Python Self-Study Notes

Python Self-Study Notes

MIT OpenCourseWare 6.0001 Problem Set Explanations & Notes

Tingrui Mary Xie

Python Self-Study Notes

MIT OpenCourseWare 6.0001 Problem Explanations & Notes

Copyright © 2024 by Tingrui Mary Xie

Published by Sisters of Impact Press
Hattiesburg, Mississippi 39402

ISBN 979-8-9911505-4-5

Names: Xie, Tingrui Mary, — author.
Title: Python Self-Study Notes.
Subtitle: MIT OpenCourseWare 6.0001 Problem Explanations & Notes.
Description: Hattiesburg, Mississippi: Sisters of Impact Press, 2024.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means — electronic, mechanical, digital, photocopy, recording, or any other — except for brief quotations and printed reviews, without the prior permission of the publisher.

Printed in the United States of America.

To every curious and motivated learner, whether you're passionate about computer science, just beginning, or a professional.

Table of Contents

Chapter 1: intro to Python

- 1.1 Integers, floats, and Strings
- 1.2 Mathematical operations
- 1.3 For, while, and Nested loops
- 1.4 If/else
- 1.5 Docstrings and Comments
- 1.6 Data Structures
- 1.7 Defining Functions

Chapter 2: House Problem

- 2.1 Getting Started
- 2.2 Part A - House Hunting
- 2.3 Part B - Saving with a Raise
- 2.4 Part C - Finding the right amount to save away

Chapter 3: Hangman Problem

- 3.1 Getting Started
- 3.2 Helper Functions
- 3.3 The Game
- 3.4 Hangman with Hints

Problem 4: The Word Game

- 4.1 Getting Started
- 4.2 Problem 1 - Word Scores
- 4.3 Problem 2 - Dealing with hands
- 4.4 Problem 3 - Valid Words
- 4.5 Problem 4 - Wildcards
- 4.6 Problem 5 - Playing a hand
- 4.7 Problem 6 - Playing a Game

Problem 5: Recursion

- 5.1 Getting Started
- 5.2 Part A - Permutations of a string
- 5.3 Part B - Caesar Cipher
- 5.4 Part C - Substitution Cipher

Problem 6: News Monitor

- 6.1 Getting Started
- 6.2 Phrase Trigger
- 6.3 Time Triggers
- 6.4 Composite Triggers
- 6.5 Filtering
- 6.6 User-Specified Triggers
- 6.7 Modify

Author's note

Throughout my junior year of high school, I have self studied advanced python problems for MIT students from the MIT OpenCourseware 6.0001 Problem set. I have found that writing notes has been very beneficial to understanding the problem and learning skills. Because of this, I was inspired to take a different approach and incorporate my note-taking style with problem explanations and organizing it into a book. I hope to create an educational and interesting book to explain Python problems.

This book is simply my own notes and explanations of these Python problems. I wanted to share my interpretation and Python coding journey with all of you. Please keep in mind that there are many different ways to code a problem.

In addition to these notes, I have created video explanations for each problem in the Python Series on my YouTube channel, Mary's Study Journey.

Chapter 1

What is Python

Python is a popular programming language that is used to create a variety of different programs and often used to build websites and software, automate tasks and conduct data analysis.



With Python, you can do:

- Data analysis and machine learning
- web development
- Automation or scripting
- Software testing or prototyping
- Everyday tasks

we all start somewhere and this is my summary of everything you need to know as a beginner coder in Python.

1.1 Integers, Floats and Strings

Integers: numbers without decimal points.
- -indicated with int()
- -if you want to change the float to an integer, use the int() function.

Floats: numbers with decimal points.
- -indicated with float()

Int(x): conversion of string or float x to an integer
float(x): conversion of string or integer x to float

In Python, it is important to differentiate between floats and integers. This is because Python represents floats as powers of 2, so many decimal numbers can not be exactly represented.

For example, if you had a loop adding 0.1+0.1+0.1, it won't be equal to 0.3, but rather 0.30000000000004. When you are using simple elements, always try to use integers to avoid this type of problem.

Ex: if you want to do integer division of 5 and 7, you should cast one or both of the integers to a float before doing the division, float(3)/5 or 3/float(5). Remember that Python evaluates inner expressions first so float(3/5) wouldn't produce the wanted result.

```
1
2 n = input("Enter a number")
3 n = int(n)
4 annual_salary = float(input("Enter your annual salary: ")
5
6
```

Example: Float: 6.2 ⟶ Integer: 6

int(6.2) = 6

Strings

A string is a sequence of characters enclosed within either single quotes (' ') or double quotes (" "). Strings are one of the fundamental data types in Python and are used to represent text. o.

Both single and double quotes can be used interchangeably to define strings, but you should be consistent on your usage within a particular code base.

Example:

```
single_quoted_string = 'Hello, World!'
double_quoted_string = "Python Programming"
```

Strings in Python are immutable, which means once a string is created, you cannot modify its contents. However, you can perform various operations on strings, such as concatenation, slicing, and formatting.

Concatenation:

```
str1 = "Hello"
str2 = "World"
result = str1 + " " + str2
print(result) #results in "Hello World"
```

Concatenation combines two or more strings or sequences into a single string or sequence.

Slicing

```
my_string = "Python"
print(my_string[1:4])  # Slices from index 1 to 3, prints 'yth'
print(my_string[:3])   # Slices from the beginning to index 2, prints 'Pyt'
print(my_string[2:])   # Slices from index 2 to the end, prints 'thon'
print(my_string[::2])  # Slices with a step of 2, prints 'Pto'
```

Slicing is the operation of extracting a portion of the sequence. It is done using the colon operator. The syntax is "startstopstep". "Start" is the index of the first element to include, "stop" is the index of the first element you want to exclude, and "step" is the step size between elements. If any is omitted, if defaults to the beginning, the end, and a step size of 1.

Indexing

```
my_string = "Python"
print(my_string[0])  # Accesses the first character, prints 'P'
print(my_string[1])  # Accesses the second character, prints 'y'
print(my_string[-1])  # Accesses the last character, prints 'n'
print(my_string[-2])  # Accesses the second-to-last character, prints 'o'
```

Indexing refers to accessing individual elements in a sequence In Python, indexing starts at o

Working with strings

```
>>> my_string = " Hello World! "
>>> print(my_string.strip())
Hello World!
>>> print(my_string.lower())
 hello world!
>>> print(my_string.upper())
 HELLO WORLD!
>>> print(my_string.replace("Hello", "Hi"))
 Hi World!
```

Formatting strings

```
>>> name = "Alice"
>>> age = 30
>>> print("My name is %s and I am %d years old." %(name, age))
My name is Alice and I am 30 years old.
>>> print("My name is {} and I am {} years old.".format(name,age))
My name is Alice and I am 30 years old.
>>> print(f"My name is {name} and I am {age} years old.")
My name is Alice and I am 30 years old.
```

You can format and strings using different methods, such a the "%" operator.

1.2 Mathematical Operations

Operator	Name	Example
+	Addition	`sum_result = 3 + 5  # Result is 8`
-	Subtraction	`difference_result = 8 - 3  # Result is 5`
*	Multiplication	`product_result = 3 * 5  # Result is 15`
/	Division	`result = 5 / 2  # Result is 2.5, not 2`
**	Exponential	`result = 2 ** 3  # Result is 8`
//	Floor division Returns the largest integer less than or equal to the division result	`result = 5 // 2  # Result is 2`
%	Modulus Returns the remainder of the result	`remainder = 7 % 3  # Result is 1, because 7 divided by 3 is 2 with a remainder of 1`

Integers or Floats

Always Integers

Mixed-Type Operations: If you perform operations with both integers and floats, Python will generally promote the result to a float to avoid losing precision.

```
result = 5 * 2.0  # Result is 10.0
```

In this operation, Python handles the mixed types and returns the most general type involved in for operation (generally a float if any operant is a float). If you want to explicitly convert the result to an integer or float, you can use "int()" or "float()" functions

You can convert between integers and floats using the int() and float() functions, as shown in section 1.1.

```python
int_result = int(2.5)  # Converts the float 2.5 to the integer 2
float_result = float(5)  # Converts the integer 5 to the float 5.0
```

Inequality signs	Names	Example
<	Less than	`result = 5 < 10  # Result is True`
>	Greater than	`result = 15 > 7  # Result is True`
<=	Less than or equal to	`result = 10 <= 10  # Result is True`
>=	Greater than or equal to	`result = 20 >= 15  # Result is True`

1.3 For, While, and Nested Loops

Looping: repeating something over and over until a particular condition is satisfied.

Loop in Python: control flow statement flag is used to repeatedly execute a group of statements as long as the condition is satisfied.

There are 3 types of loops:

For loops

In general, use for loops when you know the number of iterations you need to do. For example, I'd you need 400 trials, one operation per character in a string, or an action on every element in a list. If you can describe the problem you are trying to solve in terms of each or every element in an iteration, use a loop. This will decrease the risk of writing an infinite loop and help prevent errors.

Example:

```
1
2 for i in range(400):
3     print('hello')
```

Prints "hello" 400 times

while loops

If you want a certain condition to be satisfied, use the while loop. These are useful when you can determine the number of iterations in terms of a Boolean variable. Use while loops if you are waiting for a user to enter an input correctly or for a randomly generated value to exceed a certain amount. If you can use "until" to describe the problem, use while loops.

Boolean variable: variable that have 2 values (true or false)

Example:

```
num = float(raw_input("Enter a positive number: "))
while num <= 0.0:
    num = float(raw_input("Enter a POSITIVE number: "))
```

Loop until the user enters a positive number

To exit out of loops as soon as you find your answer, use "break" at the end of the loop. This helps you improve the average-case performance of your code.

Example:

```
my_list = [1,2,3,4,5,6,7,8]
greater_than_five = False
for elem in my_list:
    if elem > 5:
        greater_than_five = True
        break
```

Nested Loops

A nested loop is a loop inside another loop. This means you have one loop (the inner loop) running multiple times inside another loop (outer loop).

```
for i in range(1, 6):  # Outer loop for rows
    for j in range(1, 6):  # Inner loop for columns
        print(i * j, end='\t')  # Print the product, separated by tabs
    print()  # Move to the next line after completing the inner loop
```

The code above results in a multiplication table of numbers 1 to 5!!

```
1       2       3       4       5
2       4       6       8       10
3       6       9       12      15
4       8       12      16      20
5       10      15      20      25
>>>
```

The "range" function

generates numbers up to, but not including the specified upper bound.

1.4 If/Else Conditions

In Python, "if" and "else" are used to create conditional statements that allow your program to make decisions based on whether a given condition is true or false.

```
if x > 5:
    print("x is greater than 5")
else:
    print("x is less than 5")
```

```
if my_function(): #my_function returns True or False
    return True
else:
    return False
```

In the code above, it states that if the code shows my_function, then return True, otherwise return False.

If you want to return "False" when "my_function" is True and "True" when "my_function" is False, you can code as shown below:

```
if my_function() == True
    return False
else:
    return True
```

In Python, the "==" operator is used for equality comparison. It checks whether the values on the left and right sides of the operator are equal and returns a Boolean result: "True" if they are equal, and "False" if they are not.

1.5 Docstrings and Comments

Docstrings

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. It is used to provide documentation about the purpose of your code. When writing new classes and functions, it is important to document your intent by using docstring.

These are indicated by 3 double or single quotes before and after the statements. (""" or ''')

Function: a block of code which only runs when it is called
Class: a code template for creating objects

```
def greet(name):
    """
    This function takes a name as an argument and
    prints a greeting message with the provided name.
    """
    print(f"Hello, {name}!")

# Calling the function
greet("Alice")
```

#Comments

Comments are the lines in the code that are ignored by the interpreter during the execution of the program. Comments enhance the readability of the code and helps programmers understand the code very carefully

Comments are indicated with # signs before the text. As shown in the code above, the programmer wrote "calling the function" as a comment, indicating what the code below's purpose is.

1.6 Data Structures

List

A list is a mutable and ordered collection of elements. They can contain elements of different data types and they allow for easy manipulation and access of the data they hold.

Lists are defined by enclosing the elements within square brackets '[]' and separating them with commas.

```
my_list = [1, 2, 3, 4, 5]
```

Mutable: lists can be modified after creation.
Ordered: the elements in a list are stored in a specific order and you can access them by their index. The index starts at 0 for the first element.

Using lists:

Accessing elements:

```
first_element = my_list[0]  # Access the first element (index 0)
```

Remember that the first value in the list has a index of 0!

Slicing:

```
sub_list = my_list[1:4]  # Get a sublist from index 1 to 3
```

Adding Elements:

```
my_list.append(6)  # Add an element to the end of the list
```

Inserting Elements:

```
my_list.insert(2, 7)  # Insert the value 7 at index 2
```

Removing Elements:

```
my_list.remove(3)  # Remove the first occurrence of the value 3
```

Length of list

```
length = len(my_list)  # Get the number of elements in the list
```

{ Sets }

In python, a set is an unordered collection of unique elements. It is defined by placing a comma-separated sequence of elements inside curly braces "{}", and duplicates are not allowed. The order of elements in a set is not guaranteed, and sets are mutable, meaning you can add or remove elements from them.

Ex:

```
my_set = {1, 2, 3, 4, 5}
print(my_set)
```

Sets are useful when you need an unordered collection of unique elements and wand to perform operations like checking membership, intersections, unions, etc. below are some common use cases for sets:

```
set1 = {1, 2, 3, 4, 5}
set2 = {3, 4, 5, 6, 7}

# Union
union_set = set1 | set2
print(union_set)  # Output: {1, 2, 3, 4, 5, 6, 7}

# Intersection
intersection_set = set1 & set2
print(intersection_set)  # Output: {3, 4, 5}

# Difference
difference_set = set1 - set2
print(difference_set)  # Output: {1, 2}

# Adding and Removing Elements
my_set.add(6)
print(my_set)  # Output: {1, 2, 3, 4, 5, 6}

my_set.remove(3)
print(my_set)  # Output: {1, 2, 4, 5, 6}
```

Unions sets contain of elements is both "set1" and "set2"

Intersection sets contain only the elements that are common to both "set1" and "set2"

Difference sets will contain elements from "set1" that are not present in "set2"

You can add or remove elements of a set with the "add()" or "remove()" method.

(Tuples)

A tuple is an ordered and immutable collection of elements. Tuples are used to group together related data, and they are similar to lists in many ways, but with a key difference: once a tuple is created, its elements cannot be modified, added, or removed. Tuples are defined using parentheses "()".

Example:

```
my_tuple = (1, "apple", 3.14)
```

In this example, "my_tuple" contains three elements: an integer ('1'), a string (' "apple" '), and a floating point number ('3.14')

Key characteristics of tuples:

Ordered: tuples maintain the order of elements, meaning the order in which elements are added to the tuple is preserved.

Immutable: once a tuple is created, you cannot modify its elements. You cannot add, remove, or change the order of elements within a tuple.

Heterogeneous: tuples can contain elements of different data types, allowing you to create a collection of diverse values.

Access by index: elements in a tuple can be accessed using their index(position) within the tuple. Indexing starts at 0.

1.7 Defining Functions

In Python, a function is a reusable block of code that performs a specific task. Functions are defined using the "def" keyword, followed by the function name, a pair of parentheses containing optional parameters, and a colon. The function body is indented, and it contains the code that gets executed when the function is called.

Example:

```
my_set = {1, 2, 3, 4, 5}
def add_numbers(x, y):
  result = x + y
  return result
```

"add_numbers" is the name of the function.
"(x, y)" are the parameters (input values) that the function accepts.
The colon ":" indicates the beginning of the function body.
"result = x + y" is the code inside the function that calculates the sum.
"return result" specifies the value that the function will output.

You can then call functions with specific values:

```
sum_result = add_numbers(3, 5)
print(sum_result)  # Output: 8
```

This is a basic function, but Python supports more advanced features for functions, such as default parameter values, variable length argument lists, and the ability to return multiple values using tuples.

Chapter 2

2.1 Getting Started

In the following chapters, we will be completing the 6.0001 MIT OpenCourseWarse Python coding problems. In this chapter, we will begin with the first problem, or problem set one.

The first step in writing this program is to download the instructions from the website. This can be found on the MIT OpenCourseware website and we will write the code in a separate python text document.

In this problem, we will be using Python to solve the problem. There will be three sections to this problem where we use coding to calculate how long it will take for you to save enough money to make the down payment of a house given certain variables. This is a real life application, which shows we can use coding in real life situations. A lot of times, it is much easier to use coding to solve problems than it is if we had to calculate it by hand. Today, we will be going through each problem and learning how to code them.

Before we start, we are going to first understand where we will write code and how it will work. You can use a TexShop file and save it as a Python file or you can use xcode. Both will create Python files that we can run. Name your file ps1 for problem set one, or whatever name you prefer, and we will begin!

Save the downloaded instructions file and your Python file to the same directory on your computer. I suggest you make a folder dedicated to this game. It will be much easier to navigate!

Terminal

A terminal is a text-based interface where you can interact with a computer by typing commands. It provides a way to execute commands. Navigate the file system, and perform various tasks without using a graphical user interface.

This is where we will be testing our code. To get here, you must first download the terminal app. When you are in the base level of the terminal, you can then enter different files on your computer or simply use python to test codes.

This is what the terminal app should look like!



Once you open the terminal app, you should be in the base. To enter the folder that your files are in, you type "cd" and then the name of your folder. To enter the file, type "python3" and then the name of your file with ".py" at the end to indicate it is a Python file.

2.2 Part A - House Hunting

Before we begin, let's read the instructions and see what we are given. It tells us that we will make the following assumptions:

1. Call the cost of your dream home **total_cost**.
2. Call the portion of the cost needed for a down payment **portion_down_payment**. For simplicity, assume that portion_down_payment = 0.25 (25%).
3. Call the amount that you have saved thus far **current_savings**. You start with a current savings of $0.
4. Assume that you invest your current savings wisely, with an annual return of **r** (in other words, at the end of each month, you receive an additional **current_savings*r/12** funds to put into your savings – the 12 is because **r** is an annual rate). Assume that your investments earn a return of r = 0.04 (4%).
5. Assume your annual salary is **annual_salary**.
6. Assume you are going to dedicate a certain amount of your salary each month to saving for the down payment. Call that **portion_saved**. This variable should be in decimal form (i.e. 0.1 for 10%).
7. At the end of each month, your savings will be increased by the return on your investment, plus a percentage of your **monthly salary** (annual salary / 12).

we will be writing a program to calculate how many months it will take to save up enough money for a down payment. Your main variables should be floats, so you should cast user inputs to floats. Your program should ask the user to enter the following variables:

1) the starting annual salary (annual_salary)
2) the portion of the salary to be saved (portion_saved)
3) the cost of your dream home (total_cost)

The instructions tell us that our variables should be inputs, and tell us the name of the variables we should set it to. we should clarify what the user is entering by labeling the input.

```
annual_salary = float(input("Enter your annual salary: "))
portion_saved = float(input("Enter the percent of your salary to save, as a decimal:" ))
total_cost = float(input("Enter the cost of your dream home:"))
```

After this, we should initialize some state variables. We will decide what information we need. Be careful about value of represent annual amounts, and those that represent monthly amounts. In the beginning, there were several assumptions made. We will translate this to code.

```
monthly_salary = (annual_salary)/12
monthly_saving = (monthly_salary)*(portion_saved)
current_savings = 0
portion_down_payment = (total_cost)*(0.25)
annual_return = 0.04
```

Now we will use what we know and write the code to actually calculate how much we need to save to make a down payment.

To determine how many months we will have to save up for, we will test the number of months one by one starting from zero. When the current savings is less than amount for the down payment, we will save for another month. Until our savings are more than the amount for the down payment, we will continue incrementing the number of months. But first, we have to determine the formula for our current savings.

Our current savings are what we have for our monthly savings plus our current savings multiplied by one plus our annual return divided by 12.

```
current_savings = (monthly_saving) + (current_savings)*(1 + annual_return/12)
```

To increment the number of months as we go, I will set the number of months, or n for short, to zero before we begin. Then, while our current savings is less than the down payment, we will update our current savings and increment the number of months by one.

Using a while loop we can ensure that our loop will end once our current savings is more than the down payment.

```
n = 0
while current_savings < portion_down_payment:
    current_savings = (monthly_saving) + (current_savings)*(1 + annual_return/12)
    n = n + 1
```

Finally, we will print out our result which is the number of months.

```
result = n
print("Number of months:", result)
```

The instructions give us test cases in which we can try out to see if our code works.

Below is the full code and an example of what our output should look like when we run it.

```
annual_salary = float(input("Enter your annual salary: "))
portion_saved = float(input("Enter the percent of your salary to save, as a decimal:" ))
total_cost = float(input("Enter the cost of your dream home:" ))

monthly_salary = (annual_salary)/12
monthly_saving = (monthly_salary)*(portion_saved)
current_savings = 0
portion_down_payment = (total_cost)*(0.25)
annual_return = 0.04

n = 0
while current_savings < portion_down_payment:
    current_savings = (monthly_saving) + (current_savings)*(1 + annual_return/12)
    n = n + 1

result = n
print("Number of months:", result)
```

```
Enter your annual salary: 120000
Enter the percent of your salary to save, as a decimal:.10
Enter the cost of your dream home:1000000
Number of months: 183
```

2.3 Part B - Saving, with a raise

In part A, we unrealistically assumed that our salary won't change. But we will change this by building onto our solution to part A by factoring in a raise every six months. We will modify our code from part A to include the following:

1) have the user input a semi annual salary raise semi_annual_raise (as a decimal percentage)
2) after the sixth month, increase your salary by that percentage. Do the same after the 12th month, the 18th month, and so on.

We will write a program to calculate how many months it will take for you to save up enough money for a down payment. Like before, we will assume that the investments have a return rate of 0.04 and the required down payment percentage is 0.25. The user will enter the following variables:

1) the starting annual salary (annual_salary)
2) the percentage of salary to be saved (portion_saved)
3) the cost of your dream home (total_cost)
4) the semi annual salary raise (semi_annual_raise)

First, we will receive the user input just like we did in the beginning of part A.

```
annual_salary = float(input("Enter your annual salary: "))
portion_saved = float(input("Enter the percent of your salary to save, as a decimal:" ))
total_cost = float(input("Enter the cost of your dream home:"))
semi_annual_salary_raise_rate = float(input("Enter your semi annual salary raise rate, as a decimal:"))
```

Now we will initialize our state variables. This part should be the same as the previous code.

```
monthly_salary = (annual_salary)/12
monthly_saving = (monthly_salary)*(portion_saved)
current_savings = 0
portion_down_payment = (total_cost)*(0.25)
annual_return = 0.04
```

Now you will write the statements that determine how many months we will need before we can pay the down payment. Remember, we will be increasing our salary, which should only happen after the 6th, 12th, 18th month, and so on.

The conditional should still be the same when our current saving is less than our down payment. we will still be in incrementing the number of months by one. However, we will add another conditional in which if it is a month that is divisible by six, then we will be updating the monthly saving and the monthly salary.

```
n = 0
while current_savings < portion_down_payment:
    current_savings = (monthly_saving) + (current_savings)*(1 + annual_return/12)
    n = n + 1
    if n%6 == 0:
        monthly_saving = (monthly_salary)*(1 + semi_annual_salary_raise_rate)*(portion_saved)
        monthly_salary = (monthly_salary)*(1 + semi_annual_salary_raise_rate)
```

Now we will print the result which is a number of months just like we did before. we will also test our program with the test cases from the instructions.

our final code and an example of the output when we run it is shown below

```
annual_salary = float(input("Enter your annual salary: "))
portion_saved = float(input("Enter the percent of your salary to save, as a decimal:" ))
total_cost = float(input("Enter the cost of your dream home:"))
semi_annual_salary_raise_rate = float(input("Enter your semi annual salary raise rate, as a decimal:"))

monthly_salary = (annual_salary)/12
monthly_saving = (monthly_salary)*(portion_saved)
current_savings = 0
portion_down_payment = (total_cost)*(0.25)
annual_return = 0.04

n = 0
while current_savings < portion_down_payment:
    current_savings = (monthly_saving) + (current_savings)*(1 + annual_return/12)
    n = n + 1
    if n%6 == 0:
        monthly_saving = (monthly_salary)*(1 + semi_annual_salary_raise_rate)*(portion_saved)
        monthly_salary = (monthly_salary)*(1 + semi_annual_salary_raise_rate)

result = n
print("Number of months:", result)
```

```
Enter your annual salary: 150000
Enter the percent of your salary to save, as a decimal:.07
Enter the cost of your dream home:900000
Enter your semi annual salary raise rate, as a decimal:.04
Number of months: 135
```

2.4 Part C - Finding the right amount to save away

In part B, we had a chance to explore how both the percentage of our salary that we save each month and our annual raise affect how long it takes for you to save for a down payment. But if you wanted to set a particular goal, for example, to be able to afford the down payment in three years, how much would you need to save each month to achieve this? In this problem, we will be writing a program to answer that question.

There are a few things that we will assume to simplify this problem.

1. Your semi-annual raise is .07 (7%)
2. Your investments have an annual return of 0.04 (4%)
3. The down payment is 0.25 (25%) of the cost of the house
4. The cost of the house that you are saving for is $1M.

we are now going to try to find the best rate of savings to achieve a down payment of $1 million house in 36 months. Since hitting this exactly is a challenge, we simply want your savings to be within $100 of the required down payment.

we will write a program to calculate the best savings rate, as a function of our starting salary. The instructions tell us to use bisection search to help us do this efficiently. we should keep track of the number of steps it takes our bisection search to finish. we should be able to use some parts of part B in this problem.

Because we are searching for a value that is a float, we are going to limit ourselves to do two decimals of accuracy. This means we can search for an integer between zero and 10,000 and then convert it to a decimal percentage to use when we are calculating the current savings after 36 months.

The only input we need is the annual salary, so we will ask the user to enter their annual salary.

```
starting_annual_salary = float(input("Enter your annual salary: "))
```

Now we will write the code for the things we assumed in the beginning and also defined what we will use in this problem.

```
total_cost = 1_000_000
current_savings = 0
portion_down_payment = total_cost*0.25
semi_annual_salary_raise = 0.07
annual_return = 0.04
num_months = 36
current_savings = 0
annual_salary = starting_annual_salary
```

Because we are searching for a value that is in principle of float, we are going to limit ourselves to two decimals of accuracy. This means we can search for an integer between 0 and 10,000. We will also be counting the number of bisections it takes to finish the search so I'm going to define a variable and set it to 0.

```
low = 0
high = 10_000
n_bisec = 0
```

In part B of this problem, we were asked how many months it would take for us to be able to put down the down payment of a house. However, in this part of the problem, we are given the number of months and we are trying to find how much money we need to save per month to reach this goal.

Because the instructions tell us that the savings amount is within $100 of the required amount, we must write code as to when our bisection search should be considered complete. When the current savings is more than the down payment amount but less than the amount plus 100, we know we have found the mid point within the bisection.

```
while low <= high and (
 not (
      current_savings >= portion_down_payment
      and
      current_savings <= portion_down_payment + 100
     )
):
 mid = (low + high) // 2
 portion_saved = mid / 10000
```

Now we will write the code to update the annual salary. For every month, we know the formula for the monthly salary, monthly saving, investment dividend, and current savings. If the month is the 6th, 12th, 16th, etc., month, then we update the annual salary to contain the semi annual salary raise. Then, we update how many bisections we used.

```
for i in range(num_months):
 monthly_salary = annual_salary / 12
 monthly_saving = monthly_salary * portion_saved
 investment_dividend = current_savings * (1 + annual_return / 12)
 current_savings = monthly_saving + investment_dividend

 if (i+1) % 6 == 0:
  annual_salary = annual_salary * (1 + semi_annual_salary_raise)

n_bisec = n_bisec + 1
```

To test our code, we can always print out some statements. This can be deleted once we confirm our code is correct. The test code could be something as follows:

```
print('low = {}, high = {}, mid = {}, current_savyings = {}'
        .format(low, high, mid, current_savings));
```

If the current savings is more than the down payment, then our high value becomes our middle value. Otherwise, our low value is the middle value for our bisection search.

```
if current_savings >= portion_down_payment:
 high = mid
else:
 low = mid
```

Finally, we can print out our current savings, best savings rate, and the number of steps in the bisection search that it took.

```
print("current savings:", current_savings)
print("best savings rate:", portion_saved)
print("steps in bisection search: ", n_bisec)
```

Now we have just finished the first problem set. In this problem, we have learned how to use python in real life problems and also how to use a bisection search to efficiently program code.

The final code is below

```
starting_annual_salary = float(input("Enter your annual salary: "))

total_cost = 1_000_000
current_savings = 0
portion_down_payment = total_cost*0.25
semi_annual_salary_raise = 0.07
annual_return = 0.04
num_months = 36
current_savings = 0
annual_salary = starting_annual_salary

low = 0
high = 10_000
n_bisec = 0

while low <= high and (
 not (
      current_savings >= portion_down_payment
      and
      current_savings <= portion_down_payment + 100
     )
):
 mid = (low + high) // 2
 portion_saved = mid / 10000

 for i in range(num_months):
  monthly_salary = annual_salary / 12
  monthly_saving = monthly_salary * portion_saved
  investment_dividend = current_savings * (1 + annual_return / 12)
  current_savings = monthly_saving + investment_dividend

  if (i+1) % 6 == 0:
   annual_salary = annual_salary * (1 + semi_annual_salary_raise)
 n_bisec = n_bisec + 1

 print('low = {}, high = {}, mid = {}, current_savyings = {}'
         .format(low, high, mid, current_savings));

 if current_savings >= portion_down_payment:
  high = mid
 else:
  low = mid

print("current savings:", current_savings)
print("best savings rate:", portion_saved)
print("steps in bisection search: ", n_bisec)
```

An example output of our code is shown below:

```
Enter your annual salary: 300000
low = 0, high = 10000, mid = 5000, current_savyings = 566774.4123135424
low = 0, high = 5000, mid = 2500, current_savyings = 283387.2061567712
low = 0, high = 2500, mid = 1250, current_savyings = 141693.6030783856
low = 1250, high = 2500, mid = 1875, current_savyings = 212540.40461757834
low = 1875, high = 2500, mid = 2187, current_savyings = 247907.1279459436
low = 2187, high = 2500, mid = 2343, current_savyings = 265590.489610126
low = 2187, high = 2343, mid = 2265, current_savyings = 256748.80877803464
low = 2187, high = 2265, mid = 2226, current_savyings = 252327.96836198907
low = 2187, high = 2226, mid = 2206, current_savyings = 250060.87071273496
best savings rate: 0.2206
steps in bisection search:  9
```

Chapter 3

3.1 Getting Started

Hangman is a very common game played by many, but did you know you can use Python to create this game? It is a guessing game where one player thinks of a word, phrase, or sentence and the other player tries to guess it by suggesting letters within a certain number of guesses. In this version, we will create hangman where a player will be guessing a word generated by the computer.

This chapter is separated into several sections that correspond with YouTube video episodes on my channel and website.

The first step in writing this program is to download the word file and template we will be using. This game is modeled after the MITOPENCOURSEWARE Introduction To Computer Science And Programming In Python Problem Set 2 Hangman problem.

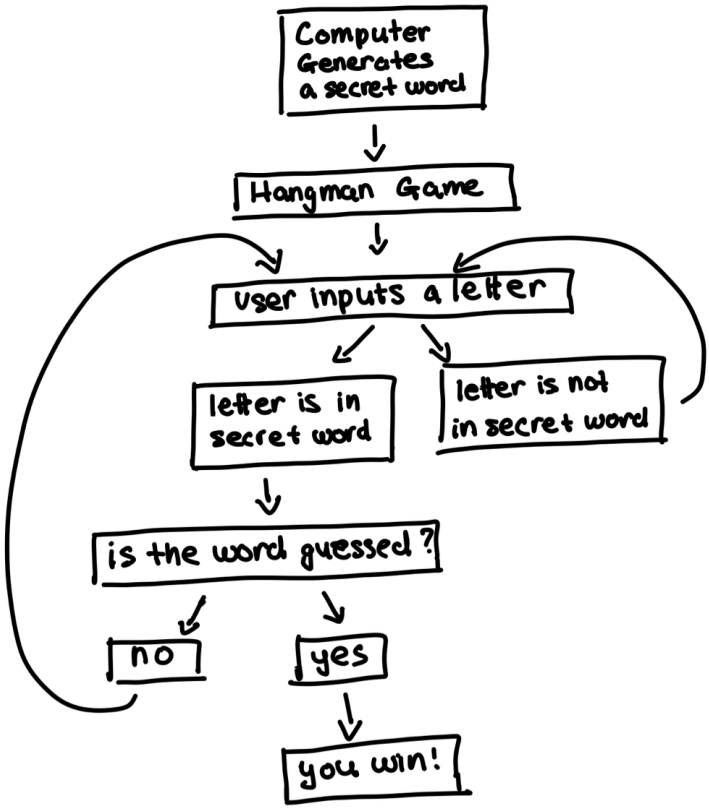
We will be using the file provided to download the hangman.py and words.txt files. The words.txt is a file that contains all of the words the computer will choose from to generate the word the player guesses. The hangman.py is what we will be using as our template to write the main functions of hangman.

Once you have downloaded these files, save them to the same directory on your computer. I suggest you make a folder dedicated to this game. It will be much easier to navigate!

To find these links, go to herstemjourney.com and they will be under the section titled Python Notes!

Flow Chart

To get a gist of what we will be doing to write this game, I have created a rough flowchart of the process.



This is how the game should run when we program it. By drawing out a flow chart, we have created a template for what small functions we need to write to accomplish this.

Transferring flow chart to code

Open the file hangman.py. This is where we will be writing the main code this program uses to function. The two helper codes "load_words()" and "choose_word(wordlist)" are on the top of the function because these are what the computer uses to pick a word from the wordlist file we downloaded in the first step.

Now we can scroll down to where it says def hangman(secret_word). "Def" is short for define and in Python, we must first define a function to use it. Below is where you will write the code that will run when we call the hangman function.

Using the flow chart we have already created, we will be converting it to code. The first step in the hangman function is to let the user input a guess. Next we determine whether or not the letter guessed is in the secret word.

to let user input something

program prints out what is in quotes

```
def hangman(secret_word):
    current_guess = input("Please guess a letter: ")
    if is_letter_guessed(secret_word, current_guess):
        print("Good guess")
    else:
        print("Oops! That letter is not in my word.")
```

if the current guess is a letter in secret word

then

otherwise

Note that we are using the function "is_letter_guessed" but we have not yet written the function so it will not run. In order for this program to work, you must define is_letter_guessed (a function that will determine whether or not the letter you have just guessed is a letter in the secret word.)

You must update the letters_guessed after the user guesses the word so it contains all of the letters the user has guessed. We can do this by setting letters_guessed to letters_guessed + current_guess.

```
letters_guessed = letters_guessed + current_guess
```

The next step is determining whether or not the word has been guessed. Note that the word will only be guessed if ALL of the letters in secret_word have been guessed.

out of the letters guessed, if all letters in secret word are guessed, then you win the game.

```
if is_word_guessed(secret_word, letters_guessed):
    won = True          → if the secret word is guessed, the player wins the game.
    print("word guessed!")
    break               → break out of the loop
```

However, we realize that everyone can guess the right letters as long as we have enough guesses. If you guess all the letters of the alphabet, it is guaranteed that you'll get the word right. However, to make this game more engaging, we don't want that. Instead, we can use a for loop (refer to chapter 1.3) to limit how many guesses the user has.

you can set the number of guesses the user has to however many you like

```
for remaining_guesses in range(10):
    print("You have " +str(10 - remaining_guesses) + " guesses left.")
    print("Available letters :", get_available_letters(letters_guessed))
```

tells what the available letters are

when printing out sentence, you have to separate out text from string.

Similar to the function on the last page, "is_letter_guessed", "remaining_guesses" is not yet defined. We must define this function to use it.

Everything else that we have written should go INSIDE the for loop. This is because the if conditions hold when remaining_guesses is in a range of 10. Make sure the indentions match the level each function is on. Refer to the next page for the full function.

Notice that when the word was guessed, we set won to true. However, in the beginning of the function, when the user has not guessed the word yet, we must set won to false.

In addition, we use "letters_guessed" in the function, but it is also not defined. Because in the beginning of the game, the user has not guessed any letters, letters_guessed should be blank. Therefore, we can set letters_guessed = "" which indicates that it is an empty set.

For testing purposes, we can always print out the secret word at the beginning of the function to make sure our program is running as it is supposed to. This is a great way to test our program.

```
print(secret_word)
letters_guessed = ""
won = False
```

we can put these lines above and in line with the for loop because these go before the loops.

The last step is what happens if the user does not guess the word in the designated number of tries. This next part will go outside of the for loop because it is what happens if remaining_guesses is out of the range.

we can write another if statement at the end of the function that is aligned with the for loop. This last statement should convey that if the game is not won, the computer should print out that the user has lost.

```
if not won:
    print("YOU LOST!!!")
```

→ you can always modify what you want to be printed

when you are done, your program should look something like this.

```
letters_guessed=""
won = False
for remaining_guesses in range(6):
    print("You have " + str(6 - remaining_guesses) + " guesses left.")
    print("Available letters :", get_available_letters(letters_guessed))
    current_guess = input("Please guess a letter:")
    if is_letter_guessed(current_guess, secret_word):
      print("Good guess")
    else:
      print("Oops! That letter is not in my word.")
    letters_guessed = letters_guessed + current_guess
    if is_word_guessed(secret_word, letters_guessed):
        won = True
        print("word guessed!")
        break

    print(get_guessed_word(secret_word, letters_guessed))
if not won:
    print("LOSER!!!")
```

→ these highlighted functions are not yet written, so this function can't be run. Our next step is to write these helper functions so we can use them in our code

↳ notice you don't always need an "else" after a "if".

↳ always make sure you are consistent with your indentation. I use one tab for each level but you can use the same number of spaces, etc.

However, because the highlighted functions are not defined, if you try to run this program, it should say "name 'get_available_letters' is not defined". This is because the first function it runs into that is not defined is the get_available_letters.

3.2 Helper Functions

Helper functions are small, specialized functions that are created to perform a specific task within a program. They are designed to be reusable and assist in solving larger problems by breaking them down into smaller, more manageable parts. They are an essential part of programming because they promote readability and organization.

In this section, we will be writing the three helper functions this program needs to function. In the last section, we used the functions is_word_guessed, get_guessed_word, and get_available_letters, and in this section we will be defining them. As we saw, these code did not function when we ran it because these functions are not defined.

This section corresponds to episodes 3,4,5 on YouTube MaryStudyJourney.

is_word_guessed

In this helper function, we will be determining whether or not the secret word has been guessed. To do this, we assume that all of the letters guessed are lowercase. what this function will do is return True is all of the letters that are in the secret word have been guessed and False otherwise.

The first step is to determine whether a letter of the secret word is guessed. In order to do so, we will be writing another helper function called "is_letter_guessed".

1a is_letter_guessed

By defining is_letter_guessed, we are defining a helper function for a helper function. This continues to simplify our code and make it more readable. is_letter_guessed determines whether a single letter that has been guessed is in secret word. This function should return a true or a false because that is what we need in our main function. The parameters of this function is the letter that is being guessed and the secret word, so when we are defining our function, it should look like this:

```
def is_letter_guessed(secret_letter, letters_guessed):
    return secret_letter in letters_guessed
```

In is_word_guessed, we will be determining whether each letter in the secret word has been guessed. In other words, we are determining if all of the letters in the secret_word are in letters_guessed. To do so, we have to use the length of the secret word and compare all of its letters to those is letters_guessed. So in our function, the first thing we will write is length_sw = len(secret_word). This assigns the actual length of the secret word to length_sw. Now we can use length_sw as a parameter in our code.

Now we can use the for loop to compare each letter of the secret word to the letters guessed.

If all of the letters have been guessed, the computer should return "True", otherwise the computer should return "False".

The best way to do this is to use "if not" instead of the "if" loop. In other words, if all of the letters have not been guessed, the computer will return "False", otherwise it will be true. This is much simpler than the first choice because it is difficult to compare whether all of the letters in the secret word are in letters guessed, but much easier to determine if the word has not been guessed because only one letter has to not be guessed for it to return "False". Therefore, our code should look something like this:

```
def is_word_guessed(secret_word, letters_guessed):
    length_sw = len(secret_word)
    for i in range(length_sw):
        if not is_letter_guessed(secret_word[i], letters_guessed):
            return False
    return True
```

2. get_guessed_word

The next step is to get the result the computer shows when the user guesses a letter. If the letter that is guessed is in the word, the computer will print out the word with underscores to represent letters not guessed and the letter that has been guessed correctly in the correct place. If the letter is not guessed correctly, the computer will print out what it has so far with the underscore.

we will call this function get_guessed_word. The two parameters that it is taking in are the secret word we are trying to guess and also the letters guessed so far.

This is similar to the last helper function in the sense that we must also determine the number of letters we are comparing using the length of secret word. We first set the length of secret word, or length_sw = len(secret_word). Now we can write a for loop to determine what to do if the letter is guessed and what to do when the letter is not.

If the letter is guessed, then the result will be updated to show the correct letter in the word. If it is not, the letter will just be an underscore and a space.

Because we are updating the result, we must first define the result beforehand so the computer knows what we are updating. Since in the beginning, before the user has input their guess, our result is empty, we can define result as two quotes (either " or '') The program should look as follows:

```
def get_guessed_word(secret_word, letters_guessed):
    result = ''
    Length_sw = len(secret_word)
    for i in range(Length_sw):
        if is_letter_guessed(secret_word[i], letters_guessed) :
            result = result + secret_word[i]
        else:
            result = result + '_ '
    return(result)
```

get_available_letters

In the last helper function, we will be writing code that will show what the available letters that the user can guess from. This function should take in one parameter (letters_guessed) and return a list of English lowercase letters - all letters that have not been guessed (not in letters_guessed).

```
>>> letters_guessed = ['e', 'i', 'k', 'p', 'r', 's']
>>> print get_available_letters(letters_guessed)
abcdfghjlmnoqtuvwxyz
```

First you must import the string of letters, or the string of all letters in the English alphabet.

```
>>> import string
>>> print(string.ascii_lowercase)
abcdefghijklmnopqrstuvwxyz
```

Next, because we are updating the available letters the user can guess from in this function, we must define what the available letters are before we run a loop. Similar to defining "results" in Helper Function 2, we can set available_letters = "", which shows that in the beginning it is an empty string.

```
def get_available_letters(letters_guessed):
    import string
    available_letter = ""
```

Now we can write our loop that defines our string of all available letters. For each letter of our alphabet, this loop should compare it to the letters guessed to determine whether or not it has already been guessed. If it has been guessed, the computer should not add it to the available letters list. If the letter has not been guessed, the computer should add it to the available letters.

Because we are starting off with a blank string, if the letter has already been guessed, the string of available letters stays the same. Therefore, we only have to account for when the letter has not been guessed.

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

Don't forget to return the available letters at the end of the code!

Now that we have all of our helper functions written, we can implement them into our main function, test it out, and update it as needed.

3.3 The Game

In 3.1, we have already wrote a basic structure of our code and in 3.2, we wrote the helper functions that were used in the code. Now we should begin to modify the code and check if what we wrote in 3.1 is accurate.

User computer interaction

The first step we are going to compare our code to is "User computer interaction" in the instructions. If we run our code and compare it to the instructions we will notice that there are some minor differences. Because this can be your own code, you can modify it to your liking. However, it is also important to learn how to debug your code, or fix it to what it is supposed to be.

Their code:

```
You have 6 guesses left.
Available letters: abcdefghijklmnopqrstuvwxyz
Please guess a letter: a
Good guess: _ a_ _
------------
You have 6 guesses left.
Available letters: bcdefghijklmnopqrstuvwxyz
Please guess a letter: b
Oops! That letter is not in my word: _ a_ _
```

Our code:

```
You have 6 guesses left.
Available letters : abcdefghijklmnopqrstuvwxyz
Please guess a letter:a
Good guess
a_ _ _ _
You have 6 guesses left.
Available letters : bcdefghijklmnopqrstuvwxyz
Please guess a letter:d
Oops! That letter is not in my word.
```

Right now, the computer prints out the function of get_guessed_word after the while loop and so the result is on a separate line that the "good guess" or "Oops! That letter is not in my word." If we want the get_guessed_word to be right after those phrases, we need to change the position of get_guessed_word.

The demo code also shows a line of dashes in between each guess to separate the code and make it more readable. We can do the same by adding a line of dashes at the end of the while loop.

```
96      letters_guessed=""
97      won = False
98      remaining_guesses = 6
99      while (remaining_guesses > 0):
100         print("You have " + str(remaining_guesses) + " guesses left.")
101         print("Available letters :", get_available_letters(letters_guessed))
102         current_guess = input("Please guess a letter:")
103         if is_letters_guessed(current_guess, secret_word):
104             print("Good guess:", get_guessed_word(secret_word, letters_guessed))
105         else:
106             print("Oops! That letter is not in my word:", get_guessed_word(secret_word, letters_guessed))
107         print("-----------")
108         letters_guessed = letters_guessed + current_guess
109         if is_word_guessed(secret_word, letters_guessed):
110             won = True
111             print("word guessed!")
112             break
113     if not won:
114         print("YOU LOST!!!")
```

Now if we run our code, we see that it now prints out the dashes after the phrases. However, when we guess a letter that is in the word, the letter does not show up after "good guess" like it is supposed to and it only shows up after our next guess. This is because we updates the letters guessed after we print out the get_guessed_word. we can modify this by moving line 108 in the code above to a line before we use it.

```
letters_guessed=""
won = False
remaining_guesses = 6
while (remaining_guesses > 0):
    print("You have " + str(remaining_guesses) + " guesses left.")
    print("Available letters :", get_available_letters(letters_guessed))
    current_guess = input("Please guess a letter:")
    letters_guessed = letters_guessed + current_guess
    if is_letters_guessed(current_guess, secret_word):
        print("Good guess:", get_guessed_word(secret_word, letters_guessed))
    else:
        print("Oops! That letter is not in my word:", get_guessed_word(secret_word, letters_guessed))
    print("-------------------")
    if is_word_guessed(secret_word, letters_guessed):
        won = True
        print("word guessed!")
        break
if not won:
    print("YOU LOST!!!")
```

User input requirements

1) guess is not a letter

Before, we have always assumed that the guess the user input is a letter. However, this is not guaranteed. Sometimes there could be misclicks and we have to account for those.

The user can only guess an uppercase or lowercase letter. If the user inputs anything else, tell the user that they can only input an alphabet. In addition to this, the user should get 3 warnings at the beginning of the game. Each time they put in an invalid input, they should lose a warning. If there are no warnings left and they input another invalid input, the user should lose a guess.

In your code, add in the remaining warnings and print out what it is.

```
letters_guessed=""
won = False
remaining_warnings = 3
remaining_guesses = 6
print("You have " + str(remaining_warnings) + " warnings left.")
while (remaining_guesses > 0):
```

We will write another helper function that determines what the computer should do if the user input is not a letter. It is best to define functions and implement them in the main function because it is easier to read and simpler. I will name this code not_alpha_warning. The two parameters this function should take in are the remaining_guesses and the remaining_warnings because those are the ones we will use and update.

In this code, there are two possibilities we must account for. First is if there are warnings left and the other is when there are no more warnings left. If there aren't any warnings left, we deduct a guess and print out the statement "Oops, that is not a letter. You have no warnings left," followed by a statement telling how many guesses the user has left. If there are still warnings left, we just subtract a warning and print out "Oops, that is not a letter," followed by a statement saying how many warnings there are left.

```
def not_alpha_warning(remaining_warnings, remaining_guesses):
    if remaining_warnings == 0:
        remaining_guesses = remaining_guesses -1
        print("Oops! That is not a valid letter. You have no warnings left so you lose one guess: ", get_guessed_word(secret_word, letters_guessed))
    else:
        remaining_warnings = remaining_warnings - 1
        print("Oops! That is not a valid letter. You have " + str(remaining_warnings) + " warnings left: ", get_guessed_word(secret_word, letters_guessed))
    return remaining_warnings, remaining_guesses
```

Because we used get_guessed_word and it includes the parameters secret_word and letters_guessed, we must also add them to the parameters of the function not_alpha_warning.

```
def not_alpha_warning(remaining_warnings, remaining_guesses, secret_word, letters_guessed):
    if remaining_warnings == 0:
        remaining_guesses = remaining_guesses -1
        print("Oops! That is not a valid letter. You have no warnings left so you lose one guess: ", get_guessed_word(secret_word, letters_guessed))
    else:
        remaining_warnings = remaining_warnings - 1
        print("Oops! That is not a valid letter. You have " + str(remaining_warnings) + " warnings left: ", get_guessed_word(secret_word, letters_guessed))
    return remaining_warnings, remaining_guesses
```

Now we can implement this into the main function.

```
letters_guessed=""
won = False
remaining_warnings = 3
remaining_guesses = 6
print("You have " + str(remaining_warnings) + " warnings left.")
while (remaining_guesses > 0):
    print("You have " + str(remaining_guesses) + " guesses left.")
    print("Available letters :", get_available_letters(letters_guessed))
    current_guess = input("Please guess a letter:")
    if not str.isalpha(current_guess):
        [remaining_warnings, remaining_guesses] = not_alpha_warning(remaining_warnings, remaining_guesses, secret_word, letters_guessed)
    else:
        if is_letters_guessed(current_guess, letters_guessed):
```

2) repeated guess warning

This is similar to the previous helper function because in both, each time the user input in an invalid input, they lose a warning. If there are no warnings left and they input another invalid input, the user loses a guess.

we can basically copy the code from the previous function into this one because the contents are the same, just what we type out and when we use it is different.

In this code, it should print out "oops, that letter has already been guessed." Along with the remaining warnings/guesses left.

```
def repeated_guess_warning(remaining_warnings, remaining_guesses, secret_word, letters_guessed):
    if remaining_warnings == 0:
        remaining_guesses = remaining_guesses -1
        print("Oops! You've already guessed that letter. You have no warnings left so you lose one guess: ", get_guessed_word(secret_word, letters_guessed))
    else:
        remaining_warnings = remaining_warnings - 1
        print("Oops! You've already guessed that letter. You have " + str(remaining_warnings) + " warnings left: ", get_guessed_word(secret_word, letters_guessed))
    return remaining_warnings, remaining_guesses
```

Now we update the main function so we can implement this code. we are updating the remaining_warnings and remaining_guesses, so we have to do this after implementing the function.

```
121     letters_guessed=""
122     won = False
123     remaining_warnings = 3
124     remaining_guesses = 6
125     print("You have " + str(remaining_warnings) + " warnings left.")
126     while (remaining_guesses > 0):
127         print("You have " + str(remaining_guesses) + " guesses left.")
128         print("Available letters :", get_available_letters(letters_guessed))
129         current_guess = input("Please guess a letter:")
130         if not str.isalpha(current_guess):
131             [remaining_warnings, remaining_guesses] = not_alpha_warning(remaining_warnings, remaining_guesses, secret_word, letters_guessed)
132         else:
133             if is_letters_guessed(current_guess, letters_guessed):
134                 [remaining_warnings, remaining_guesses] = repeated_guess_warning(remaining_warnings, remaining_guesses, secret_word, letters_guessed)
135             else:
136                 letters_guessed = letters_guessed + current_guess
137                 if is_letters_guessed(current_guess, secret_word):
138                     print("Good guess:", get_guessed_word(secret_word, letters_guessed))
139                 else:
140                     print("Oops! That letter is not in my word:", get_guessed_word(secret_word, letters_guessed))
141         print("------------------")
142         if is_word_guessed(secret_word, letters_guessed):
143             won = True
144             print("word guessed!")
145             break
146     if not won:
147         print("YOU LOST!!!")
```

Game rules

There are a few more rules to our game in addition to the ones we already have. Previously, we have just discussed what the computer should do when the input is not a letter or is a letter that has already been guessed. Now we are going to determine what to do when the letter guessed is a letter that hasn't been guessed, but is not in the secret word.

As stated in the instructions, if the incorrect guess is a vowel, the user loses 2 guesses and if it is a consonant, the user loses 1 guess. Now we just have to write a separate function that does this. I will name the function wrong_letter_guessed.

In this function, we have to compare the current guessed to the secret word and determine if it is already in the letters guessed. Then, we update the remaining guesses per the instructions. Because of this, the parameters of this function are current_guess, remaining_guesses, secret_word, and letters_guessed.

we can use an "if, else" loop to do this. Because it is easier to define what a vowel is, we can say "if the input is a vowel" then subtract 2 guesses, and otherwise, subtract one guess. The instructions also say to print out "Oops! That letter is not in my word," so that is what we will also do.

```
92
93 def wrong_letter_guessed(current_guess, remaining_guesses, secret_word, letters_guessed):
94         vowel = "aeiou"
95         if is_letters_guessed(current_guess, vowel):
96             remaining_guesses = remaining_guesses -2
97             print("That letter is not in my word: ", get_guessed_word(secret_word, letters_guessed))
98         else:
```

If the letter is not a vowel, then that means it is a consonant, so it is easy to use an "if, else" loop in this situation.

```
92
93 def wrong_letter_guessed(current_guess, remaining_guesses, secret_word, letters_guessed):
94         vowel = "aeiou"
95         if is_letters_guessed(current_guess, vowel):
96             remaining_guesses = remaining_guesses -2
97             print("That letter is not in my word: ", get_guessed_word(secret_word, letters_guessed))
98         else:
99             remaining_guesses = remaining_guesses -1
100            print("That letter is not in my word: ", get_guessed_word(secret_word, letters_guessed))
101        return remaining_guesses
```

Make sure to return the remaining_guesses at the end because that is what we are updating when running this function. Now that we have this, we can implement it into our main function.

If the guess is a letter and it is not a repeated guess, than this function should occur. Because of this, we add this as an "else" condition, the "if" condition being the repeated guess. In other words, if the guess implemented is a wrong letter that has not been guessed, then we first update the letters_guessed to hold this letter and then implement wrong_letter_guessed to determine how many guesses to take away from the user.

```
letters_guessed=""
won = False
remaining_warnings = 3
remaining_guesses = 6
print("You have " + str(remaining_warnings) + " warnings left.")
while (remaining_guesses > 0):
    print("You have " + str(remaining_guesses) + " guesses left.")
    print("Available letters :", get_available_letters(letters_guessed))
    current_guess = input("Please guess a letter:")
    if not str.isalpha(current_guess):
        [remaining_warnings, remaining_guesses] = not_alpha_warning(remaining_warnings, remaining_guesses, secret_word, letters_guessed)
    else:
        if is_letters_guessed(current_guess, letters_guessed):
            [remaining_warnings, remaining_guesses] = repeated_guess_warning(remaining_warnings, remaining_guesses, secret_word, letters_guessed)
        else:
            letters_guessed = letters_guessed + current_guess
            if is_letters_guessed(current_guess, secret_word):
                print("Good guess:", get_guessed_word(secret_word, letters_guessed))
            else:
                remaining_guesses = wrong_letter_guessed(current_guess, remaining_guesses, secret_word, letters_guessed)
    print("------------")
    if is_word_guessed(secret_word, letters_guessed):
        won = True
        print("Congratulations, you won!")
```

Game Termination/ Total Score

when the game ends - either the user has run out of guesses or the word has been guessed - we must write out a congratulatory message to the user if the word is guessed and tell them the total score or tell them they have lost and the secret word if they did not guess the word. The instructions include the following example implementation. we will be writing code that mimics these results.

Example Implementation:

```
You have 3 guesses left.
Available letters: bcdfghijklnopquvwxyz
Please guess a letter: c
Good guess: tact
------------
Congratulations, you won!
Your total score for this game is: 9
```

I am going to write a separate code that calculates the total score by defining total_score with the parameters of remaining_guesses and secret_word. This is because the instructions tell us that the formula to calculate the secret word is the remaining_guesses multiplied by the number of unique letters in secret_word.

We know that we can use remaining_guesses for the first portion, but the number of unique letters in secret_word is something we do not have yet. Because we know that in a set, all of the proponents are unique characters, we can utilize this to find the number of unique letters by putting the secret_word into a set and finding the length of it.

```
104 def total_score(remaining_guesses, secret_word):
105         total_score = remaining_guesses * (len(set(secret_word)))
106         return total_score
107
```

Now that we have the formula to find the total score, we can implement this into the main function.

```
        if is_word_guessed(secret_word, letters_guessed):
            won = True
            print("Congratulations, you won!")
            print("Your total score for this game is: ", total_score(remaining_guesses, secret_word))
            break
    if not won:
        print("Sorry, you ran out of guesses. The word was", secret_word + ".")
```

Now that we have all of the portions of the main function, we should test out our code and check to see that all of the portions work. Remember, you can always define your secret_word as a word to test out the function so you know that the code works!

```
letters_guessed=""
won = False
remaining_warnings = 3
remaining_guesses = 6
print("You have " + str(remaining_warnings) + " warnings left.")
while (remaining_guesses > 0):
    print("You have " + str(remaining_guesses) + " guesses left.")
    print("Available letters :", get_available_letters(letters_guessed))
    current_guess = input("Please guess a letter:")
    if not str.isalpha(current_guess):
        [remaining_warnings, remaining_guesses] = not_alpha_warning(remaining_warnings, remaining_guesses, secret_word, letters_guessed)
    else:
        if is_letters_guessed(current_guess, letters_guessed):
            [remaining_warnings, remaining_guesses] = repeated_guess_warning(remaining_warnings, remaining_guesses, secret_word, letters_guessed)
        else:
            letters_guessed = letters_guessed + current_guess
            if is_letters_guessed(current_guess, secret_word):
                print("Good guess:", get_guessed_word(secret_word, letters_guessed))
            else:
                remaining_guesses = wrong_letter_guessed(current_guess, remaining_guesses, secret_word, letters_guessed)
    print("------------------")
    if is_word_guessed(secret_word, letters_guessed):
        won = True
        print("Congratulations, you won!")
        print("Your total score for this game is: ", total_score(remaining_guesses, secret_word))
        break
if not won:
    print("Sorry, you ran out of guesses. The word was", secret_word + ".")
```

3.4 Hangman With Hints

If you have played the computer game Hangman before, you'll realize that it's not so easy, especially if it is a word you do not know. Then, you would ask the computer for a hint, such as a list of words that match what you have currently guessed.

For example, if the word is "tent" and you have already guessed "t", the solution is "t_ _ t" and if you asked for a guess that gives you a list, you would get words such as "tact, tent, test, text..."

We are going to written a variation of hangman called "Hangman with Hints" that will find all the words from its loaded list that might match your current guessed word and print them out. This will happen if you guess the special character "*".

There will be two helper functions to help us achieve this.

1) match_with_gaps

This function should take in 2 parameters: my_word and other_word.

my_word is an instance of a guessed word and may have _'s in places. other_word is a normal English word.

This function should return "True" if the guessed letters of my_word match with the corresponding letters of other_word. It should return "False" if the two words do not correspond in length or if a guessed letter in my_word does not match the corresponding letter in other_word.

Remember that when a letter is guessed, your code updates "_" to a hidden letter that you have guessed correctly and this cannot be one of the letters in the word that has already been revealed.

```
>>> match_with_gaps("te_ t", "tact")
False
>>> match_with_gaps("a_ _ le", "banana")
False
>>> match_with_gaps("a_ _ le", "apple")
True
>>> match_with_gaps("a_ ple", "apple")
False
```

The first thing we need to do to make this happen is to match the length of our secret word to the length of the word we have already guessed.

However, we must remember that the word we have guessed currently consists of underscores and spaces " _ " to represent the letters we have not guessed. The spaces are here to help us tell apart the different letters that have not been guessed. Refer back to the beginning when we first wrote this code.

This was convenient for us to tell apart letters missing, but now that we are trying to compare the word that is being guessed to the possible secret words, we must first compare whether or not these two words are the same length. The spaces in our word that is being guessed adds extra characters (spaces in this case) to our word and does not give us an accurate representation of how many characters our word has. To overcome this, we will first write a code that removes spaces.

1a) remove_space

The first sub-function we will define is remove_space. The goal of this function is to get rid of the spaces in our current word that we are guessed, or my_word. Because the only thing we are modifying is my_word, our only parameter to this function is my_word.

One way we can think about this function is to write a function that only takes in the characters of my_word that are not a space. This can be done by comparing every character of our current my_word to a space and if it is not a space, we update it into our my_word.

To do this, we can first set up a new variable my_word_ns, standing for my word no space. we can first let this equal an empty string or "". By doing this, we can now update the new my_word into my_word_ns. So for every character in my_word, if the character is not a space, then we update my_word_ns so it contains the character. This will continue until all of the characters have gone through this cycle and in the end we will return my_word_ns, which is our new word without the spaces.

Our code should look like this

```
def remove_space(my_word):
    my_word_ns = ""
    for i in my_word:
        if i != " ":
            my_word_ns = my_word_ns + i
    return my_word_ns
```

Now that we have the updated my_word, we can write the function match_with_gaps. This function still has to consider whether or not the letters that are already in our my_word match up with those in the potential secret_word.

To do this, our first step is updating my_word so it removes the spaces. Next, we will compare the lengths of my_word (the one without spaces) and the length of other_word. Currently, my_word and other_word are both strings and we cannot directly compare their lengths. To change this, we must compare the lengths of my_word and other_word.

```
def match_with_gaps(my_word, other_word):
    my_word = remove_space(my_word)
    length_my_word = len(my_word)
    length_other_word = len(other_word)
```

Only if the lengths of these two words are the same will our code keep going. If the lengths of these two words are not the same, then there is no way they will be the sam word, hence it is false.

But if the lengths are the same, our code will go on further to compare the letters of the characters in my_word that haven't been guessed. In other words, for all the characters in my_word (now that spaces are not in the equation), if the character is not an underscore "_", we will compare that letter to the corresponding place value letter of other_word.

It is always easier to compare using what it isn't. So instead of seeing whether the corresponding letters are the same, we can simply return False when the corresponding letters are not the same.

Our finished code should look like this.

```
def match_with_gaps(my_word, other_word):
    my_word = remove_space(my_word)
    length_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] != "_":
                if my_word[i] != other_word[i]:
                    return False
        return True
    else:
        return False
```

2) show_possible_matches

In hangman with hints, the goal is to allow the user to ask for a hint - the hint being a list of all words that could potentially be the secret word. This is done through match_with_gaps where we determined whether the other word could potentially be the secret word by comparing the lengths of the other word with the current word being guessed and the correct letters already guessed in my_word.

Now that we have this, the next step is to show all of the possible words. This function should print out all words in our wordlist (we have to define what the word list is in the beginning of this function - it's the words we load into the game) that match my_word.

show_possible_matches should have one parameter, my_word, because we will be comparing my_word with the possible secret word through match_with_gaps and then printing out the possible words.

The first step is to define what the wordlist that we will be getting the possible words from is. In this case, wordlist = load_words(), which is defined in the very beginning of our game.

```
def show_possible_matches(my_word):
    wordlist = load_words()
```

Now that we have the wordlist, for every possible secret word, or possible_word, in the wordlist, we will print out the possible_word if it passes match_with_gaps.

In other words, if we run match_with_gaps comparing my_word and the possible_word and it goes through, then we will print out the possible word. Once all of the possible_word is wordlist have been through this loop, we will have all of the possible matches printed out.

```
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)
```

Now to implement this into our code, we will make a small addition that will allow the instance that the user can guess an asterisk (*), and the computer will print out all of the words that match that guess. The user should not lose a guess if the guess is an asterisk.

```
if current_guess == "*":
    show_possible_matches(get_guessed_word(secret_word, letters_guessed))
```

Now that we have the hangman with hints code, we must comment out the lines of code that we used to play the original Hangman game and uncomment out these new lines of code at the bottom of the file to play our new game Hangman with Hints.

```
200
201     # To test part 2, comment out the pass line above and
202     # uncomment the following two lines.
203
204     #secret_word = choose_word(wordlist)
205     #hangman(secret_word)
206
207 ###############
208
209     # To test part 3 re-comment out the above lines and
210     # uncomment the following two lines.
211
212     secret_word = choose_word(wordlist)
213     hangman_with_hints(secret_word)
214
```

Congratulations! we have just completed our first game Hangman with Hints! To check your results, we can compare them to the sample output on the problem set 2. Try to test all aspects of your code and make sure the different possibilities work. Remember you can always define what the secret word is so it is easier to test. You can also modify your code so the user has more guesses, warnings, etc. This is your code and now you can modify it however you'd like!

Below is an example of what a possible output could look like.



Chapter 4

4.1 Getting Started

This game is a lot like scrabble. Letters are dealt to player, 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. The rules of the game are as follows:

Dealing:

- a player is dealt a hand of hand_size letters of the alphabet, chosen at random. This may include multiple instances of a particular letter.
- the player arranges the hand into as many words as they want out of the letters, but using each letter at most once.
- some letters may remain unused, though the size of the hand when a word is played does affect its score.

Scoring:

-the score of the hand is the sum of the score for each word formed.

- the score for a word is the product of two components:

  first component: the sum of the points for letters in the word.

  second component: either [7*word_length - 3*(n-word_length)] or 1, whichever value is greater, where word_length is the number of letters used in the word and n is the number of letters available in the current hand. .

Letters are scored as in Scrabble, A is worth 1, B is worth 3, C is worth 3, D is worth 2, E is worth 1, and so on. This is already defined in the dictionary scrabble_letters_values that maps each lowercase letter to its Scrabble letter value.

Before we begin coding, we will go to MITOPENCOURSEWARE Introduction To Computer Science And Programming In Python Problem Set 3 to download the items needed. This is the same place where we downloaded the files needed for the Hangman Problem in chapter 2. You can find these links on the MITOPENCOURSEWARE website or on my website www.herstemjourney.com.

We will download and save ps3.zip, which includes the python file ps3.py, which should contain all of your code, initial procedures, and templates for new procedures and also a file for testing your code test_ps3.py. The last file in the list of legitimate words words.txt. You do not have to change anything here unless specified.

Make sure to save the words.txt in the same directory as ps3.py and then run ps3.py to make sure it runs smoothly.

Because we have not written the code for our program yet, we should see the following printed out:

```
Loading word list from file...
        83667 words loaded.
        play_game not yet implemented.
```

we also have a test code file and it is there so we can test our code along the way instead of waiting to finish the code before checking it. If our code passes the unit tests, there will be a "SUCCESS" message. Otherwise, there will be a "FAILURE" message indicating we need to fix our code.

Now that we understand a little bit about this problem, we can start writing the code!

4.2 Problem 1 - Word Scores

The first step is to implement a function that calculates the score for a single word. This is going to go under the code get_word_score. The instructions give us these rules that we must implement into our code:

As a reminder, here are the rules for scoring a word:

- The score for a word is the **product** of two components:
  - First component: the sum of the points for letters in the word.
  - Second component: either [7 * *word_length* - 3 * (*n*-*word_length*)] or 1, whichever value is greater, where:
    - *word_length* is the number of letters used in the word
    - *n* is the number of letters available in the current hand

It also tells us to use the scrabble_letter_values dictionary defined at the top of ps3.py and to not assume that there are always 7 letters in a hand. The parameter n is the total number of letters in the hand when the word was entered.

1) get_word_score

Because there are two components of the scoring for a word, we can write two helper codes to make each code simpler.

1a) c1_score

The first part of the score is the sum of the points for letters in the word. To make this happen, we have to use the SCRABBLE_LETTER_VALUES defined in the beginning of the code. This is a dictionary, so we will have to get the value of the letter and then do the same for each letter in the word.

we will be using the ".get()" method to access the value associated with the key (in this case the letter) in a dictionary. This allows us to retrieve the value of a given key and provides a default value if the key is not present in the dictionary. The general syntax is:

```
value = dictionary.get(key, default)
```

In our code, we will be updating the sum of our score, so we must first let our initial sum = 0. Now we can update our sum as we get the value of each letter from our dictionary.

Next, we will go through all of the letters in the word and retrieve its value using the ".get()" method.

```
def c1_score(word):
    sum = 0
    for letter in word:
        SCRABBLE_LETTER_VALUES.get(letter,0)
```

Now that we have the value, we will update the sum so that it contains the value of the letter as well. After each letter has gone through the "for" loop, we will return the sum and that will be the value of our first component

```
def c1_score(word):
    sum = 0
    for letter in word:
        sum = sum + SCRABBLE_LETTER_VALUES.get(letter,0)
    return sum
```

1b) c2_score

The score of the second component is either the word_length * 7 - (n - word_length) * 3 or 1, whichever one is greater, where word_length is the number of letters used in the word and n is the umber of letters available in the current hand.

In this case, we will be writing the code for the instance that the value is greater than one. This is basically a python interpretation of the expression given. We know that the two parameters of this function will be "word" and "n" because that is what we will be using in the function. To find the length of the word, we need to use the "word" and the "len()" function.

```
def c2_score(word, n):
    word_length = len(word)
```

This will give us the length of the word. Now that we have the length of the word, we can write the expression for the score.

```
def c2_score(word, n):
    word_length = len(word)
    score = (7*word_length) - (3*(n - word_length))
    return score
```

Once we find the score, we can return the score and running this code will give us the score of the second component.

1c) Putting it together

Now that we have each section of the total score, the instructions say that the total score is the product of the two components. Going back to get_word_score, we know that the parameters of this function are "word" and "n" because those are the inputs. Notice that we are using the SCRABBLE_LETTER_VALUES defined at the top of our file. This only contains lowercase letters, meaning if the word contains an uppercase letter, our dictionary will not have a value for it. To get around this, we can first convert the word into lowercase letters and then retrieve the value from the dictionary.

```
def get_word_score(word, n):
    word = word.lower()
```

Now that the word is lowercase, we can calculate the total score. This is where we will be using the c1_score and c2_score that we have defined previously. Before we compare these, we can set them to a value so it is easier to compare.

```
def get_word_score(word, n):
    word = word.lower()
    score1 = c1_score(word)
    score2 = c2_score(word, n)
```

This is so when we are using the first and second components of the score, we can use score1 and score2 instead of c1_score(word) and c2_score(word, n). This is much simpler, especially in larger codes that refer to helper codes multiple times.

Recall that we only use score2 if its value is greater than 1. To determine this, we can use an "if" loop.

```
def get_word_score(word, n):
    word = word.lower()
    score1 = c1_score(word)
    score2 = c2_score(word, n)
    if score2 >= 1:
        total_score = score1*score2
```

If it is greater than 1, than the total score is the first component times the second. If it is not, then the total score is the first component times 1, which is just the first component. We can write the second part as the "else" statement.

```
def get_word_score(word, n):
        word = word.lower()
        score1 = c1_score(word)
        score2 = c2_score(word, n)
        if score2 >= 1:
                total_score = score1*score2
        else:
                total_score = score1
        return total_score
```

After we retrieve the total score, we will return it and then our code is written! To determine whether or not we have written the code correctly, we can run it through the test_ps3.py file.

we can do this by first opening our terminal and changing our directory into the one that contains the PS3 folder. Once we have entered the folder, we can run the test file and look at the results. The first portion of our code should look similar to what I have below. Don't worry about the Failure messages below, as those are the codes that have not been written yet.

```
Last login: Mon Feb 19 14:44:06 on [redacted]
(base) [redacted]-MacBook-Pro ~ % cd cspractice
(base) [redacted]-MacBook-Pro cspractice % cd PS3
(base) [redacted]-MacBook-Pro PS3 % python3 test_ps3.py
Loading word list from file...
   83667 words loaded.
----------------------------------------------------------------------
Testing get_word_score...
SUCCESS: test_get_word_score()
----------------------------------------------------------------------
```

4.3 Problem 2 - Dealing with Hands

A hand is the set of letters held by a player during the game. The player starts off with a random set of letters, which is the hand. In our program, a hand will be represented as a dictionary: the keys are lowercase letters and the values are the number of times the particular letter is repeated in that hand. For example, if we were dealt with a hand consisting of: a, q, l, m, u, i, l, our dictionary would be represented as:

```
hand = {'a':1, 'q':1, 'l':2, 'm':1, 'u':1, 'i':1}
```

The repeated letter "l" is represented with the value of 2 because it occurs twice in this hand. With a dictionary representation, the usual way to access a value is hand['a'], where 'a' is the key we want to find. However, this only works if the key is in the dictionary. To avoid this, we can instead use hand.get("a", 0). This is the safe way to access a value if we are not sure the key is in the dictionary.

At the top of the ps3.py, there is already a function defined called get_frequency_dict, which returns a dictionary where the keys are elements of the sequence and the values are integer counts, for the number of times that an element is repeated in the sequence.

GIVEN FUNCTIONS:

1) display hand

when we have a hand represented as a dictionary, we want to display it to the user in a way that they can read it. The display_hand function has been defined in ps3.py and it prints out the hand for us.

2) generate a random hand

we have the code to display the hand to the user, but we need code that generates a set of random letters. The parameter for this function is "n" because it takes in a positive integer "n" and returns a new dictionary representing a hand of "n" lowercase letters. This code has also been given to us under display_hand so we can implement it as needed.

WRITE FUNCTIONS:

1) update hand

The player starts with a full hand of "n" letters. When the player spells out words, they use letters and the letters from the set are used up. For example, if the player started with "h, e, n, a, k, d," and the player chose the word "hand", the hand will be left with "e, k".

We would need to write a function that takes a hand and a word as inputs, so our parameters are "hand" and "word". The function would use the letters from the hand to spell out word and return a new hand that only contains the remaining letters.

Note that when we write our code, it does not assume that the hand contains every letter in our word at least as many times as the letter appears in the word. If the letters in the word don't appear in hand, they should be ignored. If the user uses a letter more times than it is in the hand, the result should not be negative. Instead, the count for the letter in the hand should be set to 0.

Because our dictionary only contains lower case letters, we must ensure that the word the user inputs is converted into lowercase letters before implemented. To do this, we set word = word.lower().

Before we update the hand, we want to make a copy of the original hand so we do not alter the original hand. This is because we will be using the original hand later on. Using new_hand = hand.copy(), we now have a copy of the hand that we will update.

To update the hand, we will be subtracting the letter count by one if the letter has been used and keep it the same otherwise. we will do this by using the for loop.

```
for letter in word:
    if letter in new_hand:
```

If the letter is in the hand, then we have to determine whether or not the number of times the letter has appeared in the hand is greater than or equal to one. If it is, then we will subtract a count. Remember the count should never be negative. This can be done with the ".get()" function.

our code should look like this.

```
for letter in word:
        if letter in new_hand:
                if new_hand.get(letter,0) >= 1:
                        new_hand[letter] = new_hand.get(letter,0) - 1
return new_hand
```

4.4 Problem 3 - Valid Words

This is the third problem of this game. We still haven't written any code to verify that a word given by a player obeys the rules of the game. A valid word is in the word list and it is composed entirely of letters from the current hand. We will be implementing the is_valid_word function and test it.

In the function is_valid_word, it should return True if the word is in the word_list and it is entirely composed of letters in the hand. Otherwise, it will return False. To do this, we have to first determine if the word is in the word list and then determine if the word is in the hand.

is_valid_word

This function will take in the parameters word, hand and word_list because those are what we will be using in this function. Our word list generates all the words in lowercase letters. To compare whether or not our word is in this word list, we must first ensure our word is written in all lowercase. We can do this by setting word = word.lower().

Just like what we have done before, we must make a copy of our hand before we alter its information. This is done by setting new_hand = hand.copy(). Now we can compare our word to the words in the word list. If the word is in the word list, then we will determine whether the word is in the hand. This is done by comparing every letter of the word to the letters in the hand. It will be easier if we write a separate code for this. If the word is not in the word list then we will return False.

Our code should look like this so far.

```
def is_valid_word(word, hand, word_list):
    word = word.lower()
    new_hand = hand.copy()
    if word in word_list:
        return is_word_in_hand(word, new_hand)
    else:
        return False
```

Notice that we have not written the function is_word_in_hand. That is what we will be doing next.

Is_word_in_hand will help us determine whether or not all of the letters in the word are in the hand.

The parameters of this function are the word, and also the new_hand. Recall that in is_valid_word, we made a copy of the hand. Since this function is going to be run within is_valid_word, we can use that as our parameter.

In this function, we must first determine whether or not every letter is in the hand. If the letter is not in the hand, we simply return False. If the letter is in the hand, then we update the value of the letter (how many times it appears in the hand) by subtracting it by one, similar to how we did it in update_hand.

The first loop we write is the "for" loop: for every letter in the word... The next condition is the "if... else..." condition . If the letter is not in the hand, then we return False.

```
def is_word_in_hand(word, new_hand):
    for letter in word:
        if letter not in new_hand:
            return False
        else:
```

If the letter is in the word, then we update the value of the letter. This is done by subtracting one to the value if the value is greater than or equal to one and returning False if the count is less than one.

```
else:
        if new_hand.get(letter,0) >= 1:
                new_hand[letter] = new_hand.get(letter,0) - 1
        else:
                return False
```

our code for is_word_in_hand should look like the following when we are done with Problem 3:

```
def is_word_in_hand(word, new_hand):
        for letter in word:
                if letter not in new_hand:
                        return False
                else:
                        if new_hand.get(letter,0) >= 1:
                                new_hand[letter] = new_hand.get(letter,0) - 1
                        else:
                                return False
        return True
```

4.5 Problem 4 - Wildcards

During this game, we want to allow hands to contain wildcard letters that players can use to replace a vowel. This is so the player can more flexibly create a word of their choice. A wildcard letter will be denoted by an asterisk (*) and can ONLY replace vowels. The player won't receive any points for using the wildcard though it does count as a used or unused letter when scoring. This is the drawback of using wildcards, but in a larger scope, it is beneficial to the player.

When playing the game, a player who wants to use a wildcard should enter the asterisk instead of the intended letter. The word-validation code shouldn't assume what the vowel will be, but rather verify that at least one word can be made with the wildcard as a vowel in the desired position.

To do this, we will be editing the function is_valid_word and the helper function we wrote for is_valid_word: is_word_in_hand. So far, is_valid_word lets us know if the word (that only consists of letters) is a valid word. We will update it so that an asterisk could make a word potentially valid if one of the vowels replacing the asterisk will create a possible word.

In our code, we must first determine if there is an asterisk in the word. If there is not an asterisk, then we will run the code of is_valid_word that we have already written.

If there is an asterisk, then we will replace the asterisk with each vowel and see if it is in the word list. If it is in the word list, then we will use is_word_in_hand to determine whether or not the word is in the hand. Once we have done this, we will return whatever value is_word_in_hand gives us, whether it is True (if the word is in the hand) or False (if the word is not in the hand). If the word is not in the word list, meaning the vowel replacing the asterisk will not create a valid word, then we go through the loop again to try another vowel. Once all of these vowels are tried will the loop end.

```
if "*" in word:
    for vowel_letter in VOWELS:
        new_word = word.replace("*", vowel_letter)
        if new_word in word_list:
            return is_word_in_hand(word, new_hand)
```

Our "else" statement is for if there is not an asterisk in the word.

If there is not an asterisk, then we will run the code that we wrote initially for is_valid_word. Putting them together, we get the final code for is_valid_word.

```
def is_valid_word(word, hand, word_list):
    word = word.lower()
    new_hand = hand.copy()
    if "*" in word:
        for vowel_letter in VOWELS:
            new_word = word.replace("*", vowel_letter)
            if new_word in word_list:
                return is_word_in_hand(word, new_hand)
    else:
        if word in word_list:
            return is_word_in_hand(word, new_hand)
        else:
            return False
```

Now we have to modify our helper function is_word_in_hand so it consists the asterisk condition as well. For every letter in the word, if the letter is not an asterisk, then we run our code to determine if the letter is in the hand.

```
def is_word_in_hand(word, new_hand):
    for letter in word:
        if letter !="*":
            if letter not in new_hand:
                return False
            else:
                if new_hand.get(letter,0) >= 1:
                    new_hand[letter] = new_hand.get(letter,0) - 1
                else:
                    return False
    return True
```

The last part we need to update for our wildcard code is the deal_hand function. Recall that this is one of the functions that has already been provided for us. Our instructions tell us to modify the deal_hand function to support always giving one wildcard in each hand. Note that deal_hand currently ensure that one third of the letters are vowels and the rest are consonants. Leave the consonant count the same and replace one of the vowel slots with the wildcard. You will also need to modify one or more constants defined at the top of the file to account for wildcards.

we do this by setting the hand = {"*":1}. This ensures one asterisk in each hand. Now we must subtract the number of vowels by one because the asterisk accounts for a vowel. Our final updated code should look as follows.

```
def deal_hand(n):

    hand={'*':1}
    num_vowels = int(math.ceil(n / 3))

    for i in range(num_vowels-1):
        x = random.choice(VOWELS)
        hand[x] = hand.get(x, 0) + 1

    for i in range(num_vowels, n):
        x = random.choice(CONSONANTS)
        hand[x] = hand.get(x, 0) + 1

    return hand
```

4.6 Problem 5 - Playing a hand

Now that we have the helper codes, we can write the function that will allow the user to play out a single hand. Before we do this, we must first write the helper function calculate_handlen, which will return the number of letters in the current hand.

calculate_handelen

This function will help us determine the total number of letters that are in our current hand. Remember that our hand is a dictionary with the values being how many times each letter appears. To determine the total length of our hand, we must add up all these values.

The parameter of this function is the hand because that is what we are using to determine the total length of the hand. we can determine the length by adding each letter's value into this count. When we first run the function, the hand_length is 0. Then, for every letter that is in the hand, we update the hand_length to be the sum of the current hand_length and the value of the letter. Remember to get the value of the letter, we must ensure the letter is lowercase, hence letter.lower(). Once all of the letters have gone through this loop, we will return the hand.

Our code for calculate_handlen should look as follows:

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

Now that we have our helper function ready, we can play the hand.

play_hand

The instructions give us a pseudocode, which is a detailed yet readable description of what a computer program should do. We will follow this pseudocode to write the actual code of our program.

The first thing our pseudocode says is to keep track of the total score. We can set the total_score = 0 to begin with because there is no value for the total score to begin with.

To make our code easier to read, we can set the number of letters in the hand, or calculate_handlen(hand), to n. In other words, n = calculate_handlen(hand). Now when we want to use this function, we can just call n.

The next pseudocode tells us to display the hand, so we will call the display_hand(hand) function. The "end = ' '" in the print current hand section is so that "current hand:" and the hand will be on the same row.

Remember to make a copy of the current hand so we don't alter it. We will be updating the hand when the user guesses a word to what is remaining. Then, we will let the user guess another word if they wish. Our instructions tell us that the user can finish playing the hand by inputting two exclamation points "!!" instead of a word. Otherwise, if the hand is not empty (the number of letters in the hand is greater than 0), we will run this loop.

There are two ways for our code to stop. Either the user stops it with "!!" or there are no more letters in our hand. Because of this, I will let the stop_value = 1 to begin with and when the stop_value is 0, our code will stop. This is the most simple way to program our code to end.

```
def play_hand(hand, word_list):
    n = calculate_handlen(hand)
    total_score = 0
    print("Current hand: ", end= ' ')
    display_hand(hand)
    current_hand = hand.copy()
    stop_value=1
```

The pseudocode uses the "if...else" conditions to determine what to do. we can also use the "while" loop for another way of thinking about the same problem. Both ways work, but for this game we will be using the "while" loop. If you would like to use the "if...else" conditions, you can follow the pseudocode in the instructions.

while the stop_value is not 0, then we will ask the user to input a word. To do this, we use the input function and type what we want to be printed in quotes.

```
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()
    stop_value=1
    while stop_value != 0:
        input_word = input("Enter word, or '! !' to indicate that you are finished: ")
```

The user will input something and we have to determine what to do with the input. If the input is not "!!" Then we will run the code to determine if the word is a valid word. To do this, we will call is_valid_word. Remember the parameters for this function is (word, hand, word_list). In our case, when we are calling this function, we are comparing the input_word with the current_hand and the word_list.

```
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):
```

If the word is a valid word, we will print out the score of the word. We get the total score by adding each of the scores of the words together. Because the user can guess multiple words, our total score should be updated to contain the value of the new words guessed by the user. our total_score = get_word_score(input_word, num_letters) + total_score.

Now that we have the total score, we will print it out for the player to see. Using the print function we are going to print out the sentence that says "*our input word* earned xxx points. Total: xxxxx points."

After we have this, the current hand will be updated using the update_hand function. we can now print out our current hand and the user is able to guess another word by going through this loop again. we will want to display the hand to the user and let the user know that this is our current hand. Because of this, we will call print("Current hand:", end = ' ') so that the current hand displayed will be on the same row as its label.

If the input word is not a valid word, the we will print out "That is not a valid word. Please choose another word."

```
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, end= ' ')
        else:
            print("That is not a valid word. Please choose another word.")
```

we have another "if" condition which determines if the input is "!!" meaning the user would like to be finished. If the user's input is "!!", we will simply print out the total score for the hand and set the stop_value to 0. When the stop value is 0, we will be out of this while loop and can end the game.

```
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, end= ' ')
        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
```

As we can see, we have set the stop value to 0, which means the game will end. This is one of the 2 possible ways to end the game.

The second way is if there are no more letters in the hand. our next code will determine what to do if the length of our hand becomes 0. If there are no more letters in the hand, our game will end. We will print out "Ran out of letters" and the total score. After that, we will set the stop value to 0, indicating a way out of the while loop.

Once our wild loop is finished, we will return the total score, as that is the output of this function.

Our final code should look as follows:

```
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()
    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, end= ' ')
            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 we have this, we can test our code to make sure it works. At the bottom of ps3.py, we can make sure the play_game is commented out and write a code to run play hand.

```
hand = {'a':1, 'm':1, 't':1, 'h':1}
if __name__ == '__main__':
    word_list = load_words()
    play_hand(hand, word_list)
    #play_game(word_list)
```

4.7 Problem 6 - Playing a Game

Now that we know how to play a hand, we will add this together to play a game. Playing a game consists of playing multiple hands and we will need to implement 2 more functions to complete our word game. These two functions are substitute_hand and play_game.

1) Substitute Hand

This function will allow the user to replace all copies of one letter in the hand with a new letter chosen from the alphabet at random. The new letter should be different from the user's choice and should not be any of the letters already in the hand. If the user provides a letter not in the hand, the hand should be the same. The first thing we must do is determine what letters are in the hand and also what the available letters are. This is because we cannot choose to replace a letter if it is already in the hand. Our available letters should only contain letters that were not previously in the hand. We can set two sub functions for substitute hand: letters_in_hand and get_available_letters.

1a) get_available_letters

As we have mentioned before, this helper function is to help decide which letters the user can choose from if they decide to substitute a letter. These available letters should be all of the letters that are not in the current hand. we can write this function by comparing every letter in the alphabet to the letters of the hand, if the letter is not in the hand, then we will add it into the available letters.

The parameter of this function is the letters_in_hand, or the letters that are in the hand. we can first set the available_letter = "" because there are currently no available letters when we begin. For every letter that is in the lowercase string of letters, or string.ascii_lowercase, if the letter is not in letters_in_hand (if the letter is not in the hand), then we will update available_letters to add that letter. Once we have this, we will return the available_letter string.

```
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
```

1b) letters_in_hand

Notice that we used letters_in_hand, which was supposed to give us the letters that are in the current hand. we now have to write this helper function. The parameter will be the hand because we are determining what letters are in the hand.

we must write this extra function instead of just using hand because remember that hand is a dictionary and we are comparing letters, so we must use a string. This means we must receive the "key" values of the hand dictionary, or the letters in the hand, and turn that into a string. we can do this by first letting hand_letter = "" and add in all of the keys in the hand to hand_letter.

> For every letter in hand.keys(), for every letter that is in the hand, we will update hand_letter to also contain that letter. Finally, we will return hand_letter.

```
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 there, we can write substitute_hand.

As always, we must first make a copy of our hand before altering it. With substitute hand, we will first determine the possible letters to choose from. We can do this by setting possible_letters to get_available_letters with the parameter of letters_in_hand. Now that we have this, the letter the game uses to substitute is a random letter from the list of possible letters. From this, we can get the new hand with the substitute letter by updating new hand with the substitute letter. Now that we have this we set new hand to 0 and then return the new hand.

```
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
```

Play game

In the play game function, we allow the user to play a series of hands. It first asks the user to input a total number of hands and then accumulates the scores for each hand into a total score for the entire series. Finally, it returns the total score for the series of hands.

For each hand, before playing, we ask the user if they want to substitute one legged for another. If the user inputs "yes", then ask them for their desired letter. This can only be used once during the game. Once the substitute hand option is used, the user should not be asked if they want to substitute letters in the future.

For each hand, we also ask the user if they would like to replay the hand. If the user inputs, yes, they will replay the hand and keep the better of the two scores for that hand. 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 that does not count as one of the total number of hands the user initially wanted to play.

If you replay the hand, you do not get the option to substitute a letter.

To write this function, we can see that the play game function is going to take in word_list. To ask the user for to input the total number of hands, we use the input function and label it by saying "Enter total number of hands:". Because the result of this is a whole number, we can ensure this by making the result an integer. we're also going to set this to num_hands so we can use this number in the future.

Because you can only replay the hand once, we are going to create a replay_option and set it to 1. Once this replay option becomes 0 we will no longer show the replay option.

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

Because it wants us to accumulate the score for each hand into a total score for the entire series, we will be updating the total score. To do this, we're going to let the total score first equal zero. From there, we will accumulate the total score based off the score for each hand.

For however many hands we have, we are going to deal_hand using the hand size. Then we are going to print out the hand so the user knows what letters they can use. We want to label this as "Current hand" but with the hand on the next row so it's more clear. To do this, we will print("current hand: ", end = ' '). Then we can display the hand.

```
def play_game(word_list):
    num_hands = int(input("Enter total number of hands: "))
    replay_option = 1
    total_score = 0
    for i in range(num_hands):
        hand = deal_hand(HAND_SIZE)
        print("Current hand: ", end = ' ')
        display_hand(hand)
```

Once the user has their hand, we must ask if they would like to substitute a letter. this is going to be an input and we can set the result of this to ask_substitute because we were going to use the result in the future. If the user answers "yes", then we have to ask which letter they would like to replace and this is also going to be an input. We are going to use the answer of this input later so we're going to set this result to "letter". Using "letter", we are going to update the hand with the replaced letter.

Now we can calculate the hand score. We know that if we play the hand, it gives us a hand score so we're going to play the hand with the hand and the word list as its parameters. Then, we can update the total score to be what the total score used to be plus the updated hand score.

This was if the user wanted to substitute the letter. If the user did not want to substitute a letter, then we allow the user to play the hand and we just have to calculate the hand score and total score just like we did above.

```
if ask_substitute == 'yes':
    letter = input("Which letter would you like to replace: ")
    hand = substitute_hand(hand, letter)
    hand_score = play_hand(hand, word_list)
    total_score = total_score + hand_score
else:
    hand_score = play_hand(hand, word_list)
    total_score = total_score + hand_score
```

As we mentioned before, we can only replay the hand once. The replay_option is 1 to begin with and should turn to 0 once it has been used. If the replay_option is 1 then we can ask the user whether or not they want to substitute the hand. We are also going to use the answer of this input so we're going to set this input equal to replay. If the answer is yes then we're going to allow the user to replay the hand and set the score to replay_score. This is because we are going to take the greater score from the replay score and the original score. If the replay score is greater than the original score, then we're going to update the total score to be the total score plus the replay score minus the hand score. Then we will set the replay option to 0.

Finally, we will tell the user what the total score is and print out "the total score overall hands: " and the total score.

Now we can put this all together and test out our game! Remember to change the play_hand(hand, word_list) to play_game(word_list) because we are no longer just testing the play_hand function, but rather the play_game function.

```
def play_game(word_list):
    num_hands = int(input("Enter total number of hands: "))
    replay_option = 1
    total_score = 0
    for i in range(num_hands):
        hand = deal_hand(HAND_SIZE)
        print("Current hand: ", end = ' ')
        display_hand(hand)
        ask_substitute = input("Would you like to substitute a letter?")
        if ask_substitute == 'yes':
            letter = input("Which letter would you like to replace: ")
            hand = substitute_hand(hand, letter)
            hand_score = play_hand(hand, word_list)
            total_score = total_score + hand_score
        else:
            hand_score = play_hand(hand, word_list)
            total_score = total_score + hand_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)
```

Below is an example output of what our game should look like.

```
Loading word list from file...
   83667 words loaded.
Enter total number of hands: 2
Current hand:  * u o t s r p
Would you like to substitute a letter? no
Current hand:  * u o t s r p
Enter word, or '!!' to indicate that you are finished: torps
That is not a valid word. Please choose another word.
Enter word, or '!!' to indicate that you are finished: tops
"tops" earned  114  points. Total:  114
Current hand:  * u r
Enter word, or '!!' to indicate that you are finished: r*u
That is not a valid word. Please choose another word.
Enter word, or '!!' to indicate that you are finished: u*r
That is not a valid word. Please choose another word.
Enter word, or '!!' to indicate that you are finished: ru*
"ru*" earned  18  points. Total:  132
Current hand:
Enter word, or '!!' to indicate that you are finished: !!
Total score for this hand:  132
Would you like to replay the hand?no
Current hand:  * a o n r p v
Enter word, or '!!' to indicate that you are finished:
```

Chapter 5

5.1 Getting Started

In this problem set, we will be using our knowledge of recursion and solving problems. This problem has two parts. The first part allows you to process thinking about problems in a recursive fashion, and the second part will give you experience thinking about problems in terms of classes. First, download problem set four from MIT open courseware and make sure all the files are saved in the same folder.

This problem says it broken down into three sections. In ps4a.py, You will write a recursive function that takes as input a string and figures out all the possible re-orderings of the characters in the string. What this means is, you will write a function that defines all the mutations of a given sequence of letters.

The second part consists of 2 sections. In ps4b.py, you will use object oriented programming to write a Caesar cipher. In ps4c.py, you will use object oriented programming to write a very simple substitution cipher.

As always, don't rename any of the files and make sure to keep all of the files in the same folder.

5.2 Part A - Permutations of a String

A permutation is simply a name for a re-ordering. So the presentations of the string "abc" are "acb", "bac", "bca", "cab", "cba". Know that a sequence is a presentation of itself. For this problem set you'll need to write a recursive function get_permutations that takes a string and returns a list of all its permutations.

A few requirements are that you must use recursion, and please avoid returning duplicates in your final list. Additionally, it is OK to use loops to code the solution and the order of the returned permutations does not matter.

Suggested approach

While there are multiple ways to solve this problem, here is a suggested approach. In order to solve any recursive problem, we must have at least one base case and a recursive case or cases. The base case is the simplest input we could have. For this approach, our base case is if sequence is a single character.

For sequences longer than one character, we need to identify a simple version of that problem. When that is solved, it should help us easily find all permutations for sequence.

The pseudocode below gives us what approach to recursively solving this problem.

Given an input string sequence:

Base case:

If sequence is a single character, there's only one way to order it

Return a single list containing sequence

Recursive case:

Supposed we have a method that can give us a list of all permutations of all, but the first character in sequence. (Think recursion)

Then the permutations of all characters and sequence would be all the different ways we can insert the first character into each permutation of the remaining characters

we're going to implement the get_permutation(sequence) function found in ps4a.py according to the specifications in the doc string. Then we will write three test cases for our function to make sure it works.

we will be going off the pseudocode and make it into actual code so that it can run in the function.

First, we were writing the base case, which is if sequence is a single character and there's only one way to order it.

```
def get_permutations(sequence):
    if len(sequence) == 1:
        return sequence
```

Next, we will write the recursive case. First, we define the permutations list which is an empty list in the beginning. Then we identify the first character in the sequence, which has a index of 0. Now we will write the code for the permutations of all characters in sequence except the first character. we do this by inserting the first character into all of the permutations of the rest of the string.

```
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
```

Now that we have the code for this function, we can write the test case for it. In `__name__ == '__main__'`, we will write three test cases for it. We will print out the input, expected output, and actual output. To make writing the test case simpler, limit your inputs to be three characters or fewer, as you will have n! permutations for a sequence of length n.

We can write the test cases to have three characters, two characters, and one character. This is so we have a variety of test cases.

```
if __name__ == '__main__':
    input_1 = 'cde'
    print("input: ", input_1)
    print("expected output: ", ["cde", "ced", "dce", "dec", "ecd", "edc"])
    print("actual output: ", get_permutations(input_1))

    input_2 = 'ms'
    print("input: ", input_2)
    print("expected output: ", ["ms", "sm"])
    print("actual output: ", get_permutations(input_2))

    input_3 = 'k'
    print("input: ", input_3)
    print("expected output: ", ["k"])
    print("actual output: ", get_permutations(input_3))
```

Now, we can run this code and test our function.

On the next page is an example of what the output should look like.

```
input:  cde
expected output:  ['cde', 'ced', 'dce', 'dec', 'ecd', 'edc']
actual output:  ['cde', 'dce', 'dec', 'ced', 'ecd', 'edc']
input:  ms
expected output:  ['ms', 'sm']
actual output:  ['ms', 'sm']
input:  k
expected output:  ['k']
actual output:  k
```

5.3 Part B - Caesar Cipher

Caesar cipher is a way to encrypt messages and decrypt messages by picking an integer and shifting every letter of your message by that integer. But first, we will review some vocabulary.

- Encryption - the process of obscuring or encoding messages to make them unreadable
- Decryption - making encrypted messages readable again by decoding them
- Cipher - algorithm for performing encryption and decryption
- Plaintext - the original message
- Ciphertext - the encrypted message. Note: a ciphertext still contains all of the original message information, even if it looks like gibberish.

we will treat uppercase and lowercase letters individually, so that uppercase letters are always mapped to uppercase letters, and lowercase letters are always mapped to lowercase letters. If an uppercase letter maps to "A" then the same lowercase letters should map to "a". Punctuation and spaces should be retained and not changed. For example, a plaintext message with a comma should have a corresponding cipher with a comma in the same position.

Examples:

plaintext	shift	ciphertext
'abcdef' 'Hello, World!' ''	2 4 any value	'cdefgh' 'Lipps, Asvph!' ''

we will be using classes in our code. we will have a Message class with two sub classes, PlaintextMessage and CiphertextMessage. Message contains methods that could be used to apply a cipher to a string either to encrypt or decrypt a message. Caesar cipher decrypts and encrypts with the same action. PlaintextMessage has methods to encode a string using a specified shift value. our class will always create an encoded version of the message and we will have method for changing the encoding. CiphertextMessage contains the method used to decode a string.

when you have completed your implementation, you can either create a CiphertextMessage instance, using an encrypted string that someone provides you and try to decrypt it; or you can encrypt your own PlaintextMessage instance, then create a CiphertextMessage instance from the encrypted message within the PlaintextMessage instance, and try to decrypt it and see if it matches the original plain text message.

Your job will be to fill in the methods for all three of these classes, according to the specifications given in the docstring of ps4b.py. You do not have to use recursion in this part but you're welcome to. There are a couple of helper functions that we have implemented for you: load_words and is_word. You may use these in your solution and you do not need to understand completely, but should read the associated comments. You should read and understand the helper code and the rest of the file and use it you guide your solution.

Part 1: Message

Fill in the method of the message class found in ps4b.py according to the specifications in the docstring. First, we must initialize a message object. The message object has two attributes, self.message_text and the self.valid_words

```
def __init__(self, text):
    self.message_text = text
    self.valid_words = load_words(WORDLIST_FILENAME)
```

The get_valid_words(self) should return an immutable version of the message text we added to this object in init. Luckily, strings are already immutable objects, so we can simply return that string. This is used to access self.message_text outside of the class. It should return self.message_text.

```
def get_message_text(self):
    return list(self.valid_words)
```

Next, get_vaild_words(self) should return a copy of self.valid_words to prevent someone from accidentally mutating the original list.

```
def get_valid_words(self):
    return list(self.valid_words)
```

The build_shift_dict(self, shift) should create 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): is the amount by which to shift every letter of the alphabet. This should range from 0 to 26. This function should return additionally mapping a letter to another letter.

```
def build_shift_dict(self, shift):
    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
```

The apply_shift(self, shift) function applies the Caesar cipher to self.message_text with the input shifts. It creates a new string that is self.message_text shifted down the alphabet by some number of characters, determined by the input shift. Returns the message text in which every character is shifted down the alphabet by the input shift.

```
def apply_shift(self, shift):
    shift_dict = self.build_shift_dict(shift)
    shifted_text=""
    for char in self.message_text:
        if char.isalpha():
            shifted_text = shifted_text+shift_dict[char]
        else:
            shifted_text = shifted_text+char
    return shifted_text
```

Part 2: PlaintextMessage

Fill in the methods of the PlaintextMessage class found in ps4b.py according to the specifications in the docstring.

__init__(self, text, shift) initializes a PlaintextMessage object. This 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)
```

```
def __init__(self, text, shift):
    super().__init__(text)
    self.shift = shift
    self.encryption_dict = self.build_shift_dict(shift)
    self.message_text_encrypted = self.apply_shift(shift)
```

The get_shift(self) function is used to safely access self.shift outside of the class. It should return self.shift.

The get_encryption_dict(self) is used to safely access a copy of self.encryption_dict outside of the class. It should return a copy of self.encryption_dict.

```
def get_shift(self):
    return self.shift
def get_encryption_dict(self):
    return dict(self.encryption_dict)
```

The get_message_text_encrypted(self) is used to safely access self.message_text_encrypted outside of the class. It should return self.message_text_encrypted.

```
def get_message_text_encrypted(self):
    return self.message_text_encrypted
```

The change_shift(self, shift) changes self.shift of the PlaintextMessage and updates other attributes determined by shift. It should return nothing.

```
def change_shift(self, shift):
    self.shift = shift
    self.encryption_dict = self.build_shift_dict(shift)
    self.message_text_encrypted = self.apply_shift(shift)
```



Part 3: CiphertextMessage



Given an encrypted message, if you know the shift used to encode the message, decoding it is essential. Message is encrypted message, and s is the shift used to encrypt the message, then apply_shift(message, 26-s) gives you the original plain text message. Fill in the method of the CiphertextMessage according to the specifications in the docstrings.

__init__(self, text) initializes a CiphertextMessage object.

```
def __init__(self, text):
    super().__init__(text)
```

decrypt_message(self) decrypts self.message_text by trying every possible shift value and finding the best one. we will determine the best one as a shift that creates the maximum number of real words when we use apply_shift(shift) on the message text. Note that if multiple shifts are equally good such that they all create the maximum number of words, you may choose any of those shifts. This function should return a tuple of the best shift value used to decrypt the message and the decrypted message text using that shift value.

```
def decrypt_message(self):
    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
```

Part 4: testing

Now we can write two test cases for PlaintextMessage and two test cases for CiphertextMessage. Used to displayed the input, expected output, and actual output. Below is an example we can test with.

```
if __name__ == '__main__':
    plaintext = PlaintextMessage('hello', 2)
    print('Expected Output: jgnnq')
    print('Actual Output:', plaintext.get_message_text_encrypted())

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

Below is an example of what the output should look like.

```
Loading word list from file...
   55901 words loaded.
Expected Output: jgnnq
Actual Output: jgnnq
Loading word list from file...
   55901 words loaded.
Expected Output: (24, 'hello')
Actual Output: (24, 'hello')
Loading word list from file
```

5.4 Part C - Substitution Cipher

Substitution cipher is a better way to hide your messages. In the approach, you create a hidden coding scheme, in which you substitute a randomly selected letter for each original letter. The number of possibilities to test if you wanted to decrypt substitution cipher message is much larger than for a Caesar cipher. So for this problem, we are going to just consider substitution cipher in which the vowels are encrypted. '

Similar to the Caesar Cypher, we are going to use classes to explore this idea. we will have a class with general functions for handling substitution message of this kind. we will also write a class with more specific implementation and specification that inherit from this class.

Part 1: SubMessage

There is a skeleton code for the following method in this class and your task is to implement them.

The __init__(self, text) initializes a SubMessage object. It has two attributes: self.message_text, self.valid_words.

```
def __init__(self, text):
    self.message_text = text
    self.valid_words = load_words(WORDLIST_FILENAME)
```

get_message_text(self) is used to safely access self.message_text outside of the class. It should return self.message_text.

```
def get_message_text(self):
    return self.message_text
```

The get_valid_words(self) it used to safely access a copy of self.valid_words outside of the class. This helps you avoid accidentally mutating class attributes. It should return a copy of self.valid_words.

```
def get_valid_words(self):
    return self.valid_words
```

The build_transpose_dict(self, vowels_permutation) creates a dictionary that can be used to apply a cipher to letter. The dictionary maps every uppercase a lowercase letter to an uppercase in 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 constants remain the same. The dictionary should have 52 keys of all the uppercase letters and all the lowercase letters. This function should return a letter mapping a letter to another letter.

```
def build_transpose_dict(self, vowels_permutation):
    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
```

apply_transpose(self, transpose_dict) returns and encrypted version of the message text, based on the dictionary.

```
def apply_transpose(self, transpose_dict):
    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
```

Part 2: EncryptedSubMessage

we have a lot of different options for substitutions for the vowels. As with the Caesar cipher, we can use the tricks of testing to see if the decoded words are real English words. we then keep the decryption that uses the most valid English words.

The __init__(self, text) initializes an EncryptedSubMessage object.

```
def __init__(self, text):
    super().__init__(text)
```

decrypt_message(self) attempts to decrypt the encrypted message. The 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 encrypted text are valid English words, and return the decrypted message with the most English words. If no mutations are found, return the original string. If there are multiple mutations that have the maximum number of words, return any one of them. This function should return the best decrypted message.

This is done by going through all of the different permutation ways in vowel_permutations and finding the one with the most valid words.

```
def decrypt_message(self):
    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
```

Part 3: Testing

Now, we can write two test cases for SubMessage and two test cases for EncryptedMessage.

Below is an example of a text case.

```
if __name__ == '__main__':
    message = SubMessage("Hello World! I love you!")
    permutation = "eaiuo"
    enc_dict = message.build_transpose_dict(permutation)
    print("Original message:", message.get_message_text(), "Permutation:", permutation)
    print("Expected encryption:", "Hallu Wurld!")
    print("Actual encryption:", message.apply_transpose(enc_dict))
    enc_message = EncryptedSubMessage(message.apply_transpose(enc_dict))
    print("Decrypted message:", enc_message.decrypt_message())
```

we have just finished problems set 4. we have learned recursion and classes. Below is an example of the output.

```
Loading word list from file...
   55901 words loaded.
Original message: Hello World! I love you! Permutation: eaiuo
Expected encryption: Hallu Wurld!
Actual encryption: Hallu Wurld! I luva yuo!
Loading word list from file...
   55901 words loaded.
Decrypted message: ('eiauo', 'Hello World! A love you!')
Original message: Hello World! I love you! Permutation: uieao
Expected encryption: es thes thi carrict cadi
Actual encryption: Hilla Warld! E lavi yao!
Loading word list from file...
   55901 words loaded.
is this the correct code
```

Chapter 6

6.1 Getting Started

In this problem set, we will use object-oriented programming to build a program to monitor news feeds over the Internet. Our program will filter the news, alerting the user when it notices a news story that matches the user's interest. This problem does not require many lines of code.

First, download problem set 5 from MIT open courseware problems. Read over the instructions and we will begin with problem one.

Problem 1

Parsing the feed is the process of turning a data stream into a structured format that is more convenient to work with. Parsing all of this information is not easy, so let's tackle an easy part first.

If someone has already done this specific parsing and has left you with variables that contain information for a news story, then we want to sort this information in an object that we can then pass around in the rest of our program. In this problem, we will write a class starting with a constructor and stores arguments appropriately.

The class, NewsStory, should take in (guid, title, description, link, pubdate) as arguments.

It should also contain the following methods.

```
get_guid(self)
get_title(self)
get_description(self)
get_link(self)
get_pubdate(self)
```

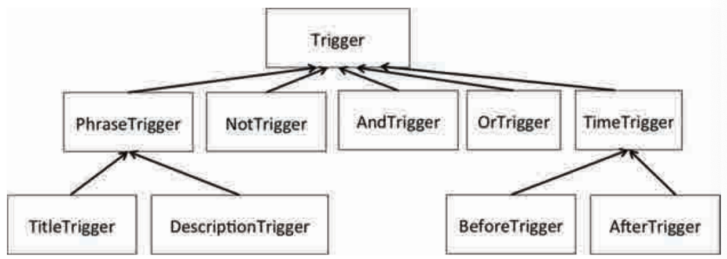
```
class NewsStory(object):
    def __init__(self, guid, title, description, link, pubdate):
        self.guid = guid
        self.title = title
        self.description = description
        self.link = link
        self.pubdate = pubdate
    def get_guid(self):
        return self.guid
    def get_title(self):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

Given a set of news stories, your problem will generate alerts for a subset of these stories. We will represent alerting rules as triggers. A trigger is a rule that is evaluated over a single news story and may fire to generate an alert. In order to simplify our code, we will use object polymorphism. We will define a trigger interface and then implement a number of different classes that implement that trigger interface in different ways. Every trigger class you define should implement the following interface either directly or transitively. We will not directly use the implementation of the Trigger class, which is why it raises it an exception should anyone attempt to use it.

The class below implements the Trigger interface. Any subclass that inherits from it will have an evaluate method. By default, they will use the evaluate method from Trigger, the superclass, unless they define their own evaluate function, which would then be used instead.

```
class Trigger(object):
    def evaluate(self, story):
        """
        Returns True if an alert should be generated
        for the given news item, or False otherwise.
        """
        raise NotImplementedError
```

we will define a number of classes that inherit from Trigger. In the figure below, Trigger is a superclass, from which all other classes inherit. The arrow from PhraseTrigger to trigger means that phrase trigger inherence from trigger. A PhraseTrigger is a Trigger.



6.2 Phrase Trigger

A phrase trigger is a specific type of trigger. A phrase is one or more words, separated by a single space between the words. Given some text, the trigger should only fire when each word in the phrase is present in its entirety and appear consecutively in the text, separated by spaces or punctuation. For example, a phrase trigger with the phrase "purple cow" should only fire on the following text snippets.

- `'PURPLE COW'`
- `'The purple cow is soft and cuddly.'`
- `'The farmer owns a really PURPLE cow.'`
- `'Purple!!! Cow!!!'`
- `'purple@#$%cow'`
- `'Did you see a purple     cow?'`

But it should not fire on these text snippets:

- `'Purple cows are cool!'`
- `'The purple blob over there is a cow.'`
- `'How now brown cow.'`
- `'Cow!!! Purple!!!'`
- `'purplecowpurplecowpurplecow'`

we will implement a phrase trigger abstract class, PhraseTrigger. it should take a string phrase as an arguments in the classes constructor. This trigger should not be case sensitive. PhraseTrigger should be a subclass of trigger. It has one new method, is_phrase_in, which takes in one string argument text. It returns true if the whole phrase is present and test, false otherwise. This method should not be case sensitive.

To do this, we will create a translation table that makes all punctuation marks spaces. This will help us identify whether or not it is a real phrase. Then, we have to determine whether the phrase has anything else attached to the end of it. For example, if there is a plural S after a word. This is because "purple cow" and "purple cows" are not the same thing.

we will also define the method is_phrase_in so we can use it in the two subclasses of PhraseTrigger: TitleTrigger and DescriptionTrigger.

```
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
```

TitleTrigger

TitleTrigger is a subclass of PhraseTrigger. It should fire when a news item's title contains a given phrase. As it was in PhraseTrigger, the phrase should be an argument to the class's constructor, and the trigger should not be case sensitive.

we should implement evaluate, and use the method is_phrase_in.

```
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)
```

Make sure to run your code through the unit test. If it says OK, then it means it has passed.

Description Trigger

DescriptionTrigger is another subclass of PhraseTrigger. It should fire when a new item's description contains the given phrase. This phrase should be an argument to the class's constructor and the trigger should not be case sensitive. 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)
```

Once this has been implemented, the unit test for DescriptionTrigger should pass.

6.3 Time Triggers

We'll move on from PhraseTrigger. Now we want to have triggers that is based on when the new story was published, not on its content. We will implement a time trigger abstract class, TimeTrigger, that is a subclass of Trigger. The class's constructer should take in time in EST as a string in the format of "3 Oct 2016 17:00:10". Make sure to convert time from string to a date time before saving it as an attribute. Because this is an abstract class, we will not directly instantiating any TimeTrigger.

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

Now we will implement two subclasses of TimeTrigger: BeforeTrigger and AfterTrigger. BeforeTrigger fires when a story is published directly before a trigger's time, and AfterTrigger fires when a story is published, strictly after the trigger's time. Time zones may be a problem, so for now we can set it to none. Then we will retrieve the pubdate and compare it to the trigger time to determine whether or not the story is published before or after.

The code for BeforeTrigger and AfterTrigger are similar, and we will use evaluate for both.

```
class BeforeTrigger(TimeTrigger):
    def __init__(self, trigger_time):
        super().__init__(trigger_time)
    def evaluate(self, story):
        pubdate = story.get_pubdate()
        pubdate = pubdate.replace(tzinfo=None)
        return pubdate < self.trigger_time

class AfterTrigger(TimeTrigger):
    def __init__(self, trigger_time):
        super().__init__(trigger_time)
    def evaluate(self, story):
        pubdate = story.get_pubdate()
        pubdate = pubdate.replace(tzinfo=None)
        return pubdate > self.trigger_time
```

Once these are implemented, the BeforeAndAfterTrigger unit test in our test suite should pass.

6.4 Composite Triggers

These are more powerful in that these triggers contain conditionals as to when it fires. There are three different types of composite triggers that we will be writing today: NotTrigger, AndTrigger, and OrTrigger.

NotTrigger

This trigger should produce its output by converting the output of another trigger. It should take this other trigger as an argument to its constructor. When it's done, its unit test should pass.

```
class NotTrigger(Trigger):
    def __init__(self, other_trigger):
        self.other_trigger = other_trigger
    def evaluate(self, story):
        return not self.other_trigger.evaluate(story)
```

AndTrigger and OrTrigger

These two triggers are similar in that they should take two triggers as arguments. The AndTrigger should fire on a news story only if both of the inputted triggered fire on that item. The OrTrigger should fire on a news story if either one or both of its inputted triggers should fire on that item.

```
class AndTrigger(Trigger):
    def __init__(self, trigger_one, trigger_two):
        self.trigger_one = trigger_one
        self.trigger_two = trigger_two
    def evaluate(self, story):
        return self.trigger_one.evaluate(story) and
            self.trigger_two.evaluate(story)

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

Once we have this, both unit tests should pass.

6.5 Filtering

You can now run ps5.py but it will display all Google and Yahoo news items. We will write a function filter_stories(stories, triggerlist) that 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 should take in a list of NewsStory instances, and returns a list of only the stories for which a trigger and triggerlist fires.

We will first create the empty list filtered_stories and for every story in which the trigger fires, we will add it to this list and return it at the end.

```
def filter_stories(stories, triggerlist):
    filtered_stories = []
    for story in stories:
        for trigger in triggerlist:
            if trigger.evaluate(story):
                filtered_stories.append(story)
                break
    return filtered_stories
```

6.6 User-Specified Triggers

Now we want to write code to read our trigger configuration from triggers.txt instead of from our Python code. There is already code that eliminates blank lines and comments. All we have to do is finish the implementation and return a list of trigger specified by the file.

First, we will have a dictionary to hold the name, the triggers, and then we will have a list to hold the final triggers. In trigger definitions, it states that the first element in a trigger definition is either of the keyword ADD or the name of the trigger. The second element of a trigger definition is a keyword that specifies the type of trigger being defined. The remaining elements of the definition are the trigger arguments. It states that the elements following ADD are names of one or more previously defined triggers.

In other words, if the first part is ADD, then the part after it should just be the names of the triggers.

```
# Dictionary to hold named triggers
triggers = {}

# List to hold the final triggers
trigger_list = []

for line in lines:
    parts = line.split(',')

    if parts[0] == 'ADD':
        for name in parts[1:]:
            trigger_list.append(triggers[name])
```

otherwise, if the first element is not ADD, then we will split it up into two elements which are the names of the trigger and the type of the trigger. what arguments are required depends on the trigger type. The instructions tell you what arguments each triggered requires.

- **TITLE**: one phrase
- **DESCRIPTION**: one phrase
- **AFTER**: one correctly formatted time string
- **BEFORE**: one correctly formatted time string
- **NOT**: the name of the trigger that will be NOT'd
- **AND**: the names of the two triggers that will be AND'd.
- **OR**: the names of the two triggers that will be OR'd.

we will simply transfer this into code language.

```
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]])
```

Finally, we will return the trigger_list. The final code is on the next page.

```
    # Dictionary to hold named triggers
    triggers = {}

    # List to hold the final triggers
    trigger_list = []

    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]])

    return trigger_list
```

Now we can modify the code within the function main_thread to use the trigger list specified in our configuration file.

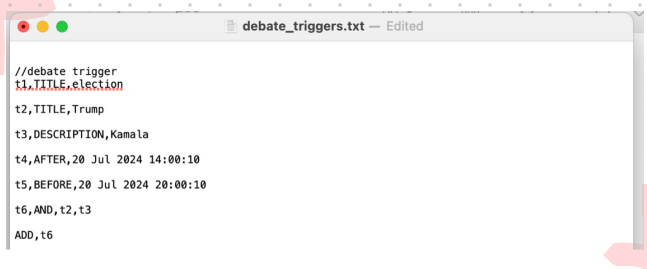
```
    # Problem 11
    # TODO: After implementing read_trigger_config, uncomment this line
    triggerlist = read_trigger_config('triggers.txt')
```

6.7 Modify

Problem 12 tells us to write a debate_triggers.txt file so that we can get news that is published within a three hour window of the presidential debate. we can do this by modifying our triggers.txt file. Since the election has not happened yet, our code will not run since there is no news. we will have to wait until the election happens to see the results of our code. This is written really similar to the code already in triggers.txt. You can also feel free to try the code and see if it gives you news from a specific time period or with a specific title.

Below is an example if we try to find news containing "Kamala" and "Trump" with a specific time window.

what we put in:



what we receive:



Acknowledgements

First of all, I would like to thank all of you for taking interest in computer science and coding. Self studying these problems has been tremendously rewarding for me and I hope you all have felt the same.

I would like to thank Dr.Chen for your motivation and encouragement. After long days of coding and debugging, your expertise has been a light in my path, pointing me in the right direction.

My journey would not have been complete without the encouragement and help from my parents and younger sister. I would like to express my deepest gratitudes for my mom, dad, and Michelle. Thank you all for encouraging me to not only pursue my passion in computer science, but also motivating me to write this notes book.