Homework 4

Intro to Programming (Term III/2017–18)

built on 2018/05/17 at 11:28:30 **due:** *fri may 25 @ 11:59pm*

This assignment will give you more practice on strings and lists, together with the concepts of iteration. You will write some code and hand it in electronically. As with the last assignment:

- For every function you write that returns a value, write at least 5 test cases using `assert`. These will go into `unittests.py`.
- 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 it, you shouldn't print it.

Help us get rid of **input**. If your program waits for console input, the grader will think it's stuck in an infinite loop, kill your program, and move on.

Overview:

Problem	File Name
1.	`reverse.py`
2.	`altsum.py`
3.	`aloud.py`
4.	`hidden.py`

Problem	File Name
5.	`cipher.py`
6.	`jogging.py`
7. (extra)	`pme.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 `a4.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: Reverse This String (10 points)

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

Write a function `rev_str(s)` that takes a string `s` (maybe empty) and returns a string that is the letters of `s` read backward. Don't use the built-in reverse mechanism. Also, your code must work directly with strings. The one-liner `s[::-1]` is *unacceptable*.

Examples:

- `rev_str("nowhere")` will return `"erehwon"`.
- `rev_str("gnimmargorpevoli")` will return `"iloveprogramming"`.
- `rev_str("y")` will return `"y"`.

Task 2: Alternating Sum (10 points)

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

Using the accumulator pattern we saw in class or otherwise, you'll write a function `altSum(lst)` 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 3: 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)` 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 4: Hidden String (10 points)

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

Meow is playing a game with her friend. She gives her friend a string `s` and asks her whether another string `t` "hides" inside `s`. Let's be a bit more precise about hiding. Say you have two strings `s` and `t`. The

string `t` hides inside `s` if all the letters of `t` appear in `s` in the order that originally appear in `t`. There may be additional letters between the letters of `s`, interleaving with them.

Under this definition, every string hides inside itself: `'cat'` hides inside `'cat'`. For a more interesting example, the string `'cat'` hides inside the string `'afciurfdasctxz'` as the diagram below highlight the letters of `'cat'`:

`afciurfdasctxz`.

Importantly, these letters must appear in the order they originally appear. Hence, this means that `'cat'` does not hide inside the string `'xaytpc'`. In this case, although the letters `c`, `a`, and `t` individually appear, they don't appear in the correct order.

As another example, the string `'moo'` does not hide inside `'mow'`. This is because although we can find an `m` and an `o`, the second `o` doesn't appear in `'mow'`.

In this problem, you'll help Meow's friend by implementing a function `is_hidden(s, t)` that determines whether `t` hides inside `s`. The function returns a Boolean value: `True` if `t` hides inside `s` and `False` otherwise.

Here are some more examples:

```
is_hidden("welcometothehotelcalifornia","melon") == True
is_hidden("welcometothehotelcalifornia","space") == False
is_hidden("TQ89MnQU3IC7t6","MUIC") == True
is_hidden("VhHTdipc07","htc") == False
is_hidden("VhHTdipc07","hTc") == True
```

Task 5: Writing A Cipher (10 points)

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

Hiding secrets from unwanted eyes has fascinated the humankind since ancient times. In the digital age, this is usually done by cipher algorithms. A cipher algorithm takes a string message called the *plaintext* and returns another string called the *ciphertext* that is (a) apparently hard to read, but (b) can be converted back into the original plaintext message. Turning the plaintext into the ciphertext is called encryption; reversing this process is called decryption.

In this task, you'll learn about a classic cipher called the transposition cipher and implement it.

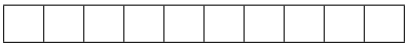
Transposition Cipher. As the name suggests, the basic idea of the transposition cipher is to take the message's symbols and jumble them up so that they become unreadable. The cipher decides how to move the symbols around using what's known as the *key*, denoted by k. To encrypt a text of length n, the key k is usually chosen to be a number between 2 and $k-1$ (inclusive). The exact encryption process is best described using an example.

Consider encrypting the text

```
Common␣sense␣in␣an␣uncommon␣degree␣is␣what␣the␣world␣calls␣wisdom
```

which contains 65 symbols, including spaces. The spaces are indicated by the ␣ symbol.

We'll use $k = 10$. First, we'll draw k boxes in a row—hence, drawing 10 boxes in our case.



Following that, we'll start filling the boxes from left to right using the symbols (letters and punctuation marks) from the input text. When you run out of boxes, you create a new row of k boxes and continue filling them until you accommodate the text entirely:

C	o	m	m	o	n	␣	s	e	n
s	e	␣	i	n	␣	a	n	␣	u
n	c	o	m	m	o	n	␣	d	e
g	r	e	e	␣	i	s	␣	w	h
a	t	␣	t	h	e	␣	w	o	r
l	d	␣	c	a	l	l	s	␣	w
i	s	d	o	m	×	×	×	×	×

We include the ×'s in the last row only to remind ourselves that they aren't part of the input text and should be ignored in later processing.

To complete the encryption, we'll read the symbol in the first (leftmost) column from top to bottom, then the second column from top to bottom, so on so forth. Notice that when we hit an ×, we simply skip over it onto the next column. The resulting encrypted string is

```
Csngalioecrtdsm␣oe␣␣dmimetcoonm␣hamn␣oiel␣ans␣lsn␣␣wse␣dwo␣nuehrw
```

YOUR TASK: Implement a function `enc(msg, key)` that takes a message string `msg` and a key `key`, $1 \leq$ `key` $<$ `len(msg)`, and returns a string representing the message encrypted using the transposition cipher (as explained above). Here are some more examples:

- `enc("abc",2)== 'acb'`
- `enc("monosodium glutamate", 7)=='mitouanmmo asgtoledu'`
- `enc("polylogarithmicsubexponential", 3)=='pygimseonaolatiuxntllorhcbpei'`

Task 6: Jogging Log (10 points)

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

Data geeks log their daily activities for no good reasons. In this problem, you are to process data in a logbook. The logbook is given to you as a list of strings. These are the lines from the logbook. Our goal is to derive this person's average jogging speed (ignoring data about all other activities).

The author logged data into her book carefully following a specific format. Hence, your input looks similar to the following example:

```
log_book = [
   "cycling;time:1,49;distance:2",
   "jogging;time:40,11;distance:6",
   "swimming;time:1,49;distance:1.2",
   "jogging;time:36,25;distance:5.3",
   "hiking;time:106,01;distance:8.29"
 ]
```

There can be as many lines (entries) as the logbook may contain. But each line always contains *three* pieces of information:

(1) the type of activity;
(2) the duration of that activity (time in minutes); and
(3) the distance involved in that activity (in km).

These three pieces are semicolon(`;`)-separated, and there will not be extra spaces anywhere. Therefore, for example, entry 0 in the example shows a cycling activity that lasted for 1 minute and 49 seconds involving 2 km of distance.

The author always uses a comma (`,`) to separate minutes from seconds. The seconds will always have 2 digits. For the distance component, she uses a dot (`.`) between the integral and the fraction parts.

This dot is omitted if there is only the integral part. The jogging activity—the only activity relevant to us—is every line where the activity field is exactly `"jogging"`. This means, if the activity field is, for example, `'joggingandsprinting'`, it will not be counted.

YOUR TASK: It helps to break this big task down into smaller functions, representing different logical units. Your main goal is to write a function `jogging_average(activities)` that takes a list of strings similar to the example above and returns a floating-point number that equals the author's average jogging speed in m/sec (meter per second)[1]. To be very precise, your answer only has to be correct up to ±0.0001. We promise also that there will be at least one jogging entry with a positive duration and distance.

Now to implement this function, you may wish to implement the following helper functions:

- a function `parse_time(line)` that takes a string (a line in the logbook) and returns the duration mentioned in this line (in seconds). For example, `parse_time('jogging;time:40,11;distance:6')` should return $60 \times 40 + 11 = 2411$.
- a function `parse_dist(line)` that takes a string (a line in the logbook) and returns the distance mentioned in this line (in km). For example, `parse_dist('jogging;time:40,11;distance:7.1')` should return 7.1.

EXAMPLES: Running `jogging_average` on the input described above, we should get 2.4586597. This is because there were *two* jogging activities:

- `"jogging;time:40,11;distance:6"`, which translates to 2411 seconds and 6 km; and
- `"jogging;time:36,25;distance:5.3"`, which translates to 2185 seconds and 5.3 km.

Therefore, the author has covered a distance of 11.3 km—that is, 11300.0 meters, in $2411 + 2185 = 4596$ seconds. Hence, the average jogging speed is

$$\text{Avg. Speed} = \frac{\text{Distance [m]}}{\text{Time [s]}} = \frac{11300.0\,[\text{m}]}{4596\,[\text{s}]} = 2.4586597\,[\text{m/s}],$$

so the function returns the floating-point number 2.4586597.

TIPS: Checking floating-point numbers for strict equality (e.g., `math.exp(1)==2.718`) is almost always a bad idea. Because of round-off errors and other approximation nuances, floating-point numbers are usually slightly different from what they look—even though the number displays 3.14 when printed, it might be 3.13999999999999 inside. Hence, we need a more robust test to compare them. You should use the **round** function as we did in previous assignments.

Task 7: EXTRA: Prime Meat Extract (0 points)

For this task, save your work in `pme.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.

[1] We know this is an usual unit for jogging speed, but we hope it helps everyone's sanity.

First, before you even begin, you'll want a fast `isPrime(n)` function. This tells us whether a given number n is prime. We have implemented one before in class, but it wasn't particularly fast. For this task, can you think of a way to make `isPrime` faster than trying all the possible divisors? *(Hint: Given a number n, if a is going to divide n, then there's another number n/a which also divides n.)*

The meat of a number is found by adding together the digits that make up the number. If the resulting number has more than one digit, the process is repeated until a single digit remains. For example, the meat of 245 is $2+4+5=11$ but because 11 has more than one digit, we repeat the same steps again. This time we have $1+1=2$. Therefore, the meat of 245 is 2.

For this problem, you'll work with a variation of this–prime meat extract. Instead of stopping only when there's only one digit left, it will also stop when the number being processed is a prime number. When the process stops, if the number at that point is a prime number, then that number is the prime meat extract (PME) of the original number; otherwise, the original number contains no prime meat.

- 1 is not a prime number, so 1 has no prime meat.
- 3 is a prime number, so the PME of 3 is 3.
- 4 is not a prime number, so 4 has no prime meat.
- 11 is a prime number, so the PME of 11 is 11.
- 642 is not a prime number, so adding its digits gives $6+4+2=12$. This is not a prime number, so adding again gives $1+2=3$. This is a prime number, so the PME of 642 is 3.
- 128 is not a prime number, so adding its digits gives $1+2+8=11$. This is a prime number, so the PME of 128 is 11.
- 886 is not a prime number, so adding its digits gives $8+8+6=22$. This is not a prime number, so adding again gives $2+2=4$. This is not a prime number, so 886 contains no prime meat.

You'll write a function `primeMeatExtract(x)` that takes a positive integer (i.e., >0) and returns its PME. If the input number contains no prime meat, you'll return the special value `None`.

Using this function, you'll implement `allPME(n)` which takes an integer $n \geq 1$ and returns a list of the unique PMEs of the numbers $1, 2, \ldots, n$. That is, the return value should contain every number in the following set exactly once:

$$\left\{ \texttt{primeMeatExtract}(x) \,\middle|\, x = 1, 2, \ldots, n \wedge x \text{ has a PME.} \right\}.$$

The numbers may be listed in any order.