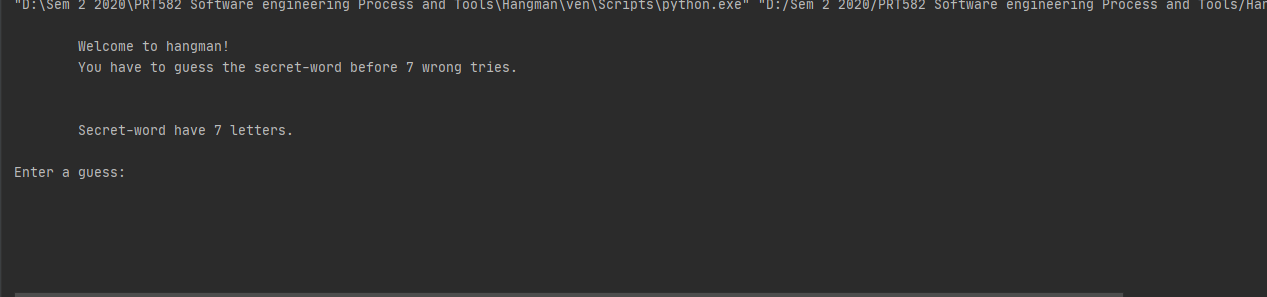
# Introduction

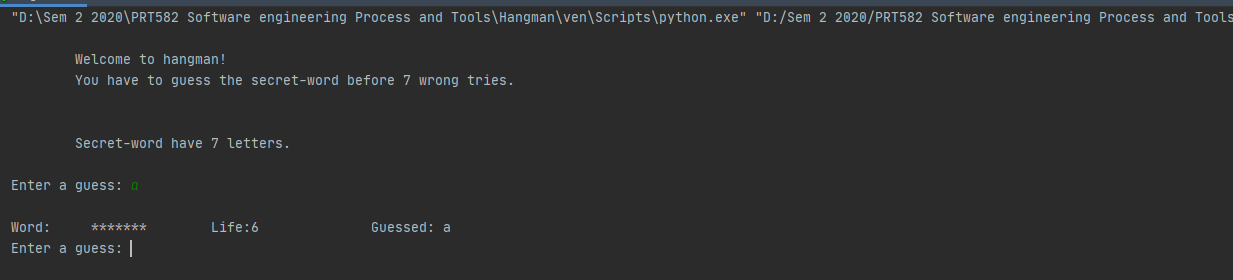
This report is about the Hangman Application which is developed on the Python language using the test-driven Development. The focus of the app development was to learn and implement the test-driven development i.e. software testing. Test driven development. Test-driven development is a software development process that relies on the repetition of a very short development cycle: requirements are turned into very specific test cases, then the code is improved so that the tests pass.

# Features of the Hangman Game

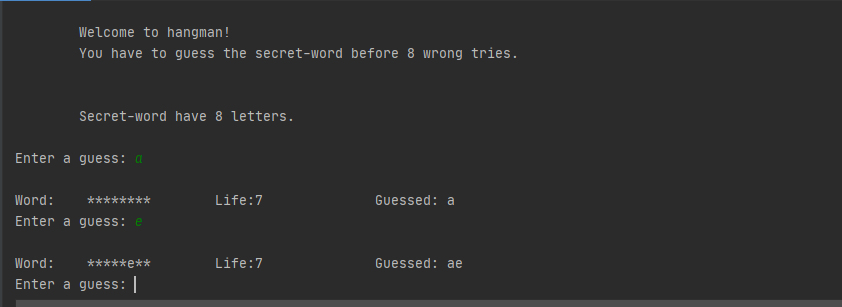
1. One word will be generated randomly



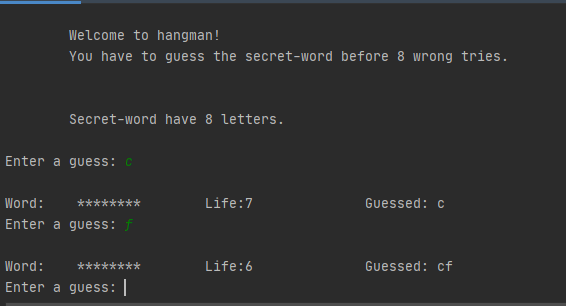
1. Player will be presented with several blank spaces representing the missing letters the player needs to find.



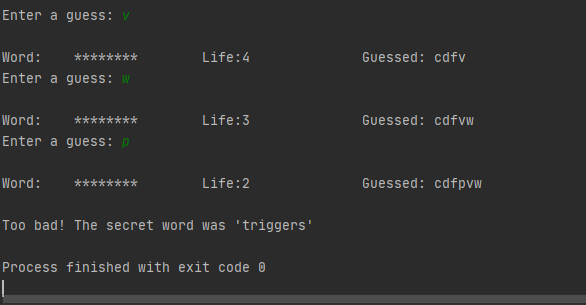
1. If the player’s chosen letter exists in the answer, then all places in the answer where that letter appear will be revealed.



1. Every time the player guesses a letter wrong, the player’s life will be deducted.



1. The player must find the missing word before the player’s life becomes zero.



# Code before refactoring

import random  
import time  
  
  
def mask\_word(s\_word, guessed):  
 *"masks letters in 's\_word' unless they are in 'guessed'"* mask = []  
 for i in s\_word:  
 if i in guessed:  
 mask.append(i)  
 else:  
 mask.append("\*")  
 return "".join(mask)  
  
  
def check\_cond(guessed):  
 *"checks input-condition 1: One letter guesses"* if len(guessed) == 1:  
 return "", True  
 return "Only a single letter is allowed", False  
  
  
def upper\_to\_lower(guess):  
 *"input-condition 4: changes uppercase to lowercase"* if guess.isupper():  
 return guess.lower()  
 return guess  
  
  
def guess\_evaluator(guessed, guess\_list):  
 *"guess\_evaluator: checks all the input-conditions"* guessed = upper\_to\_lower(guessed)  
  
 if not check\_cond(guessed)[1]:  
 return check\_cond(guessed)  
  
 return "", True  
  
  
def guess\_manager(guessed, guess\_list):  
 *"""guess\_manager: prints string for failed input-condition  
and asks for new input"""* while not guess\_evaluator(guessed, guess\_list)[1]:  
 print(guess\_evaluator(guessed, guess\_list)[0])  
 guessed = input("Enter another guess: ")  
 return guessed  
  
  
def wrong\_guesses(s\_word, guess\_list):  
 *"wrong\_guess: returns the number of wrong guesses"* count = 0  
 for i in guess\_list:  
 if i not in s\_word:  
 count += 1  
 return count  
  
  
def main():  
 with open("./test\_data/words") as words:  
 good\_words = []  
 for i in words:  
 i = i.strip()  
 if len(i) <= 6: # No short words  
 continue  
 if not i.isalpha(): # No punctuation  
 continue  
 if i[0].isupper(): # No proper nouns  
 continue  
 good\_words.append(i)  
 secret\_word = random.choice(good\_words)  
 print("""  
 Welcome to hangman!  
 You have to guess the secret-word before 6 wrong tries.  
 Secret-word have {} letters  
 .  
 """.format(len(secret\_word)))  
 formatter = "\nWord: {:^15} Life:{:<12} Guessed: {:<10}"  
 guesslist = ""  
 game\_won = "no"  
  
 # Game losses when 6 guesses are wrong  
 while not wrong\_guesses(secret\_word, guesslist) == 6:  
 newguess = input("Enter a guess: ")  
 newguess = guess\_manager(newguess, guesslist) # Checking conditions  
  
 guesslist += newguess # guesslist updates  
 guesslist = "".join(sorted(set(guesslist))) # sorting guesslist  
  
 life = (len(secret\_word) - wrong\_guesses(secret\_word, guesslist))  
 print(formatter.format(mask\_word(secret\_word, guesslist), life, guesslist))  
  
 if secret\_word == mask\_word(secret\_word, guesslist):  
 print("\n\nCongratulations!")  
 game\_won = "yes"  
 break  
  
 if game\_won == "no":  
 time.sleep(0.5)  
 print("\nToo bad! The secret word was '{}'".format(secret\_word))  
  
  
# import-guard  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

# Test cases

# test\_hangman.py  
  
import hangman  
  
  
# A. Getting secret word  
# 1. Secret word should have atleast 6 letters,no punctautaion,proper noun  
  
  
def test\_secret\_word\_6\_letters():  
 assert all(hangman.get\_secret\_word("./test\_data/1.words") == "policeman" for \_ in range(100))  
  
  
def test\_secret\_word\_no\_punctuation():  
 assert all(hangman.get\_secret\_word("./test\_data/2.words") == "fireman" for \_ in range(100))  
  
  
def test\_secret\_word\_no\_proper\_nouns():  
 assert all(hangman.get\_secret\_word("./test\_data/3.words") == "policeman" for \_ in range(100))  
  
  
# B. Masking secret word   
# 1. Masks entire word when not guessed.  
# 2. Unmasks entire word when fully guessed.  
# 3. Unmasks for words with repetitive letters.  
# Do not masks for wrong guesses.  
  
def test\_mask\_word():  
 word = "gangman"  
 guess = ""  
 assert hangman.mask\_word(word, guess) == "\*\*\*\*\*\*\*"  
  
  
def test\_mask\_word\_guessed():  
 words = ["python", "tigers", "whales"]  
 assert all([hangman.mask\_word(i, i) == i for i in words])  
  
  
def test\_mask\_word\_repetitive():  
 words = ["deadpool", "batman", "greenlantern"]  
 guess = ["kvwvkod", "qwewqa", "zxyxzen"]  
 masks = ["d\*\*d\*oo\*", "\*a\*\*a\*", "\*\*een\*\*n\*e\*n"]  
 assert all([hangman.mask\_word(words[i], guess[i]) == masks[i] for i in range(len(words))])  
  
  
# C. Checking inputs test cases  
# 1. Condition1: One letter guesses  
# 2. Condition2: No repetitive guesses  
# 3. Condition3: Only alphabets  
# 4. Changes uppercase to lowercase  
  
def test\_check\_cond1():  
 guess = ["you", "we", "u", "i", "v"]  
 result = [("Only a single letter is allowed", False), ("Only a single letter is allowed", False), ("", True),  
 ("", True), ("", True)]  
 assert all([hangman.check\_cond1(guess[i]) == result[i] for i in range(len(guess))])  
  
  
def test\_check\_cond2():  
 guess\_f = ["a", "b", "c"]  
 guess\_t = ["d", "e", "f"]  
 guess\_list = "qawbrc"  
 assert all([hangman.check\_cond2(guess\_t[i], guess\_list) == ("", True) for i in range(len(guess\_t))])  
 assert all(  
 [hangman.check\_cond2(guess\_f[i], guess\_list) == ("Already guessed '{}'".format(guess\_f[i]), False) for i in  
 range(len(guess\_f))])  
  
  
def test\_check\_cond3():  
 guess = ["1", "?", "n", "o", "f"]  
 result = [("Only alphabets are allowed", False), ("Only alphabets are allowed", False), ("", True), ("", True),  
 ("", True)]  
 assert all([hangman.check\_cond3(guess[i]) == result[i] for i in range(len(guess))])  
  
  
def test\_upper\_to\_lower():  
 guess = ["A", "B", "C", "d", "e", "f"]  
 result = ["a", "b", "c", "d", "e", "f"]  
 assert all([hangman.upper\_to\_lower(guess[i]) == result[i] for i in range(len(guess))])  
  
  
def test\_guess\_evaluator():  
 guess = ["A", "e", "f"]  
 guess\_list = "zxy"  
 result = [("", True), ("", True), ("", True)]  
 assert all([hangman.guess\_evaluator(guess[i], guess\_list) == result[i] for i in range(len(guess))])  
  
  
def test\_guess\_evaluator\_cond1():  
 guess1 = ["wou", "sss", "123", "zy"]  
 guess\_list = "zxy"  
 result = ("Only a single letter is allowed", False)  
 assert all([hangman.guess\_evaluator(guess1[i], guess\_list) == result for i in range(len(guess1))])  
  
  
def test\_guess\_evaluator\_cond2():  
 guess2 = ["x", "z", "y"]  
 guess\_list = "xzy"  
 assert all(  
 [hangman.guess\_evaluator(guess2[i], guess\_list) == ("Already guessed '{}'".format(guess2[i]), False) for i in  
 range(len(guess2))])  
  
  
def test\_guess\_evaluator\_cond3():  
 guess3 = ["?", "1", "#"]  
 guess\_list = "zxy"  
 result = ("Only alphabets are allowed", False)  
 assert all([hangman.guess\_evaluator(guess3[i], guess\_list) == result for i in range(len(guess3))])  
  
  
# E. Counting wrong guesses  
def test\_wrong\_guesses():  
 words = ["python", "tigers", "whales", "elephant"]  
 guesslist = ["lthwea", "qwerty", "asdfe", "qwdtfe"]  
 result = [4, 3, 2, 4]  
 assert all([hangman.wrong\_guesses(words[i], guesslist[i]) == result[i] for i in range(len(words))])

# Code after test cases validation and refactoring

*"hangman word game"*import random  
import time  
  
def get\_secret\_word(word\_file="./test\_data/words"):  
 *"finds secret\_word that satisfies certain conditions"* with open(word\_file) as words:  
 good\_words = []  
 for i in words:  
 i = i.strip()  
 if len(i) <= 6: # No short words  
 continue  
 if not i.isalpha(): # No punctuation  
 continue  
 if i[0].isupper(): # No proper nouns  
 continue  
 good\_words.append(i)  
 return random.choice(good\_words)  
  
def mask\_word(s\_word, guessed):  
 *"masks letters in 's\_word' unless they are in 'guessed'"* mask = []  
 for i in s\_word:  
 if i in guessed:  
 mask.append(i)  
 else:  
 mask.append("\*")  
 return "".join(mask)  
  
def check\_cond1(guessed):  
 *"checks input-condition 1: One letter guesses"* if len(guessed) == 1:  
 return "", True  
 return "Only a single letter is allowed", False  
  
def check\_cond2(guessed, guess\_list):  
 *"checks input-condition 2: No repetitive guesses"* if not guessed in guess\_list:  
 return "", True  
 return "Already guessed '{}'".format(guessed), False  
  
def check\_cond3(guessed):  
 *"checks input-condition 3: Only alphabets"* if guessed.isalpha():  
 return "", True  
 return "Only alphabets are allowed", False  
  
def upper\_to\_lower(guess):  
 *"input-condition 4: changes uppercase to lowercase"* if guess.isupper():  
 return guess.lower()  
 return guess  
  
def guess\_evaluator(guessed, guess\_list):  
 *"guess\_evaluator: checks all the input-conditions"* guessed = upper\_to\_lower(guessed)  
 if not check\_cond1(guessed)[1]:  
 return check\_cond1(guessed)  
 if not check\_cond2(guessed, guess\_list)[1]:  
 return check\_cond2(guessed, guess\_list)  
 if not check\_cond3(guessed)[1]:  
 return check\_cond3(guessed)  
  
 return "", True  
  
def guess\_manager(guessed, guess\_list):  
 *"""guess\_manager: prints string for failed input-condition  
and asks for new input"""* while not guess\_evaluator(guessed, guess\_list)[1]:  
 print(guess\_evaluator(guessed, guess\_list)[0])  
 guessed = input("Enter another guess: ")  
 return guessed  
  
def wrong\_guesses(s\_word, guess\_list):  
 *"wrong\_guess: returns the number of wrong guesses"* count = 0  
 for i in guess\_list:  
 if i not in s\_word:  
 count += 1  
 return count  
  
  
  
def playloop(secret\_word):  
 formatter = "\nWord: {:^15} Life:{:<12} Guessed: {:<10}"  
 guesslist = ""  
 game\_won = "no"  
  
 # Game losses when 6 guesses are wrong  
 while not wrong\_guesses(secret\_word, guesslist) == 6:  
 newguess = input("Enter a guess: ")  
 newguess = guess\_manager(newguess, guesslist) # Checking conditions  
  
 guesslist += newguess # guesslist updates  
 guesslist = "".join(sorted(set(guesslist))) # sorting guesslist  
  
 life = (len(secret\_word) - wrong\_guesses(secret\_word, guesslist))  
 print(formatter.format(mask\_word(secret\_word, guesslist), life, guesslist))  
  
 if secret\_word == mask\_word(secret\_word, guesslist):  
 print("\n\nCongratulations! you have guessed correctly")  
 game\_won = "yes"  
 break  
  
 if game\_won == "no":  
 time.sleep(0.5)  
 print("\nToo bad! The secret word was '{}'".format(secret\_word))  
  
  
def main():  
 secret\_word = get\_secret\_word()  
 print("""  
 Welcome to hangman!  
 You have to guess the secret-word before {} wrong tries.  
 """.format(len(secret\_word)))  
 print("""  
 Secret-word have {} letters.  
 """.format(len(secret\_word)))  
 playloop(secret\_word)  
  
  
# import-guard  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

# Implementation of TDD

TDD has been used to create the program. The process involved Write a failing unit test, Making the unit test pass and refactoring. The several test cases have been written to test each of the requirements and refactor has been done as required.

def main():  
 secret\_word = get\_secret\_word()  
 print("""  
 Welcome to hangman!  
 You have to guess the secret-word before {} wrong tries.  
 """.format(len(secret\_word)))  
 print("""  
 Secret-word have {} letters.  
 """.format(len(secret\_word)))  
 playloop(secret\_word)  
  
  
# import-guard  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

# How refactoring has been done, the code smell, issues, and solutions

The refactoring has been done in following ways

I. The duplication code has been removed and called with the single function

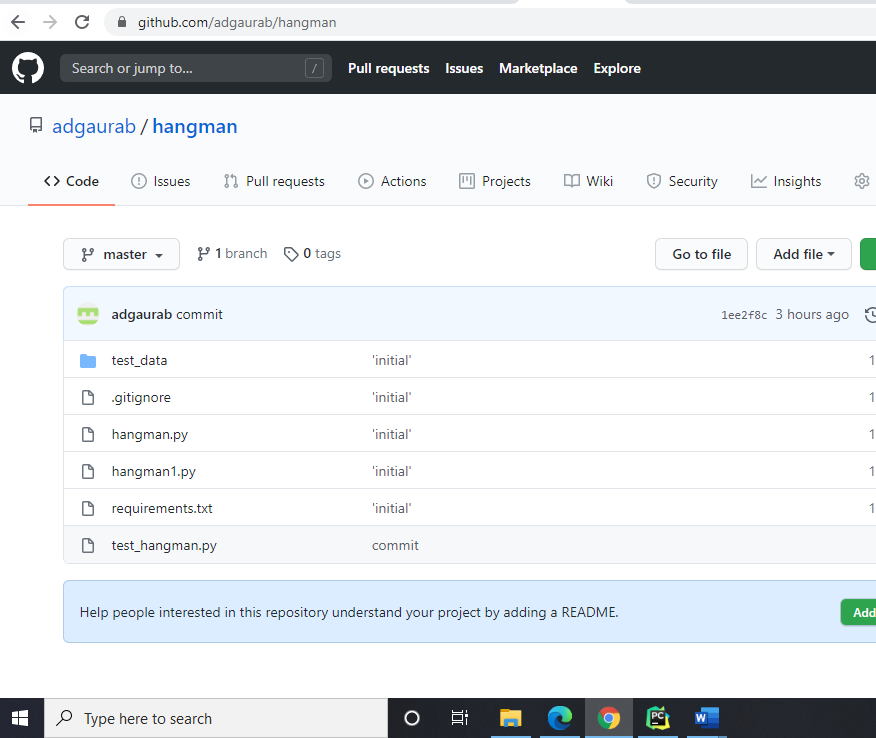
ii. The variables were not self-explanatory which was all in the main method. Now the different function has been created

iii. The long parameter has been limited and object has been to combine the parameters.

iv. long methods have been refacoted to shorter methods which is easy to read.

v. The method, variables and function are well defined.

# GitHub Link



<https://github.com/adgaurab/hangman>