 'm': 'q', 'M': 'Q', 'n': 'r', 'N': 'R',
'o': 's', 'O': 'S', 'p': 't', 'P': 'T', 'q': 'u', 'Q': 'U', 'r': 'v',
'R': 'V', 's': 'w', 'S': 'W', 't': 'x', 'T': 'X', 'u': 'y', 'U': 'Y',
'v': 'z', 'V': 'Z', 'w': 'a', 'W': 'A', 'x': 'b', 'X': 'B', 'y': 'c',
'Y': 'C', 'z': 'd', 'Z': 'D'}
End of Example 3.

Example 4:
Loading word list from file...
   55901 words loaded.
original message:  Happy Chinese New Year! Be grateful!
```

```
encrypted message:  Lettc Glmriwi Ria Ciev! Fi kvexijyp!
Loading word list from file...
   55901 words loaded.
(0, '')
End of Example 4.
```

5.3 Code for Problem Set 4C

```
    # Problem Set 4C
# Name: <your name here>
# Collaborators:
# Time Spent: x:xx

import string
from ps4aZF import get_permutations

### HELPER CODE ###
def load_words(file_name):
        '''
    file_name (string): the name of the file containing
    the list of words to load

    Returns: a list of valid words. Words are strings of
    lowercase letters.

    Depending on the size of the word list, this function may
    take a while to finish.
        '''

        print("Loading word list from file...")
        # inFile: file
        inFile = open(file_name, 'r')
        # wordlist: list of strings
        wordlist = []
        for line in inFile:
                wordlist.extend([word.lower() for word
                    in line.split(' ')])
                print("  ", len(wordlist), "words loaded.")
        return wordlist

def is_word(word_list, word):
        '''
    Determines if word is a valid word, ignoring
    capitalization and punctuation
```

```
    word_list (list): list of words in the dictionary.
    word (string): a possible word.

    Returns: True if word is in word_list, False otherwise

    Example:
    >>> is_word(word_list, 'bat') returns
    True
    >>> is_word(word_list, 'asdf') returns
    False
        '''
        word = word.lower()
        word = word.strip(" !@#$%^&*()-_+={}[]|\:;'<>?,./\"")
        return word in word_list


### END HELPER CODE ###

WORDLIST_FILENAME = 'words.txt'


# you may find these constants helpful
VOWELS_LOWER = 'aeiou'
VOWELS_UPPER = 'AEIOU'
CONSONANTS_LOWER = 'bcdfghjklmnpqrstvwxyz'
CONSONANTS_UPPER = 'BCDFGHJKLMNPQRSTVWXYZ'

class SubMessage(object):
        def __init__(self, text):
                '''
        Initializes a SubMessage object

        text (string): the message's text

        A SubMessage object has two attributes:
            self.message_text (string, determined by input text)
            self.valid_words (list, determined using helper
            function load_words)
                '''
        #pass #delete this line and replace with your code here
                self.message_text = text
                self.valid_words = load_words(WORDLIST_FILENAME)

        def get_message_text(self):
```

```
        '''
    Used to safely access self.message_text outside of the class

    Returns: self.message_text
        '''
        #pass #delete this line and replace with your code
    here return self.message_text

def get_valid_words(self):
        '''
    Used to safely access a copy of self.valid_words outside
    of the class.
    This helps you avoid accidentally mutating class attributes.

    Returns: a COPY of self.valid_words
        '''
        #pass #delete this line and replace with your code here
        return self.valid_words

def build_transpose_dict(self, vowels_permutation):
        '''
    vowels_permutation (string): a string containing a permutation
    of vowels (a, e, i, o, u)

    Creates a dictionary that can be used to apply a cipher to
    a letter. The dictionary maps every uppercase and lowercase
    letter to an uppercase and lowercase letter, respectively.
    Vowels are shuffled according to vowels_permutation. The
    first letter in vowels_permutation corresponds to a, the second
    to e, and so on in the order a, e, i, o, u.
    The consonants remain the same. The dictionary should have 52
    keys of all the uppercase letters and all the lowercase letters.

    Example: When input "eaiuo":
    Mapping is a->e, e->a, i->i, o->u, u->o
    and "Hello World!" maps to "Hallu Wurld!"

    Returns: a dictionary mapping a letter (string) to
             another letter (string).
        '''

        #pass #delete this line and replace with your code here
        cipher_dict = {}

        for i in range(5):
```

```
                        cipher_dict[VOWELS_LOWER[i]]=
                                        vowels_permutation[i]
                        cipher_dict[VOWELS_LOWER[i].upper()]=
                                      vowels_permutation[i].upper()
                for i in range(21):
                        cipher_dict[CONSONANTS_LOWER[i]]=
                                            CONSONANTS_LOWER[i]
                        cipher_dict[CONSONANTS_UPPER[i]]=
                                            CONSONANTS_UPPER [i]
                return cipher_dict



        def apply_transpose(self, transpose_dict):
                '''
        transpose_dict (dict): a transpose dictionary

        Returns: an encrypted version of the message text, based
        on the dictionary
                        '''

                #pass #delete this line and replace with your code here
                encrypted_word=""
                for letter in SubMessage.get_message_text(self):
                        if letter.isalpha():
                                encrypted_word = encrypted_word+
                                    transpose_dict[letter]
                        else:
                                encrypted_word = encrypted_word +letter
                return encrypted_word

class EncryptedSubMessage(SubMessage):
        def __init__(self, text):
                '''
        Initializes an EncryptedSubMessage object

        text (string): the encrypted message text

        An EncryptedSubMessage object inherits from SubMessage and
            has two attributes:
            self.message_text (string, determined by input text)
            self.valid_words (list, determined using helper function
            load_words)
                '''
                #pass #delete this line and replace with your code here
```

```
        super().__init__(text)

    def decrypt_message(self):
        '''
    Attempt to decrypt the encrypted message

    Idea is to go through each permutation of the vowels and test it
    on the encrypted message. For each permutation, check how many
    words in the decrypted text are valid English words, and return
    the decrypted message with the most English words.

    If no good permutations are found (i.e. no permutations result in
    at least 1 valid word), return the original string. If there are
    multiple permutations that yield the maximum number of words,
    return any one of them.

    Returns: the best decrypted message

    Hint: use your function from Part 4A
        '''
        #pass #delete this line and replace with your code here
        vowels_permutation = get_permutations(VOWELS_LOWER)
        Best_vowels_permutation = vowels_permutation[0]
        max_valid_words =0
        decrypted_message = ""
        num_perm=0
        for vowels_permu in vowels_permutation:
            num_perm =num_perm+1
            valid_words_count =0
            decry_dict = self.build_transpose_dict(vowels_permu)
            decrypted_text = self.apply_transpose(decry_dict)
            #print(num_perm, vowels_permu, decrypted_text)
            for word in decrypted_text.split():
                word = word.translate(str.maketrans('', '',
                    string.punctuation))
                if word.lower() in self.valid_words:
                    valid_words_count =valid_words_count +1
            #print(valid_words_count, max_valid_words)
            if valid_words_count >max_valid_words:
                max_valid_words = valid_words_count
                Best_vowels_permutation = vowels_permu
                decrypted_message = decrypted_text

        return Best_vowels_permutation, decrypted_message
```

```
if __name__ == '__main__':

    # Example test case
    print("Test Case 1:")
    message = SubMessage("Hello World! I love you!")
    permutation = "eaiuo"
    print("Used Permutation: ", permutation)
    enc_dict = message.build_transpose_dict(permutation)
    print("Original message: ", message.get_message_text())
    print("Expected encryption: ", "Hallu Wurld! I luva yuo!")
    print("Actual encryption:", message.apply_transpose(enc_dict))
    enc_message = EncryptedSubMessage(message.apply_transpose(enc_dict))
    print("Decrypted message:", enc_message.decrypt_message())
    print("End of Test Case 1")

    #TODO: WRITE YOUR TEST CASES HERE

    print("Test Case 2:")
    message = SubMessage("Is this the correct code?")
    permutation = "uieao"
    print("Used Permutation: ", permutation)
    enc_dict = message.build_transpose_dict(permutation)
    print("Original message:", message.get_message_text())
    print("Expected encryption:", "Es thes thi carrict cadi")
    print("Actual encryption:", message.apply_transpose(enc_dict))
    encrypt_message = message.apply_transpose(enc_dict)
    #enc_message = EncryptedSubMessage(message.apply_transpose(
                    enc_dict))
    #print("Decrypted message:", enc_message.decrypt_message())
    print("End of Test Case 2")

    print("Test Case 3: use inverse permutation of above to
                reverse it.")
    inverse_perm = "oieua"
    print("Used Inverse Permutation: ", inverse_perm)
    message = SubMessage("Es thes thi carrict cadi?")
    inverse_enc_dict = message.build_transpose_dict(inverse_perm)
    print("Original message:", message.get_message_text())
    print("Expected message:", "Is this the correct code?")
    print("reversed message:", message.apply_transpose(
                                inverse_enc_dict))
    print("End of Test Case 3")
```

Here are the results of the code implementation.

```
Test Case 1:
```

```
Used Permutation:  eaiuo
Original message:  Hello World! I love you!
Expected encryption:  Hallu Wurld! I luva yuo!
Actual encryption: Hallu Wurld! I luva yuo!
Decrypted message: ('eiauo', 'Hello World! A love you!')
End of Test Case 1

Test Case 2:
Used Permutation:  uieao
Original message: Is this the correct code?
Expected encryption: Es thes thi carrict cadi
Actual encryption: Es thes thi carrict cadi?
End of Test Case 2

Test Case 3: use inverse permutation of above to reverse it.
Used Inverse Permutation:  oieua
Original message: Es thes thi carrict cadi?
Expected message: Is this the correct code?
reversed message: Is this the correct code?
End of Test Case 3
```

Chapter 6

Filtering News Feed

6.1 Introduction

In this problem set, you will filter through the news, alerting the user when it finds something that matches the user's interest. So for example, if the user is interested in news stories that contains information the Olympics, then an alert will generate when it filters through the news, and finds a story that is related to the Olympics.
First, you can go to the website "https://ocw.mit.edu/courses/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/download/", click on assignments, download ps5.zip. This will give you access to the full set of instructions as well as the other files needed to complete this problem set. Please keep in mind that you should always try to figure out the problems yourself before looking at my code. Also, there are many ways to write the solutions, so yours might not be the same as mine! Through out the problem set, we will be running the ps5_test.py to make sure that each problem will run correctly. Now that you're all set, feel free to read through the instructions first before following along!
In this problem set, we will define different kinds of triggers that will fire accordingly. We have a superclass, Trigger, where all of the other sub-classes will inherit from. Looking at the evaluate method under the Trigger superclass, it says "Returns True if an alert should be generated for the given news item, or False otherwise." Therefore we will define an evaluate method for each subclass. You can think of the evaluate method as a function that contains the instructions for when each different trigger should fire.

6.2 Problem 1

The instructions already tell you exactly what to do. We must define a class, Newstory, starting with a constructor that takes (guid, title, description, link, pubdate) as arguments and stores them in an object.

```
Class NewsStory(object):
    def __init__(self, guid, title, description, link, pubdate):
```

In the code above, I defined a class called Newstory. The "___init___" method, also known as the constructor, is called when an object of the class is created. It is short for initialize, and allows us to initialize the object's attributes (in this case, the guid, title, description... are the attributes).

```
self.guid = guid
self.title = title
self.description = description
self.link = link
self.pubdate = pubdate
```

Next, we set self.guid to guid and the same thing applys to the other attributes. The ___init___ method receives parameters like guid, title, description, link, and pubdate when an instance of the class is created. Self.guid, self.title, self.description, self.link, and self.pubdate are instance variables. These variables are specific to each instance of the NewsStory class. We set self.guid = guid to store the guid parameter value in the instance variable self.guid. This makes the guid value specific to the instance and allows it to be accessed and used by other methods within the same instance.

```
def get_guid(self):
    return self.guid
def get_title(self):
    return self.title
def get_description(self):
    return self.description
def get_link(self):
    return self.link
def get_pubdate(self):
    return self.pubdate
```

The code above is known as Getter methods. Getter methods are used to access the values of an object's private attributes from outside the class. This is important as we will be using these values later on in the program.
Here is the complete code for problem 1:

```
class NewsStory(object):
    def __init__(self, guid, title, description, link, pubdate):
        self.guid = guid
        self.title = title
        self.description = description
        self.link = link
        self.pubdate = pubdate
    def get_guid(self):
        return self.guid
    def get_title(self):
        return self.title
    def get_description(self):
        return self.description
    def get_link(self):
```

```
        return self.link
    def get_pubdate(self):
        return self.pubdate
```

If you run the ps5_test.py, it should say ok!

6.3 Phrase Trigger

Now let's move on to phrase triggers. Phrase triggers will fire when a specific phrase appears in the text. Valid phrases should not include punctuation or multiple spaces between the words in the phrase. The trigger should only fire when the phrase appears in its entirety and consecutively. Keep in mind that purple cow and purple cows are not the same thing, and this trigger is NOT case sensitive (meaning purple cow and Purple Cow should be treated the same). In the instructions, it also says that the phrase trigger class should be a subclass of Trigger and has a new method: is_phrase_in, which will return True if the whole phrase is in the text and False otherwise. Now that we know what to do, let's get started!

```
class PhraseTrigger(Trigger):
  def __init__(self, phrase):
        self.phrase = phrase.lower()
```

Here we defined the class and it takes a single parameter phrase and initializes an instance variable self.phrase with the lowercase version of the given phrase. This ensures that the phrase is case-insensitive.

```
  def is_phrase_in(self, text):
      phrase = self.phrase.lower()
      text = text.lower()
```

Next, we defined the function which checks if the phrase is in the text. I set both the phrase and input text to lowercase to make sure the comparison is case-insensitive.

```
  translation_table = str.maketrans(string.punctuation,
                                    ' ' * len(string.punctuation))
  text = text.translate(translation_table)
  text = re.sub(r"\s+", " ", text)
```

This is where it gets a little tricky. Starting from the top, I called a variable "translation_table". This is where we will translate/replace each punctuation character with a space. str.maketrans(fromstr, tostr) is a static method that creates a translation table, which is a dictionary mapping each character in fromstr to the character at the same position in tostr. In this case, fromstr is string.punctuation and tostr is a string of spaces. String.punctuation is a predefined string in the string module that contains all the punctuation characters. Len(string.punctuation) gives the number of punctuation characters. Multiplying a single space ' ' by this length creates a string of spaces that is the same length as string.punctuation. So to put it simply, this translation table is used to preprocess the input text by removing punctuation. This ensures that the phrase matching is done on a clean

version of the text, where punctuation does not interfere with the detection of phrases.
Now let's move on to the next line of code: text = text.translate(translation_table). .translate is a method is a built-in string method that takes a translation table (a dictionary) as its argument. This line of code simply goes through each character in the text, and if the character is in the translation table, it will be replaced by the corresponding value from the table.
text = re.sub(r"\s+", " ", text) uses the module "regular expression". It's a function that will search for a pattern in a string and replace it. It will return a new string with the replacements made. "\s" matches any whitespace characters. + means "one or more of the preceding element", so "\s+" matches one or more consecutive whitespace characters. " " is the replacement string, so all matches of the pattern r"\s+" in text will be replaced with a single space. Finally, text is the input string in which the operation is performed. So to put it simply, this will replace sequences of whitespace characters in the string text with a single space.

```
    if self.phrase in text:
        pos = text.index(self.phrase) + len(self.phrase)
```

Now we have an "if" statement. If the phrase is in the text, then I have sent a variable called pos, which is short for position. text.index(self.phrase) returns the starting index of the first occurrence of self.phrase in text. len(self.phrase) returns the length of the phrase. By adding the length of the phrase to the starting index, this gives us the index position immediately after the end of the found phrase in the text.

```
    if len(text) > pos and text[pos] >= 'a' and text[pos] <= 'z':
            return False
    else:
            return True
else
    return False
```

So the first part of the if statement checks if the length of the string text is greater than the index pos. This ensures that 'text[pos]' is a valid index in the string 'text'. Then, text[pos] >= 'a' and text[pos] <= 'z' checks if the character at index pos in the string text is a lowercase English letter. If the above conditions are met, it means the found phrase is immediately followed by a lowercase letter, indicating that the phrase is part of a larger word. Therefore, the function returns False, indicating that the phrase is not a standalone phrase in the text. If the conditions are not met, it means the found phrase is either at the end of the text or followed by a non-letter character, ensuring that it is a standalone phrase. Therefore, the method returns True. Finally, if the phrase is not found in the text, the method returns False.
Here is the full code for PhraseTrigger:

```
class PhraseTrigger(Trigger):
    def __init__(self, phrase):
        self.phrase = phrase.lower()
    def is_phrase_in(self, text):
        phrase = self.phrase.lower()
```

```
        text = text.lower()
        translation_table = str.maketrans(string.punctuation,
                                          ' ' * len(string.punctuation))
        text = text.translate(translation_table)
        text = re.sub(r"\s+", " ", text)
        if self.phrase in text:
            pos = text.index(self.phrase) + len(self.phrase)
            if len(text) > pos and text[pos] >= 'a' and
                                  text[pos] <= 'z':
                return False
            else:
                return True
        else:
            return False
```

6.3.1 Title Trigger

Now let's talk about title triggers. This trigger should only fire when a news story's TITLE contains the given phrase. This trigger should not be case-sensitive either.
This trigger should inherit from Phrase Trigger, and let's think about what we should use. In Phrase Trigger, we defined the function is_phrase_in, which sees if the phrase is in the text. Well in Title Trigger, we want to fire a trigger when the phrase is in the title. Therefore, we need to get the title from the story, and then we can use the is_phrase_in method from Phrase Trigger.

```
class TitleTrigger(PhraseTrigger):
    def __init__(self, phrase):
         super().__init__(phrase)
```

Here, we created the Title Trigger class, and the super() is a built-in function that gives you access to the methods and properties of the parent class, which in this case is Phrase Trigger.

```
    def evaluate(self, story):
        title = story.get_title().lower()
        return PhraseTrigger.is_phrase_in(self, title)
```

Next, we can define the evaluate method in Title Trigger, and then get the title using the get method. Finally, let's look at the last line of code here. We have PhraseTrigger.is_phrase_in(self, title), which refers to the "is_phrase_in" method in PhraseTrigger. We passed in the self and title parameters. Title is the text that needs to be checked for the presence of the phrase. By returning this result, it would check in the phrase is in the title or not.
Here is the final code for TitleTrigger

```
class TitleTrigger(PhraseTrigger):
    def __init__(self, phrase):
        super().__init__(phrase)
```

```
    def evaluate(self, story):
        title = story.get_title().lower()
        return PhraseTrigger.is_phrase_in(self, title)
```

If you run the ps5_test.py, it should say ok!

6.3.2 DescriptionTrigger

Now that we have TitleTrigger down, let's continue with DescriptionTrigger. This trigger will fire when the news item's DESCRIPTION contains the given phrase. This is very similar to TitleTrigger

```
class DescriptionTrigger(PhraseTrigger):
    def __init__(self, phrase):
         super().__init__(phrase)
    def evaluate(self, story):
        description = story.get_description().lower()
        return PhraseTrigger.is_phrase_in(self, description)
```

If you look at this code closely, you will see that it is the exact same as TitleTrigger, but instead of getting the title, we used the get method to get the description. Then, instead of passing in the title parameter, we use description instead. If you run the ps5_test.py, it should say ok!

6.4 Time Trigger

Let's move on from phrase triggers. Now we want to have triggers that is based on when the news story was published, not on its content. For problem 5, we want to implement the abstract class, TimeTrigger, which is a subclass of Trigger. It should take the time in EST, as a string, and convert it to datetime before saving it to an attribute.

```
class TimeTrigger(Trigger):
    def __init__(self, trigger_time):
       self.trigger_time = datetime.strptime(trigger_time,
                                    "%d %b %Y %H:%M:%S")
```

This only takes a couple of lines of code. We defined the class, and converted a string representation of a date and time into a datetime. datetime.strptime(trigger_time, "%d %b %Y %H:%M:%S") parses the string according to the specified format ("%d %b %Y %H:%M:%S") and creates a datetime object representing that date and time. %d, %b, %Y, and so on are different format codes and represent year, month, day and much more. So for example, %d represents day of the month as a zero-padded decimal number (like 01, 02, ..., 30). %b represents month as locale's abbreviated name. You can find out more about what each format code represents here: https://docs.python.org/3.5/library/datetime.html#strftime-and-strptime-behavior.

6.4.1 Before Trigger

This trigger is a subclass of TimeTrigger and should fire when a story is published strictly before the trigger's time.

```
class BeforeTrigger(TimeTrigger):
    def __init__(self, trigger_time):
        super().__init__(trigger_time)
```

We defined the BeforeTrigger class and used the super() method, just like before. Next, we need to define another evaluate method.

```
def evaluate(self, story):
    pubdate = story.get_pubdate()
    return pubdate < self.trigger_time
```

This calls the get_pubdate method on the story object to retrieve the publication date and stores it in the variable pubdate. Keep in mind that pubdate is a variable that holds the publication date that is retrieved from the story object, and self.trigger_time is an attribute that represents a specific time that serves as the trigger. I compared these two, and returned the value of the comparison between pubdate and self.trigger_time. Therefore, if pubdate is less than self.trigger_time (pubdate is earlier than self.trigger_time), it will return true; else it will return false.

6.4.2 After Trigger

Implementing the After Trigger is almost the same as Before Trigger, but instead of letting the trigger fire when a story is published before the trigger's time, it should fire strictly after the trigger's time.

```
class AfterTrigger(TimeTrigger):
    def __init__(self, trigger_time):
        super().__init__(trigger_time)
    def evaluate(self, story):
        pubdate = story.get_pubdate()
        return pubdate > self.trigger_time
```

Here, after defining the class, the only change we made was to change the less sign to a greater sign on line.
Let's run ps5_test.py. We will see two tests for before and after trigger: test3BeforeAndAfterTrigger and test3altBeforeAndAfterTrigger. The regular test should pass, however the alternate test will give us an error that says: can't compare offset-naive and offset-aware datetimes. This is because we have one datetime object that does not include any information about time zone or UTC offset while the other does. To fix this error, we can add this line of code in front of the comparison: pubdate

= pubdate.replace(tzinfo=None) This gets rid of the time zone of pubdate, so we can effectively compare the two.

6.5 Composite Trigger

Now let's move on to Composite Triggers. For example, we may want to raise a trigger when two words appear in the news item. We will implement a NotTrigger, AndTrigger, and OrTrigger. These three triggers will not be a subclass of PhraseTrigger, it should inherit from the superclass Trigger.

6.5.1 Not Trigger

First let's talk about NotTrigger. This should produce its output by inverting the output of another trigger.

```
class NotTrigger(Trigger):
    def __init__(self, another_trigger):
        self.another_trigger = another_trigger
```

Here, I defined the class, and I passed in a parameter called another_trigger. This is because the instructions for writing the Not Trigger class is that it should produce its output by inverting the output of another trigger. Therefore we must have another trigger to invert its output.

```
    def evaluate(self, story):
        return not self.another_trigger.evaluate(story)
```

So I defined the evaluate method, and I returned a value. Not is a logical operator that negates the expression that follows it. Self.another_trigger tells us that the class instance has an attribute named another_trigger, and .evaluate calls on the "evaluate" method on the another_trigger object, passing the story parameter to it. This should return a boolean value (true or false).

6.5.2 And Trigger

This trigger should take two triggers as an input and the news story should fire when both of the inputted triggers fire on that item. So in this case, instead of having one inputted trigger, we need to have two.

```
class AndTrigger(Trigger):
    def __init__(self, trigger1, trigger2):
        self.trigger1 = trigger1
        self.trigger2 = trigger2
```

Here, I assigned the two parameters to an instance variable (self.trigger1, self.trigger2).

```
    def evaluate(self, story):
        return self.trigger1.evaluate(story) and
                    self.trigger2.evaluate(story)
```

Then, I defined the evaluate method and returned the boolean value of the expression that follows it. Instead of using the "or" logical operator, we use "and", which will return True if both operands are true and False otherwise.

6.5.3 Or Trigger

Moving onto our last composite trigger: or trigger. This should fire if either one (or both) of its inputted triggers would fire on the news item. Therefore, this trigger should also take two triggers and use the - you guessed it! - or logical operator!

```
class OrTrigger(Trigger):
    def __init__(self, trigger1, trigger2):
        self.trigger1 = trigger1
        self.trigger2 = trigger2
    def evaluate(self, story):
        return self.trigger1.evaluate(story) or
                self.trigger2.evaluate(story)
```

6.6 Filtering

As of right now, you can run ps5.py and it will fetch and display all of the news items. However, the goal of this problem set was the filter through the news stories, only displaying the ones we want. Therefore, we can write a function, filter_stories(stories, triggerlist), takes in a list of news stories and a list of triggers, and returns a list of only the stories for which a trigger fires. This function shouldn't take more than a few lines of code to write.

```
    def filter_stories(stories, triggerlist):
```

Here, I defined the function, with the two parameters, stories, and triggerlist. The thought process for writing this function is as follows: first I would like to define an empty list that stores the stories. Then, if any trigger fires for a story, that story will be saved to the empty list. Now that we know what we are doing, let's turn this into code!

```
    filtered_stories = []
    for story in stories:
        for trigger in triggerlist:
```

```
            if trigger.evaluate(story):
                filtered_stories.append(story)

    return filtered_stories
```

We initialized an empty list to store the stories that meet the criteria set by the triggers. Then we looped through each story in the stories list. For each story, it loops through each trigger in the triggerlist. For each trigger, it checks to see if the current trigger fires (returns True) for the current story by calling its evaluate method. If any trigger fires for the story, the story is added to the filtered_stories list. Finally, after all stories and triggers have been checked, it returns the list of filtered stories that met the criteria of at least one trigger.

6.7 User-Specified Triggers

Now we will modify the read_trigger_config(filename) function. Our goal here is to return a list of trigger objects specified by the trigger configuration file. It has already given us a few lines of code to start with.

```
lines = []
for line in trigger_file:
    line = line.rstrip()
    if not (len(line) == 0 or line.startswith('//')):
        lines.append(line)
```

Here, it will go through each line in the trigger file and eliminate blank lines and comments. Now, let's begin our own code!

```
triggers = {}
trigger_list = []
```

We set an empty dictionary (triggers) that will store trigger objects with their names as keys. Then, the trigger_list is also currently empty, and will eventually contain the final sequence of trigger objects to be used.

```
for line in lines:
parts = line.split(',')
if parts[0] == 'ADD':
    for name in parts[1:]:
        trigger_list.append(triggers[name])
else:
    trigger_name = parts[0]
    trigger_type = parts[1]

    if trigger_type == 'TITLE':
        triggers[trigger_name] = TitleTrigger(parts[2])
    elif trigger_type == 'DESCRIPTION':
        triggers[trigger_name] = DescriptionTrigger(parts[2])
```

```
    elif trigger_type == 'AFTER':
        triggers[trigger_name] = AfterTrigger(parts[2])
    elif trigger_type == 'BEFORE':
        triggers[trigger_name] = BeforeTrigger(parts[2])
    elif trigger_type == 'NOT':
        triggers[trigger_name] = NotTrigger(triggers[parts[2]])
    elif trigger_type == 'AND':
        triggers[trigger_name] = AndTrigger(triggers[parts[2]],
        triggers[parts[3]])
    elif trigger_type == 'OR':
        triggers[trigger_name] = OrTrigger(triggers[parts[2]],
        triggers[parts[3]])
```

This is a lot, so we will break it down. Starting off with the first line of code, it will go through each line in the lines list. Then, the line is split by commas, creating a list called parts. Then, we have an if/else conditional. Keep in mind that the first element of parts (parts[0]) indicates the type of action to perform. So, to understand this more, you can open the triggers.txt file. Looking at the first two lines, we see

```
// title trigger named t1
t1,TITLE,election.
```

So here, we see that the second line is split up by two commas. The first element (in this case, t1) is at place 0. Now scroling down of the file, we have this.

```
// the trigger list contains t1 and t4
ADD,t1,t4
```

The first element of this is not a trigger, but the keyword add. The ADD command indicates that the following elements are names of triggers that should be added to the trigger_list. So, beneath the if statement, we have a for loop. It loops through each element in parts, starting from the second element (index 1) to the end. This retrieves the trigger object from the triggers dictionary using the name (name) and appends it to the trigger_list. This means that for each name specified after ADD, the corresponding trigger object is added to the list of active triggers. The else: block handles lines that are not ADD commands. These lines define new triggers. trigger_name = parts[0] assigns the first element of parts to trigger_name. trigger_type = parts[1] assigns the second element of parts to trigger_type. This indicates the type of trigger being defined (e.g., TITLE, DESCRIPTION, AFTER, etc.)
Now let's move onto the last section of code. It's basically the same concept for all of them, so if you get one then you'll get the rest.

```
if trigger_type == 'TITLE':
    triggers[trigger_name] = TitleTrigger(parts[2])
```

Let's just use the first one as an example. So here, we have an if conditional that checks to see if the trigger type is title. If it is, then it assigns this TitleTrigger object to triggers under the key trigger_name. In other words, parts[2] is the keyword to be matched in the title. Keep in mind that parts[2] is the third word. So if we have t2,DESCRIPTION,Trump, then parts[2] would be Trump.

It creates a TitleTrigger that activates when a news title contains the keyword from parts[2]. The same concept goes for the rest of the elif statements. Now, scroll down to the main_thread function and uncomment this line: #triggerlist = read_trigger_config('triggers.txt').

6.8 Wrapping Up

At this point, we have finished writing all of the code for the problems in this problem set. The last problem (12) is for you to create your own triggers.txt file with keywords of your own choice. Feel free to create a file of whatever that interests you. We won't be going into detail for that.
However, we do need to make some changes to ps5.py. Because the problem sets from MIT that we have been doing were from a long time ago, the latest version of python that I have been using has gotten rid of some old features. So if you ran your ps5.py code in the terminal and it gave you an error, it might not be because your code is wrong, but rather the ps5.py file that we have been coding on is outdated. So to fix these problems, at the top of the code where it has imported many modules, we will import re, collections, and add an additional line of code below.

```
import re
import collections
collections.Callable = collections.abc.Callable
```

You don't have to understand all of this for now. We will then scroll down to the process function, and we will comment out line 46.

```
#pubdate = datetime.strptime(pubdate, "%a, %d %b %Y %H:%M:%S %Z")
```

We will also comment out line 52.

```
#pubdate = datetime.strptime(pubdate, "%a, %d %b %Y %H:%M:%S %z")
```

On line 53, we will add a new line that basically just changes the lowercase z in line 52 to an uppercase z.

```
pubdate = datetime.strptime(pubdate, "%a, %d %b %Y %H:%M:%S %Z")
```

Congratulations! You have finished the problem set 5 of the MIT 6.0001 introduction-to-computer-science-and-programming-in-python course! Give yourselves a pat on the back, as this problem is a more difficult problem set compared to the others.