WEEK 7

Files and Exceptions & Lists and Tuples

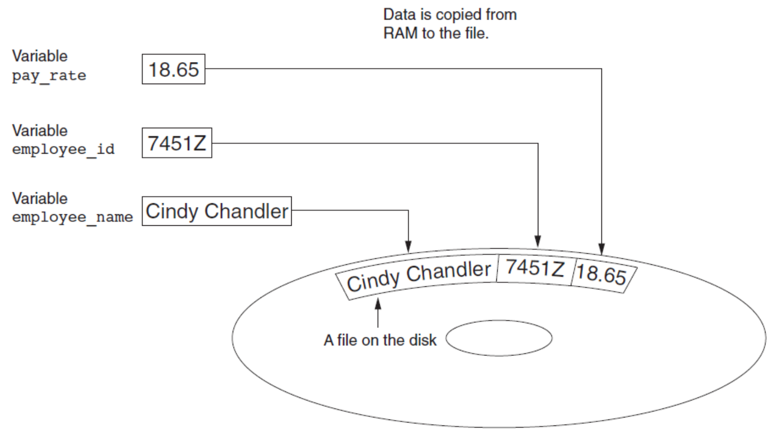
Topics

- Introduction to File Input and Output
- Using Loops to Process Files
- Processing Records
- Exceptions

Introduction to File Input and Output

- For program to retain data between the times it is run, you must save the data
  - Data is saved to a file, typically on computer disk
  - Saved data can be retrieved and used at a later time
- "**Writing data to**": saving data on a file
- **Output file**: a file that data is written to

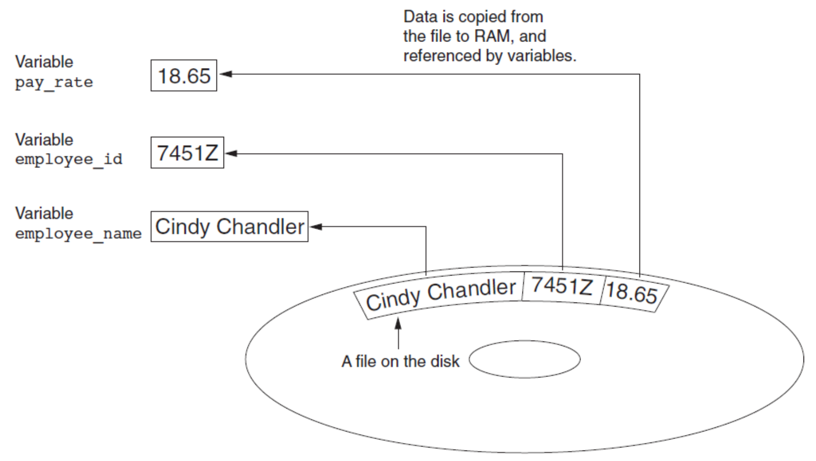
Figure 6-1 Writing data to a file



Introduction to File Input and Output (cont'd.)

- "Reading data from": process of retrieving data from a file
- Input file: a file from which data is read
- Three steps when a program uses a file
  - Open the file
  - Process the file
  - Close the file

Figure 6-2 Reading data from a file



Types of Files and File Access Methods

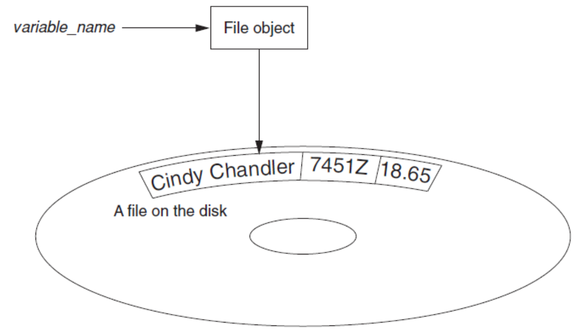
- In general, two types of files
  - Text file: contains data that has been encoded as text
  - Binary file: contains data that has not been converted to text
- Two ways to access data stored in file
  - Sequential access: file read sequentially from beginning to end, can't skip ahead
  - Direct access: can jump directly to any piece of data in the file

Filenames and File Objects

- Filename extensions: short sequences of characters that appear at the end of a filename preceded by a period
  - Extension indicates type of data stored in the file
- File object: object associated with a specific file
  - Provides a way for a program to work with the file: file object referenced by a variable

Filenames and File Objects (cont'd.)

Figure 6-4 A variable name references a file object that is associated with a file



Opening a File

- **`open` function: used to open a file**
  - Creates a file object and associates it with a file on the disk
  - General format:
  - `file_object = open(filename, mode)`
- **Mode: string specifying how the file will be opened**
  - Example: reading only (`'r'`), writing (`'w'`), and appending (`'a'`)

Specifying the Location of a File

- **If `open` function receives a filename that does not contain a path, assumes that file is in same directory as program**
- **If program is running and file is created, it is created in the same directory as the program**
  - Can specify alternative path and file name in the `open` function argument
    - Prefix the path string literal with the letter `r`

Writing Data to a File

- **Method: a function that belongs to an object**
  - Performs operations using that object
- **File object's `write` method used to write data to the file**
  - Format: *`file_variable`*`.write(`*`string`*`)`
- **File should be closed using file object `close` method**
  - Format: *`file_variable`*`.close()`

Reading Data From a File

- **`read` method: file object method that reads entire file contents into memory**
  - Only works if file has been opened for reading
  - Contents returned as a string
- **`readline` method: file object method that reads a line from the file**
  - Line returned as a string, including `'\n'`
- **Read position: marks the location of the next item to be read from a file**

Concatenating a Newline to and Stripping it From a String

- **In most cases, data items written to a file are values referenced by variables**
  - Usually necessary to concatenate a `'\n'` to data before writing it
    - Carried out using the `+` operator in the argument of the `write` method
- **In many cases need to remove `'\n'` from string after it is read from a file**
  - `rstrip` method: string method that strips specific characters from end of the string

Appending Data to an Existing File

- **When open file with `'w'` mode, if the file already exists it is overwritten**
- **To append data to a file use the `'a'` mode**
  - If file exists, it is not erased, and if it does not exist it is created
  - Data is written to the file at the end of the current contents

Writing and Reading Numeric Data

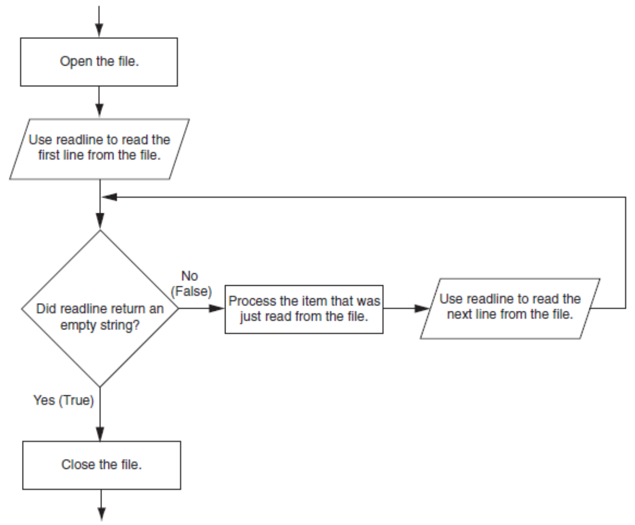
- **Numbers must be converted to strings before they are written to a file**
- **`str` function: converts value to string**
- **Number are read from a text file as strings**
  - Must be converted to numeric type in order to perform mathematical operations
  - Use `int` and `float` functions to convert string to numeric value

Using Loops to Process Files

- **Files typically used to hold large amounts of data**
  - Loop typically involved in reading from and writing to a file
- **Often the number of items stored in file is unknown**
  - The `readline` method uses an empty string as a sentinel when end of file is reached
    - Can write a while loop with the condition

```
      while line != ''
```

Figure 6-17 General logic for detecting the end of a file



Open the file.
Use readline to read the first line from the file.
Did readline return an empty string?
No (False)
Process the item that was just read from the file.
Use readline to read the next line from the file.
Yes (True)
Close the file.

Using Python's `for` Loop to Read Lines

- **Python allows the programmer to write a `for` loop that automatically reads lines in a file and stops when end of file is reached**
  - Format: `for line in file_object:`
  - `statements`
  - The loop iterates once over each line in the file

Processing Records

- **<u>Record</u>: set of data that describes one item**
- **<u>Field</u>: single piece of data within a record**
- **Write record to sequential access file by writing the fields one after the other**
- **Read record from sequential access file by reading each field until record complete**

Processing Records (cont'd.)

- **When working with records, it is also important to be able to:**
  - Add records
  - Display records
  - Search for a specific record
  - Modify records
  - Delete records

Exceptions

- **<u>Exception</u>: error that occurs while a program is running**
  - Usually causes program to abruptly halt
- **<u>Traceback</u>: error message that gives information regarding line numbers that caused the exception**
  - Indicates the type of exception and brief description of the error that caused exception to be raised

Exceptions (cont'd.)

- **Many exceptions can be prevented by careful coding**
  - Example: input validation
  - Usually involve a simple decision construct
- **Some exceptions cannot be avoided by careful coding**
  - Examples
    - Trying to convert non-numeric string to an integer
    - Trying to open for reading a file that doesn't exist

Exceptions (cont'd.)

- **<u>Exception handler</u>: code that responds when exceptions are raised and prevents program from crashing**
  - In Python, written as `try/except` statement
    - General format:
      ```
      try:
          statements
      except exceptionName:
          statements
      ```
    - <u>Try suite</u>: statements that can potentially raise an exception
    - <u>Handler</u>: statements contained in `except` block

Exceptions (cont'd.)

- **If statement in try suite raises exception:**
  - Exception specified in except clause:
    - Handler immediately following except clause executes
    - Continue program after try/except statement
  - Other exceptions:
    - Program halts with traceback error message
- **If no exception is raised, handlers are skipped**

Handling Multiple Exceptions

- **Often code in try suite can throw more than one type of exception**
  - Need to write `except` clause for each type of exception that needs to be handled
- **An `except` clause that does not list a specific exception will handle any exception that is raised in the try suite**
  - Should always be last in a series of `except` clauses

Displaying an Exception's Default Error Message

- **Exception object: object created in memory when an exception is thrown**
  - Usually contains default error message pertaining to the exception
  - Can assign the exception object to a variable in an `except` clause
    - Example: `except ValueError as err:`
  - Can pass exception object variable to `print` function to display the default error message

The `else` Clause

- **`try/except` statement may include an optional `else` clause, which appears after all the `except` clauses**
  - Aligned with `try` and `except` clauses
  - Syntax similar to `else` clause in decision structure
  - <u>Else suite</u>: block of statements executed after statements in try suite, only if no exceptions were raised
    - If exception was raised, the else suite is skipped

The `finally` Clause

- **`try/except` statement may include an optional `finally` clause, which appears after all the `except` clauses**
  - Aligned with `try` and `except` clauses
  - General format: `finally:`
  - `statements`
  - <u>Finally suite</u>: block of statements after the `finally` clause
    - Execute whether an exception occurs or not
    - Purpose is to perform cleanup before exiting

What If an Exception Is Not Handled?

- **Two ways for exception to go unhandled:**
  - No except clause specifying exception of the right type
  - Exception raised outside a try suite
- **In both cases, exception will cause the program to halt**
  - Python documentation provides information about exceptions that can be raised by different functions

Summary

- **This chapter covered:**
  - Types of files and file access methods
  - Filenames and file objects
  - Writing data to a file
  - Reading data from a file and determining when the end of the file is reached
  - Processing records
  - Exceptions, including:
    - Traceback messages
    - Handling exceptions

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
- 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.)

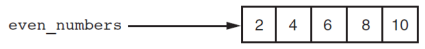


Figure 7-1 A list of integers

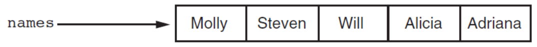


Figure 7-2 A list of strings

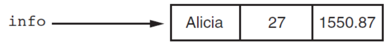


Figure 7-3 A list holding different types

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**
- **`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**

Table 7-1 A few of the list methods

Method	Description
`append(item)`	Adds *`item`* to the end of the list.
`index(item)`	Returns the index of the first element whose value is equal to item. A `ValueError` exception is raised if item is not found in the list.
`insert(index, item)`	Inserts *`item`* into the list at the specified *`index`*. When an item is inserted into a list, the list is expanded in size to accommodate the new item. The item that was previously at the specified index, and all the items after it, are shifted by one position toward the end of the list. No exceptions will occur if you specify an invalid index. If you specify an index beyond the end of the list, the item will be added to the end of the list. If you use a negative index that specifies an invalid position, the item will be inserted at the beginning of the list.
`sort()`	Sorts the items in the list so they appear in ascending order (from the lowest value to the highest value).
`remove(item)`	Removes the first occurrence of *`item`* from the list. A `ValueError` exception is raised if item is not found in the list.
`reverse()`	Reverses the order of the items in the list.

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

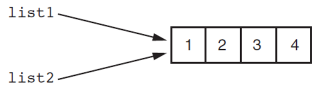
- **`del` statement: removes an element from a specific index in a list**
  - General format: `del list[i]`
- **`min` and `max` functions: built-in functions that returns the item that has the lowest or highest value in a sequence**
  - The sequence is passed as an argument

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.)

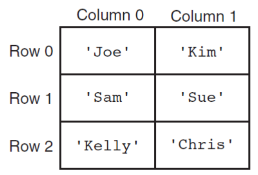
- **A function can return a reference to a list**
- **To save the contents of a list to a file:**
  - Use the file object's `writelines` method
    - Does not automatically write `\n` at then end of each item
  - Use a `for` loop to write each element and `\n`
- **To read data from a file use the file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

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.)

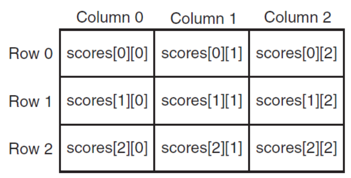
Figure 7-5 A two-dimensional list



	Column 0	Column 1
Row 0	`'Joe'`	`'Kim'`
Row 1	`'Sam'`	`'Sue'`
Row 2	`'Kelly'`	`'Chris'`

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**
  - Very similar to a list
  - Once it is created it cannot be changed
  - Format: `tuple_name = (item1, item2)`
  - Tuples support operations as lists
    - Subscript indexing for retrieving elements
    - Methods such as `index`
    - Built in functions such as `len, min, max`
    - Slicing expressions
    - The `in`, `+`, and `*` operators

Tuples (cont'd.)

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

Tuples (cont'd.)

- **Advantages for using tuples over lists:**
  - Processing tuples is faster than processing lists
  - Tuples are safe
  - 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`

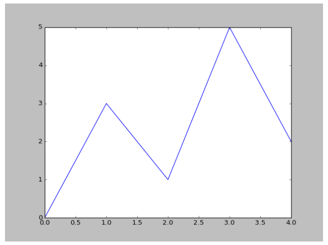
- The `matplotlib` package contains a module named `pyplot` that you will need to import.
- Use the following `import` statement to import the module and create an alias named `plt`:

```
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 7-19 (line_graph1.py)

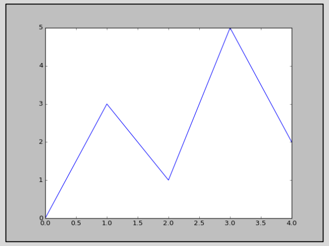
```
# This program displays a simple line graph.
import matplotlib.pyplot as plt

def main():
    # Create lists with the X and Y coordinates of each data point.
    x_coords = [0, 1, 2, 3, 4]
    y_coords = [0, 3, 1, 5, 2]

    # Build the line graph.
    plt.plot(x_coords, y_coords)

    # Display the line graph.
    plt.show()

# Call the main function.
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 7-24

```
# This program displays a simple line graph.
import matplotlib.pyplot as plt

def main():
    # Create lists with the X,Y coordinates of each data point.
    x_coords = [0, 1, 2, 3, 4]
    y_coords = [0, 3, 1, 5, 2]

    # Build the line graph.
    plt.plot(x_coords, y_coords, marker='o')

    # Add a title.
    plt.title('Sales by Year')

    # Add labels to the axes.
    plt.xlabel('Year')
    plt.ylabel('Sales')

```

Continued…

Program 7-24 (continued)

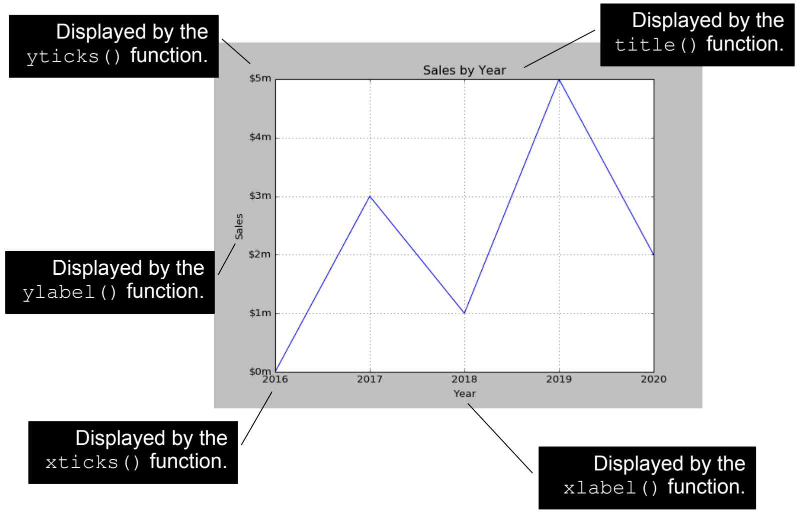
```
    # Customize the tick marks.
    plt.xticks([0, 1, 2, 3, 4],
               ['2016', '2017', '2018', '2019', '2020'])
    plt.yticks([0, 1, 2, 3, 4, 5],
               ['$0m', '$1m', '$2m', '$3m', '$4m', '$5m'])

    # Add a grid.
    plt.grid(True)

    # Display the line graph.
    plt.show()

# Call the main function.
main()
```

Output of Program 7-24



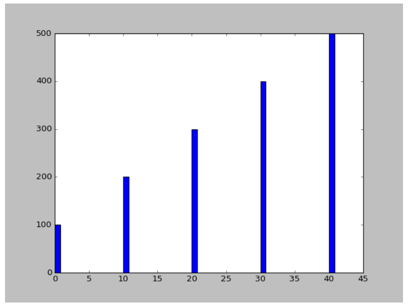
Plotting a Bar Chart

- 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()
```

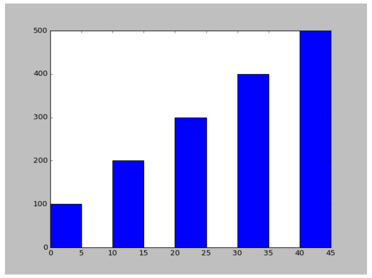


Plotting a Bar Chart

- 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.**

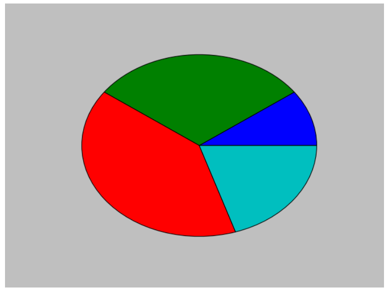
Plotting a Pie Chart

- **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()
```



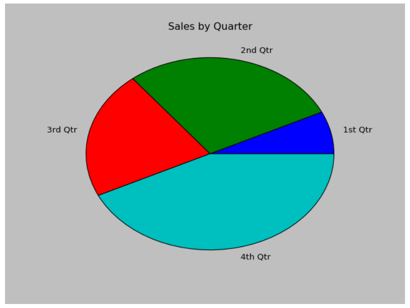
Plotting a Pie Chart

- **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