Project Description

You have the following two options for the project:

1) you can complete and submit the project as a group of two students. In that case, you will complete both part 1 and part 2 as outlined below. You must also submit a contribution form confirming (a template will be provided and linked in the assignment dropbox) that each of the group-mates roughly contributed equally to different aspects of the project such as the code design, documentation, implementation and testing.

2) alternatively, you can work individually and submit your own implementation. In that case, you will complete part 1 only.

For groups, you will pick your own group-mate, see announcements on updates on how to register your group.

Please decide early, as you will not be able to change whether you are going to work individually or a group after Friday March 26.

Fully follow the project specification.

The project is due April 8, by the end of the day (the last day of classes). Please plan ahead, no late submission will be accepted.

Academic honesty and standard:

As always, remind yourself of the **UBC academic honesty and standard (http://www.calendar.ubc.ca/Vancouver/index.cfm?tree=3,286,0,0)** and submit work that is yours or your own group's work only. Do not share/discuss your design thinking or any part of your code with anybody outside your own group. Using any outside snippet of code that is neither from the course materials posted on Canvas, nor the Python's official online documentation is not allowed. Any code snippet or code design from the the online documentation must be properly cited.

==========

Part 1: Implementing a multithreaded game with graphic user interface

We are to implement a simplified snake game in this project. Here are some characteristics of the program:

- Python's `multithreading` module is used to implement multithreading in our program.
- Python's `Tkinter` module (**https://docs.python.org/3/library/tkinter.html (https://docs.python.org/3/library/tkinter.html)**) is used to create the graphic user interface for our program. It doesn't affect the code you would write directly, but FYI, under the hood Tkinter uses

event-driven programming to implement concurrency. We have already used `turtle` (which is a toy and educational GUI module). In contrast, `Tkinter` is a popular and much more capable Python GUI module that comes as a part of Python's standard library (so no need to install anything additional).

- Python's `queue` module (**https://docs.python.org/3/library/queue.html (https://docs.python.org/3/library/queue.html)**) is used as an excellent thread-safe multi-producer, multi-consumer queue (so we would not need any additional synchronization mechanism). Different tasks in our program are put in the FIFO queue and handled accordingly.
- The game is played by the four arrow keys of the keyboard bound to the corresponding directions, e.g. left arrow for the left direction, down arrow for the downward direction, ... This is done by the `Tkinter`'s provision of binding keys.
- The game is a variety of the snake game (**https://en.wikipedia.org/wiki/Snake_(video_game_genre) (https://en.wikipedia.org/wiki/Snake_(video_game_genre))** which is a classic arcade game dating back to the early generations of computer games. The gamer controls a snake using the keyboard's arrow keys and the objective is to collect as many preys as possible (so the displayed game score is the number of collected preys). Each prey is stationary and is placed randomly on the game windows. As the snake feeds, it become longer (one additional block each time), progressively making it harder to control without losing.
- The game is over if the snake hits any of the walls or if it hits itself (technically biting itself).
- Many game related values are set using constants. For example, the speed of the game (the snake speed) is set to 0.15 (sec). Of course, you can adjust that during your debugging (if you need to slow it down or speed it up for your amusement). There are a few magic numbers left in the game but it should rather be clear what they are.

Please read the docstring for each of the methods or portion of the program that is left for you to implement, complete the `project_part1.py` file and submit to the associated submission dropbox by the deadline.

Some notes:

- The snake's current position and length is represented by the field `snakeCoordinates` of the `Game` class. It is a list of tuples. Each tuple is an (x, y) coordinate. In the GUI, the snake is represented by the `snakeIcon` which is a Tkinter canvas line with the width `SNAKE_ICON_WIDTH` based on the `snakeCoordinate` list.
- The new prey is created at the beginning of the game and each time the snake captures one. In the GUI, the prey is represented by the `preyIcon` which is a Tkinter canvas rectangle. The method createNewPrey randomly generates an x and a y, and then sets the prey `rectnagleCoordinates` as (x - 5, y - 5, x + 5, y + 5) which is added as a task to the queue. On the GUI, the prey is a 10 by 10 rectangle.
- The queue is a FIFO queue of all the pending tasks that each portion of the program created and added to the queue. Each item in the queue is a dictionary.

    - The keys in the dictionary is one of `"game_over"`, `"move"`, `"prey"`, or `"score"`.
    - The value depends on the key.
        - The value for the key `"game_over"` is a Boolean value (`True` or `False`)
        - The value for the key `"score"` is an integer representing the new score.
        - The value for the key `"prey"` is the new `rectangleCoordinates` of the form (x1, y1, x2, y2).
        - The value for the key `"move"` is the `snakeCoordinates` which is a list of tuples.
- The implementations of the *GUI* and the *Queue* classes, as well as the main thread is given to you. You are to complete the implementation of some of the methods in the *Game* class. Those two types and their operations must be used by the *Game* class whenever needed.
- Note that the * in an argument of the form *var in a function call is an *unpacking operator*. **From the documentation (https://docs.python.org/3/reference/expressions.html#expression-lists)**: "An asterisk * denotes iterable unpacking. Its operand must be an iterable. The iterable is expanded into a sequence of items, which are included in the new tuple, list, or set, at the site of the unpacking." Also in **PEP 448 (https://peps.python.org/pep-0448/)**. (A note that * has different meaning for the method's formal parameter as opposed to a function argument.)

Here is a video of the sample run of the game:

https://youtu.be/kFzkDpd5qNo (https://youtu.be/kFzkDpd5qNo)

(https://youtu.be/kFzkDpd5qNo)

For a general but much longer and sped-up demo of a generic snake game see here:
https://en.wikipedia.org/wiki/Snake_(video_game_genre)#/media/File:Snake_can_be_completed.gif
(https://en.wikipedia.org/wiki/Snake_(video_game_genre)#/media/File:Snake_can_be_completed.gif)

The amount of code that you will be writing is not much, but you need to understand the problem and figure out how to work with the provided classes and code to complete the project. This is a common situation when you join a company in real-life and need to continue on and complete a project based on present code and client requirements.

You are allowed to add local functions to any of the methods but do not add any new type, or new methods.

Make sure that your code is readable and add comments whenever needed to explain your code.

Here is the code template:

```
# Name:
# Student number:

"""
    This program implements one variety of the snake
    game (https://en.wikipedia.org/wiki/Snake_(video_game_genre))
"""

import threading
import queue        #the thread-safe queue from Python standard library

from tkinter import Tk, Canvas, Button
import random, time

class Gui():
    """
        This class takes care of the game's graphic user interface (gui)
        creation and termination.
    """
    def __init__(self, queue, game):
        """
            The initializer instantiates the main window and
            creates the starting icons for the snake and the prey,
            and displays the initial gamer score.
        """
        #some GUI constants
        scoreTextXLocation = 60
        scoreTextYLocation = 15
        textColour = "white"
        #instantiate and create gui
        self.root = Tk()
        self.canvas = Canvas(self.root, width = WINDOW_WIDTH,
            height = WINDOW_HEIGHT, bg = BACKGROUND_COLOUR)
        self.canvas.pack()
        #create starting game icons for snake and the prey
        self.snakeIcon = self.canvas.create_line(
            (0, 0), (0, 0), fill=ICON_COLOUR, width=SNAKE_ICON_WIDTH)
        self.preyIcon = self.canvas.create_rectangle(
            0, 0, 0, 0, fill=ICON_COLOUR, outline=ICON_COLOUR)
        #display starting score of 0
        self.score = self.canvas.create_text(
            scoreTextXLocation, scoreTextYLocation, fill=textColour,
            text='Your Score: 0', font=("Helvetica","11","bold"))
        #binding the arrow keys to be able to control the snake
        for key in ("Left", "Right", "Up", "Down"):
            self.root.bind(f"<Key-{key}>", game.whenAnArrowKeyIsPressed)

    def gameOver(self):
        """
            This method is used at the end to display a
            game over button.
        """
        gameOverButton = Button(self.canvas, text="Game Over!",
            height = 3, width = 10, font=("Helvetica","14","bold"),
            command=self.root.destroy)
        self.canvas.create_window(200, 100, anchor="nw", window=gameOverButton)


class QueueHandler():
    """
        This class implements the queue handler for the game.
    """
    def __init__(self, queue, gui):
        self.queue = queue
        self.gui = gui
        self.queueHandler()
```

```
    def queueHandler(self):
        '''
            This method handles the queue by constantly retrieving
            tasks from it and accordingly taking the corresponding
            action.
            A task could be: game_over, move, prey, score.
            Each item in the queue is a dictionary whose key is
            the task type (for example, "move") and its value is
            the corresponding task value.
            If the queue.empty exception happens, it schedules
            to call itself after a short delay.
        '''
        try:
            while True:
                task = self.queue.get_nowait()
                if "game_over" in task:
                    gui.gameOver()
                elif "move" in task:
                    points = [x for point in task["move"] for x in point]
                    gui.canvas.coords(gui.snakeIcon, *points)
                elif "prey" in task:
                    gui.canvas.coords(gui.preyIcon, *task["prey"])
                elif "score" in task:
                    gui.canvas.itemconfigure(
                        gui.score, text=f"Your Score: {task['score']}")
                self.queue.task_done()
        except queue.Empty:
            gui.root.after(100, self.queueHandler)


class Game():
    '''
        This class implements most of the game functionalities.
    '''
    def __init__(self, queue):
        """
           This initializer sets the initial snake coordinate list, movement
           direction, and arranges for the first prey to be created.
        """
        self.queue = queue
        self.score = 0
        #starting length and location of the snake
        #note that it is a list of tuples, each being an
        # (x, y) tuple. Initially its size is 5 tuples.
        self.snakeCoordinates = [(495, 55), (485, 55), (475, 55),
                                 (465, 55), (455, 55)]
        #initial direction of the snake
        self.direction = "Left"
        self.gameNotOver = True
        self.createNewPrey()

    def superloop(self) -> None:
        """
            This method implements a main loop
            of the game. It constantly generates "move"
            tasks to cause the constant movement of the snake.
            Use the SPEED constant to set how often the move tasks
            are generated.
        """
        SPEED = 0.15     #speed of snake updates (sec)
        while self.gameNotOver:
            #complete the method implementation below
            pass #remove this line from your implemenation

    def whenAnArrowKeyIsPressed(self, e) -> None:
        """
            This method is bound to the arrow keys
            and is called when one of those is clicked.
```

```
        It sets the movement direction based on
        the key that was pressed by the gamer.
        Use as is.
    """
    currentDirection = self.direction
    #ignore invalid keys
    if (currentDirection == "Left" and e.keysym == "Right" or
        currentDirection == "Right" and e.keysym == "Left" or
        currentDirection == "Up" and e.keysym == "Down" or
        currentDirection == "Down" and e.keysym == "Up"):
        return
    self.direction = e.keysym

def move(self) -> None:
    """
        This method implements what is needed to be done
        for the movement of the snake.
        It generates a new snake coordinate.
        If based on this new movement, the prey has been
        captured, it adds a task to the queue for the updated
        score and also creates a new prey.
        It also calls a corresponding method to check if
        the game should be over.
        The snake coordinates list (representing its length
        and position) should be correctly updated.
    """
    NewSnakeCoordinates = self.calculateNewCoordinates()
    #complete the method implementation below


def calculateNewCoordinates(self) -> tuple:
    """
        This method calculates and returns the new
        coordinates to be added to the snake
        coordinates list based on the movement
        direction and the current coordinate of
        head of the snake.
        It is used by the move() method.
    """
    lastX, lastY = self.snakeCoordinates[-1]
    #complete the method implementation below


def isGameOver(self, snakeCoordinates) -> None:
    """
        This method checks if the game is over by
        checking if now the snake has passed any wall
        or if it has bit itself.
        If that is the case, it updates the gameNotOver
        field and also adds a "game_over" task to the queue.
    """
    x, y = snakeCoordinates
    #complete the method implementation below

def createNewPrey(self) -> None:
    """
        This methods randomly picks an x and a y as the coordinate
        of the new prey and uses that to calculate the
        coordinates (x - 5, y - 5, x + 5, y + 5).
        It then adds a "prey" task to the queue with the calculated
        rectangle coordinates as its value. This is used by the
        queue handler to represent the new prey.
        To make playing the game easier, set the x and y to be THRESHOLD
        away from the walls.
    """
    THRESHOLD = 15   #sets how close prey can be to borders
    #complete the method implementation below
```

```
if __name__ == "__main__":
    #some constants for our GUI
    WINDOW_WIDTH = 500
    WINDOW_HEIGHT = 300
    SNAKE_ICON_WIDTH = 15

    BACKGROUND_COLOUR = "green"
    ICON_COLOUR = "yellow"

    gameQueue = queue.Queue()     #instantiate a queue object using python's queue class

    game = Game(gameQueue)        #instantiate the game object

    gui = Gui(gameQueue, game)    #instantiate the game user interface

    QueueHandler(gameQueue, gui)  #instantiate our queue handler

    #start a thread with the main loop of the game
    threading.Thread(target = game.superloop, daemon=True).start()

    #start the GUI's own event loop
    gui.root.mainloop()
```

==========

Part 2: UDP Pinger (client and server)

This is a socket programming program for UDP using Python's socket module as discussed in the lecture (**https://docs.python.org/3/library/socket.html (https://docs.python.org/3/library/socket.html)**).

Write one python3 file as the client program (`clinet.py`) and one python3 file as the server program (`server.py`). Please submit the two files to the associated submission dropbox by the deadline. The code must include sufficient comment statements. No email or late submission is accepted, please do plan ahead accordingly.

The two processes normally run on two separate host computers on the Internet, but obviously you should develop/debug them both running on one machine (your computer). So, you can use **localhost** (or alternatively 127.0.0.1) as the hostname. **127.0.0.1** (**https://en.wikipedia.org/wiki/Localhost (https://en.wikipedia.org/wiki/Localhost)**) is called the loopback IP address and is used to identify the same machine without the need to know any machine's real network interface IP addresses.

The actual communication between the two processes is very straightforward. The client sends a simple greeting message including the message number, for example “PING 1 - hello world”) to the server using UDP. Second message would be for example “PING 2 - hello world”, etc.

The server listens on the UDP port 12000 and responds back by simply replacing the greeting with "ditto" and then sends back the message. For example, in response to the above message, it echoes back "PING 1 - ditto").

The client then calculates the Round Trip Time (RTT) and prints out the RTT result, and the received message itself (since it has the message number). RTT is a common delay measure and it is the delay from the time the client sent the ping message and the time it received the echoed response message.

The client then repeat the above 4 more times (that is, we send 5 ping messages and calculate the corresponding 5 RTT values).

- To simulate the variability of the RTT delay, the server program must randomly wait for some time between 5 to 50 ms before responding back.
- To simulate packet loss (that is, the messages that are lost for any reason along the way in an actual IPC communication over the Internet), the server must randomly ignore a message (i.e. not responding back) with a probability of 10%. If the client does not hear back within 1 sec for a sent message it must print a 'request timed out' message.
- You may begin by using the code provided in the slides as a starting template for the programs.

To implement variable delay and loss and calculating the delay, use the methods in the *random* module (to create random variables) and *time* module (for timing related functionalities) in Python.