

Lab 4

This lab provides additional exercises on conditional reasoning and an introduction to lists.

Download lab 4 files from the link on Canvas, place it in your `cpe101` directory.

Conditionals

This part of the lab is structured a bit differently from what you have been asked to do in the past. The given files, in the `patterns` directory, include a "driver" and multiple pattern files. You will not (and should not) need to modify these files. The driver provides a mostly complete program but it needs a function to work properly. This is the function that you are to write. The pattern files provide text-based "images".

letter

Each pattern file consists of an "image" made up of letters. For a given pattern, you must write a function that returns the correct letter for the row/column position specified (as the function's arguments). This function will be called multiple times; once for each position in the pattern image.

As an example, `pattern00` is made up entirely of repeated 'A' characters. As such, the `letter` function could be written as follows.

```
def letter(row, col):  
    return 'A'
```

In general, however, the `letter` function must determine which letter to return based on the row and column provided. In the pattern files, rows and columns are counted beginning at 0. As such, the top-left character is at `row == 0` and `col == 0`. So if the pattern consisted of all 'A' characters but with a single 'G' character in the top-left corner, then the `letter` function might look as follows.

```
def letter(row, col):  
    if (row == 0 and col == 0):  
        return 'G'  
    else:  
        return 'A'
```

Multiple Files and All That

You will write multiple versions of this `letter` function. Since the `driver.py` file will remain unchanged, you must place your `letter` function in a separate file. The name for this file is given below (for convenience, it is the name of the pattern).

Of course, for a given execution, the driver must be "told" which specific `letter` function to use. You will do this by importing the driver in your file and calling the `comparePatterns` function in the driver. To this function, you must pass your `letter` function (by specifying the name as you would a variable). The following code completes the "all A" pattern example discussed above.

```
import driver  
  
def letter(row, col):  
    return 'A'  
  
if __name__ == '__main__':  
    driver.comparePatterns(letter)
```

Patterns - What you need to do.

pattern00

In pattern00.py, write the letter function that will generate the pattern image in pattern00.

When you run this program, you must tell the program which pattern to compare against. You can do so as follows.

```
python pattern00.py < pattern00
```

pattern01

In pattern01.py, write the letter function that will generate the pattern image in pattern01.

pattern02

In pattern02.py, write the letter function that will generate the pattern image in pattern02.

pattern03

In pattern03.py, write the letter function that will generate the pattern image in pattern03.

pattern04

In pattern04.py, write the letter function that will generate the pattern image in pattern04.

pattern05

In pattern05.py, write the letter function that will generate the pattern image in pattern05.

pattern06

In pattern06.py, write the letter function that will generate the pattern image in pattern06.

pattern07

In pattern07.py, write the letter function that will generate the pattern image in pattern07.

Lists

This part of the lab introduces indexing lists. You should **not** use any loops in your functions. Though they would be useful to generalize your code (which we will do in a later lab), the goal of this lab exercise is to understand the mechanics of using lists so you should focus only on that.

In the poly directory create a file named poly.py. Place your test cases in poly_tests.py (this file is not provided; you should be comfortable writing this by now).

You must provide at least two test cases for each of these functions.

This part will be executed with: **python poly_tests.py**

Polynomial Arithmetic

For this part of the lab you will develop two functions that perform basic arithmetic on polynomials. A [polynomial](#) will be represented as a list. The values in the list will represent the coefficients of the terms whereas the indices will represent the exponents for the terms.

This means that the polynomial $2.7x^2 + 3.1x + 2$ will be represented by the following list. Notice that the term with exponent 0 is first in the list while the term with exponent 2 is last (i.e., the terms in the list are in reverse order of how they are typically written in mathematics; this is done so that an element's index represents that term's exponent).

```
poly = []  
poly.append(2)  
poly.append(3.1)  
poly.append(2.7)
```

This list can also be created directly as follows. This is convenient when, for instance, writing test cases.

```
poly = [2, 3.1, 2.7]
```

You may think this mapping of a polynomial to a list is a bit odd. In fact, attributing meaning to indices of a list (and not just the values within the list) is a pretty important skill that allows a list to be used as more than just a substitution for a bunch of variables.

`poly_add2`

In `poly.py`, develop the `poly_add2` function. This function takes two polynomials of degree two (lists of length three) as arguments. This function must return a new list (i.e., do not modify the contents of the input lists) representing the sum of the input polynomials.

Though the testing framework does work with lists, it does not support an "almost" equal check on the contents of a list. In the provided testing file you will find `assertListAlmostEqual`. It can be used, in a testing function, as follows.

```
def test_poly(self):
    poly1 = [2.3, 4.7, 1.0]
    poly2 = [1.2, 2.1, -3.2]

    poly3 = poly.poly_add2(poly1, poly2)
    self.assertListAlmostEqual(poly3, [3.5, 6.8, -2.2])
```

`poly_mult2`

Develop the function `poly_mult2`. This function will take two polynomials of degree two and compute the product of the two polynomials. Polynomial multiplication is not a simple multiplication of values at the same index; instead, think of the distributive law (of which the FOIL method is a special, simple case).

Note carefully: The polynomial resulting from a multiplication will, in general, be of degree greater than the argument polynomials. In this case, the result can be of at most degree four, so your result list (checked against in your test cases) may be larger than initially expected.

Again, though the use of loops would allow one to generalize this function, for this lab you cannot use any loops. Think carefully about how to compute the product of polynomials and how that relates to the representation of polynomials in this lab.

Demonstration

Demonstrate the test cases from each part of the lab to your instructor to have this lab recorded as completed. In addition, be prepared to show the source code to your instructor.