Name: _____

Please answer the following questions within the space provided on the online template. Format your solutions as well as you are able within the online text editor. While you are not required to document your code here, comments may help me to understand what you were trying to do and thus increase the likelihood of partial credit should something go wrong. If you get entirely stuck somewhere, explain in words as much as possible what you would try.

Each question clearly shows the number of points available and should serve as a rough metric to how much time you should expect to spend on each problem. You can assume that you can import any of the common libraries we have used throughout the semester thus far.

The exam is partially open, and thus you are free to utilize:

- The text
- Your notes
- Class slides
- Any past work you have done as part of sections, problem sets, or projects, provided it has been uploaded, and you access it through GitHub.
- The in-Canvas calculator on problems that offer it.

While you are allowed to use a computer for ease of typing and accessing the above resources, you are prohibited from accessing and using any editor or terminal to run your code. Visual Studio Code or any similar editor should never be open on your system during this exam. Additionally, you are prohibited from accessing outside internet resources beyond the webpages described above. *Your work must be your own on this exam, and under no conditions should you discuss the exam or ask questions to anyone but myself.* Failure to abide by these rules will be considered a breach of Willamette's Honor Code and will result in penalties as set forth by Willamette's academic honesty policy.

Please sign and date the below lines to indicate that you have read and understand these instructions and agree to abide by them. *Failure to abide by the rules will result in a 0 on the test.* Good luck!!

_____ _____

Signature Date

(10) 1. **Tracing Functions**

The code below defines three different function definitions:

```
def mystery(x):

    def puzzle(x, y=5):
        return x * y

    def enigma(y):
        return y ** x

    return enigma(puzzle(x=2)) + enigma(puzzle(3,x))

if __name__ == '__main__':
    print(mystery(3))
```

For each function, every time that function would *return* a value, indicate what that value would be. If the function returns multiple times, put each value on a new line, in the order they would be returned.

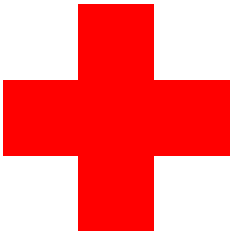
mystery	puzzle	enigma
1729	10	1000
	9	729

Solution: The final output value would be 1729.

(20) 2. **Interactive Graphics**

Write a graphical program that using the PGL library to accomplish the following:

1. Creates the below cross as a `GCompound` containing two filled rectangles:



   The color of the cross should be red, as in the emblem of the International Red Cross. The horizontal rectangle should be 60×20 pixels in size and the vertical one should be 20×60. They cross at their centers.

2. Add the cross to the graphics window so that it appears in the exact center of the window.

3. Moves the cross at a speed of 2 pixels every 20 milliseconds in a constant random direction, which is specified as a random real number between 0 and 360 degrees.

4. Every time you click *inside the cross*, its direction changes to some new constant random direction–again chosen as a random real number between 0 and 360 degrees–but its speed remains the same. Clicks outside the cross have no effect.

If you were actually working on a program with this setup, you would presumably supply some means of making it stop, such as when the cross moves off the screen. For this problem, just have the program run continuously without worrying about objects moving off-screen or how to stop the timer.

Solution:

```
from pgl import GWindow, GCompound, GRect
from random import uniform

WIDTH = 600
HEIGHT = 600

def create_cross():
    """ Creates the cross GCompound """
    c = GCompound()
    hrect = GRect(60, 20)
    hrect.set_filled(True)
    hrect.set_color("red")
    vrect = GRect(20, 60)
```

```
    vrect.set_filled(True)
    vrect.set_color("red")

    c.add(hrect, -30, -10)
    c.add(vrect, -10, -30)
    return c

def step():
    """ Animates the cross movement """
    cross.move_polar(2, gw.heading)

def click_action(event):
    """ Changes the heading is mouse clicked
    within cross object. """
    mx, my = event.get_x(), event.get_y()
    if cross.contains(mx, my):
        gw.heading = uniform(0, 360)


gw = GWindow(WIDTH, HEIGHT)
cross = create_cross()
gw.add(cross, WIDTH/2, HEIGHT/2)
gw.heading = uniform(0, 360)

gw.set_interval(step, 20)
gw.add_event_listener("mousedown", click_action)
#click would be fine for the event as well
```

(20) 3. **Working with Arrays**

There is a simple mobile game that I discovered a while back that relies on placing red and blue squares onto a grid. The puzzle lies in the fact that:

- no row or column can ever be duplicated. That is to say, you can't have any two rows with the same sequence of red and blue squares, or any two columns with the same sequence of red and blue squares.
- there can never be three adjacent colors of one type in any row or in any column.

Generally the game starts with some initial population of squares filled in as either red or blue, and then you, the player, need to fill in the rest according to the rules. While the strategies to implement solving the puzzle are beyond this class, we absolutely have the necessary machinery to *check* a given puzzle to ensure it obeys all the rules. To that end, in this problem you should write two predicate functions:

```
def no_duplicate_rows(puzzle):

def no_3petes_in_rows(puzzle):
```

Each will take as input the `puzzle`, which will be a 2D array with strings (`"R"` or `"B"`) as the elements. `no_duplicate_rows` should check all the rows in the puzzle and return `False` if any two rows are exact copies. Otherwise, it should return `True`. `no_3petes_in_rows` should check the elements of each row to ensure that three red or blue squares are never adjacent. If this is true for *all* the rows, then `no_3petes_in_rows` should return `True`, otherwise it should return `False`.

I'm only asking you to check the rows here, despite the fact that columns are also part of the game rules. My reasoning is that you already wrote a function to rotate a 2D array by 90 degrees in Imageshop. So presumably, after checking the rows, you could just rotate the puzzle by 90 degrees (so that the columns are now rows) and then check the columns using the exact same functions. *You do not need to do this here! I am merely pointing out that it would be simple and saves having to write more functions.*

As an example, the below 3×3 puzzle is valid and should thus return `True` when passed into both functions.

```
puzzle = [ ["R", "B", "R"],
           ["R", "B", "B"],
           ["B", "R", "R"] ]
```

whereas this 4×4 puzzle would fail the duplicate rows check and the 3-pete check:

```
bad_puzzle = [  ["R", "B", "R", "B"],
                ["R", "B", "B", "R"],
                ["R", "R", "R", "B"],
                ["R", "B", "B", "R"] ]
```

Solution: There are many ways these could be approached, but here are two:

```
def no_duplicate_rows(puzzle):
    """ Checks that no two rows are exact
    duplicates of one another. """
    for i in range(len(puzzle)):
        row = puzzle[i]
        if row in puzzle[i+1:]:
            return False
    return True


def no_3petes_in_rows(puzzle):
    """ Checks that 3 blue or red squares
    never appear adjacently in a row. """
    for row in puzzle:
        cons_str = ""
        for letter in row:
            if cons_str == "" or letter == cons_str[-1]:
                cons_str += letter
            else:
                cons_str = letter
            if cons_str == "RRR" or cons_str == "BBB":
                return False
    return True
```