More on Lists

List comprehensions

- A list comprehension is a way of specifying an expression that you would like to apply to every element of a list without using a `for` loop to iterate over the elements. The result is a new list.

Example

- Here is an example, say you have a list of integers and you want to produce a list of the squares of those integers. You could use a `for` loop:

```
>>> # create list of squares of 1-5 using cumulative algorithm
>>> nums1 = [1, 2, 3, 4, 5]
>>> nums2 = []
>>> for n in nums1:
...     nums2.append(n * n)
...
>>> nums2
[1, 4, 9, 16, 25]
```

- A list comprehension provides a neater, more elegant way to express this:

```
>>> # create list of the squares of 1-5 using list comprehension
>>> nums1 = [1, 2, 3, 4, 5]
>>> nums2 = [n * n for n in nums1]
>>> nums2
[1, 4, 9, 16, 25]
```

2D lists

- Most of the lists we have encountered so far have been *one dimensional*.
- We can create lists of lists - multidimensional lists, in this section we will look at 2-dimensional lists:

```
name = [[value, value, ..., value],
        [value, value, ..., value],
         ...,
        [value, value, ..., value]]
```

2D list example

```
>>> # create a multi-dimensional list (first syntax)
>>> temps = [[0, 0, 0, 0, 0],
             [0, 0, 0, 0, 0],
             [0, 0, 0, 0, 0]]
>>>
>>> # create a multi-dimensional list (second syntax)
>>> temps = [[0] * 5, [0] * 5, [0] * 5]
```

```
                0    1    2    3    4
           +----+----+----+----+----+
        0  |  0 |  0 |  0 |  0 |  0 |
           +----+----+----+----+----+
temps   1  |  0 |  0 |  0 |  0 |  0 |
           +----+----+----+----+----+
        2  |  0 |  0 |  0 |  0 |  0 |
           +----+----+----+----+----+
```

Using a 2D array

- Accessing elements:
- `temps` is the entire grid
- `temps[2]` is the entire third row
- `temps[2][0]` is the first element of the third row
- Finding the number of rows and column

```
>>> # use the len function on a multidimensional list
>>> temps = [[0] * 5, [0] * 5, [0] * 5]
>>> len(temps)       # number of rows
3
>>> len(temps[0])    # length of first row, number of columns
5
```

Ragged Lists

- In a jagged list, the number of columns varies from row to row.
- Example: Pascal's Triangle. The numbers in the triangle have many useful mathematical properties. For example, row ‘n of Pascal's triangle contains the coefficients obtained when you expand the equation:

$(x + y)^n$

Here are the results for n between 0 and 4:

```
(x + y)0 = 1
(x + y)1 = x + y
(x + y)2 = x2 + 2xy + y2
(x + y)3 = x3 + 3x2y + 3xy2 + y3
(x + y)4 = x4 + 4x3y + 6x2y2 + 4xy3 + y4
```

Example (continued):

- Writing the coefficients as a triangle, you get:

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

Example (continued):

- These rows of numbers form a five-row Pascal's triangle. You can compute a row from the one above, adding 1s at the front and back:

```
1   4   6   4   1

(1 + 4) (4 + 6) (6 + 4) (4 + 1)
|_____| |_____| |_____| |_____|
   |       |       |       |
   5      10      10       5
```

```
           1
         1   1
       1   2   1
     1   3   3   1
   1   4   6   4   1
 1   5  10   10  5   1
```

Writing the code

```
                +----+
             0  |  1 |
                +----+----+
triangle     1  |  1 |  1 |
                +----+----+----+
             2  |  1 |  2 |  1 |
                +----+----+----+----+
             3  |  1 |  3 |  3 |  1 |
                +----+----+----+----+----+
             4  |  1 |  4 |  6 |  4 |  1 |
                +----+----+----+----+----+----+
             5  |  1 |  5 | 10 | 10 |  5 |  1 |
                +----+----+----+----+----+----+
Figure: Pascal's Triangle as multi-dimensional list
```

Pseudocode

- High-level

  ```
  for i in range(len(triangle)):
    define triangle[i] using triangle[i - 1].
  ```

- The first and last values in each row should be 1:

  ```
    for i in range(len(triangle)):
        triangle[i] = [0] * (i + 1) # create an empty row
        triangle[i][0] = 1
        triangle[i][i] = 1
        fill in the middle of triangle[i] using triangle[i - 1].
  ```

Middle values

- Generally, each of these middle values is the sum of the two values from the previous row that appear just above and to the left:

  ```
  triangle[i][j] = (value above and left) + (value above).
  ```

  or:

  ```
  triangle[i][j] = triangle[i - 1][j - 1] + triangle[i - 1][j]
  ```

- The `for` loop is the final step:

  ```
  for i in range(len(triangle)):
      triangle[i] = [0] * (i + 1)
      triangle[i][0] = 1
      triangle[i][i] = 1
      for j in range(1, i):
          triangle[i][j] = triangle[i - 1][j - 1]
                               + triangle[i - 1][j]
  ```

- Complete program is in folder `src`