Homework 3

Intro to Programming (Term III/2017–18)

built on 2018/05/10 at 16:54:06 ***due:*** *thu may 17 @ 11:59pm*

This assignment will give you more practice on the concepts of repetition, iteration, and functions. You will write some code, answer some questions, and hand them in electronically. The following instructions are **new to this assignment:**

- For every function you write that returns a value, write at least 5 test cases using `assert`. Put all these tests (`asserts`) in a separate file called `unittests.py`, which you will also hand in. Use the **import** command to access code you write in a different file.
- Before you hand in your solutions, delete or comment out all extraneous **print** statements. As a rule of thumb, if the question doesn't ask you to print you shouldn't print. Do not use **input** in this assignment.

Overview:

Problem	File Name
1.	`powerloop.py`
2.	`draw.py`
3.	`allodd.py`
4.	`abc.txt`

Problem	File Name
5.	`happy.py`
6.	`avgno.py`
7. (extra)	`anagram.py`
▷**Test Cases**	`unittests.py`

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Be sure to indicate who you have worked with (refer to the hand-in instructions).

Logistics

We're using a script to grade your submission before any human being looks at it. Sadly, the script is not as forgiving as we are. *So, make sure you follow the instructions strictly.* It's a bad omen when the course staff has to manually recover your file because the script doesn't like it. Hence:

- Save your work in a file as described in the task description. This will be different for each task. **Do not save your file(s) with names other than specified.**
- You'll zip these files into a single file called `a3.zip` and you will upload this one zip file to Canvas before the due date.
- Before handing anything in, you should thoroughly test everything you write.
- At the beginning of each of your solution files, write down the number of hours (roughly) you spent on that particular task, and the names of the people you collaborated with as comments. As an example, each of your files should look like this:

```
# Assignment XX, Task YY
# Name: Eye Loveprograming
# Collaborators: John Nonexistent
# Time Spent: 4:00 hrs

... your real program continues here ...
```

- The course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online on Canvas discussion.

Task 1: Power Loop (10 points)

For this task, save your work in `powerloop.py`

You'll implement a function `powerLoop(upto)` that takes an integer parameter `upto` ≥ 0 and returns nothing. However, when the function is called, it will print the value of $11^i \bmod 101$—that is, the remainder of dividing 11^i by 101—for $i = 0, 1, \ldots,$ `upto` in the following format:

```
0 1
1 11
2 20
3 18
(... more lines to follow ...)
```

Notice that there will be a total of `upto` + 1 lines, where on the i-th line, we display the value i followed by the value of $11^i \bmod 101$, separated by a single space.

NOTE: Because your function in this task doesn't return any meaningful value, you do *not* need asserts for this task.

Task 2: Draw, Let's Draw (10 points)

For this task, save your work in `draw.py`

Like the previous task, this task involves writing functions that take in some arguments and print to the display (not return). For each function, you'll draw a shape in text mode (think the old days when there was no graphical display).

FOR BOTH SUBTASKS: It is *imperative* that what you print looks exactly like what's asked of you, without any extra spaces, nor any extra lines. It must not contain any control characters (such as a back space, a tab, etc.)—if you have no idea what control characters are, you are probably not using it. *The grader script is comparing your answers with the modeled solutions byte for byte.* You do *not* need to write any assert statements for this task.

Subtask I: Implement a function `triangle(k)` that takes a nonnegative integer k denoting the size of the triangle and prints to the screen a triangle of size `k`.

A triangle of size k contains a total of k lines, where every line contains a total of $2k - 1$ characters. Each character is either the # character or the * character.

Here are a few examples:

```
*
```

Only 1 line for $k = 1$.

```
#*#
***
```

Only 2 lines for $k = 2$.

```
##*##
#***#
*****
```

Only 3 lines for $k = 3$.

Figure 1: Examples of triangles of various sizes.

Subtask II: Implement a function `diamond(k)` that takes a nonnegative integer `k` denoting the size of the diamond and prints to the screen a diamond of size `k`.

A diamond of size k contains a total of $2k$ lines, where every line contains a total of $2k + 1$ characters (each character is either a # or a *). A few examples are given in Figure 2 for you to discover the pattern.

```
##*##
#***#
#***#
##*##
```

$k = 2$

```
###*###
##***##
#*****#
#*****#
##***##
###*###
```

$k = 3$

```
####*####
###***###
##*****##
#*******#
#*******#
##*****##
###***###
####*####
```

$k = 4$

Figure 2: Examples of diamonds of different sizes.

Task 3: Is It All Odd? (10 points)

For this task, save your work in `allodd.py`

Remember that a number n is odd if it is *not* divisible by 2—that is, `n%2 != 0`. For this task, you'll write a function `is_all_odd(lst)` that takes in a list of integers and returns a Boolean value indicating whether all of the numbers in the list are odd.

For example:

- `is_all_odd([1,2,3])` should return `False` (because 2 is even).
- `is_all_odd([3,1,7])` should return `True` (all odd).
- `is_all_odd([])` should return `True` because it's vacuously true that all numbers in the input list are odd.

Task 4: A, B, C, D, AC (10 points)

For this task, save your work in `abc.txt`

In this task, you will study the following Python code and answer a few questions about it:

```python
# prints ch without starting a new line.
def pp(ch):
    print(ch, end='')

def a():
    pp('a')

def b():
    pp('b')
```

```python
def c():
    a()
    pp('c')

def d():
    pp('d')
    b()

def e():
    c()
    pp('e')
    d()
```

For each of the following strings, write *all* sequence of function calls that will produce the following output or say *impossible*:

1. abacdb
2. aacbba
3. dbacba
4. acedbb
5. bdbbca

Task 5: Happy and Sad Numbers (12 points)

For this task, save your work in `happy.py`

Happy numbers are a nice mathematical concept. Just as only certain numbers are prime, only certain numbers are happy—under the mathematical definition of happiness. We'll tell you exactly what happiness means.

The concept of happy numbers is only defined for whole numbers $n \geq 1$. To test whether a number is happy, we can follow a simple step-by-step procedure:

(1) Write that number down
(2) Stop if that number is either 1 or 4.
(3) Cross out the number you have now. Write down instead the sum of the squares of its digits.
(4) Repeat Step (2)

When you stop, if the number you have is 1, the initial number is *happy*. If the number you have is 4, the initial number is *sad*. There are only two possible outcomes, happy or sad.

By this definition, 19 is a happy number because $1^2 + 9^2 = 82$, then $8^2 + 2^2 = 68$, then $6^2 + 8^2 = 100$, and then $1^2 + 0^2 + 0^2 = 1$. However, 145 is sad because $1^2 + 4^2 + 5^2 = 42$, $4^2 + 2^2 = 20$, and $2^2 + 0^2 = 4$.

Subtask I: First, you'll implement a function `sumOfDigitsSquared(n)` that takes a positive number `n` and returns the sum the squares of its digits. For example,

- `sumOfDigitsSquared(7)` should return `49`
- `sumOfDigitsSquared(145)` should return `42` (i.e., `1**2 + 4**2 + 5**2 = 1 + 16 + 25`)
- `sumOfDigitsSquared(199)` should return `163` (i.e., `1**2 + 9**2 + 9**2 = 1 + 81 + 81`)

Subtask II: Then, you'll write a function `isHappy(n)` that takes as input a positive number `n` and tests if `n` is happy. You may wish to use what you wrote in the previous subtask.

- `isHappy(100)` should return `True`
- `isHappy(111)` should return `False`
- `isHappy(1234)` should return `False`
- `isHappy(989)` should return `True`

IMPORTANT: Your `isHappy` function must **not** call itself.

Subtask III: There are in fact many levels of happiness. The larger the number, the happier we feel about that number. The k-th happy number is the k-th smallest happy number. This means, the 1st happy number is the smallest happy number, which is 1. The 2nd happy number is the second smallest happy number, which is 7. Below is a list of the first few happy numbers:

> 1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97, 100, 103, 109, 129, 130, 133, 139, 167, 176, 188, 190, ...

Implement a function `kThHappy(k)` that computes and returns the k-th happy number. For example:

- `kThHappy(1)` returns 1
- `kThHappy(3)` returns 10
- `kThHappy(11)` returns 49
- `kThHappy(19)` returns 97

TIP: Use what you have already written. Don't repeat yourself.

Task 6: On Average, No Outlier (8 points)

For this task, save your work in `avgno.py`

Because the minimum and the maximum in a dataset are often outliers, a standard practice in data processing is to exclude the minimum value and the maximum value when computing the average. In this problem, you'll implement a function `robust_avg(lst)`, where

- the input is a list `lst` of *distinct* numbers (in any combination of **float** or **int**); and
- the function returns a floating-point number representing the average of the numbers in this dataset, excluding the smallest and the largest values.

You should recall that the average of a dataset is the sum of the values in that dataset divided by the count of the items in that dataset. Mathematically, if the dataset consists of n numbers $x_1, x_2, \dots, x_n$, then the average is $\frac{1}{n}(x_1 + x_2 + \cdots + x_n)$.

As a concrete example, say your input list is `[3, 1, 2, 5, 9, 11,4]`. The minimum is 1 and the maximum is 11. Once the extreme values have been excluded, we are left with 3, ~~1~~, 2, 5, 9, ~~11~~, 4, so we can compute the average by calculating $\frac{1}{5}(3+2+5+9+4)$, which equals 4.6 Therefore, `robust_avg([3, 1, 2, 5, 9, 11,4])` should return the number 4.6.

Your answer must be correct up to 4 decimal places.

Remarks on Testing: Floating-point numbers in Python are an approximation of the reals. For example, Python is unable to store the outcome of 22/7 precisely. Worse yet, how such approximation is done can vary from machine to machine. For this reason, to test functions that return floating-point quantities, it is generally prudent to never compare the outcomes directly. For example, if you're testing a function `foo`, do **not** write `assert(foo(42)==1.242529)`

It is more robust to write:

```
asseert(round(foo(42), 4)==1.2425)
```

which asserts that after rounding the outcome of `foo` to 4 decimal places, the outcome should be 1.2425. If this test passes, we know that `foo` does the right thing up to 4 decimal places. Read the documentation for **round** for more information.

Task 7: EXTRA: Anagram For The Purist (0 points)

For this task, save your work in `anagram.py`

READ THIS BEFORE YOU ATTEMPT IT: This is an extra problem for those who want a more challenging task. It is worth 0 points on the assignment; however, you'll earn

- bragging rights;
- a place on the course's wall of fame;
- brownie points that may be exchanged for real points if needed at the end; and
- above all, an opportunity to practice programming.

Anagrams are a geek's heaven. An *anagram* of a word w is a word formed by rearranging the letters of the word w, such as `iceman` formed from `cinema`. Similarly, `angel` is an anagram of `glean`. By this definition, two words must have the same length to be each other's anagrams. In addition, some letters can be repeated, e.g., `agree` vs. `eager`.

In this problem, you'll implement a function `isAnagram(word1, word2)` that takes in two words (as strings) and returns a Boolean indicating if `word2` is an anagram of `word1`. But **read on**; as this is a challenge, there are some restrictions!

You *cannot* use the built-in sorting functions. You *cannot* use a list data structure—or any built-in collection (dict, set, etc.) for that matter. In fact, you cannot use anything beyond simple loops. It is, however, guaranteed that both input words will only contain letters in the English alphabet.