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	20	15	10	10	20	85
Score:							

(10) 1. **Simple Python** Write a Python function called `contains_triple` that takes a single integer input `n` and returns `True` if that number contains three of the same digit consecutively. For example, calling

```
contains_triple(1000)
```

should return `True` because the number 1000 contains 3 consecutive 0's. By contrast, calling:

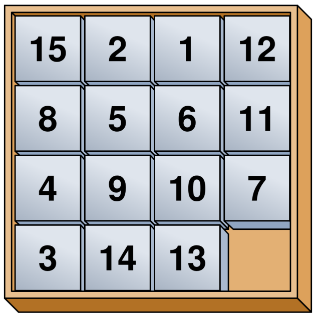
```
contains_triple(122112)
```

should return `False` even though it contains three 1s and three 2s, because these digits never appear three times in a row.

Solution:

```
def contains_triple(n):
    n = str(n)
    cons_counter = 1
    for i in range(len(n)):
        if i > 0 and n[i] == n[i - 1]:
            cons_counter += 1
            if cons_counter == 3:
                return True
        else:
            cons_counter = 1
    return False
```

(20) 2. **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):



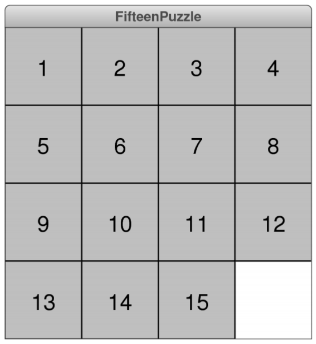
One of the squares is missing from the $4x4$ grid. The puzzle is constructed so that you can slide any of the adjacent squares into the position taken up by the missing square. The object of the game is to restore a scrambled puzzle to its original ordered state. Your task here is to simulate the Fifteen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

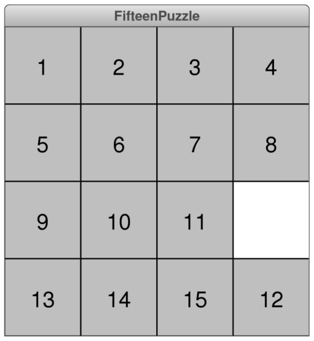


Step 2:

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:



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

```python
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(GCompound):
    def __init__(self, num):
        GCompound.__init__(self)
        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.add(square)
        self.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, cy = mx + x * SQUARE_SIZE, 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.move(SQUARE_SIZE * (i % 4), SQUARE_SIZE * (i // 4))
    gw.add(p)
gw.add_event_listener("click", click_action)
```

(15) 3. **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

```python
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.

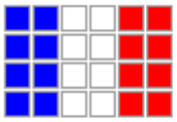
Solution: Most of the difficulty here just comes from getting the spacing right on either side of the leader. So to that end, I just directly worked out that spacing by looking at which cases there was a double space on the left and in which cases there was a double space on the right, remembering that my `". "` repeated leader ends with a space as well.

```python
TOC_LINE_LENGTH = 60

def create_toc_entry(title, page):
    if len(title) % 2 == 0:
        fpad = 2
    else:
        fpad = 1
    if len(str(page)) % 2 == 0:
        epad = 0
    else:
        epad = 1
    leader_num = TOC_LINE_LENGTH - len(title) - fpad - epad - len(str(page))
    leader = ". " * (leader_num // 2)
    line = title + fpad * " " + leader + epad * " " + str(page)
    return line
```

(10) 4. **Working with Arrays** Write a function `double_image(old_image)` that takes an existing `GImage` and returns a new `GImage` that is twice as large in each dimension as the original. Each pixel in the old image should be mapped into the new image as a 2×2 square in the new image where each of the pixels in that square match the original.

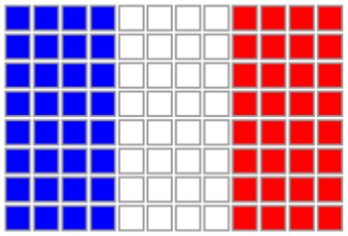
As an example, suppose that you have a `GImage` from the file `TinyFrenchFlag.png` that looks like the image below, where the scale has been expanded so that you can see the individual pixels, each of which appears as a small outlined square:



This 6×4 rectangle has two columns of blue pixels, two columns of white pixels, and two columns of red pixels. Calling

```
bigger_french_flag = double_image(GImage("TinyFrenchFlag.png"))
```

should create a new image with the following 12×8 pixel array:



The blue pixel in the upper left corner of the original has become a square of 4 blue pixels, the pixel to its right has become the next 2×2 square of blue pixels, and so on.

Keep in mind that your goal is to write an implementation of `double_image` that works with *any* `GImage`, and not just the flag image used in this example.

Solution: I probably could have done some sort of looping thing to "paint" all four new pixels in each case here, but I just figured with only 4 it would probably be more clear and about the same number of lines to just let each one explicitly.

```
def double_image(old_image):
    ow, oh = old_image.get_width(), old_image.get_height()
    new_image = [[0 for c in range(2 * ow)] for r in range(2 * oh)]
    old_array = old_image.get_pixel_array()
    for r in range(oh):
        for c in range(ow):
            pixel = old_array[r][c]
            new_image[2 * r][2 * c] = pixel
            new_image[2 * r + 1][2 * c] = pixel
            new_image[2 * r][2 * c + 1] = pixel
            new_image[2 * r + 1][2 * c + 1] = pixel
    return GImage(new_image)
```

(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** In recent years, the globalization of the world economy has put increasing pressure on software developers to make their programs operate in a wide variety of languages. That process used to be called *internationalization*, but is now more often referred to (perhaps somewhat paradoxically) as *localization*. In particular, the menus and buttons that you use in a program should appear in a language that the user knows.

Your task in this problem is to write a definition for a class called `Localizer` designed to help with the localization process. The constructor for the class has the form:

```
class Localizer:
    def __init__(self, filename):
```

The constructor creates a new `Localizer` object and initializes it by reading the contents of the data file. The data file consists of an English word, followed by any number of lines of the form

```
xx=translation
```

where *xx* is a standardized two-letter language code, such as `de` for German, `es` for Spanish, and `fr` for French. Part of such a data file, therefore, might look like this:

Localizations.txt

```
Cancel
de=Abbrechen
es=Cancelar
fr=Annuler
Close
de=Schließen
es=Cerrar
fr=Fermer
OK
fr=Approuver
Open
de=Öffnen
es=Abrir
fr=Ouvrir
```

This file tells us, for example, that the English word `Cancel` should be rendered in German as `Abbrechen`, in Spanish as `Ayudar`, and in French as `Annuler`.

Beyond the implementation of the constructor, the only method you need to define for `Localizer` is

```
def localize(self, word, language):
```

which returns the translation of the English word as specified by the two-letter language parameter. For example, if you have initialized a variable `my_localizer` by calling:

```
my_localizer = Localizer("Localizations.txt")
```

you could then call

```
my_localizer.localize("Open", "de")
```

and expect it to return the string `"Öffnen"`. If no entry appears in the table for a particular word, `localize` should return the English word unchanged. Thus, `OK` becomes `Approuver` in French, but would remain as `OK` in Spanish or German.

As you write your answer to this problem, here are a few points to keep in mind:

- You can determine when a new entry starts in the data file by checking for a line without an equal sign. As long as an equal sign appears, what you have is a new translation for the most recent English word into a new language.
- For this problem, you don't have to worry about distinctions between uppercase and lowercase letters and may assume that the word passed to `localize` appears exactly as it does in the data file.
- The data file shown above is just a small example; your program must be general enough to work with a much larger file. You may not assume that there are only three languages or, worse yet, only four words.
- You don't have to do anything special for the characters in other languages that are not part of standard English, such as the ö and ß that appear in this data file. They are all characters in the expanded Unicode set that Python uses.
- It may help you solve this problem if you observe that it is the *combination* of an English word and a language code that has a unique translation. Thus, although there are several different translations of the word `Close` in the localizer and German translations of many words, there is only one entry for the combination of `Close+de`.

Solution: Taking the advice of the last bullet point, I decided to make my keys to the dictionary a tuple that held both the word and the language. You could have also do something similar by concatenating both together in a string or similar. Most of the rest of the complexity is just from reading in the file to the desired structure.

```python
class Localizer:
    def __init__(self, filename):
        self._translations = self.read_file(filename)

    def read_file(self, filename):
        word = ""
        lang = ""
        trans = {}
        with open(filename) as fh:
            for line in fh:
                line = line.strip()
                eqloc = line.find("=")
                if eqloc == -1:
                    word = line
                else:
                    lang = line[:eqloc]
                    new = line[eqloc + 1 :]
                    trans[(word, lang)] = new
        return trans

    def localize(self, word, lang):
        return self._translations.get((word, lang), word)
```