

Files and Exceptions

Topics

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

2

Introduction to File Input and Output (1 of 4)

- 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

3

Introduction to File Input and Output (2 of 4)

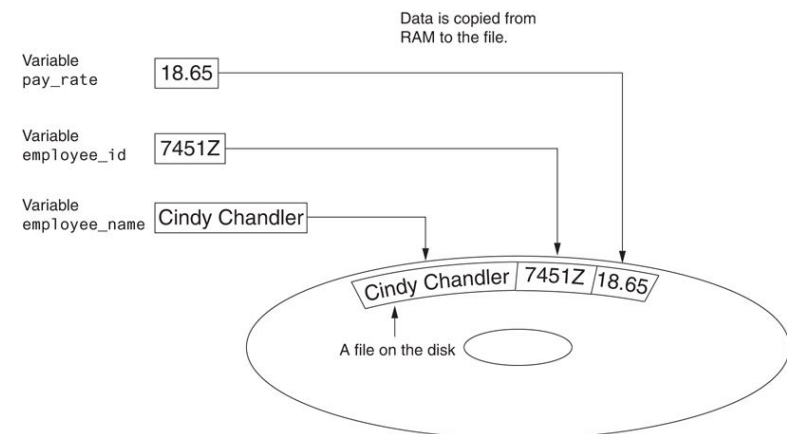


Figure 6-1 Writing data to a file

4

Introduction to File Input and Output (3 of 4)

- “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

5

Introduction to File Input and Output (4 of 4)

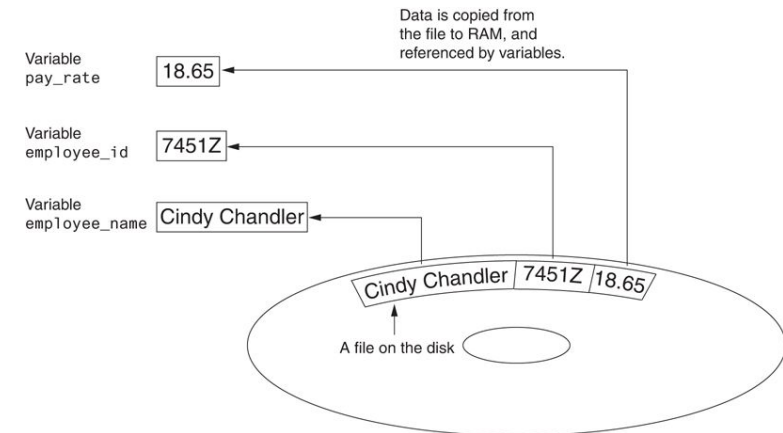


Figure 6-2 Reading data from a file

6

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

7

Filenames and File Objects (1 of 2)

- 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

8

Filenames and File Objects (2 of 2)

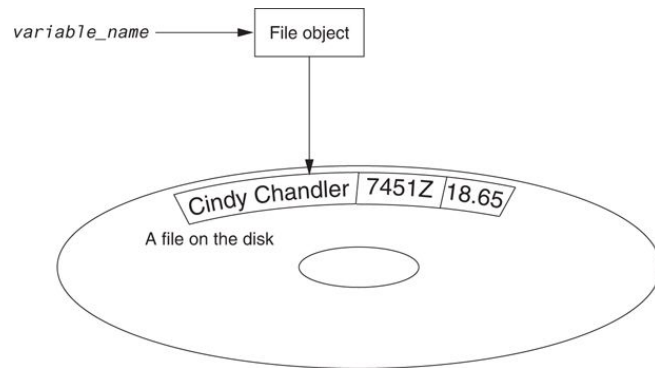


Figure 6-4 A variable name references a file object that is associated with a file

9

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')

10

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`

11

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()`

12

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

13

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

14

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

15

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

16

Using Loops to Process Files (1 of 2)

- 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 != ''
```

17

Using Loops to Process Files (2 of 2)

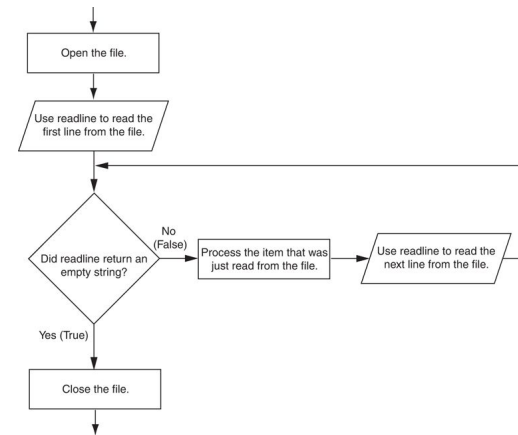


Figure 6-17 General logic for detecting the end of a file

18

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

19

## Processing Records (1 of 2)

- **Record**: set of data that describes one item
- **Field**: 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

20

## Processing Records (2 of 2)

- 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

21

## Exceptions (1 of 4)

- Exception: error that occurs while a program is running
  - Usually causes program to abruptly halt
- Traceback: 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

22

## Exceptions (2 of 4)

- 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

23

## Exceptions (3 of 4)

- Exception handler: 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`
    - Try suite: statements that can potentially raise an exception
    - Handler: statements contained in `except` block

24

## Exceptions (4 of 4)

- 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

25

## 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

26

## 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

27

## 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
  - Else suite: block of statements executed after statements in try suite, only if no exceptions were raised
    - If exception was raised, the else suite is skipped

28

## 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`
  - **Finally suite**: block of statements after the `finally` clause
    - Execute whether an exception occurs or not
    - Purpose is to perform cleanup before exiting

29

## 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

30

## 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

31

## List and Tuples & Introduction to Plotting



## Topics (1 of 2)

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

2

## Topics (2 of 2)

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

3

## 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

4

## Introduction to Lists (1 of 2)

- 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

5

## Introduction to Lists (2 of 2)

6



Figure 7-1 A list of integers



Figure 7-2 A list of strings



Figure 7-3 A list holding different types

6

## 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:`

7

## 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.

8

## 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

9

## 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

10

## 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

11

## 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

12

## 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

13

## List Methods and Useful Built-in Functions (1 of 4)

- `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

14

## List Methods and Useful Built-in Functions (2 of 4)

- `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

15

## List Methods and Useful Built-in Functions (3 of 4)

Table 7-1 A few of the list methods

| Method                           | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|----------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>append(item)</code>        | Adds <i>item</i> to the end of the list.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <code>index(item)</code>         | Returns the index of the first element whose value is equal to <i>item</i> . A <code>ValueError</code> exception is raised if <i>item</i> is not found in the list.                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <code>insert(index, item)</code> | Inserts <i>item</i> into the list at the specified <i>index</i> . 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 <i>index</i> , 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 <i>index</i> . If you specify an <i>index</i> beyond the end of the list, the item will be added to the end of the list. If you use a negative <i>index</i> that specifies an invalid position, the item will be inserted at the beginning of the list. |
| <code>sort()</code>              | Sorts the items in the list so they appear in ascending order (from the lowest value to the highest value).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| <code>remove(item)</code>        | Removes the first occurrence of <i>item</i> from the list. A <code>ValueError</code> exception is raised if <i>item</i> is not found in the list.                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <code>reverse()</code>           | Reverses the order of the items in the list.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

16

## List Methods and Useful Built-in Functions (4 of 4)

- `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

17

## Copying Lists (1 of 2)

- 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

18

## Copying Lists (2 of 2)

19

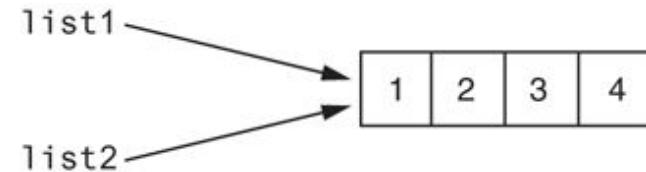


Figure 7-5 `list1` and `list2` reference the same list

19

## Processing Lists (1 of 2)

- 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

20

## Processing Lists (2 of 2)

- 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 the 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

21

## List Comprehensions (1 of 7)

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

22

## List Comprehensions (2 of 7)

- 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]
```

23

## List Comprehensions (3 of 7)

```
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.

24

## List Comprehensions (4 of 7)

```
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]`

25

## List Comprehensions (5 of 7)

```
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]

26

## List Comprehensions (6 of 7)

- 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]
```

27

## List Comprehensions (7 of 7)

```
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]

28

## Two-Dimensional Lists (1 of 3)

- 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

29

## Two-Dimensional Lists (2 of 3)

30

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

Figure 7-8 A two-dimensional list

30

## Two-Dimensional Lists (3 of 3)

31

|       | 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] |

Figure 7-10 Subscripts for each element of the scores list

31

## Tuples (1 of 3)

- **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

32

## Tuples (2 of 3)

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

33



## Tuples (3 of 3)

- 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

34

## Plotting Data with matplotlib (1 of 4)

- 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.

35

## Plotting Data with matplotlib (2 of 4)

- 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.

36

## Plotting Data with matplotlib (3 of 4)

- 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.

37

## Plotting Data with `matplotlib` (4 of 4)

- 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`:

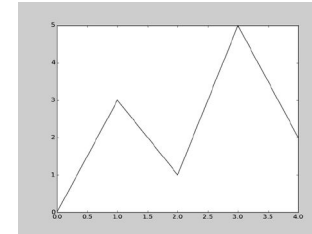
```
import matplotlib.pyplot as plt
```

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

38

## Plotting a Line Graph with the `plot` Function (1 of 4)

- 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.

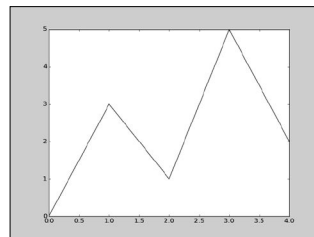


39

## Plotting a Line Graph with the `plot` Function (2 of 4)

### 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
6 # point.
7 x_coords = [0, 1, 2, 3, 4]
8 y_coords = [0, 3, 1, 5, 2]
9
10 # Build the line graph.
11 plt.plot(x_coords, y_coords)
12
13 # Display the line graph.
14 plt.show()
15
16 # Call the main function.
17 if __name__ == '__main__':
18 main()
```



40

## Plotting a Line Graph with the `plot` Function (3 of 4)

- 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

41

## Plotting a Line Graph with the plot Function (4 of 4)

- 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'])
```

42

## 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_coors = [0, 1, 2, 3, 4]
7 y_coors = [0, 3, 1, 5, 2]
8
9 # Build the line graph.
10 plt.plot(x_coors, y_coors, 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. . .

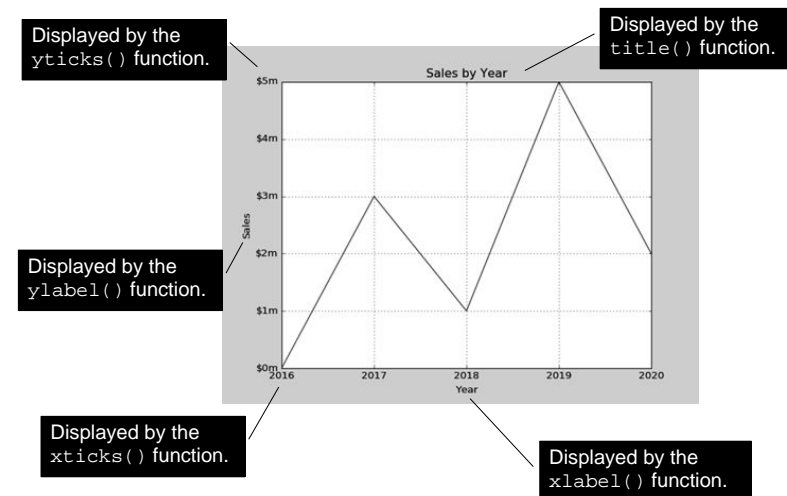
43

## Program 7-24

```
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()
```

44

## Output of Program 7-24



45

## Plotting a Bar Chart (1 of 6)

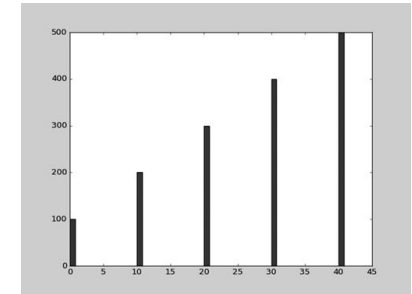
- Use the `bar` function in the `matplotlib.pyplot` module to create a bar chart.
- The function needs two lists: one with the *X* coordinates of each bar's left edge, and another with the heights of each bar, along the *Y* axis.

46

## Plotting a Bar Chart (2 of 6)

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

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



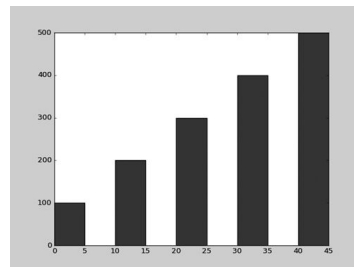
47

## Plotting a Bar Chart (3 of 6)

- 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()
```



48

## Plotting a Bar Chart (4 of 6)

- 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               |

49

## Plotting a Bar Chart (5 of 6)

- 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.

50

## Plotting a Bar Chart (6 of 6)

- 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.

51

## Plotting a Pie Chart (1 of 4)

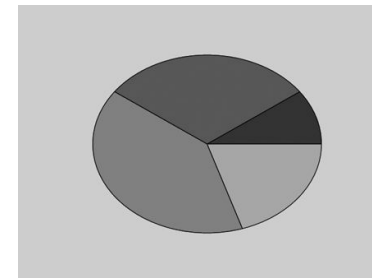
- You use the `pie` function in the `matplotlib.pyplot` module to create a pie chart.
- When you call the `pie` function, you pass a list of values as an argument.
  - The sum of the values will be used as the value of the whole.
  - Each element in the list will become a slice in the pie chart.
  -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

52

## Plotting a Pie Chart (2 of 4)

- Example

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



53

## Plotting a Pie Chart (3 of 4)

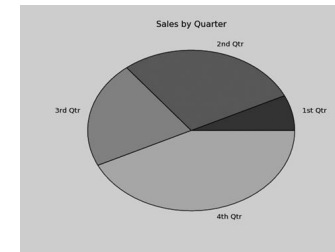
- The `pie` function has a `labels` parameter that you can use to display labels for the slices in the pie chart.
- The argument that you pass into this parameter is a list containing the desired labels, as strings.

54

## Plotting a Pie Chart (4 of 4)

- 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()
```



55

## 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.

56

## 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

57