YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

CS106AP Final Exam

Summer 2019

Your exam number:

Instructions (PLEASE READ CAREFULLY)

This printed exam has been provided to you as space for scratch work and as a reference for you as you work through the problems on your laptop. **We will not grade or look at any of the work written here.** The Exam Reference Sheet is printed at the end of the exam.

However, since we have make-up exams scheduled on different days, you must turn in this paper copy of the final at the end of the exam period. **Do not throw it away or take it with you.** In BlueBook, the first problem asks you for the number printed above ("Your exam number") so that we can confirm everyone has returned their hard copy. It also asks you for an electronic signature agreeing that you abided by the Stanford Honor Code. **Any exam that does not provide a complete answer to this problem (exam number + signature) will automatically receive a 0.**

There are six total problems on the exam with 180 points total, and some have multiple parts:

- Problem 1: Trace ..... 8 points
- Problem 2: Counting word frequency ..... 30 points (two parts)
- Problem 3: Structuring article data ..... 40 points (two parts)
- Problem 4: Dodger game ..... 50 points (six parts)
- Problem 5: Managing newspaper subscribers ..... 40 points
- Problem 6: One-liners ..... 12 points

Some general tips for tackling problems:

- We recommend looking over the entire exam before getting started so that you can budget your time well.
- Pay attention to which functions ask you to print a value versus return a value.
- Please type up your work and write down your ideas, even if you get stuck on a problem. We will do our best to give you partial credit, as long as we can understand what you wrote.
- You do not need to comment your functions (unless you find it helpful as you work). We will not be grading on style.

Good luck!

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

1. Trace (8 points)

Below is a program that tests your understanding of lists and mutability. Consider the following three lines of code:

```
a = [1, 2]
b = [3, 4]
bar(a, b)
```

Given the functions below, which will run without any crashes or syntax errors, what will be printed out when the above three lines of code are executed?

```
def foo(lst1):
    lst1.append(1 + lst1[1])
    print(lst1)

def baz(lst2):
    lst2.append(2 + lst2[1])
    lst2 = []
    print(lst2)

def bar(lst1, lst2):
    foo(lst2)
    print(lst1)
    print(lst2)
    baz(lst1)
    print(lst1)
    print(lst2)
    lst1, lst2 = lst2, lst1
    print(lst1)
    print(lst2)
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

You are now a part of an independent, local newspaper: the PYTimes! The following questions pertain to our newspaper and the articles that we produce.

2. Counting Word Frequency (30 points)

a) (15 points)
We're conducting an analysis on a PYTimes article and want to see which words occur most frequently within the article. We don't want to count common words (like "a", "and", "the", etc.) because they don't tell us very much about what the article is discussing.

GOAL: Write a function **`get_word_frequencies()`** that reads each word in a file indicated by **`filename`** and returns a counts dictionary that maps from a word to the number of times that that word appeared in the file. You should ignore words that are contained in a given list of **`common_words`**, and you should handle words case-insensitively (so that 'Science' and 'science' are treated as the same word).

Input: **`filename (string), common_words (list[string])`**
Returns: **`word_counts (dict: string -> int)`**

Assumptions:

- The file has multiple lines, which are separated by the newline character ('\n').
- Each line has no punctuation, and all words are separated by spaces (' ').
- All words in **`common_words`** are completely lowercase.
- Note that in the example below, 'Science' and 'science' are counted as the same (case-insensitive) word.

Example with a sample file:

File contents:

```
1 Computer Science
2 is a fun science
```

```
common_words = ['is', 'a', 'the']
```

Given the above file contents and **`common_words`** list, your function should have the following return value:

```
{'computer': 1, 'science': 2, 'fun': 1}
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

```
def get_word_frequencies(filename, common_words):
    """
    Reads each word in a file and returns a dictionary of the
    number of times each word occurs in a file, ignoring all
    words contained in the list common_words. This function
    should be case-insensitive.

    Input:
        filename (string): name of file to be processed
        common_words (list of strings): list of words to be ignored

    Returns:
        word_counts (dict: string -> int): dict containing the # of
                                           times each word appears
    """
     # YOUR CODE HERE
     pass
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

b) (15 points)

Now we want to figure out which words are used the most. In this function, you will figure out which ***n*** words from **`word_counts`** are used the most (e.g. if ***n* = 5**, you should return the information corresponding to the words with the **five** highest word counts).

GOAL: Write a function **`get_n_most_frequent()`** that takes in an integer **`n`** and a dictionary of word occurrences (**`word_counts`**) and returns a list of ***n* tuples**, which each contain a word and the number of times that that word occurred. This list should correspond to the ***n*** words with the **highest counts**, and it should be **sorted in descending order** (i.e. the tuple with the highest count should come first in the list).

Input: **`word_counts (dict: string -> int), n (int)`**
Returns: **`most_frequent_words (list of tuples)`**

Assumptions:

- We don't care how you handle tie-breaking if there are multiple words with the same number of counts.
- There will be at least ***n*** unique words in **`word_counts`**.
- You can assume that the **`word_counts`** dictionary passed into your function is completely accurate, and you won't need to call your function from part a).

Example:

Given your function, someone should be able to declare the following **`word_counts`** dictionary and get the output in blue below by running these lines of code:

```
>>> word_counts = {'science': 2, 'fun': 1, 'karel': 1, 'computer': 3, 'gates': 1}
>>> get_n_most_frequent(word_counts, 2)
[('computer', 3), ('science', 2)]
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

```
def get_n_most_frequent(word_counts, n):
    """
    Returns a sorted list containing tuples corresponding to the n
    words that have the highest counts in the word_counts dictionary.
    Each tuple should be formatted (word, word_frequency), where
    word is a string and word_frequency is an int. Output list is
    sorted in descending order.

    Input:
        word_counts (dict: string -> int): dict containing the # of
                                           times each word appears
        n (int): number of word tuples to return

    Returns:
        most_frequent_words (list of tuples): sorted list of the most
                                              frequent tuples of
                                              (word, word_frequency)
    """
    # YOUR CODE HERE
    pass
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

3. Structuring Article Data (40 points)

a) (20 points)

We want to learn more about what makes an article popular. In this problem, you will deal with four pieces of information pertaining to an article: article name, author, number of views, and genre. An article's "genre" is what news category the article belongs to, like "Local News," "Opinions," "Sports," etc.

GOAL: Write a function that takes in a list of strings corresponding to names of files (`filenames`) and returns a dictionary that groups articles by genre. Specifically, each of the **keys in this dictionary should be a genre**, and **each value should be a list of articles (tuples) associated with that genre.** Each article should be represented as a **tuple** that looks like this:

`(article_name, author, num_views)`

where `article_name` and `author` are strings and `num_views` is an int.

Input: `filenames (list of strings)`
Returns: `genre_data (dict: string -> list of tuples)`

Assumptions:

- Each of the files has information for a specific genre. That genre will be listed on the first line of the file. After the first line, the file is formatted:

  ArticleName,Author,Views
- No two files will contain information for the same genre (i.e. each file is unique).
- Article names and author names will not contain any commas.
- Although the example below contains a limited number of articles and genre files, there may be many more genre files and/or articles.
- The order of the list containing article tuples doesn't matter.

Example with sample files:

`opinions.txt:`

```
1 Opinions
2 Why CS106AP is the Best Class,Kylie Jue,159
3 My Love Affair with Python,Sonja Johnson-Yu,106
```

`arts.txt:`

```
1 Arts
2 Explorable Explanations Wow All,Kylie Jue,450
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

Given the files on the previous page, someone could use your `get_genre_data()` function as follows:

```
>>> filenames = ['opinions.txt', 'arts.txt']
>>> get_genre_data(filenames)
```

In the above example, the dictionary returned by your function would be the following:

```
{
 'Opinions': [('Why CS106AP is the Best Class', 'Kylie Jue', 159),
              ('My Love Affair with Python', 'Sonja Johnson-Yu', 106)],
 'Arts':     [('Explorable Explanations Wow All', 'Kylie Jue', 450)]
}
```

```
def get_genre_data(filenames):
    """
    Returns a dictionary containing article data, where the genre
    is the key and a list of tuples containing article information
    is the value. Takes in a list of filenames, where each file
    corresponds to the article data for a single genre.

    Input:
        filenames (list of strings): list of files containing genre
                                     data

    Returns:
        genre_data (dict: string -> list of tuples): dict of articles
                                                     organized by
                                                     genre
    """
    # YOUR CODE HERE
    pass
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

b) (20 points)
Now we have a dictionary that groups our articles by genre. But suppose we want to **organize the article data by author instead**!

For this problem, you can assume that we now have the correct genre-to-articles dictionary returned by your `get_genre_data()` function from part a). In particular, this dictionary is passed into the function you will write for this problem as the parameter `genre_data`. We will be manipulating this `genre_data` dictionary to group the articles by author instead of by genre.

GOAL: Write a function that takes in the genre-to-articles dictionary and returns a new dictionary that groups articles by author. Specifically, each of the **keys in this dictionary should be an author**, and **each value should be a list of articles (tuples) written by that author**. Each article should be represented as a **tuple** that looks like this:

`(article_name, num_views, genre)`

where `article_name` and `genre` are strings and `num_views` is an int.

Input: `genre_data (dict: string -> list of tuples)`
Returns: `author_data (dict: string -> list of tuples)`

Note that the tuples in the input dictionary are different from the tuples in the return (output) dictionary. The tuples in the input dictionary match those in part a), and the format of that dictionary is explained again below.

Assumptions:

- Like your output from part a), the keys in the `genre_data` dictionary are strings that represent a genre, and the value for each key is a list of articles (tuples) associated with the given genre. Each article is represented as a tuple that looks like this:

  `(article_name, author, num_views)`

  where `article_name` and `author` are strings and `num_views` is an int.
- You do not need to call your `get_genre_data()` function or use the files from part a) in order to get the input dictionary. The dictionary is passed into your function as the `genre_data` parameter.
- Author names are unique. In other words, if two article tuples contain the same author, you can safely assume that those articles were written by the same person.
- The order of the list containing article tuples doesn't matter.

Example:
Suppose we have the following `genre_data` dictionary:

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

```
genre_data = {
 'Opinions': [('Why CS106AP is the Best Class', 'Kylie Jue', 159),
              ('My Love Affair with Python', 'Sonja Johnson-Yu', 106)],
 'Arts':     [('Explorable Explanations Wow All', 'Kylie Jue', 450)]
}
```

Given the above **genre_data** dictionary, someone could use your **get_author_data()** function as follows:

```
>>> get_author_data(genre_data)
```

In the above example, the dictionary returned by your function would be the following:

```
{
 'Kylie Jue':         [('Why CS106AP is the Best Class', 159, 'Opinions'),
                       ('Explorable Explanations Wow All', 450, 'Arts')],
 'Sonja Johnson-Yu':  [('My Love Affair with Python', 106, 'Opinions')]
}
```

```
def get_author_data(genre_data):
    """
    From a dictionary structured with genres as the keys, returns
    a dictionary containing the same information, with the authors
    as the keys.

    Input:
        genre_data (dict: string -> list of tuples): dict organized by
                                                     genre

    Returns:
        author_data (dict: string -> list of tuples): dict organized
                                                      by author
    """
    # YOUR CODE HERE
    pass
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

4. Dodger Game (50 points)

To liven up non-news-articles sections of the paper, the PYTimes has decided to include an online game, Dodger, for readers to play. The game works like this:

- There are two shapes in the game: a square and a circle (the "ball"). The user controls the square via clicks and wants to prevent the ball from hitting the square.
- When the ball reaches the right side of the window, one of two conditions will occur:
    - The game ends if the ball hit the square.
    - If the ball did not hit the square, the ball should regenerate on the left side of the window, and you continue playing.

a) Fill in the constructor (10 points)

We've provided an outline for you inside the constructor for the tasks you'll need to accomplish. In this part, you'll create the ball and the square. Initializing the mouse listeners will occur in part b).

- The ball's diameter is exactly half of the size of the window height. The square's width is exactly half of the size of the window height minus 1.
- The ball starts with its left side directly against the left side of the window and in either the top half or the bottom half of the window (see Figure 1 for the two possible starting positions). At the start of the game, the ball is given a random x speed and a y speed of 0.
- The square starts with its top right corner flush with the top right corner of the window (see Figure 1).

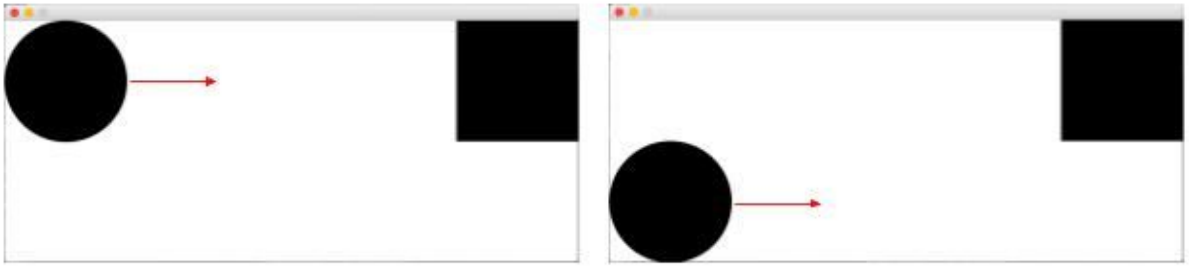

Figure 1: The two valid starting positions for the ball and the start position of the square. The red arrows would not be drawn in the actual program but indicate a random starting x speed given to the ball.

Assumptions and hints:

- If you want to generate a random boolean, you can use `random.randint(0, 1)`. The expression `random.randint(0, 1) == 0` will evaluate to `True` half of the time. Think about how you can use this to determine which half of the window the ball starts in.
- The ball's initial `x` speed is a random speed between `MIN_SPEED` and `MAX_SPEED`. Both of these constants are provided to you.
- Remember that the `x` and `y` coordinates for GRects and GOvals is their top left corner.
- The window is guaranteed to be at least two times wider than it is tall.

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

```
from campy.graphics.gwindow import GWindow
from campy.graphics.gobjects import GOval, GRect
from campy.gui.events.mouse import onmouseclicked
from campy.gui.events.timer import pause
import random

WINDOW_WIDTH = 800
WINDOW_HEIGHT = 340
MIN_SPEED = 2.0
MAX_SPEED = 4.0

class DodgerGraphics():

    def __init__(self, window_width=WINDOW_WIDTH,
                 window_height=WINDOW_HEIGHT):
        """
        Initializes the class attributes for the DodgerGraphics
        class. This class should keep track of the GWindow, the
        offset, the ball, and the square.
        """
        self.window = GWindow(width=window_width,
                              height=window_height)

        # TODO: Create and place a filled ball in the graphical window
        # (part a)

        # TODO: Create and place a filled square in the window
        # (part a)

        # TODO: Initialize any mouse listeners (part b)
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

b) Handling mouse click events (6 points)

The user can click anywhere in the window to update the square's `y` coordinate. The square can only take on two possible `y` coordinates: either the top of the window or halfway down the window.

The square's `y` coordinate depends on which half of the window (top or bottom) the user clicks in. If the user clicks in the bottom half of the window, the square should move to the bottom half of the window, and if the user clicks in the top half of the window, the square should move to the top half of the window (see Figure 2).

Write a mouse click listener method called `move_square()` that will update the square's location when the user clicks on the window.

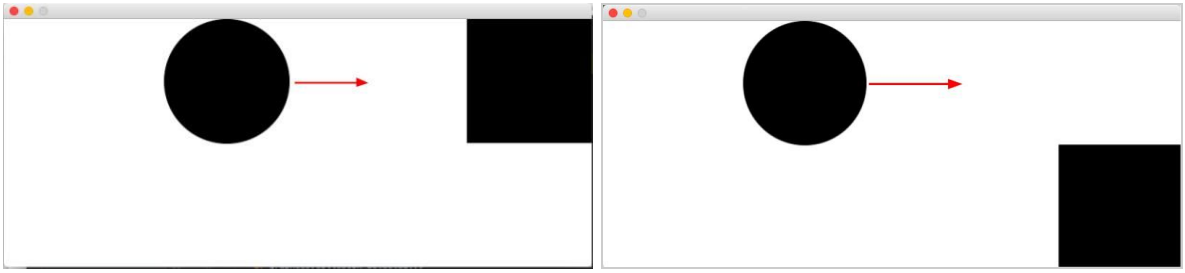

Figure 2: If the square is in the upper right corner (left), it will move to the bottom right corner (right) when the user clicks on the bottom half of the window. The square moves back to the upper right corner if the top half of the window is clicked instead.

Assumptions and hints:

- Note that the square's `x` position never changes.
- If the player clicks in the half of the window where the square already is, then the square does not change position.
- You should **initialize your mouse click listener method** using the correct campy function **inside your constructor from part a)**.

```
def move_square(self, event):
    """
    If the click is in the top half of the window, moves
    the square to the top left corner of the window.
    If the click is in the bottom half of the screen, moves
    the square to the bottom left corner.
    """
    # YOUR CODE HERE
    pass
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

c) Moving the ball (2 points)

The ball moves to the right of the window in a straight horizontal line toward the square at the random, constant speed specified in part a). Write a method called **`move_ball()`** that will get called at every step in the animation loop.

```
def move_ball(self):
    """
    Moves the ball in the x direction (to the right).
    """
    # YOUR CODE HERE
    pass
```

d) Reset the ball (4 points)

Recall that the ball will get reset if it hits the right wall of the window and did not hit the square (see Figure 3). In this case, the ball should regenerate back on the far left edge of the window with the following properties:

- It should have a new random positive x speed between **`MIN_SPEED`** and **`MAX_SPEED`**.
- It should regenerate randomly in the top half or bottom half of the window with equal probability.

Write a method called **`reset_ball()`** that does the above two tasks. You do not need to handle any wall detection in this part of the problem. **Note that some of this functionality is very similar to code you may have written in the constructor.** You can feel free to copy-paste any necessary repeated code or decompose out the code in the constructor if you find that easier.

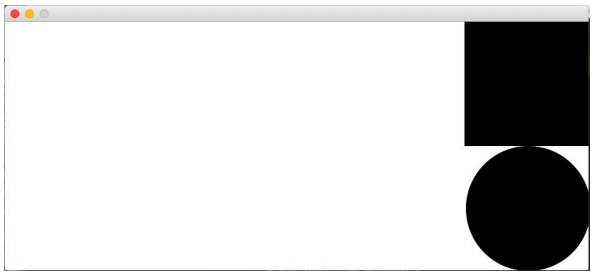

Figure 3: The ball will get reset to after hitting the right wall of the window.

```
def reset_ball(self):
    """
    Resets the ball's x speed to a random positive speed and
    Places the ball randomly in either the top half or the bottom
    half of the window, with its left side against the left wall.
    """
    # YOUR CODE HERE
    pass
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

e) Clear the screen (3 points)

Write a method that removes both of the objects from the screen. This will be called when the game ends because the ball hit the square. You do not need to handle any object collision detection in this function; just remove all of the objects.

```
def clear_screen(self):
    """
    Removes all objects from the window.
    """
    # YOUR CODE HERE
    pass
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

f) Implement the `main()` loop logic (25 points)

We have not yet asked you to write any methods that detect the two cases when the ball reaches the right side of the window. Recall that these are:

- **When the ball's right edge hits the right edge of the window**, in which case the ball should regenerate on the left side of the window (see Figure 4).
- **When the ball hits the square**, in which case the game ends and everything should be removed from the screen. (see Figure 5)

We've provided you with an initial structure for the animation loop inside **`main()`**, which calls your **`DodgerGraphics`** class constructor, as well as its **`move_ball()`** method. However, the current loop will repeat infinitely because no stop conditions have been added.

In this part of the problem, you'll need to complete two tasks:

- Create methods inside your **`DodgerGraphics`** class to detect the two cases above when the ball reaches the right side of the window.
- Use those methods inside the **`main()`** function's animation loop and add any other method calls that are necessary to finish implementing the gameplay.

Assumptions and hints:

- You will need to call some of the other methods you wrote in previous parts of this problem. You can assume all of the methods work correctly with no bugs.

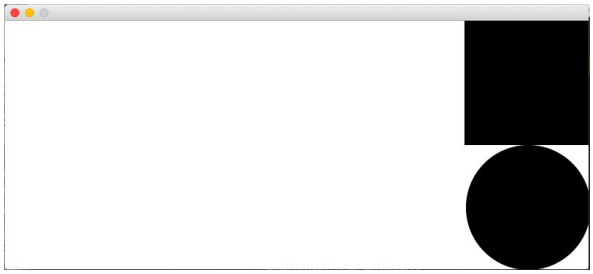

Figure 4: In this first case, the ball will get reset after hitting the right wall of the window.

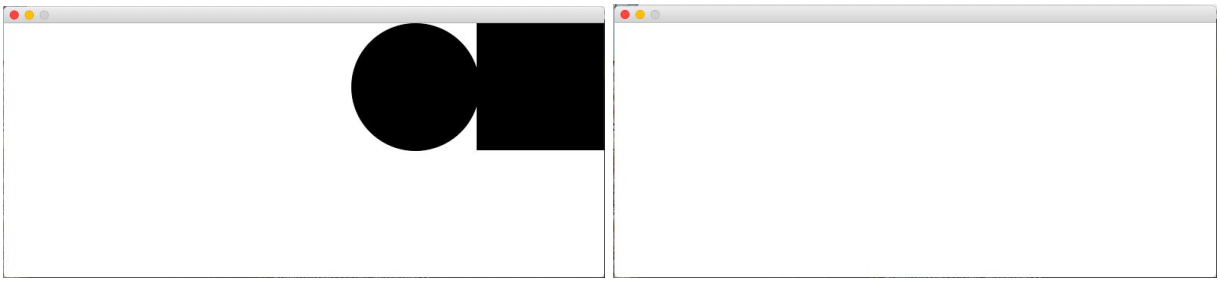

Figure 5: In this second case, the game is over once the ball has touched the square (left). You should clear the screen (right) if you detect that the collision occurred.

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

```
FRAME_RATE = 1000 / 120 # 120 frames per second.


def main():
    """
    Main gameplay loop.
    """
    graphics = DodgerGraphics()

    while True:

        # ADD STOP CONDITION(S) AND/OR ADDITIONAL LOGIC HERE

        graphics.move_ball()
        pause(FRAME_RATE)


if __name__ == '__main__':
    main()
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

5. Managing Newspaper Subscribers (40 points)

As a newspaper, we need a source of income, so we have people "subscribe" to our newspaper for one year at a time. During a year when a person is subscribed to our newspaper, they pay to have access to all of our online articles. Not only can you buy a subscription for yourself, you can also buy gift subscriptions for other users (i.e. subscriptions that you pay for and give to the other user as a gift).

GOAL: Write a **`PYTimesUser`** class, which is a class that stores all of the information related to PYTimes users, their yearly subscriptions, and the balance in their account (used to pay for subscriptions).

The class should satisfy the following requirements:

- Your constructor should take in a user's name and subscription price (how much it would cost them to subscribe to online articles for one year). It should also initialize any attributes (instance variables). (We provide the constructor header for you.)
- In addition to the qualities passed into the constructor, each **`PYTimesUser`** needs some way of keeping track of:
    - What years the user subscribed to the PYTimes. In particular, you should decide what data structure would be appropriate to store the different years that the user paid for the newspaper.
    - How much money is left in their account for paying for subscriptions.
- Your **`PYTimesUser`** class must have the following methods (make sure to check the comments in the starter code for more extensive descriptions):
    - **`add_subscription()`:** takes in an integer **`year`** and adds it to the data structure containing the user's subscription years. **Hint**: this may come handy in other methods!
    - **`subscribe()`:** takes in an integer **`year`** and attempts to buy a subscription for the given year. If the user does not have enough money in their account, prints an error message that includes the current balance (i.e. the amount of money currently in the account).
    - **`deposit()`:** takes in an integer **`amount`** and adds the amount to the user's account.
    - **`buy_gift_subscription()`:** takes in an integer **`year`** and a PYTimesUser **`other_user`** and attempts to use the money in the current user's account to buy a subscription for the other user for the given year. If the current user doesn't have enough money to buy a subscription at the **`other_user`**'s subscription price, the method prints out an error message that includes the current user's balance. **If the transaction is successful, it prints out a success message that includes both the current user and the `other_user`'s names.**
    - **`get_subscription_price()`:** returns the user's subscription price.
    - **`get_name()`:** returns the user's name.

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

Assumptions:

- Clients of the **`PYTimesUser`** class will pass in inputs of the correct type into every method.
- All values that are passed in for **`year`** will be valid values between 1990 and 2020.
- When **`add_subscription()`**, **`subscribe()`**, and **`buy_gift_subscription()`** are called, you can assume that the year passed in does not already exist in the user's subscription years data structure. In other words, you do not need to do any error handling for these three functions.

```
SUBSCRIPTION_PRICE = 50

class PYTimesUser:
    def __init__(self, name, subscription_price=SUBSCRIPTION_PRICE):
        """
        Initialize all attributes.

        Input:
            name (string): name of PYTimesUser
            subscription_price (int): cost of newspaper subscription
        """
        # YOUR CODE HERE
        pass

    def add_subscription(self, year):
        """
        Adds year to subscriptions data structure.

        Input:
            year (int): year to add to subscriptions data structure
        """
        # YOUR CODE HERE
        pass

    def subscribe(self, year):
        """
        Checks to see if PYTimesUser has enough money to purchase a
        subscription. If there is enough money, buys subscription for
        given year and adds the year to subscriptions. If
        there is not enough money, prints out an error message that
        includes the current balance (amount of money in account).

        Input:
            year (int): year for which to purchase a subscription
```

```
        """
        # YOUR CODE HERE
        pass

    def deposit(self, amount):
        """
        Deposits money into the PYTimesUser's account.

        Input:
            amount (int): amount to deposit
        """
        # YOUR CODE HERE
        pass

    def buy_gift_subscription(self, other_user, year):
        """
        Attempts to purchase a gift subscription for another
        PYTimesUser. If you don't have enough money to purchase
        a subscription at the other user's subscription price,
        then it displays an error message and your balance (amount
        of money in your account). If the transaction is successful,
        the other user now has a subscription for the given year, and
        the method displays a success message including the names
        of both users.

        Input:
            other_user (PYTimesUser): user for whom to buy gift
                                      subscription
            year (int): year for which to purchase a subscription
        """
        # YOUR CODE HERE
        pass

    def get_subscription_price(self):
        """Getter method for subscription price"""
        # YOUR CODE HERE
        pass

    def get_name(self):
        """Getter method for user name"""
        # YOUR CODE HERE
        pass
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

6. One-Liners (12 points)

The PYTimes has decided to create an alternative revenue stream by selling fruit. To help them better understand their fruit supply, you must complete each of the following four tasks in one line of code each. **Any solution that is longer than one line of code is NOT eligible to receive credit.**

a) Given a list of tuples each containing three elements, which correspond to a fruit's name, price, and number of fruit available, write a comprehension that contains the names of each of the fruits in the list, where the names are fully capitalized.

```
# input: fruits = [('banana', 0.45, 6), ('jackfruit', 4.55, 2),
                   ('kiwi', 0.15, 23)]
# output: ['BANANA', 'JACKFRUIT', 'KIWI']
# YOUR SOLUTION BELOW:
```

b) Given a list of tuples each containing three elements, which correspond to a fruit's name, price, and number of fruit available, write a comprehension that contains the names of each of the fruits in the list if the price of the fruit (the second element in the tuple) is under 0.50.

```
# input: fruits = [('banana', 0.45, 6), ('jackfruit', 4.55, 2),
                   ('kiwi', 0.15, 23)]
# output: ['banana', 'kiwi']
# YOUR SOLUTION BELOW:
```

c) Given a list of tuples each containing three elements, which correspond to a fruit's name, price, and number of fruit available, write an expression using **`max`** and a lambda function to yield the tuple with the highest number of fruit available.

```
# input: fruits = [('banana', 0.45, 6), ('jackfruit', 4.55, 2),
                   ('kiwi', 0.15, 23)]
# output: ('kiwi', 0.15, 23)
# YOUR SOLUTION BELOW:
```

d) Given a list of tuples each containing three elements, which correspond to a fruit's name, price, and number of fruit available, write an expression using **`sorted`** and a lambda function to yield a list of these tuples in **decreasing order based on the value of the fruit in stock**. The value of the fruit in stock can be calculated by multiplying the number of fruits

in stock by the price of the fruit. For example, as indicated in the provided **`fruits`** list below, there are 2 jackfruit in stock, and each one has a price of $4.55. Therefore, the total value of jackfruit in stock is 2 * $4.55 = $9.10.

```
# input: fruits = [('banana', 0.45, 6), ('jackfruit', 4.55, 2),
                   ('kiwi', 0.15, 23)]
# output: [('jackfruit', 4.55, 2), ('kiwi', 0.15, 23),
           ('banana', 0.45, 6)]
# YOUR SOLUTION BELOW:
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

Exam Reference Sheet

General Python

Built-in functions

`len()`	Returns the length of a collection (e.g. string or list).
`int()/float()/str()`	Converts a Python object to an integer/float/string type.
`range()`	Generates a range of integer values.
`input()`	Presents a string prompt to the user and returns the string that they input.
`print()`	Display output to the text output area (console).
`min()`	Given an iterable, return the minimum element from that iterable. Can take an optional key function parameter.
`max()`	Given an iterable, returns the maximum element from that iterable. Can take an optional key function parameter.
`sorted()`	Given an iterable, returns the sorted iterable. Can take an optional key function parameter.
`sum()`	Given an iterable, returns the sum of its elements.

Keywords used in boolean expressions

`not`	Negates (flips) the boolean value that follows.
`in`	Indicates if an element is part of a collection of objects.
`and`	Returns True if both values are True, False otherwise.
`or`	Returns False if both values are False, True otherwise.

Strings

Remember that string functions are called using the `noun.verb()` convention. For example, `str.isalpha()` where `str` is the string literal or variable storing the string.

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

Functions that return booleans

`isalpha()/isdigit()`	Returns a boolean indicating if a string is composed of all letters/digits.
`isupper()/islower()`	Returns a boolean indicating if a string is composed of all uppercase/lowercase characters.

Functions that return strings

`upper()/lower()`	Returns a string with all characters uppercase/lowercase.
`strip()`	Returns a string with leading and trailing whitespace removed.

Functions that return ints

`find()`	Returns the index of the first occurrence of a specified character in the string.

Functions that return lists

`split()`	Returns a list containing all substrings separated by the indicated delimiter.

Lists

To create an empty list, you use square brackets **`[]`**.

Remember that list functions are called using the **`noun.verb()`** convention. For example, **`lst.append()`** where **`lst`** is the variable storing the list.

`append()`	Add an element to the end of the list.
`extend()`	Add all elements in the specified list to the end of the target list.
`pop()`	Remove an element from the list and return it.
`insert()`	Insert an element into the specified index of a list.

Slicing

Recall that you can use slicing on both strings and lists (collections). You can get a particular slice of a collection using **`collection[start_index:end_index:step]`**, where the slice starts at **`start_index`** and stops right before **`end_index`** (not inclusive). The step is optional and defaults to a value of 1.

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

Dictionaries

- To create an empty dictionary, you use curly brackets `{}`.
- You can put or re-assign a key-value pair in your dictionary `d` using `d[key] = value`.
- To check if a particular key exists inside your dictionary `d`, you can use `key in d`.

Remember that list functions are called using the `noun.verb()` convention. For example, `d.append()` where `dict` is the variable storing the dict.

`keys()`	Returns an iterable over all of the keys in the dictionary.
`values()`	Returns an iterable over all of the values in the dictionary.
`items()`	Returns an iterable over the key, value pairs in the dictionary.

Recall that you can use iterables in for-each loops. To convert them to lists, you use `list(iterable)`. For example, you would need to use `list(d.keys())` to create a list of all the keys in a dictionary `d`.

Images

SimpleImage Code Patterns

```
# create image from filename
image = SimpleImage(filename)
# create blank image
image = SimpleImage.blank(width, height)

# foreach loop
for pixel in image:
    # do something with pixel

# range/y/x loop
for y in range(image.height):
    for x in range(image.width):
        pixel = image.get_pixel(x, y)

# pixel attributes
pixel.red, pixel.green, pixel.blue
pixel.x, pixel.y
```

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

File reading

Reading lines from a file

```
with open(filename, 'r') as f:
    for line in f:
        # do something with line
```

Campy

GWindow has the following properties and functions:

window.width	A property for accessing the window's width
window.height	A property for accessing the window's height
window.add()	Add a specified object to the window. Can optionally specify the x and y coordinates to place the object add
window.remove()	Removes a specified object from the window
window.clear()	Clears all content from window and returns it to its blank state
window.get_object_at()	Gets the object (if any) located at a given location in the window

Create a GWindow by using the following constructor:

```
window = GWindow(width=100, height=100, title='Breakout')
```

GObject has the following properties and functions:

obj.width	A property for accessing the object's width
obj.height	A property for accessing the object's height
obj.x	A property for accessing the object's x-coordinate
obj.y	A property for accessing the object's y-coordinate
obj.color	A property for accessing the object's outline color. A list of colors supported by campy can be found here.

YOU MUST RETURN THIS HARD COPY OF THE FINAL EXAM AT THE END OF THE EXAM PERIOD.

`obj.fill_color`	A property for accessing the object's interior (fill) color. A list of colors supported by campy can be found here. [`GRect` and `GOval` only]
`obj.filled`	A property for accessing whether or not the object is filled in (either True or False) [`GRect` and `GOval` only]
`obj.move(dx,dy)`	Moves the object on the screen using the specified displacements

There are multiple different `GObject` shapes:

`GRect`	`rectangle = GRect(width, height, x=0, y=0)` where `(x, y)` is the upper left corner of the rectangle
`GOval`	`oval = GOval(width, height, x=0, y=0)` where `(x, y)` is the upper left corner of the bounding rectangle around the oval
`GLine`	`line = GLine(x0, y0, x1, y1)` where `(x0, y0)` and `(x1, y1)` are the starting and ending points for the line
`GLabel`	`text_label = GLabel(label, x=0, y=0)` where `label` is the string contained inside the text label and `(x, y)` is the bottom left corner of the label

Campy allows you to detect four types of mouse movements:

`onmouseclicked(fn)`	Occurs when the user **presses and then releases** any button on their mouse
`onmousereleased(fn)`	Occurs when the user **releases** any button on their mouse
`onmousemoved(fn)`	Occurs when the user **moves** the mouse in any direction
`onmousedragged(fn)`	Occurs when the user **both presses and moves** their mouse