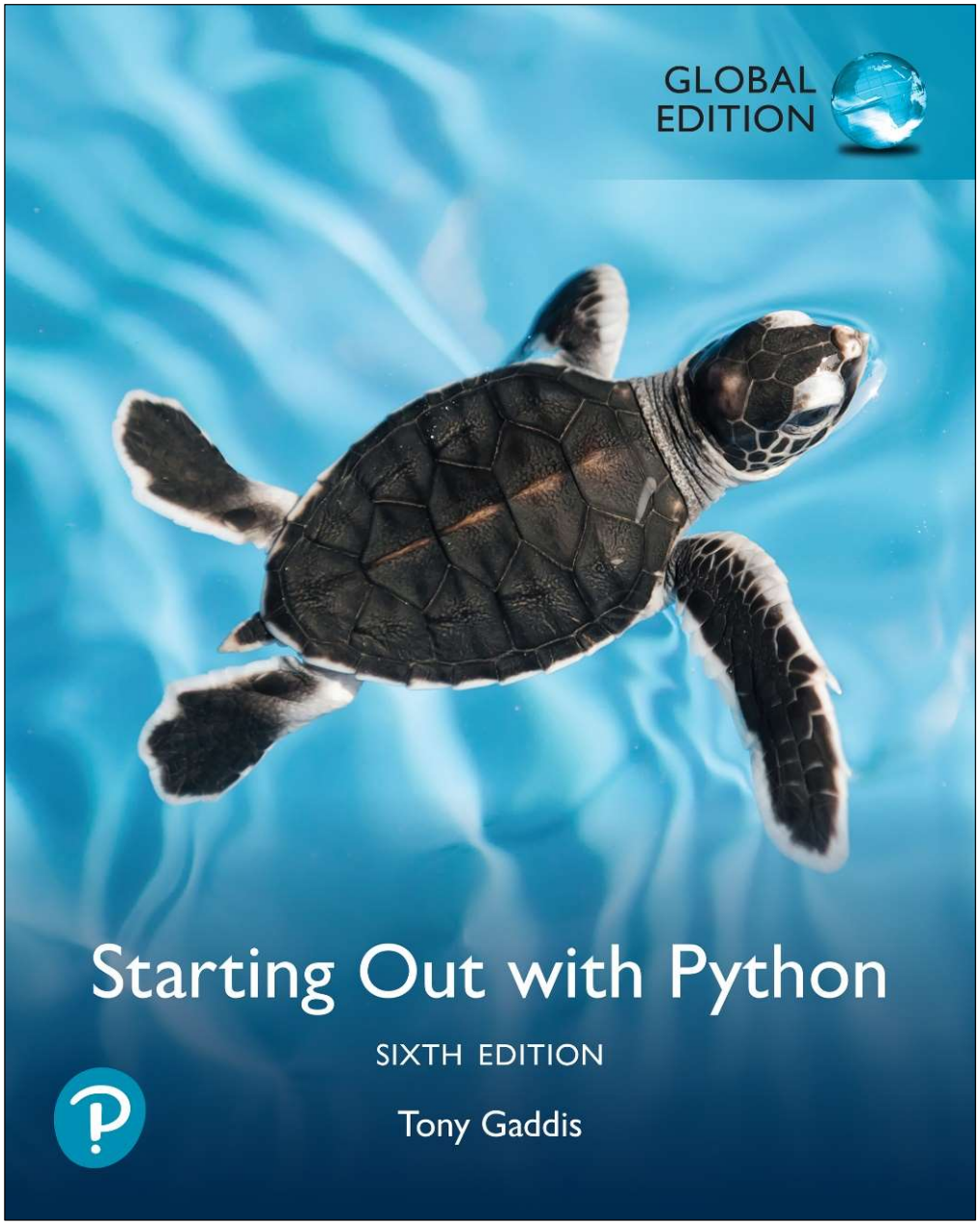
CHAPTER 7

Lists and Tuples



Topics

- Sequences
- Introduction to Lists
- List Slicing
- Finding Items in Lists with the in Operator
- List Methods and Useful Built-in Functions

Topics (cont'd.)

- Copying Lists
- Processing Lists
- List Comprehensions
- Two-Dimensional Lists
- Tuples
- Plotting List Data with the `matplotlib` Package

Sequences

- **Sequence: an object that contains multiple items of data**
  - The items are stored in sequence one after another
- **Python provides different types of sequences, including lists and tuples**
  - The difference between these is that a list is mutable and a tuple is immutable

Introduction to Lists

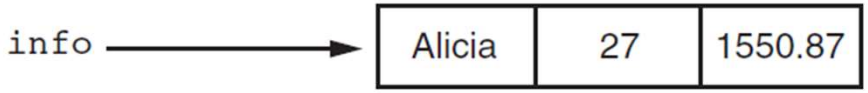
- **List: an object that contains multiple data items**
  - Element: An item in a list
  - Format: `list = [item1, item2, etc.]`
  - Can hold items of different types
- **`print` function can be used to display an entire list**
- **`list()` function can convert certain types of objects to lists**

Introduction to Lists (cont'd.)

Figure 7-1 A list of integers

Figure 7-2 A list of strings

Figure 7-3 A list holding different types



The Repetition Operator and Iterating over a List

- **Repetition operator: makes multiple copies of a list and joins them together**
  - The * symbol is a repetition operator when applied to a sequence and an integer
    - Sequence is left operand, number is right
  - General format: *`list * n`*
- **You can iterate over a list using a `for` loop**
  - Format: `for` *`x`* `in` *`list`*`:`

Indexing

- **Index: a number specifying the position of an element in a list**
  - Enables access to individual element in list
  - Index of first element in the list is 0, second element is 1, and n'th element is n-1
  - Negative indexes identify positions relative to the end of the list
    - The index -1 identifies the last element, -2 identifies the next to last element, etc.

The `len` function

- **An `IndexError` exception is raised if an invalid index is used**
- **`len` function: returns the length of a sequence such as a list**
  - Example: `size = len(my_list)`
  - Returns the number of elements in the list, so the index of last element is `len(list)-1`
  - Can be used to prevent an `IndexError` exception when iterating over a list with a loop

Lists Are Mutable

- **Mutable sequence: the items in the sequence can be changed**
  - Lists are mutable, and so their elements can be changed
- **An expression such as**
- **`list[1] = new_value` can be used to assign a new value to a list element**
  - Must use a valid index to prevent raising of an `IndexError` exception

Concatenating Lists

- **Concatenate: join two things together**
- **The + operator can be used to concatenate two lists**
  - Cannot concatenate a list with another data type, such as a number
- **The += augmented assignment operator can also be used to concatenate lists**

List Slicing

- **Slice: a span of items that are taken from a sequence**
  - List slicing format: `list[start : end]`
  - Span is a list containing copies of elements from `start` up to, but not including, `end`
    - If `start` not specified, `0` is used for start index
    - If `end` not specified, `len(list)` is used for end index
  - Slicing expressions can include a step value and negative indexes relative to end of list

Finding Items in Lists with the `in` Operator

- **You can use the `in` operator to determine whether an item is contained in a list**
  - General format: *`item` `in` `list`*
  - Returns `True` if the item is in the list, or `False` if it is not in the list
- **Similarly you can use the `not in` operator to determine whether an item is not in a list**

List Methods and Useful Built-in Functions

- **`append(item)`: used to add items to a list – *`item`* is appended to the end of the existing list**
- **`count(item)`: returns the number of times *`item`* appears in the list**
- **`index(item)`: used to determine where an item is located in a list**
  - Returns the index of the first element in the list containing `item`
  - Raises `ValueError` exception if *`item`* not in the list

List Methods and Useful Built-in Functions (cont'd.)

- `insert(index, item)`: used to insert *item* at position *index* in the list
- `sort()`: used to sort the elements of the list in ascending order
- `remove(item)`: removes the first occurrence of *item* in the list
- `reverse()`: reverses the order of the elements in the list

List Methods and Useful Built-in Functions (cont'd.)

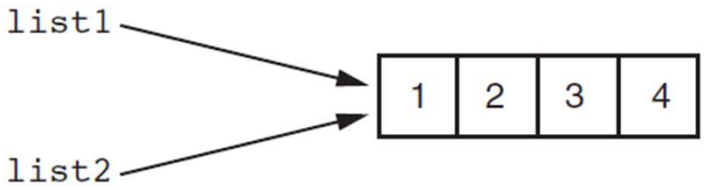
- **`del` statement: removes an element from a specific index in a list**
  - General format: `del list[i]`
- **`sum` function: returns the sum of the values in a numeric sequence**
  - Example: `total = sum(my_list)`
- **`min` and `max` functions: built-in functions that returns the item that has the lowest or highest value in a sequence**
  - The sequence is passed as an argument

Copying Lists

- **To make a copy of a list you must copy each element of the list**
  - Two methods to do this:
    - Creating a new empty list and using a `for` loop to add a copy of each element from the original list to the new list
    - Creating a new empty list and concatenating the old list to the new empty list

Copying Lists (cont'd.)

Figure 7-4 `list1` and `list2` reference the same list



Processing Lists

- **List elements can be used in calculations**
- **To calculate total of numeric values in a list use loop with accumulator variable**
- **To average numeric values in a list:**
  - Calculate total of the values
  - Divide total of the values by `len(list)`
- **List can be passed as an argument to a function**

Processing Lists (cont'd.)

- **A function can return a reference to a list**
- **To save the contents of a list to a file:**
  - Use the file object's `writelines` method
    - Does not automatically write `\n` at then end of each item
  - Use a `for` loop to write each element and `\n`
- **To read data from a file use the file object's `readlines` method**

List Comprehensions

- **List comprehension: a concise expression that creates a new list by iterating over the elements of an existing list.**

List Comprehensions

- **The following code uses a for loop to make a copy of a list:**

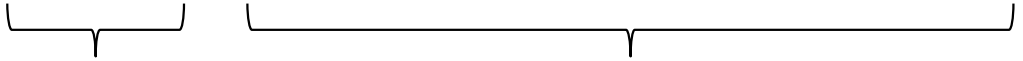
```
list1 = [1, 2, 3, 4]
list2 = []

for item in list1:
    list2.append(item)
```

- **The following code uses a list comprehension to make a copy of a list:**

```
list1 = [1, 2, 3, 4]
list2 = [item for item in list1]
```

List Comprehensions

```
list2 = [item for item in list1]
```



Result Expression Iteration Expression

- The iteration expression works like a `for` loop
- In this example, it iterates over the elements of `list1`
- Each time it iterates, the target variable `item` is assigned the value of an element.
- At the end of each iteration, the value of the result expression is appended to the new list.

List Comprehensions

```
list1 = [1, 2, 3, 4]
list2 = [item**2 for item in list1]
```

- After this code executes, `list2` will contain the values `[1, 4, 9, 16]`

List Comprehensions

```
str_list = ['Winken', 'Blinken', 'Nod']
len_list = [len(s) for s in str_list]
```

- After this code executes, `len_list` will contain the values [6, 7, 3]

List Comprehensions

- **You can use an `if` clause in a list comprehension to select only certain elements when processing a list**

```
list1 = [1, 12, 2, 20, 3, 15, 4]
list2 = []

for n in list1:
    if n < 10:
        list2.append(n)
```

Works the same as…

```
list1 = [1, 12, 2, 20, 3, 15, 4]
list2 = [item for item in list1 if item < 10]
```

List Comprehensions

```
list1 = [1, 12, 2, 20, 3, 15, 4]
list2 = [item for item in list1 if item < 10]
```

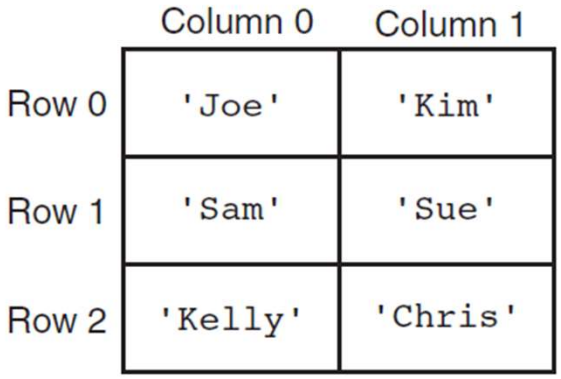
- After this code executes, `list2` will contain `[1, 2, 3, 4]`

Two-Dimensional Lists

- **Two-dimensional list: a list that contains other lists as its elements**
  - Also known as nested list
  - Common to think of two-dimensional lists as having rows and columns
  - Useful for working with multiple sets of data
- **To process data in a two-dimensional list need to use two indexes**
- **Typically use nested loops to process**

Two-Dimensional Lists (cont'd.)

Figure 7-5 A two-dimensional list



Two-Dimensional Lists (cont'd.)

Figure 7-7 Subscripts for each element of the `scores` list

	Column 0	Column 1	Column 2
Row 0	scores[0][0]	scores[0][1]	scores[0][2]
Row 1	scores[1][0]	scores[1][1]	scores[1][2]
Row 2	scores[2][0]	scores[2][1]	scores[2][2]

Tuples

- **Tuple: an immutable sequence**
  - Similar to a list
  - Tuples are immutable. Once a tuple is created:
    - New elements cannot be added to the tuple
    - Elements cannot be removed from the tuple
    - The values stored in tuple elements cannot be changed

Tuples

- **Creating a Tuple**
  - `my_tuple= (item1, item2)`
  - `my_tuple= (item1,)`

- **Tuples support many list operations**
  - Subscript indexing for retrieving elements
  - Methods such as `index`
  - Built in functions such as `len, min, max`
  - Slicing expressions
  - The `in`, `+`, and `*` operators

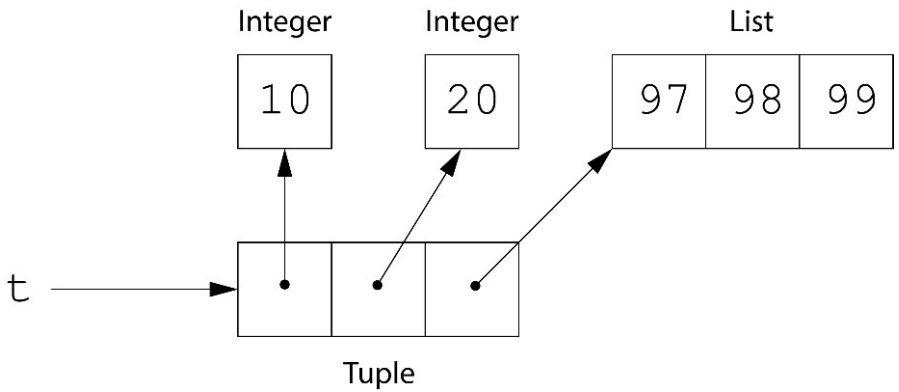
Tuples

- **Tuples do not support the methods:**
  - `append`
  - `remove`
  - `insert`
  - `reverse`
  - `sort`

Storing Mutable Objects in a Tuple

- **Although tuples are immutable, you can store mutable objects in a tuple.**
  - For example, the following statement creates a tuple that contains a list as its 3rd element:

```
t = (10, 20, [97, 98, 99])
```



Storing Mutable Objects in a Tuple

- **When a tuple element refers to a mutable object, that object's data can be changed.**

```
>>> t = (10, 20, [97, 98, 99])
>>> print(t)
(10, 20, [97, 98, 99])
>>>
>>> t[2].append(100)
>>> print(t)
(10, 20, [97, 98, 99, 100])
>>>
```

Tuples

- **Reasons for using tuples:**
  - Processing tuples is faster than processing lists
  - Tuples provide safety:
    - no elements can be added to or deleted from a tuple
    - once a tuple is created, the elements of the tuple will always refer to the same objects
  - Some operations in Python require use of tuples
- **`list()` function: converts tuple to list**
- **`tuple()` function: converts list to tuple**

Plotting Data with `matplotlib`

- The `matplotlib` package is a library for creating two-dimensional charts and graphs.

- It is not part of the standard Python library, so you will have to install it separately, after you have installed Python on your system.

Plotting Data with `matplotlib`

- **To install `matplotlib` on a Windows system, open a Command Prompt window and enter this command:**

```
pip install matplotlib
```

- **To install `matplotlib` on a Mac or Linux system, open a Terminal window and enter this command:**

```
sudo pip3 install matplotlib
```

- **See Appendix F in your textbook for more information about packages and the `pip` utility.**

Plotting Data with `matplotlib`

- **To verify the package was installed, start IDLE and enter this command:**

```
>>> import matplotlib
```

- **If you don't see any error messages, you can assume the package was properly installed.**

Plotting Data with `matplotlib`

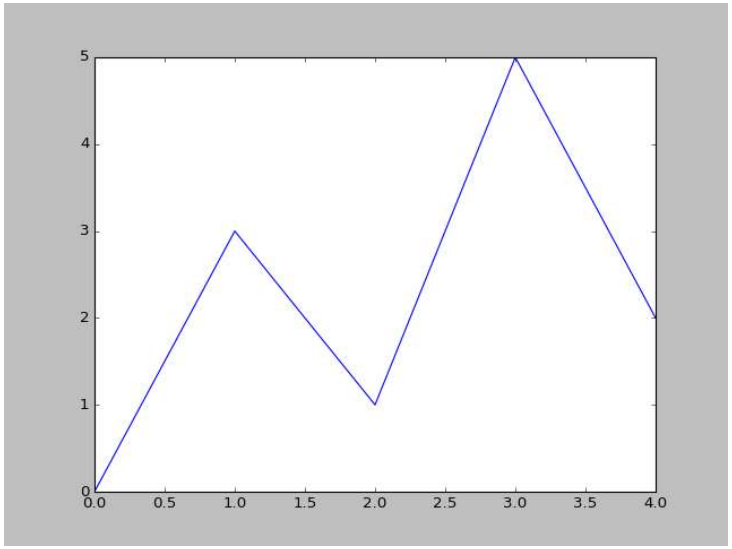
- **The `matplotlib` package contains a module named `pyplot` that you will need to import.**
- **Use the following `import` statement to import the module and create an alias named `plt`:**

```
import matplotlib.pyplot as plt
```

For more information about the `import` statement, see Appendix E in your textbook.

Plotting a Line Graph with the `plot` Function

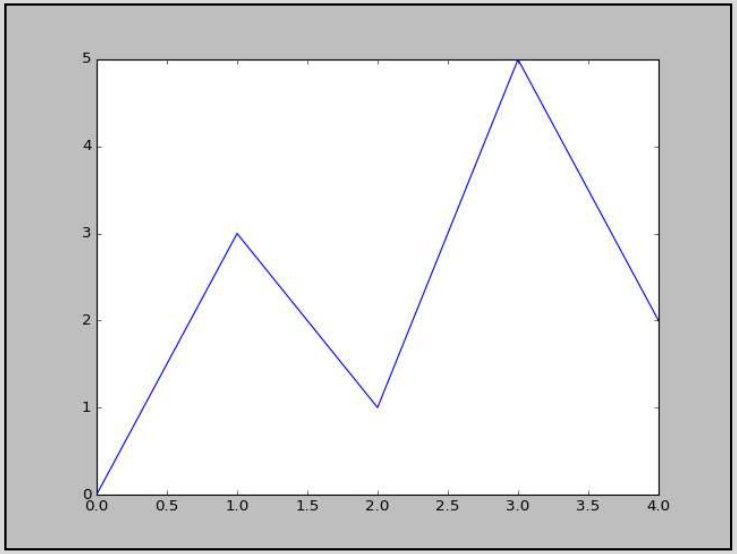
- **Use the `plot` function to create a line graph that connects a series of points with straight lines.**
- **The line graph has a horizontal *X* axis, and a vertical *Y* axis.**
- **Each point in the graph is located at a (*X*, *Y*) coordinate.**



Plotting a Line Graph with the `plot` Function

Program 7-20 (line_graph1.py)

```
 1 # This program displays a simple line graph.
 2 import matplotlib.pyplot as plt
 3
 4 def main():
 5     # Create lists with the X and Y coordinates of each data point.
 6     x_coords = [0, 1, 2, 3, 4]
 7     y_coords = [0, 3, 1, 5, 2]
 8
 9     # Build the line graph.
10     plt.plot(x_coords, y_coords)
11
12     # Display the line graph.
13     plt.show()
14
15 # Call the main function.
16 if __name__ == '__main__':
17     main()
```



Plotting a Line Graph with the `plot` Function

- **You can change the lower and upper limits of the *X* and *Y* axes by calling the `xlim` and `ylim` functions.**
  Example:

```
plt.xlim(xmin=1, xmax=100)
plt.ylim(ymin=10, ymax=50)
```

- **This code does the following:**
  - Causes the *X* axis to begin at 1 and end at 100
  - Causes the *Y* axis to begin at 10 and end at 50

Plotting a Line Graph with the `plot` Function

- **You can customize each tick mark's label with the `xticks` and `yticks` functions.**
- **These functions each take two lists as arguments.**
  - The first argument is a list of tick mark locations
  - The second argument is a list of labels to display at the specified locations.

```
plt.xticks([0, 1, 2, 3, 4],
           ['2016', '2017', '2018', '2019', '2020'])
plt.yticks([0, 1, 2, 3, 4, 5],
           ['$0m', '$1m', '$2m', '$3m', '$4m', '$5m'])
```

Program 7-20

```
1  # This program displays a simple line graph.
2  import matplotlib.pyplot as plt
3
4  def main():
5      # Create lists with the X,Y coordinates of each data point.
6      x_coords = [0, 1, 2, 3, 4]
7      y_coords = [0, 3, 1, 5, 2]
8
9      # Build the line graph.
10     plt.plot(x_coords, y_coords, marker='o')
11
12     # Add a title.
13     plt.title('Sales by Year')
14
15     # Add labels to the axes.
16     plt.xlabel('Year')
17     plt.ylabel('Sales')
18
```

Continued…

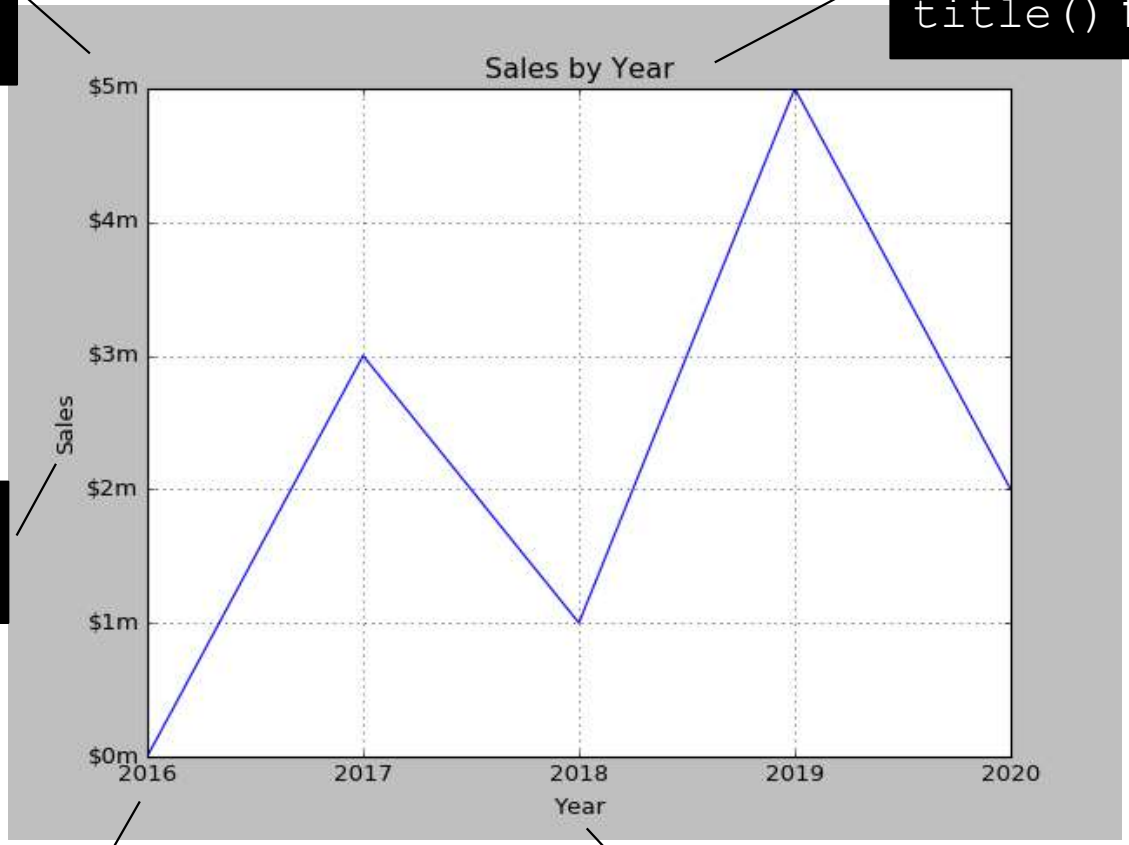
Program 7-24 (continued)

```
19      # Customize the tick marks.
20      plt.xticks([0, 1, 2, 3, 4],
21                 ['2016', '2017', '2018', '2019', '2020'])
22      plt.yticks([0, 1, 2, 3, 4, 5],
23                 ['$0m', '$1m', '$2m', '$3m', '$4m', '$5m'])
24
25      # Add a grid.
26      plt.grid(True)
27
28      # Display the line graph.
29      plt.show()
30
31  # Call the main function.
32  if __name__ == '__main__':
33      main()
```

Output of Program 7-24

Displayed by the `yticks()` function.

Displayed by the `title()` function.



Displayed by the `ylabel()` function.

Displayed by the `xticks()` function.

Displayed by the `xlabel()` function.

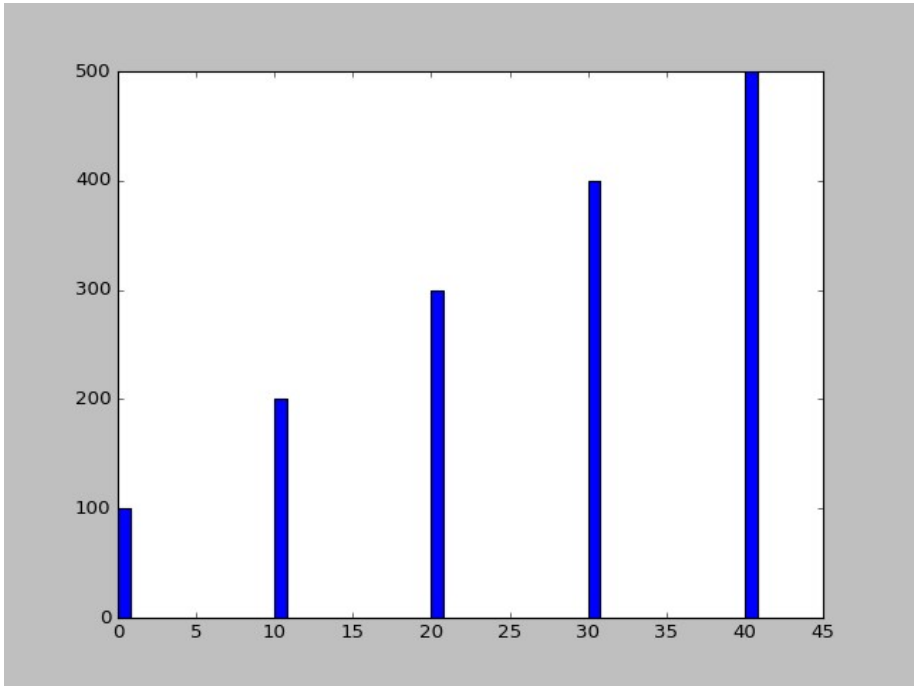
Plotting a Bar Chart

- **Use the `bar` function in the `matplotlib.pyplot` module to create a bar chart.**

- **The function needs two lists: one with the *X* coordinates of each bar's left edge, and another with the heights of each bar, along the *Y* axis.**

Plotting a Bar Chart

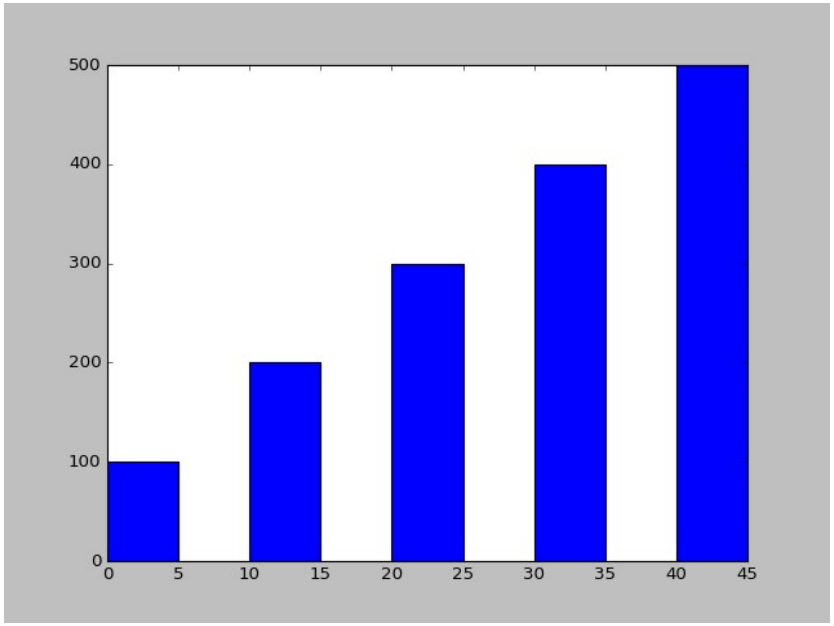
```
left_edges = [0, 10, 20, 30, 40]
heights = [100, 200, 300, 400, 500]

plt.bar(left_edges, heights)
plt.show()
```



Plotting a Bar Chart

- **The default width of each bar in a bar graph is 0.8 along the *X* axis.**
- **You can change the bar width by passing a third argument to the `bar` function.**

```
left_edges = [0, 10, 20, 30, 40]
heights = [100, 200, 300, 400, 500]
bar_width = 5

plt.bar(left_edges, heights, bar_width)
plt.show()
```



Plotting a Bar Chart

- The `bar` function has a `color` parameter that you can use to change the colors of the bars.
- The argument that you pass into this parameter is a tuple containing a series of color codes.

Color Code	Corresponding Color
`'b'`	Blue
`'g'`	Green
`'r'`	Red
`'c'`	Cyan
`'m'`	Magenta
`'y'`	Yellow
`'k'`	Black
`'w'`	White

Plotting a Bar Chart

- **Example of how to pass a tuple of color codes as a keyword argument:**

```
plt.bar(left_edges, heights, color=('r', 'g', 'b', 'w', 'k'))
```

- **The colors of the bars in the resulting bar chart will be as follows:**
  - The first bar will be red.
  - The second bar will be green.
  - The third bar will be blue.
  - The fourth bar will be white.
  - The fifth bar will be black.

Plotting a Bar Chart

- Use the `xlabel` and `ylabel` functions to add labels to the *X* and *Y* axes.
- Use the `xticks` function to display custom tick mark labels along the *X* axis
- Use the `yticks` function to display custom tick mark labels along the *Y* axis.

Plotting a Pie Chart

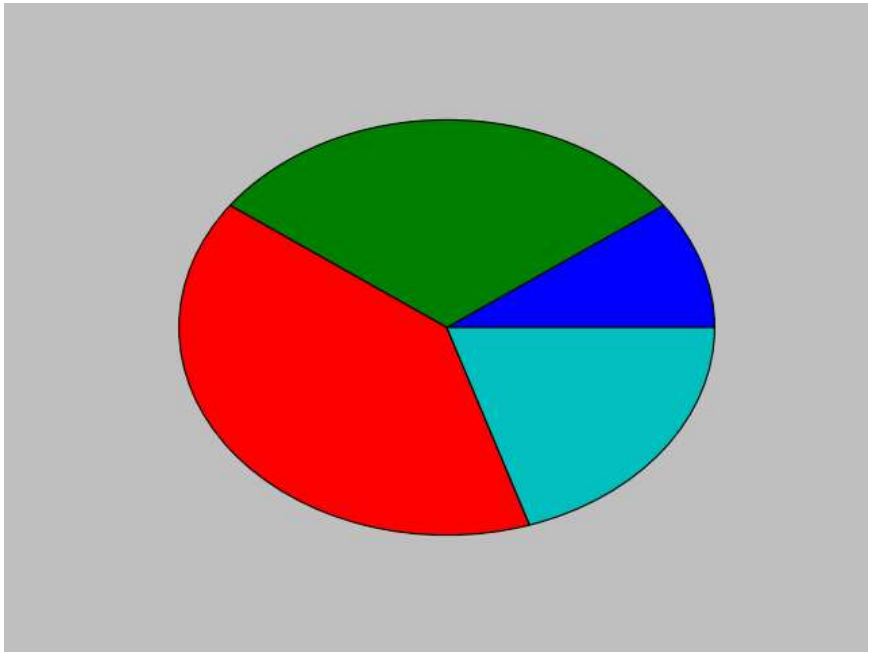
- **You use the `pie` function in the `matplotlib.pyplot` module to create a pie chart.**

- **When you call the `pie` function, you pass a list of values as an argument.**
  - The sum of the values will be used as the value of the whole.
  - Each element in the list will become a slice in the pie chart.
  - The size of a slice represents that element's value as a percentage of the whole.

Plotting a Pie Chart

- **Example**

```
values = [20, 60, 80, 40]
plt.pie(values)
plt.show()
```



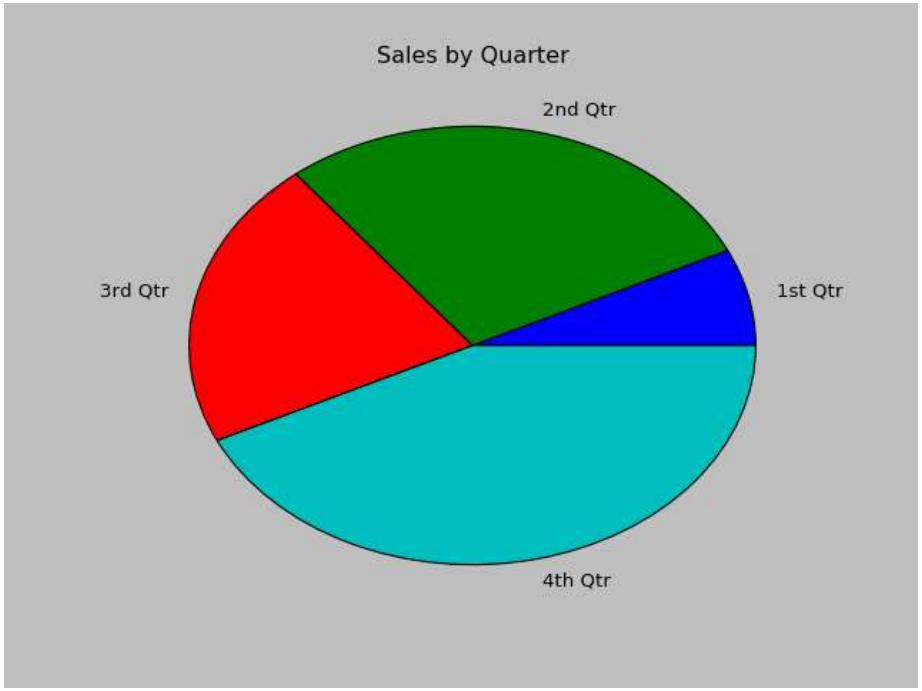
Plotting a Pie Chart

- The `pie` function has a `labels` parameter that you can use to display labels for the slices in the pie chart.
- The argument that you pass into this parameter is a list containing the desired labels, as strings.

Plotting a Pie Chart

- **Example**

```
sales = [100, 400, 300, 600]
slice_labels = ['1st Qtr', '2nd Qtr', '3rd Qtr', '4th Qtr']
plt.pie(sales, labels=slice_labels)
plt.title('Sales by Quarter')
plt.show()
```



Plotting a Pie Chart

- **The `pie` function automatically changes the color of the slices, in the following order:**
  - blue, green, red, cyan, magenta, yellow, black, and white.

- **You can specify a different set of colors, however, by passing a tuple of color codes as an argument to the `pie` function's `colors` parameter:**

```
plt.pie(values, colors=('r', 'g', 'b', 'w', 'k'))
```

- **When this statement executes, the colors of the slices in the resulting pie chart will be red, green, blue, white, and black.**

Summary

- **This chapter covered:**
  - Lists, including:
    - Repetition and concatenation operators
    - Indexing
    - Techniques for processing lists
    - Slicing and copying lists
    - List methods and built-in functions for lists
    - Two-dimensional lists
  - Tuples, including:
    - Immutability
    - Difference from and advantages over lists
  - Plotting charts and graphs with the **`matplotlib`** Package