Problem 1: Simple Python expressions (10 points)

Compute the value of each of the following expressions and write it in the space provided. If the line produces an error, write “Error” in this space. Assume that the constant ALPHABET has been initialized as in the reader and has the following internal value, listed with both positive and negative indexes:

```
alphabet = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, -26, -25, -24, -23, -22, -21, -20, -19, -18, -17, -16, -15, -14, -13, -12, -11, -10, -9, -8, -7, -6, -5, -4, -3, -2, -1]
```

1a) \(1 \times 2 \% 3 + (4 \% 5 \times 6) \times 7 \times 8 \div 9 \times 10\)

1b) \(\text{str}(5 \times 2) + \text{str}(5 + 1) \times 2\)

1c) \(\text{ALPHABET}[-9:-13] + \text{ALPHABET}[4:2:-1]\)

1d) \(\text{ALPHABET}[\text{len(ALPHABET)}]\)

Problem 2: Program tracing (10 points)

What output does the following program produce:

```
# File: Mystery.py

def mystery(x):
    def enigma(z):
        nonlocal y
        y = 2
        return z[:6] + z[y]

    y = len(x)
    z = x[1 - y]
    print(z)
    z += enigma(x)
    print(z)
    z += enigma(x)
    print(z)

# Startup code
if __name__ == "__main__":
mystery("abcdefg")
```
Problem 3: Simple Python programs (15 points)

A spoonerism is a phrase in which the leading consonant strings of the first and last words are inadvertently swapped, generally with comic effect. Some examples of spoonerisms include the following phrases and their spoonerized counterparts (the consonant strings that get swapped are underlined):

- crushing blow → blushing crow
- sons of toil → tons of soil
- pack of lies → lack of pies
- jelly beans → belly jeans
- flutter by → butter fly

In this problem, your job is to write a function

```python
def spoonerize(phrase):
```

that takes a multiword phrase as its argument and returns its spoonerized equivalent. For example, you should be able to use your function to duplicate the following console session in which all the examples come from Shel Silverstein’s spoonerism-filled children’s book Runny Babbit:

```
>>> from Spoonerize import spoonerize
>>> spoonerize("bunny rabbit")
runny babbit
>>> spoonerize("silly book")
billy soak
>>> spoonerize("take a shower")
shake a tower
>>> spoonerize("wash the dishes")
dash the wishes
>>> 
```

In this problem, you are not responsible for any error-checking. You may assume that the phrase passed to `spoonerize` contains nothing but lowercase letters along with spaces to separate the words. You may also assume that the phrase contains at least two words, that there are no extra spaces, and that each word contains at least one vowel. What your method needs to do is extract the initial consonant substrings from the first and last words and then exchange those strings, leaving the rest of the phrase alone.

Hint: Remember that you can use methods from the book. The `findFirstVowel` and `isEnglishVowel` methods from the Pig Latin program will certainly come in handy.
Problem 4: Using the Portable Graphics Library (20 points)

In this problem, your mission is to write a program that plays a simplified version of a graphical game called “Snake,” which was popular on the first generation of Nokia phones. When the program begins, there is nothing on the graphics window, but there is an invisible snake head at the center of the window. In each time step of the animation, the snake head moves 15 pixels (for which you should use the constant SQUARE_SIZE defined on page 7) in some direction, leaving behind a $15 \times 15$ filled square at the position it just left.

At the beginning of the game, the snake head is moving eastward, so after four time steps, it will have generated a trail of four squares (which run together on the screen), like this:

![Snake](image1.png)

In this implementation, you turn the snake by clicking the mouse. In this situation, with the snake moving horizontally, clicking the mouse above the current $y$ position of the snake head sends it northward, and clicking below the current $y$ position sends it to the south. In this example, let’s assume that the player has clicked above the snake, so that its direction changes to north. After three more time steps, the graphics window looks like this:

![Snake](image2.png)

When the snake is moving vertically, clicking to the left of the snake head sends it westward, and clicking to the right sends it eastward. Clicking to the left would therefore send the snake off to the west, as follows:
Eventually, after making a series of turns, the snake will end up colliding with its own path, as shown in the following diagram:

Eventually, after making a series of turns, the snake will end up colliding with its own path, as shown in the following diagram:

When this situation occurs (i.e., the position of the head of the snake already contains a square), the program should stop.

While this program might at first seem complicated, it turns out to be quite short. Here are a few general hints to get you started:

- You need to keep track of the $x$ and $y$ coordinates of the snake head, along with some indication of the direction in which it is moving, which can only be one of the four primary compass directions: North, East, South, and West.
- The function that executes in each time step must first check to see whether the current location already contains a square, in which case the program stops. If not, it creates a square whose center is at the current position, and then moves on to the next square by updating the coordinates as appropriate for the current direction.
- The listener method that responds to mouse clicks has to look at the current position and direction and then use those together with the mouse click location to determine how to update the direction.
- You need not take account of checking whether the snake remains on the window. In a more sophisticated implementation of the game, running off the window should also count as a loss.