Name: _____

Please answer the following questions within the space provided on the following pages. Should you need more space, you can use scratch paper, but clearly label on the scratch paper what problem it corresponds to. 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 printed portions of:

- The text
- Your notes
- The slides
- Any past work you have done for labs, problem sets, or projects

Computers and internet capable devices are prohibited. *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

Question:	1	2	3	4	Total
Points:	6	8	14	20	48
Score:					

(6) 1. **Evaluating Python:** For each of the below expressions, write out **both** the value of the expression and what type of object that value represents.

(a) `1 * 2 / (3 * 4 % 5) + 4 - 3 ** 2 % 1`

> **Solution:** `5.0`, a `float`

(b) `len("October"[7 // 2]) > int("31") and (2022 != "2022")`

> **Solution:** `False`, a `bool`

(8) 2. **Understanding Python:** What output would the following program print?

```
def mystery(a=4, b="skeletons"):
    c = ""
    for i in range(a-1):
        c += b[enigma(i*10)]
    return c

def enigma(a):
    a = a // len(b) + 1
    return len(c) - a

b = "Happy"
c = "Halloween"
print(mystery(b=c))
```

If you want any chance at partial credit, be sure to show your work clearly.

Solution: This would print `'neo'` at the end. Write out your different scopes so that you can carefully track which variable will be looked up in different situations.

(14) 3. **Writing Python:** Many words in the English language have doubled up letter combinations. For example, the word `committee` has three such pairings: the `m`, the `t` and the `e`. Your task in this problem is to write a function called `remove_dups` that takes a string as an argument and returns a new string where each adjacent duplicate letter has been replaced with a single copy of that letter. So in our above example, calling `remove_dups("committee")` would just return `"comite"`. Note that it is only *adjacent* duplicate letters that get simplified: `remove_dups("decided")` would return `"decided"`, without any changes, despite the fact that it contains three `d`'s and two `e`'s. Your function should, however, work with any number of adjacent repeat letters. So `remove_dups("brrr")` should return `"br"`.

Solution: One implementation might look like:

```
def remove_dups(word):
    new = ""
    prev = ""
    for letter in word:
        if letter != prev:
            new += letter
        prev = letter
    return new
```

(20) 4. **Graphics in Python:** The classic arcade game Frogger involved moving a frog around a grid while attempting to dodge horizontally moving traffic and other obstacles. In this problem, your goal is to create a very rough and approximate replica of that experience. To do so, you will need to focus on two main objects: the frog and the traffic.

The Frog:

Here you will represent the frog as a filled green circle with a diameter equal to the provided `SQ_SIZE` constant. The frog should start just touching the bottom of the window and centered horizontally on the window. To simplify the movement of the frog, you will only be letting the frog move up and down, not the left or right also available in the original game. The premise is that if you click anywhere in the window above the center of the frog, it should move upwards by a single `SQ_SIZE` distance, whereas if you click anywhere below the center of the frog, it should move downwards by a single `SQ_SIZE` distance.

The Traffic:

In the interest of keeping things simple, you will only include a single "car" in your program here. The car's dimensions are given by the constants `CAR_WIDTH` and `CAR_HEIGHT` and it should be represented as a filled red rectangle. The initial location of the upper left corner of the car is given by the constants `CAR_INIT_X` and `CAR_INIT_Y`. Note that this starting location is technically offscreen. To have your car "drive" across the screen, it should be moved to the left by `CAR_DX` pixels every 30 milliseconds. If the *entire* car has moved off the left side of the window, you should reset the position of the car back to its starting x location, but give it a new random y position, such that the entire height of the car will be visible once it drives on-screen. In this way you can have a continuous stream of cars (1 at a time) driving from right to left across the screen at different heights.

You **do not need to implement collisions** between the frog and the car for this problem, though clearly you would want to if you decided to flesh this program out a bit more. Nor do you need to worry about the victory condition. Once you have a frog that responds correctly to mouse clicks and a car that drives across the screen repeatedly at different heights, you have met all the requirements.

The next page lists the imports and constants given at the start of the program and then leaves you room to add your code.

```
from pgl import GWindow, GRect, GOval
import random

SQ_SIZE = 50  # Size of the game grid and width of the frog
GW_WIDTH = 5 * SQ_SIZE  # Width of the game window
GW_HEIGHT = 10 * SQ_SIZE  # Height of the game window
CAR_WIDTH = 2 * SQ_SIZE  # Width of the car
CAR_HEIGHT = SQ_SIZE  # Height of the car
CAR_INIT_X = GW_WIDTH  # Initial X coordinate of the car
CAR_INIT_Y = SQ_SIZE * 4  # Initial Y coordinate of the car
CAR_DX = 2  # Speed of the car (amount moved per step)
```

Solution: My implementation looked something like this:

```
def create_frog():
    sq = GOval(0, 0, SQ_SIZE, SQ_SIZE)
    sq.set_filled(True)
    sq.set_color("green")
    return sq

def create_car():
    sq = GRect(0, 0, CAR_WIDTH, CAR_HEIGHT)
    sq.set_filled(True)
    sq.set_color("red")
    return sq

def click_action(e):
    my = e.get_y()
    if my > frog.get_y() + SQ_SIZE / 2:
        frog.move(0, SQ_SIZE)
    elif my < frog.get_y() + SQ_SIZE / 2:
        frog.move(0, -SQ_SIZE)

def step():
    car.move(-CAR_DX, 0)
    if car.get_x() + CAR_WIDTH < 0:
        new_y = random.randint(0,GW_HEIGHT-CAR_HEIGHT)
        car.set_location(CAR_INIT_X, new_y)

gw = GWindow(GW_WIDTH, GW_HEIGHT)
frog = create_frog()
car = create_car()
gw.add(frog, GW_WIDTH / 2 - SQ_SIZE / 2, GW_HEIGHT - SQ_SIZE)
gw.add(car, CAR_INIT_X, CAR_INIT_Y)
gw.add_event_listener("mousedown", click_action)
gw.set_interval(step, 30)
```