Practice Final A

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. **Reading Python**

For each of the below pieces of code, evaluate what would be printed on the final line. Show as much work as you can for the potential of partial credit.

(a)
```
def mystery(x, y=10):
    z = len(x)
    return puzzle(x, y) + puzzle(w[:enigma(z,3)],y)

def enigma(x, w):
    return x - w ** 2

def puzzle(y, z):
    return y[z:]

w = "gingerbread man"
print(mystery(w, -3))
```

Solution: The string `"gingerbread man"` is passed into `mystery` and assigned to `x`, and the `-3` is passed in and assigned to `y`. The length of `"gingerbread man"` is thus 15, which is assigned to `z`. Then we evaluate `puzzle(x,y)`, which here will be `puzzle("gingerbread man", -3)`.

Thus the `"gingerbread man"` is assigned to the `y` parameter in `puzzle`, and the -3 to the `z` parameter. So we are just returning `"gingerbread man"[-3:]`, or `"man"`.

Resuming back in `mystery`, we now need to compute `puzzle(w[:enigma(z,3)], y)`, but doing so requires knowledge of what `enigma(z,3)` is. `z` here was 15, so we are evaluating `enigma(15, 3)`.

`enigma` has two parameters, `y` and `w`, and so the arguments are assigned positionally: the 15 to the `y` and the 3 to the `w`. Thus we are returning `15 - 3 ** 2`, or 6.

Back in `mystery`, we now need to evaluate `puzzle(w[:6], y)`. `w` is not defined within `mystery`, but it **is** defined in the enclosing scope (which is also the global scope here). `y` is -3. So we are evaluating `puzzle("gingerbread man"[:6], -3)`, which is the same as `puzzle("ginger", -3)`.

This is essentially the same calculation as earlier though, so the `"ginger"` is assigned to the `y` in `puzzle`, and the -3 to the `z`. So we then evaluate `"ginger[-3:]"`, which is just `"ger"`.

Back in `mystery`, we are thus returning `"man" + "ger"`, and so what would be printed to the screen would be `"manger"`.

(b)
```
class Frosty:
    def __init__(self, n, c):
```

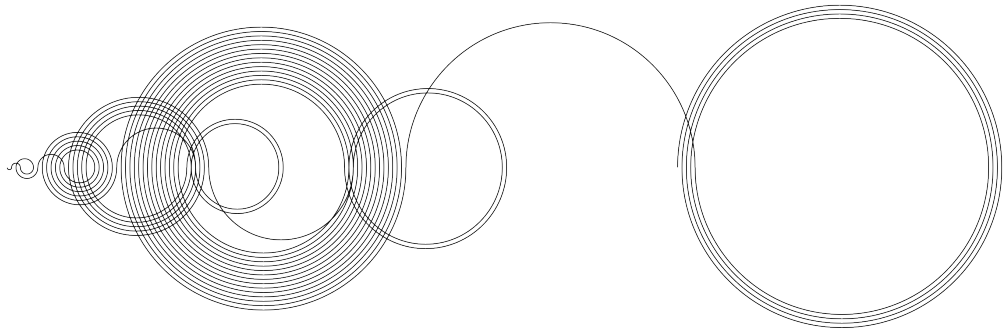
```
        self.wild = [c]
        self.n = n

    def snowball(self, h=3):
        self.n -= h
        self.wild += [self.n]

    def cap(self):
        return self.wild

f = Frosty(8, 15)
f.snowball()
f.snowball(1)
A = f.cap()
A.append(1)
print(sum(f.cap()))
```

Solution: We start by making a `Frosty` object, which passes in `5` to `n` and `12` to `c` in the constructor method. As such, we initialize the `wild` attribute to a list of a single value: `[15]`, and the `n` attribute to an integer: `8`. This newly created object gets assigned to the variable `f`.

Calling the `snowball` method of `f`, we use the default value for `h`: `3`. We thus decrement the current `n` attribute of `8` by 3, resulting in a value of 5. Then we concatenate this value to our `wild` attribute, so that now `wild` is `[15, 5]`. Calling again the `snowball` method of `f`, this time a value of `h` was provided: `1`. We thus decrement the current `n` attribute of `5` by 1, resulting in a value of 4. Then we concatenate this value to our `wild` attribute, so that now `wild` is `[15, 5, 4]`. Calling the `cap` function on `f` just returns a *reference* to the `wild` attribute of `f`, which we assign to the variable `A`. Then we go to append `1` to `A`, which would result in the `wild` attribute of `f` being updated to be `[15, 5, 4, 1]`. Thus in the end when we print out the sum of `f.cap`, since it just returns the `wild` attribute, we'd get `sum([15, 5, 4, 1])`, or 25.

(10) 2. **Fundamental Python**

In 2018 the YouTube channel Numberphile aired a special showcasing one method of visualizing the Racamán sequence, resulting in the interesting behavior shown below:



The Racamán sequence is defined to start at 0, often termed a_0, as it is the 0th term. The n^{th} future value in the sequence is determined by

$$a_n = \begin{cases} a_{n-1} - n, & \text{if } a_{n-1} - n > 0 \text{ and has not already appeared in the sequence} \\ a_{n-1} + n, & \text{otherwise} \end{cases}$$

So for the a_1 term, where $n = 1$, $a_{n-1} - n = a_0 - 1 = 0 - 1$ is less than 0, so instead we would add $0 + 1$, making $a_n = 1$. This continues for the first few terms:

$$a_0 = 0$$
$$a_1 = 1$$
$$a_2 = 3$$
$$a_3 = 6$$

At a_4, note that $6 - 4 > 0$, and the value 2 has not yet shown up in the sequence, so $a_4 = 2$. So the next few terms would be:

$$a_4 = 2$$
$$a_5 = 7$$

At a_6, $7 - 6 > 0$, but the value 1 has already shown up in the sequence (a_1), so instead we add, making $a_6 = 13$. The sequence then proceeds onward infinitely.

Write a function called `racaman` which takes as input a single integer describing the desired term n and returns the n^{th} value of the Racamán sequence. Running your function would look like:

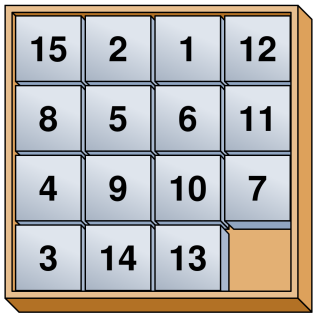
```
>>> print(racaman(3))
6
>>> print(racaman(6))
13
```

Solution: Most of the description here was in trying to describe what the math was doing in case the equations didn't make sense. But the algorithm is pretty simple. Trickiest bit is just ensuring that you get the limits of your loop correct, and realizing that you need to keep a list of all the values seen so far.

```
def racaman(num):
    if num == 0:
        return 0
    seq = [0]
    for n in range(1, num+1):
        potential = seq[n-1] - n
        if potential > 0 and potential not in seq:
            seq.append(potential)
        else:
            seq.append(seq[n-1] + n)
    return seq[-1]
```

(20) 3. **Interactive Graphics**

In all likelihood, you have at some point seen the classic "Fifteen Puzzle" which first appeared in the 1880s. The puzzle consists of 15 numbered squares in a 4×4 box that looks like the following image (taken from the Wikipedia entry):



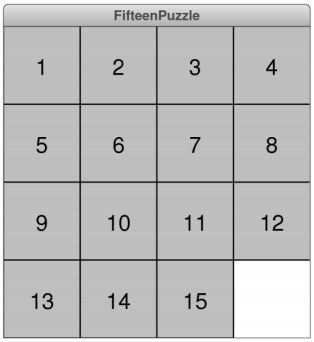
One of the squares is missing from the $4x4$ grid. The puzzle is constructed so that you can slide any of the adjacent squares into the position taken up by the missing square. The object of the game is to restore a scrambled puzzle to its original ordered state. Your task here is to simulate the Fifteen Puzzle, which is easiest to do in two steps:

Step 1:

Write a program that displays the initial state of the Fifteen Puzzle with the 15 numbered squares as shown in the diagram. Each of the pieces should be a `GCompound` containing a square filled in light gray, with a number centered in the square using an 18-point Sans-Serif font, as specified in the following constants:

```
SQUARE_SIZE = 60
GWINDOW_WIDTH = 4 * SQUARE_SIZE
GWINDOW_HEIGHT = 4 * SQUARE_SIZE
SQUARE_FILL_COLOR = "LightGray"
PUZZLE_FONT = "18px 'Sans-Serif'"
```

The completed code after Step 1 would have the graphics window looking something like this:

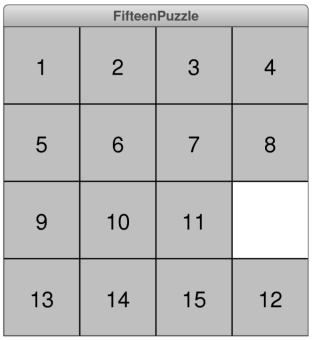


Step 2:

Animate the program so that clicking on a square moves it into the empty space, if possible. This task is easier than it sounds. All you need to do is:

1. Figure out which square you clicked on, if any, by using `get_element_at` to check for an object at that location.
2. Check the adjacent squares to the north, south, east and west. If any square is inside the window and unoccupied, move the square in that direction. If none of the directions work, do nothing.

For example, if you click on the square numbered 5 in the starting configuration, nothing should happen because all of the directions from square 5 are either occupied or outside the window. If, however, you click on square 12, your program should figure out that there is no object to the south and then move the square to that position, so that it would end look like:



Solution: As always, this is not the only way to do this. Here I decided to make the pieces in a class that stored things in an internal gcompound. And when I went to check the different directions, I just supplied them in a list of tuples to iterate over.

```
from pgl import GWindow, GRect, GLabel, GCompound

SQUARE_SIZE = 60
GWINDOW_WIDTH = 4 * SQUARE_SIZE
GWINDOW_HEIGHT = 4 * SQUARE_SIZE
SQUARE_FILL_COLOR = "LightGray"
PUZZLE_FONT = "18px 'Sans-Serif'"

class Piece:
    def __init__(self, num):
        self.compound = GCompound()
        square = GRect(SQUARE_SIZE, SQUARE_SIZE)
        square.set_filled(True)
        square.set_fill_color(SQUARE_FILL_COLOR)
        value = GLabel(str(num))
        value.set_font(PUZZLE_FONT)
        value.move(
            SQUARE_SIZE / 2 - value.get_width() / 2,
            SQUARE_SIZE / 2 + value.get_ascent() / 2,
```

```
            )
            self.compound.add(square)
            self.compound.add(value)

    def click_action(e):
        mx, my = e.get_x(), e.get_y()
        current = gw.get_element_at(mx, my)
        if current is not None:
            for x, y in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                cx = mx + x * SQUARE_SIZE
                cy = my + y * SQUARE_SIZE
                if ((0 < cx < GWINDOW_WIDTH) and
                    (0 < cy < GWINDOW_HEIGHT)):
                    elem = gw.get_element_at(cx, cy)
                    if elem is None:
                        current.move(x * SQUARE_SIZE,
                                     y * SQUARE_SIZE)
                        return

    gw = GWindow(GWINDOW_WIDTH, GWINDOW_HEIGHT)
    for i in range(15):
        p = Piece(i + 1)
        p.compound.move(SQUARE_SIZE * (i % 4),
                        SQUARE_SIZE * (i // 4))
        gw.add(p.compound)
    gw.add_event_listener("click", click_action)
```

(15) 4. **Strings and Files**

Suppose you have a file name `data.txt` that, for whatever reason, is filled with some number of ASCII characters. As an example, a few lines from one such file might look like:

```
baot234'yn8bas92*b
s2ba#9don71abis012
,fygx*@qnadb543nas
```

On each line of the file, you are guaranteed that a numeric digit will appear somewhere. Your task is, for each line, to identify the first number (which might be any number of consecutive digits) on a line, and the *last* number (which might be any number of consecutive digits) on the line and subtract the last from the first. Your function should return the total sum of all the line calculated values added together.

For instance, in the above example:

- The first number appearing in line 1 is 234, and the last number to appear is 92. $234 - 92 = 142$
- The first number appearing in line 2 is 2, and the last number to appear is 012, or 12. $2 - 12 = -10$
- The first number appearing in line 3 is 543, and the last number to appear is 543. Thus $543 - 543 = 0$

Summing up the results from each line, the function would return $142 - 10 + 0 = 132$.

Write a function `process_file(filename)` that takes a filename as an argument, and then opens that file and returns the total sum of the line scores in that file as described above.

Solution: One solution might look like:

```
def process_file(filename):
    """Opens and processes a file by finding the first and last
    numbers on each line, taking the difference, and then summing
    that difference over all lines.
    """
    diff_sum = 0
    with open(filename) as fh:
        for line in fh:
            first = find_first(line)
            last = find_last(line)
            diff = first - last
            diff_sum += diff
    return diff_sum
```

```
def find_first(line):
    """Finds the first number to occur on a line."""
    number = ""
    i = 0
    while not line[i].isdigit(): #find where num starts
        i += 1
    while line[i].isdigit(): #read num
        number += line[i]
        i += 1
    return int(number)

def find_last(line):
    """Finds the last number to occur on a line."""
    number = ""
    i = len(line) - 1
    while not line[i].isdigit(): #find where num starts
        i -= 1
    while line[i].isdigit(): #read num
        number = line[i] + number
        i -= 1
    return int(number)
```

(10) 5. **Defining Classes** For certain applications, it is useful to be able to generate a series of names that form a sequential pattern. For example, if you were writing a program to number figures in a paper, having some mechanism to return the sequence of strings `"Figure 1"`, `"Figure 2"`, `"Figure 3"`, and so on, would be very handy. However, you might also need to label points in a geometric diagram, in which case you would want a similar but independent set of labels for points such as `"P0"`, `"P1"`, `"P2"`, and so forth.

Your task in this problem is to implement a `LabelGenerator` class with the following methods:

- A constructor that takes two arguments: a string indicating the prefix for the labels and an optional starting index for the sequence number, which defaults to 1. For example, calling `LabelGenerator("Figure ")` would return a `LabelGenerator` for the figure labels described earlier, and calling `LabelGenerator("P",0)` would return a `LabelGenerator` for the points.
- A `next_label` method that returns the next label in that sequence. For example, the code sequence:

```
figures = LabelGenerator("Figure ")
print(figures.next_label())
print(figures.next_label())
print(figures.next_label())
```

would generate the following output:

```
Figure 1
Figure 2
Figure 3
```

Solution: The only real thing here to be careful of is that you can't return the label until after you've incremented the counter, so I saved it to a temporary variable.

```
class LabelGenerator:
    def __init__(self, prefix, start=1):
        self._prefix = prefix
        self._count = start

    def next_label(self):
        label = f"{self._prefix}{self._count}"
        self._count += 1
        return label
```

(20) 6. **Python Data Structures**

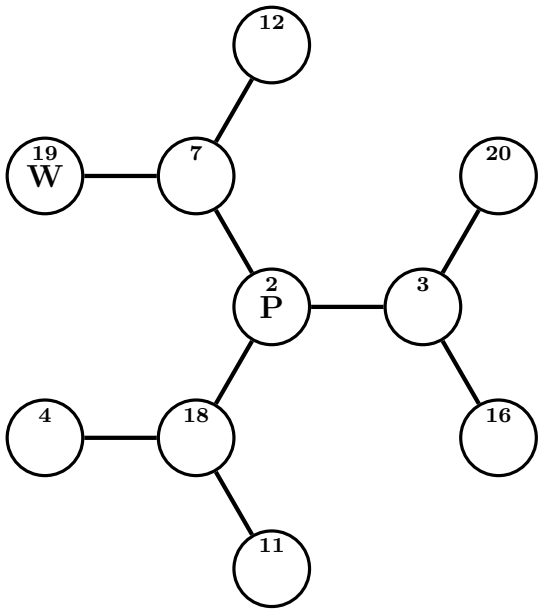
Adventure was not the first widely played computer game in which an adventurer wandered in an underground cave. As far as we know, that honor belongs to the game "Hunt the Wumpus," which was developed by Gregory Yob in 1972.

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:

```
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
```