Homework 4

Intro to Programming (Term I/2021–22)

built on 2021/10/06 at 14:26:46 **due:** *wed oct 13 @ 11:59pm*

This assignment will give you practice working with loops (for and while), as well as looping logic. In this assignment, you will solve a number of programming puzzles and hand them in.

Be sure to read this problem set thoroughly, especially the sections related to collaboration and the hand-in procedure.

Overview:

Problem	File Name
1.	`altsum.py`
2.	`powerloop.py`
3.	`draw.py`
4.	`aloud.py`

Problem	File Name
5.	`multiplek.py`
6.	`happy.py`
7.	`sansprimes.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.**
- Before handing anything in, you should thoroughly test everything you write.
- You will upload each file to our submission site `https://assn.cs.muzoo.io/` before the due date. Please use your SKY credentials to log into the submission site. Note that you can submit multiple times but only the latest version will be graded.
- For some task, you will be able to verify your submission online. Please do so as it checks if your solution is gradable or not. Passing verification does not mean that your solution is correct, but, at least, it passes our preliminary check.
- 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: Alternating Sum (10 points)

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

You'll write a function `altSum(lst: List[int]) -> int` that takes a list of numbers and returns the alternating sum of the numbers in the list (the operators alternate between addition and subtraction, starting with addition). For example:

- `altSum([])` returns 0.
- `altSum([1,3,5,2])` returns 1 (that is, $1+3-5+2$).
- `altSum([7,7,7,7])` returns 14 (that is, $7+7-7+7$).
- `altSum([31,4,28,5,71])` returns -59 (that is, $31+4-28+5-71$).

Task 2: Power Loop (10 points)

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

You'll implement a function `powerLoop(upto: int) -> None` that takes an integer parameter `upto` ≥ 0 and returns nothing. However, when the function is called, it will **print** the value of $7^i \bmod 97$—that is, the remainder of dividing 7^i by 97—for $i = 0, 1, \ldots,$ `upto` in the following format:

```
0 1
1 7
2 49
3 52
```

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 $7^i \bmod 97$, separated by a single space.

Task 3: 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.*

Subtask I: Implement a function `triangle(k: int) -> None` 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:

Subtask II: Implement a function `diamond(k: int) -> None` 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.

```
*
```

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.

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

$k = 2$

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

$k = 3$

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

$k = 4$

Figure 2: Examples of diamonds of different sizes.

Remarks: Because the grader script will check your console output, you must remove any extra print statement before submitting. Make sure your output matches with what is required **EXACTLY**.

Task 4: Read Aloud (10 points)

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

When we read aloud the list `[1,1,1,1,4,4,4]`, we most likely say four 1s and three 4s, instead of uttering each number one by one. This simple observation inspires the function you are about to implement. You're to write a function `readAloud(lst: List[int]) -> List[int]` that takes as input a list of integers (positive and negative) and returns a list of integers constructed using the following "read-aloud" method:

> Consider the first number, say m. See how many times this number is repeated consecutively. If it is repeated k times in a row, it gives rise to two entries in the output list: first the number k, then the number m. (This is similar to how we say "four 2s" when we see `[2,2,2,2]`.) Then we move on to the next number after this run of m. Repeat the process until every number in the list is considered.

The process is perhaps best understood by looking at a few examples:

- `readAloud([])` should return `[]`
- `readAloud([1,1,1])` should return `[3,1]`
- `readAloud([-1,2,7])` should return `[1,-1,1,2,1,7]`
- `readAloud([3,3,8,-10,-10,-10])` should return `[2,3,1,8,3,-10]`
- `readAloud([3,3,1,1,3,1,1])` should return `[2,3,2,1,1,3,2,1]`

Task 5: Multiples of K (10 points)

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

Remember that a number n is a multiple of another number, k, only if n is divisible by k—that is, the remainder `n%k` is 0.

For this task, you'll write a function `allMultiplesOfK(k: int, lst: List[int]) -> bool` that takes in an integer and a list of integers, and returns a Boolean value indicating whether all of the numbers in the list are multiples of k.

For example:

- `allMultiplesOfK(4, [1,10,20])` should return `False` (because 1 is not a multiple of 4).
- `allMultiplesOfK(3, [81,3,24])` should return `True` (all are multiples of 3).
- `allMultiplesOfK(11, [])` should return `True` because it's vacuously true that all numbers in the input list are multiples of k.

Task 6: 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: int) -> int` 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: int) -> bool` 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`

(Did we tell you `n` must be positive (i.e. at least 1)?)

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: int) -> int` 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 7: Sans Primes (12 points)

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

Remember that an integer n is prime if (i) $n \geq 2$ and (ii) it is only divisible by 1 and n itself.

Subtask I: You will start by implementing a function `is_prime(n: int) -> bool` that returns `True` if the number `n` is prime and `False` otherwise. For example:

- `is_prime(1)` should return `False`.
- `is_prime(2)` should return `True`.
- `is_prime(-2)` should return `False`.
- `is_prime(5)` should return `True`.
- `is_prime(41)` should return `True`.
- `is_prime(323)` should return `False`.

Subtask II: The thing is, many people believe prime numbers bring sadness and must be avoided. To this end, you'll write a function `sans_primes(numbers: List[int]) -> List[int]` that takes in a list of integers and returns the numbers in the input list in the original order, except that every prime number is banned, so it will be excluded and numbers that come immediately after a prime number will also be excluded.

For example:

- `sans_primes([1, 4, 9, 10])` should return `[1, 4, 9, 10]`.
- `sans_primes([1, 11, 9, 10, 17])` should return `[1, 10]`.
- `sans_primes([3, 10, 2, 8, 9, 4, 1, 7, 6, 5, 11])` should return `[9, 4, 1]`.
- `sans_primes([3, 1, 2, 4])` should return `[]`.
- `sans_primes([3, -2, 5, 7, 1, 42])` should return `[42]`.
- `sans_primes([1, 0, 3, 0, -2, 5, 7, 1, 42, 9])` should return `[1, 0, -2, 42, 9]`.