Practice Final

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. **Reading Python**

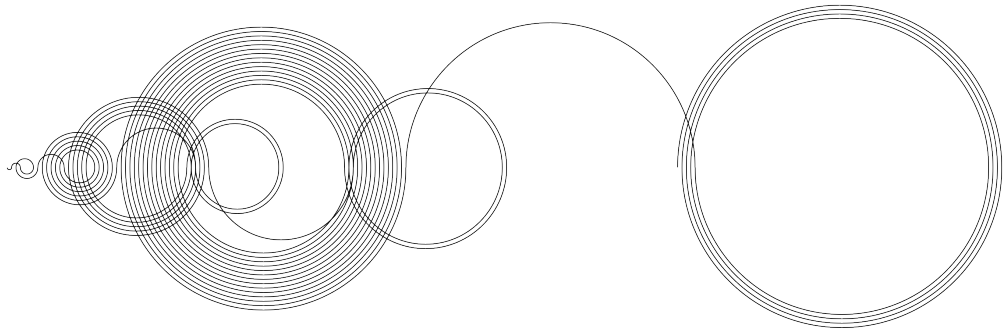
For each of the below pieces of code, evaluate what would be printed on the final line. Show as much work as you can for the potential of partial credit.

(a)
```
def mystery(x, y=10):
    z = len(x)
    return x[y:] + enigma(z,2) * "o"

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

chill = "Snowman"
print(mystery(chill, -3))
```

(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(5, 12)
f.snowball()
A = f.cap()
A.append(5)
print(f.cap())
```

(10) 2. **Early 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:

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

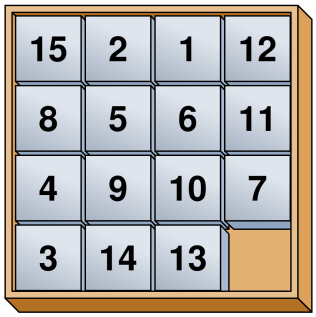
At a_6, $7 - 6 > 0$, but the value 1 has already shown up in the sequence (a_1), so instead we add, making $a_6 = 13$. The sequence then proceeds onward infinitely.

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
```

(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):



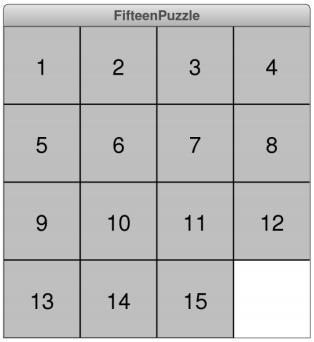
One of the squares is missing from the $4x4$ grid. The puzzle is constructed so that you can slide any of the adjacent squares into the position taken up by the missing square. The object of the game is to restore a scrambled puzzle to its original ordered state. Your task here is to simulate the Fifteen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

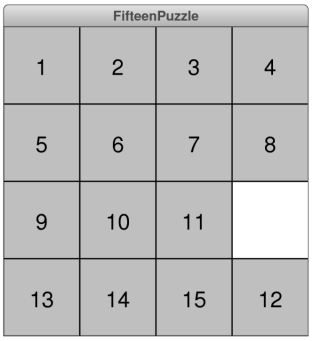


Step 2:

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:



(15) 4. **Strings**

The table of contents for a book typically consists of a list of chapter titles along the left margin of the page and then the corresponding page numbers along the right. To make it easier for your eye to match up the chapter and page, the usual approach is to tie the two visually with a line of dots called a *leader*. Using this style, the entries for the first eight chapters in the Python textbook look like this:

```
1. Introducing Python . . . . . .  1
2. Control Statements . . . . . . 41
3. Simple Graphics  . . . . . . . 79
4. Functions  . . . . . . . . . . 117
5. Writing Interactive Programs . . . . . 155
6. Strings  . . . . . . . . . . . 195
7. Lists  . . . . . . . . . . . . 231
8. Algorithmic Analysis . . . . . 269
```

Here you task is to write a function `create_toc_entry(title, page)`, which takes a chapter title (which includes the chapter number in these examples) and the page number on which that chapter begins. Your function should return a string formatted as an entry for the table of contents. Thus, if you were to call

```
create_toc_entry("6. Strings", 195)
```

the function should return the following string:

```
"6. Strings  . . . . . .  195"
```

In generating this string, your function should adhere to the following guidelines:

- The strings returned by `create_toc_entry` should all have the same length, which is given by the constant `TOC_LINE_LENGTH`.
- The chapter title must appear at the beginning of the result string and must be separated from the first dot in the leader by at least one space.
- The page number must appear at the end of the result string so that the last character of each page number lines up at the column specified by `TOC_LINE_LENGTH`. Like the title, the page number must be separated from the last dot in the leader by at least one space.
- The leader itself is comprised of alternating spaces and dots, indicated by the period character `"."`. Moreover, the dots must be arranged so that they line up vertically. If you simply start the leader one space after the chapter title, the dots would appear to weave back and forth on the page. An easy way to ensure that the dots are aligned correctly is to add an extra space after chapter titles with an even number of characters but not after those with an odd number.

- You may assume that the chapter title and page number fit in `TOC_LINE_LENGTH` character positions and need not make your function handle the situation where the title is too long for the line.

(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.

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

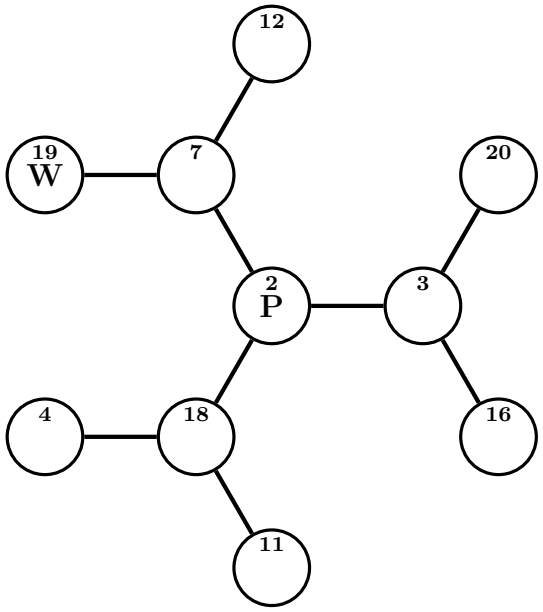
Adventure was not the first widely played computer game in which an adventurer wandered in an underground cave. As far as we know, that honor belongs to the game "Hunt the Wumpus," which was developed by Gregory Yob in 1972.

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`.