Maze Solver

Introduction
In this activity, you are going to apply all that you have learned about python and combine it with an algorithm similar to Wavefront to find solutions to mazes.

1. Download the starter code from github. Don't try to run main.py yet … you'll just get a bunch of errors until you complete the lab.

Installing packages
Python has packages of code available to assist with many tasks. One of these packages is called Pillow, and it provides objects and methods to read and manipulate images. Unfortunately, this package isn't among python's default libraries, so you must install it yourself. In a typical environment, you would only have to install it once. However, in repl.it, every time you reload the browser it needs to be re-installed. To automate this, the .replit file has been modified.

2. Confirm that you have a file called **`.replit`** (Note that the file must start with a period.) and that its contents are as follow:
   ```
   language="python3"
   run="pip install Pillow;python3 main.py"
   ```

   The first line specifies which language will be used; the second line ensures that the Python Imaging Library (PIL) is installed.

Test-driven development
There is a style of programming called test-driven development. In this style of programming, the *first* thing you do is write code to test a feature *that you have not implemented*. When you run the test, it *should* fail, because the feature has not been implemented yet. Next, modify the code to implement the feature and re-run the test. Now, the test should pass. There are several benefits to this. First, it forces you to think abstractly first: What's the input and what's the output. The details of implementation don't matter when you're writing the test. This can help you identify questions that need to be answered before you begin writing the code.

Consider the following situation: I want a method that returns a list of all the values between two parameters. What is a reasonable test?

`get_values( 0, 5 )` should return … what? Well, 1, 2, 3, and 4. But should I include the first parameter? the last parameter? You should know that answer before you start the implementation.

Another benefit of test-driven development is that you have tests for much of your code, which can be rerun anytime anything changes … if you need to change the code later on, perhaps to make it more efficient or more readable, the original tests should still pass after your modifications. This is much easier than manually testing everything.

For this project, I have written tests for you. To get full credit, *you must make the changes described in this worksheet and in the comments of the python files, and also make sure all the tests (as I have written them!) pass*. If you change the tests so that your code passes, you won't get the points. **You must change the code so that the tests pass.**

In a real production environment (or a more complex personal project) the tests would be in separate files. However, for our purposes, the tests are in the same file as the class. To run them, just run the file as if it were a stand-alone program. For example, to run the tests in dimension.py, type:

```
python dimension.py
```

If a test passes, it will say, "OK." If it fails, it will tell you what the inputs were, what the expected output was, and what the actual output was.

For example, before completing the required modifications to dimension.py:

```
python dimension.py
OK
OK
temp=Dimension(1,1); expected temp.width to be 1, got 0
temp=Dimension(1,1); expected temp.height to be 1, got 0
temp=Dimension(2,3); expected temp.width to be 2, got 0
temp=Dimension(2,3); expected temp.height to be 3, got 0
temp=Dimension(3,2); expected temp.width to be 3, got 0
temp=Dimension(3,2); expected temp.height to be 2, got 0
```

Interestingly, a couple of test cases pass before making any modifications. Sometimes this is coincidence, sometimes it is because the implementation is partially correct.

After you make the changes to dimension.py, rerun the tests to make sure everything is correct.

```
python dimension.py
OK
OK
OK
OK
temp=Dimension(2,3); expected temp.width to be 2, got 3
temp=Dimension(2,3); expected temp.height to be 3, got 2
temp=Dimension(3,2); expected temp.width to be 3, got 2
temp=Dimension(3,2); expected temp.height to be 2, got 3
```

Things are closer, but not quite right. If you look closely, you might notice that width and height values are switched. Tests that passed happened to use the same values for width and height. Often a buggy method will work for some inputs, but not all. Obviously, you want your code to work for all inputs.

After fixing that error and re-running the tests:

```
python dimension.py
OK
OK
OK
OK
OK
OK
OK
OK
```

You're ready to move on!

Correct the Dimension class

3. Open dimension.py and run the test by typing python dimension.py in the console.
   How many tests pass initially? ____________________
   How many tests fail initially? ____________________

4. On what line does the constructor for the Dimension class begin?

   LINE ____

5. Modify the constructor for the Dimension class as described in the comments. Make sure the tests pass before you move on!

Correct the Point class

6. Now open point.py, which contains the Point class. A point represents a point with screen coordinates. Like Cartesian coordinates, screen coordinates use an x and y value. Unlike Cartesian coordinates, 0, 0 is in the top left rather than center. As x increases, you move right; as y increases, you move down.
7. is_in_rect() should determine if this Point object is inside a specified rectangle. The rectangle is described by a tuple (remember, a tuple is similar to a list, but its values cannot be changed) that consists of two Point objects: The top left corner of the rectangle and the bottom right corner of the rectangle. is_in_rect() should return true if this point lies on the border of that rectangle or is within that rectangle.

8. Consider the following grid. If the rectangle is described by the following tuple:
   `( Point(1,1), Point(3,2) )`
   then the shaded squares below illustrate which points are within that rectangle.

Point(0,0)	Point(1,0)	Point(2,0)	Point(3,0)	Point(4,0)
Point(0,1)	Point(1,1)	Point(2,1)	Point(3,1)	Point(4,1)
Point(0,2)	Point(1,2)	Point(2,2)	Point(3,2)	Point(4,2)
Point(0,3)	Point(1,3)	Point(2,3)	Point(3,3)	Point(4,3)

9. Currently `is_in_rect()` always returns `True`. Fix the method so that it works as described in the comments. *Remember to run the tests before and after to make sure everything is correct.*
10. Find the `get_rect_around()` method further down on the page.

    This method works correctly as it is, but illustrates some issues that you should be aware of. First, it uses "integer division."

    In integer division, if the division results in an answer that includes a decimal part, the decimal part is dropped and the whole number is effectively rounded to the smallest whole number, regardless of how large the decimal part was. For example, 3//2 is 1 (3 / 2 is 1.5; the decimal part is dropped and the number rounded down to 1). An interesting wrinkle occurs when performing integer division with a negative denominator. 3//-2 results in -2, because 3/-2 is -1.5. You drop the .5, but round to the smallest whole number … -2 is smaller than -1, so the answer is -2.

    Also recall that modulus (%) returns the remainder. With that in mind, given the areas on the following page, what will the values for left, top, right and bottom be?

11. Feel free to type the expressions into python to confirm your results.
    For example, what does python say the result of `3 // -2 + 3 % 2` is?

`area = Dimension(3,3)`	left: -1 top: -1 right: 2 bottom: 2
`area = Dimension(1,1)`	left: ____ top: ____ right: ____ bottom: ____
`area = Dimension(3,5)`	left: ____ top: ____ right: ____ bottom: ____
`area = Dimension(4,4)`	left: ____ top: ____ right: ____ bottom: ____

Read, understand, and modify the Maze class

12. Open maze.py. This file defines the Maze class, which is where the bulk of the work happens. The key elements here are the constructor, where we identify the maze file and some other details like where the start and end points are, and the solve() method, which figures out the shortest path through the maze, draws it, and saves it as a new file.

    As you may recall, there are a lot of steps to implement the wavefront algorithm, and it can quickly become overwhelming. However, by using abstraction to hide implementation details at appropriate levels, it becomes far more manageable. Also, it becomes possible to write tests for individual abstractions (methods). If you know all the individual parts are working, it will make you more confident that the more complex algorithm that combines those parts is working[1].

13. On what line does the constructor for the Maze class begin?

    LINE ___

14. Assume the maze image is a 10x10 grid. Answer the following questions, assuming no start or end points are specified in the constructor.

    What will the start point be after the constructor runs? `Point( __ , __ )`

    What will the end point be after the constructor runs? `Point( __ , __ )`

15. Read the comments for the constructor. Explain the point of the optional crop parameter.

    THE CROP PARAMETER IS USED TO ___

16. Now find the `solve()` method. How many steps occur at this level of abstraction? Does abstraction help or hinder understanding of how the `solve()` algorithm works?

17. Find the `initialize_wave()` method and read the comments that describe how it works. Now, based on what you learned from the comments (really, I'm not kidding... *Read. The. Comments.*), use the multiplication operator to initialize a list of 100 elements with values of 1.

[1] That's not guaranteed, though! Sometimes the individual "units" are all fine, but problems arise when they are integrated. Perhaps there are flaws in the logic that integrates them, or even worse, there are "emergent behaviors" that occur only when the individual items interact. When multiple units share global variables, for example, issues will arise that won't appear in individual unit tests. Other emergent behaviors might include individual tests completing their tasks within memory and time constraints but, when taken as a whole, the overall product exceeds memory or time requirements. In this project, many of those problems are avoided by *not using global variables*. I rarely use global variables in production code for a client. I want to say "never," but as we all know, as soon as you say never, an exception arises. To the best of my recollection, however, I never have used global variables in production code. Regarding time and memory constraints, we have none that are relevant here.

18. Look through the methods used in initialization: `initialize_wave()`, `mark_blocked_areas()`, `is_blocked()`, and `area_is_blocked()`. Indicate with a T or F whether you believe the following statements are true or false.

Comments before the method indicating its overall goal make the code easier to understand.	________
Comments within the method itself, indicating what is going on at that point in the algorithm, make the code easier to understand.	________
Descriptive (and accurate) method names make the code easier to understand.	________
Descriptive (and accurate) variable names make the code easier to understand.	________
Abstraction makes the code easier to understand.	________

19. Find the `calculate_wave()` method and read the comments that describe how a queue works. Use a real-world analogy other than that of the grocery-store check-out line, to describe what a queue is and how it works.

____

____

____

20. Find the connect() method. It receives as parameters a parameter called "draw," which is a graphics context for the maze image. Read the documentation on how to draw a line at http://effbot.org/imagingbook/imagedraw.htm , then complete the method so that it draws a line between pointA and pointB. For this to work correctly, you must do a couple of things: 1. Only try to draw the line if neither pointA or pointB is None. 2. the values for pointA and pointB are Point objects, which is a class we created … not a class the PIL package is familiar with. You need to convert the x and y coordinates of the points to something PIL will recognize.
21. If you've implemented all these methods correctly, you should now be able to run main.py. It should solve the three test mazes. If you want to load your own mazes, you need to open them in an image editor to find the start and end points (specified in pixels), as well as the crop values.
22. Submit this completed worksheet as well as a link to your repl to the dropbox.