# Practice Final 2

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
- Online 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 | 5 | 6 | Total |
|---|---|---|---|---|---|---|---|
| Points: | 10 | 10 | 20 | 15 | 10 | 20 | 85 |
| Score: | | | | | | | |

(10) 1. **Short Answer**

(a) Suppose that the function `conundrum` is defined as follows:

```
def conundrum():
    array = [ ]
    for i in range(6):
        array.append(0)
        for j in range(i, 0, -1):
            array[j] += j
    return array
```

Work through the function carefully and indicate the value of each of the elements of the array returned by a call to `conundrum`.

**Solution:** A new entry is appended each iteration, so there will be 6 values in the list. Each value starts at 0 and then has its index added to it 6-index times. So index 0 has 0 added to it 6 times, index 1 has 1 added to it 5 times, index 2 has 2 added to it 4 times, etc. So the final list looks like:

```
[0, 5, 8, 9, 8, 5]
```

(b) What value is printed if you call the function `example` in the following code?

```
class MyClass:
    def __init__(self, x):
        def f(y):
            return 2 * x + y
        self.g = f

    def test(self, x):
        return self.g(x + 6)

def example():
    value = MyClass(10)
    print(value.test(-4))
```

**Solution:** The function `f` is defined within the constructor, and thus has values defined within the constructor as part of its closure. Thus, when `test` is run the value of $-4 + 6 = 2$ is passed in as `y` to `f`, resulting in a printed value of 22.

(10) 2. **Simple Python** Write a Python program that reads integers that the user inputs on the console, ending when the user enters a blank line. At that point, the program should print out two lines, one showing the average of the odd numbers and one showing the average of the even numbers. Final decimals should always show two decimal places. An example run might look like:

```
〉 python prob2.py

Enter integers:
  ? 3
  ? 1
  ? 4
  ? 1
  ? 5
  ? 9
  ? 8
  ? 2
  ?
The average of the odd numbers is 3.80
The average of the even numbers is 4.67
```

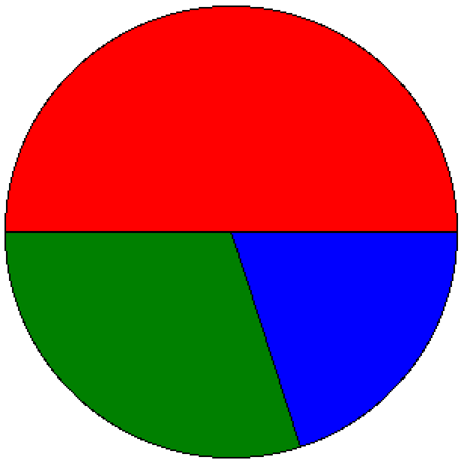
The odd numbers in the input are 3, 1, 1, 5, and 9, which average to $(3+1+1+5+9)/5 = 3.8$, and the even numbers are 4, 8, and 2, which average to $(4+8+2)/3 = 4.666666$ and thus rounds to 4.67.

**Solution:**

```
odds = []
evens = []
finished = False
print("Enter integers:")
while not finished:
    num_str = input("  ? ")
    if num_str == "":
        finished = True
    else:
        num = int(num_str)
        if num % 2 == 0:
            evens.append(num)
        else:
            odds.append(num)
odd_avg = sum(odds)/len(odds)
```

```python
even_avg = sum(evens)/len(evens)
print(f"The average of the odd numbers is {odd_avg:.2f}")
print(f"The average of the even numbers is {even_avg:.2f}")
```

(20) 3. **Interactive Graphics** A pie chart is a common visualization method used to convey approximate relative percentages among several categories. Comprised of filled colorful wedges (which can be portrayed in this problem using a filled `GArc`), the idea is that each wedge has a size or takes up a portion of the overall circle equal to it's category percentage. For instance, the below image shows a pie chart with the red category taking up 50% of the circle, the green taking up 30% of the circle and then the blue taking up 20% of the circle.



Your task in this problem will be two-fold: you'd like to both create a pie-chart from a list of category percentages, but then you'd also like to add some simple mouse interactions.

**Part 1**

Define a function

```
def create_pie_chart(list_of_percents):
```

which will take one argument: a list of the various category percentages as integers. This list could have any number of elements, but you are guaranteed that all the integers within it will add up to 100. Your program should create the necessary `GWindow` and then add the needed filled `GArcs` to create the appearance of a pie chart centered in the window. You can imagine there are several constants already defined at the top of the file that you can use:

```
COLORS = ["red", "green", "blue", "orange"] #colors
GW_WIDTH = 500 # width of window
GW_HEIGHT = 500 # height of window
CHART_RADIUS = 150 # radius of pie chart
HIGHLIGHT_THICKNESS = 10 # line thickness when clicked
```

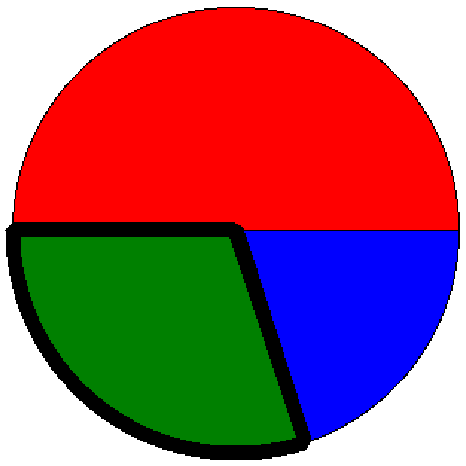
As you work through this part, a few extra things to keep in mind:

- The exact rotation of the overall pie chart doesn't matter, though it can be easiest to start the first wedge at `PGL`'s 0 angle.

- The desired colors of each wedge are given in the `COLORS` constant, and should be cycled through. If you have more wedges than colors, then the sequence should start over, so that the 5th wedge would also be red.
- The elements of the list are integers that represent a percentage, they are not angles. You will have to convert accordingly to what you need.

**Part 2**

We'd like to add some interaction to the pie chart, so that when one of the wedges is clicked, it is highlighted. To this end, you should add callback functions and corresponding listeners for both `"mousedown"` and `"mouseup"`. Upon pressing the mouse down on top of a wedge, the line width of that wedge should be increased to the specified `HIGHLIGHT_THICKNESS` and the wedge should be brought to the front of the stacking order (otherwise parts of the thick line remain behind other wedges, which looks weird). When the mouse is released, the line width should be reset back to 0 (you don't need to do anything to the stacking order). Pressing or releasing the mouse anywhere else on the screen should do nothing. An example of what the green wedge would look like while being clicked is shown below.



**Solution:**

```
from pgl import GWindow, GArc

COLORS = ["red", "green", "blue", "orange"]
GW_WIDTH = 500
GW_HEIGHT = 500
CHART_RADIUS = 150
HIGHLIGHT_THICKNESS = 10

def create_pie_chart(data):
    def down_action(event):
        elem = gw.get_element_at(event.get_x(),
                                 event.get_y())
        if elem is not None:
            elem.send_to_front()
            elem.set_line_width(HIGHLIGHT_THICKNESS)

    def up_action(event):
        elem = gw.get_element_at(event.get_x(),
                                 event.get_y())
        if elem is not None:
            elem.set_line_width(1)

    gw = GWindow(GW_WIDTH, GW_HEIGHT)
    start = 0
    i = 0
    for entry in data:
        stride = int(entry/100*360)
        x = GW_WIDTH / 2 -CHART_RADIUS
        y = GW_HEIGHT / 2 -CHART_RADIUS
        arc = GArc(x , y,
                   2*CHART_RADIUS,2*CHART_RADIUS,
                   start, stride)
        arc.set_filled(True)
        arc.set_fill_color(COLORS[i % len(COLORS)])
        gw.add(arc)
        start += stride
        i += 1

    gw.add_event_listener("mousedown", down_action)
    gw.add_event_listener("mouseup", up_action)
```

(15) 4. **Strings** In Dan Brown's best-selling novel *The Da Vinci Code*, the first clue in a long chain of puzzles is a cryptic message left by the dying curator of the Louvre. Two lines of the message are

*O, Draconian devil!*
*Oh, lame saint!*

Professor Robert Langdon (the hero of the book, played by Tom Hanks in the movie) soon recognizes that these lines are ***anagrams***–pairs of strings that contain exactly the same letters–for

*Leonardo da Vinci*
*The Mona Lisa*

Your job in this problem is to write a predicate function `is_anagram` that takes two strings as arguments and returns `True` if they contain exactly the same alphabetic characters, even though those characters might appear in any order. Thus, your function should return `True` for each of the following calls:

```
is_anagram("O, Draconian devil!", "Leonardo da Vinci")
is_anagram("Oh, lame saint!", "The Mona Lisa")
is_anagram("ALGORITHMICALLY", "logarithmically")
is_anagram("Doctor Who", "Torchwood")
```

These examples illustrate two important requirements of the `is_anagram` function:

- The implementation should look only at letters, ignoring any extraneous spaces or punctuation marks that might show up along the way.
- The implementation should ignore the case of the letters in both strings.

**Solution:** There are a variety of ways this can be approached, but I found checking the letters individually to be difficult, as you'd need to be removing them as you went to ensure you got the correct counts. So I found comparing other structures directly to be more useful. My first method looked something like:

```
def is_anagram(s1, s2):
    s1 = s1.lower()
    s2 = s2.lower()
    return letter_counts(s1) == letter_counts(s2)

def letter_counts(s):
    counts = {}
    for letter in s:
        if letter.isalpha():
            counts[letter] = s.count(letter)
    return counts
```

and my second method like:

```
def is_anagram_v2(s1, s2):
    s1 = [s.lower() for s in s1 if s.isalpha()]
    s2 = [s.lower() for s in s2 if s.isalpha()]
    return sorted(s1) == sorted(s2)
```

(10) 5. **Working with Arrays** Write a function called `rotate_array` that takes a list and an integer as two arguments. The function should have the effect of shifting every element of the list the integer number of positions. Positive integers should result in the elements being shifted toward the beginning of the list, whereas negative integers should result in the elements being shifted towards the end. Elements shifted off either end of the list should wrap around and reappear on the other end of the list. For example, if the array `digits` has the contents:

`digits`

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

then calling `rotate_array(digits, 1)` would shift each of the values one position to the left and move the first value to the end:

`digits`

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 |

Calling `rotate_array(digits, -3)` however would shift all the elements 3 positions to the right:

`digits`

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 8 | 9 | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

Note that although the example array here just had integers as its elements, your function should properly shift *any* type of data the necessary number of positions. Also note that the function should shift the elements *in place*, it should not return a new list.

**Solution:** I found it useful here to figure out how to just shift things 1 position in either direction, and then just using a loop to loop the necessary number of times:

```python
def rotate_array(array, n):
    for _ in range(abs(n)):#need abs else no loop for negs
        if n > 0:
            roll_forward(array)
        else:
            roll_backward(array)

def roll_forward(array):
    tmp = array[0]
    for i in range(len(array)-1):
        array[i] = array[i+1]
    array[-1] = tmp

def roll_backward(array):
    tmp = array[-1]
    for i in range(len(array)-1,0,-1):
        array[i] = array[i-1]
    array[0] = tmp
```

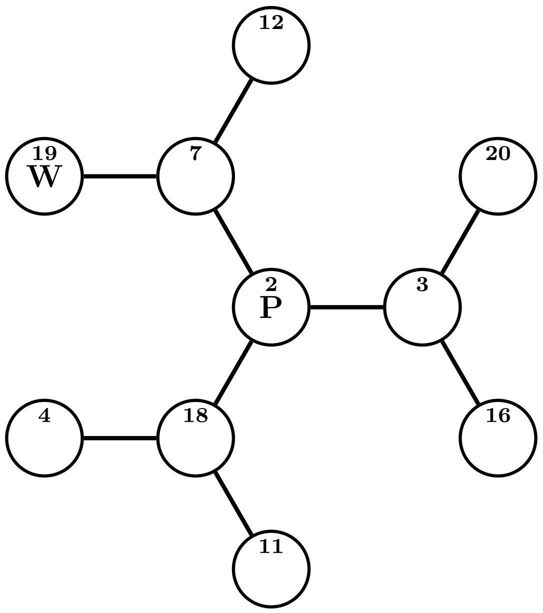
(20) 6. **Python Data Structures** Adventure was not the first widely played computer game in which an adventurer wandered in an underground cave. As far as we know, that honor belongs to the game "Hunt the Wumpus," which was developed by Gregory Yob in 1972.

In the game, the wumpus is a fearsome beast that lives in an underground cave composed of 20 rooms, each of which is numbered between 1 and 20. Each of the twenty rooms has connections to three other rooms, represented as a three-element tuple containing the numbers of the connection rooms in the data structure below. (Because the room numbers start with 1 instead of 0, the data store some irrelevant arbitrary value in element 0 of the room list.) In addition to the connections, the data structure that stores the data for the wumpus game also keeps track of which room number the player is currently occupying, and which room number the wumpus is currently in.

In an actual implementation of the wumpus game, the information in this data structure would be generated randomly. For this problem, which is focusing on whether you can work with data structures that have already been initialized, you can imagine that the variable `cave` has been initialized to the dictionary shown on the next page. The data structure shows the following:

- The player is in room 2
- The wumpus is in room 19
- Room 1 connects to rooms 6, 14, and 19; room 2 connects to 3, 7, and 18; and so on.

To help you visualize the situation, here is a `piece` of the cave map, centered on the current location of the player in room 2:



The player is in room 2, which has connections to rooms 3, 7, and 18. Similarly, room 7 has connections to rooms 2, 12, and 19, which is where the wumpus is lurking. The other connections from rooms 4, 11, 16, 20, 12, and 19 are not shown in the above image. The data structure for the wumpus cave is shown here:

```
cave = {
    "player": 2,
    "wumpus": 19,
    "connections": [
        None,          # Room  0 is not used
        [6, 14, 16],   # Room  1 connects to 6, 14, and 16
        [3,  7, 18],   # Room  2 connects to 3,  7, and 18
        [2, 16, 20],   # Room  3 connects to 2, 16, and 20
        [6, 18, 19],   # Room  4 connects to 6, 18, and 19
        [8,  9, 11],   # Room  5 connects to 8,  9, and 11
        [1,  4, 15],   # Room  6 connects to 1,  4, and 15
        [2, 12, 19],   # Room  7 connects to 2, 12, and 19
        [5, 10, 13],   # Room  8 connects to 5, 10, and 13
        [5, 11, 17],   # Room  9 connects to 5, 11, and 17
        [8, 14, 16],   # Room 10 connects to 8, 14, and 16
        [5,  9, 18],   # Room 11 connects to 5,  9, and 18
        [7, 14, 15],   # Room 12 connects to 7, 14, and 15
        [8, 15, 20],   # Room 13 connects to 8, 15, and 20
        [1, 10, 12],   # Room 14 connects to 1, 10, and 12
        [6, 12, 13],   # Room 15 connects to 6, 12, and 13
        [1,  3, 10],   # Room 16 connects to 1,  3, and 10
        [9, 19, 20],   # Room 17 connects to 9, 19, and 20
        [2,  4, 11],   # Room 18 connects to 2,  4, and 11
        [4,  7, 17],   # Room 19 connects to 4,  7, and 17
        [3, 13, 17],   # Room 20 connects to 3, 13, and 17
    ]
}
```

It is usually possible to avoid the wumpus because the wumpus is so stinky that the player can smell it from 2 rooms away. Thus, in the previous diagram, the player can smell the wumpus. If, however, the wumpus were to move to a room beyond the current boundaries of the diagram, the player would no longer be able to smell the wumpus.

Your task here is to write a predicate function `player_smells_a_wumpus`, which takes the entire wumpus data structure as an argument and returns `True` if the player smells a wumpus and `False` otherwise. Thus calling:

```
player_smells_a_wumpus(cave)
```

would return `True`, given the current values in the `cave`. The function would also return `True` if the wumpus were in rooms 3, 7, or 18, which are one room away from the player. If, however, the wumpus were in a room not shown in the above diagram (room 6, for example, which would connect to room 4), `player_smells_a_wumpus` would return `False`.

**Solution:** This is not as complicated as it might seem. We just need to check the connecting rooms to the players current location, and also check each of the connecting rooms to *those* rooms. One approach might look like:

```python
def player_smells_a_wumpus(data):
    player = data['player']
    wumpus = data['wumpus']
    connecting_rooms = data['connections']
    if player == wumpus:
        return True
    # Check immediate rooms that connect to player
    for room1 in connecting_rooms[player]:
        if room1 == wumpus:
            return True
        #Check rooms that connect to connecting room
        for room2 in connecting_rooms[room1]:
            if room2 == wumpus:
                return True
    return False
```