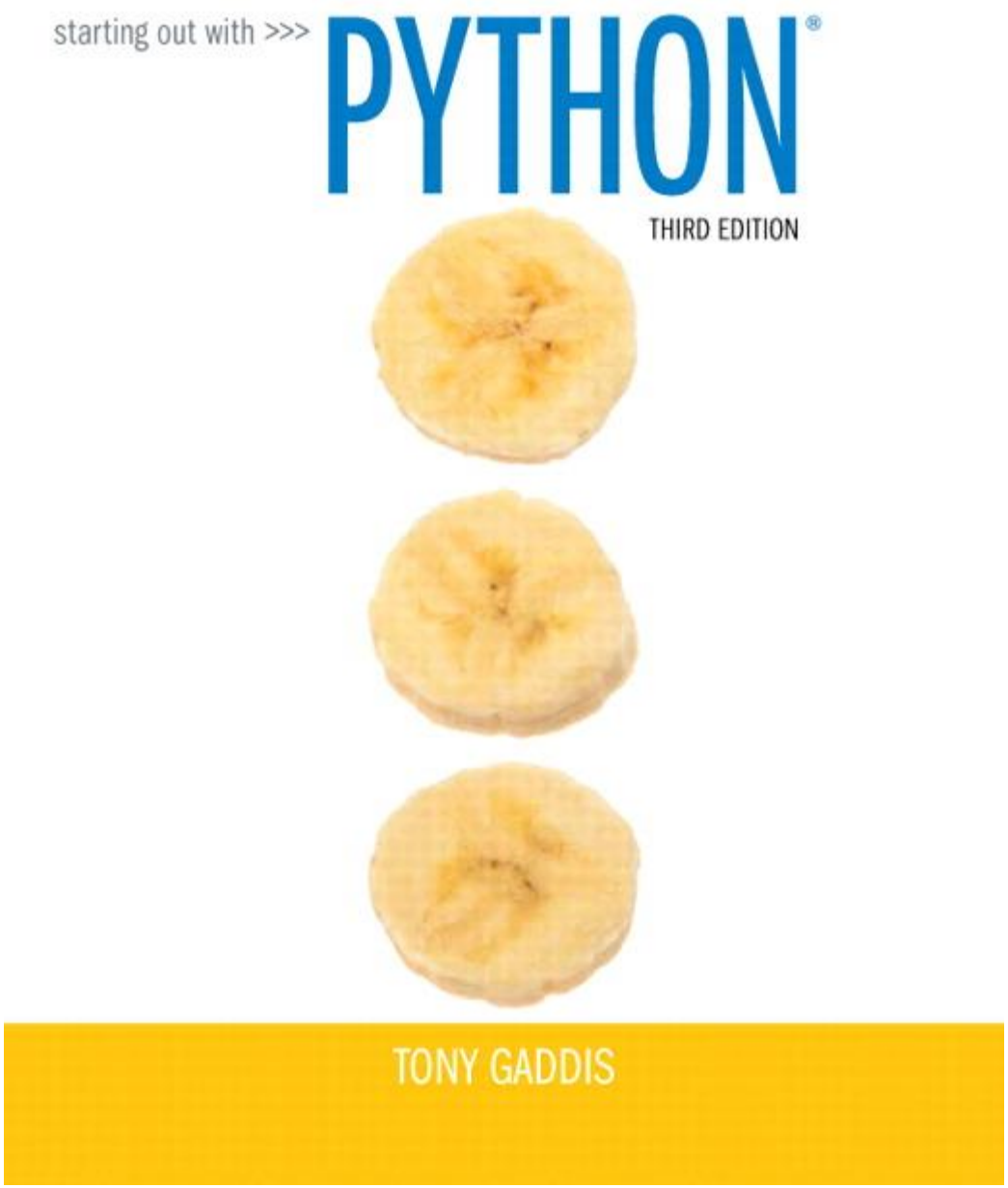
CHAPTER 6

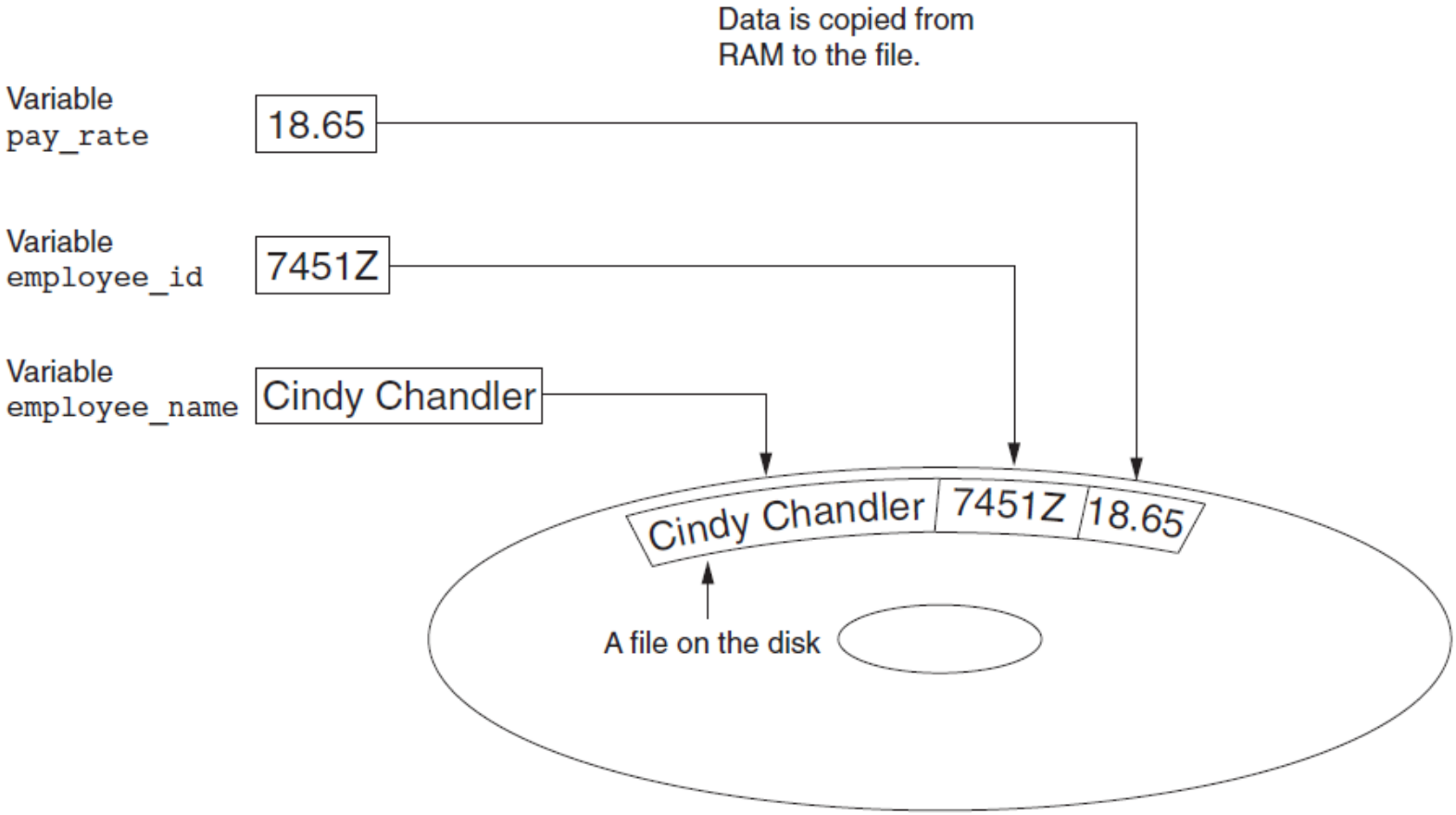
Files and Exceptions



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**

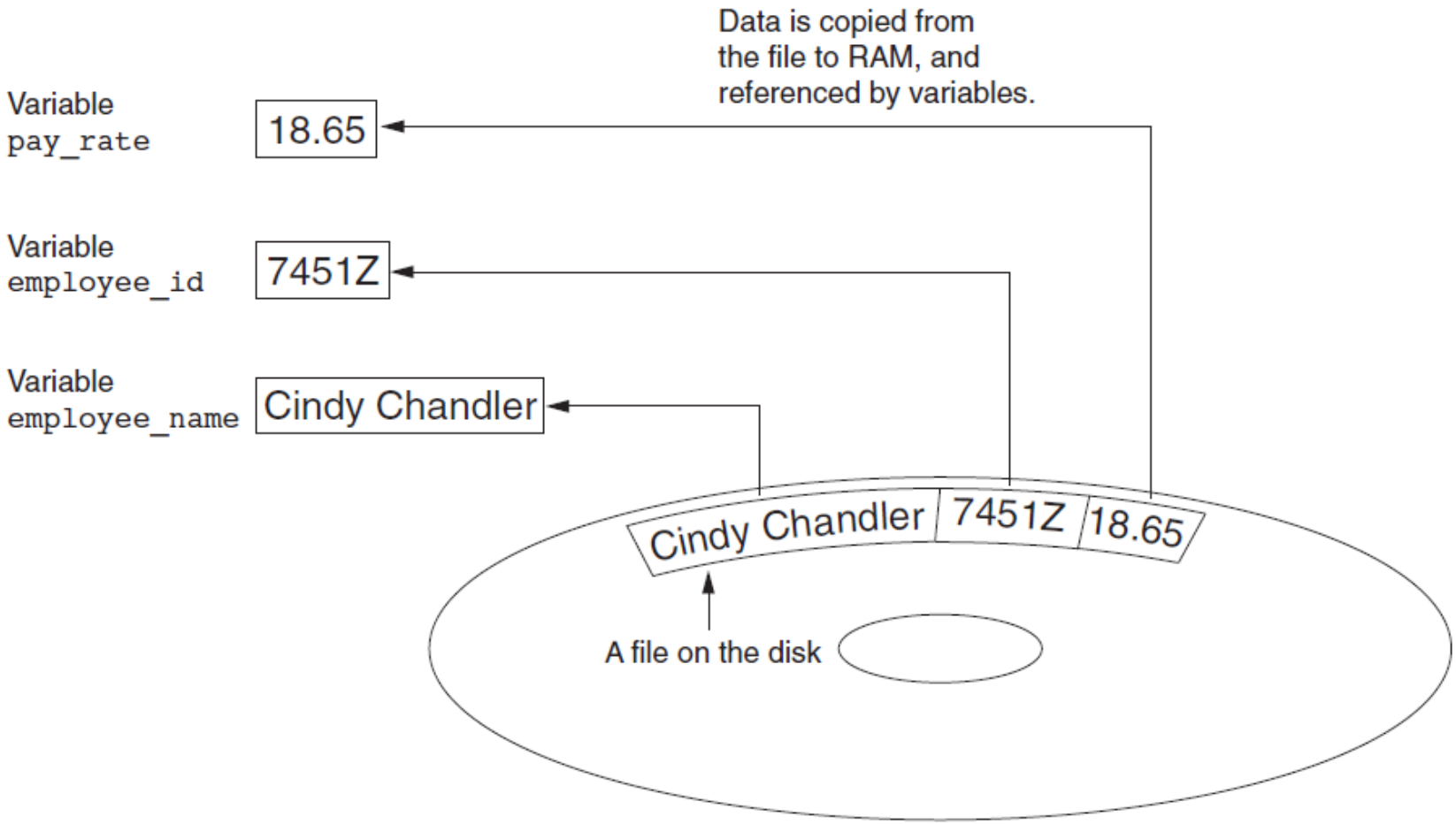
Figure 6-1 Writing data to a file



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

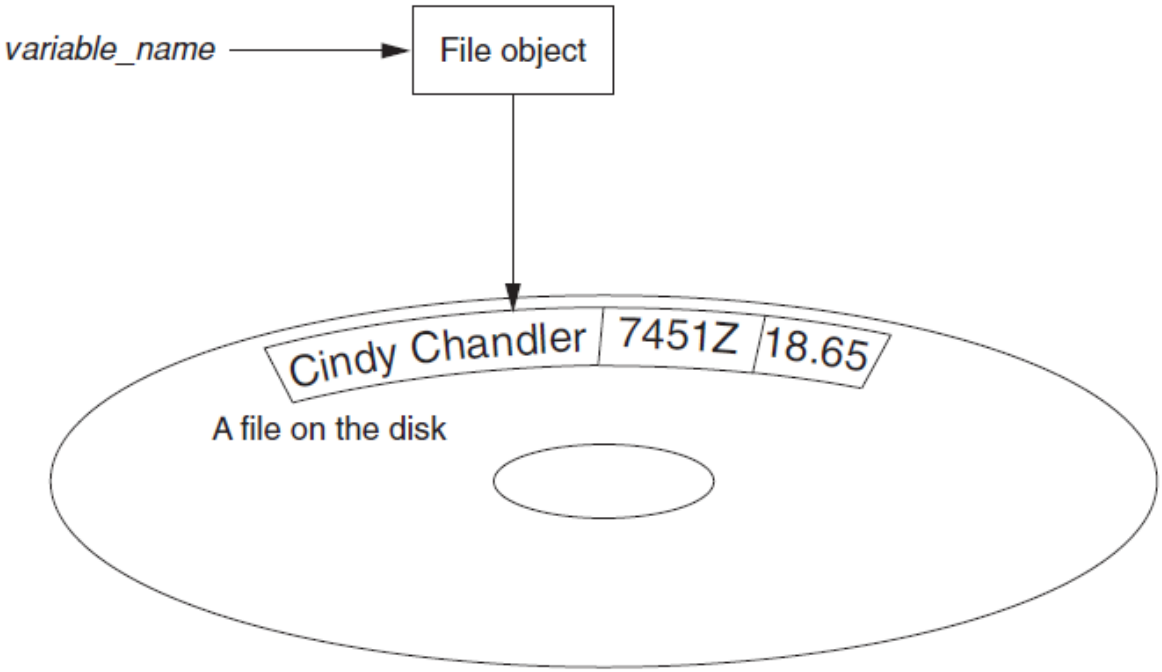
- **In general, two types of files**
  - Text file: contains data that has been encoded as text
  - Binary file: contains data that has not been converted to text
- **Two ways to access data stored in file**
  - Sequential access: file read sequentially from beginning to end, can’t skip ahead
  - Direct 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()`

```
# This program writes three lines of data
# to a file.
def main():
    # Open a file named philosophers.txt.
    outfile = open('philosophers.txt', 'w')

    # Write the names of three philosphers
    # to the file.
    outfile.write('John Locke\n')
    outfile.write('David Hume\n')
    outfile.write('Edmund Burke\n')

    # Close the file.
    outfile.close()

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

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**

```
# This program reads and displays the contents
# of the philosophers.txt file.
def main():
    # Open a file named philosophers.txt.
    infile = open('philosophers.txt', 'r')

    # Read the file's contents.
    file_contents = infile.read()    # read all file content

    # Close the file.
    infile.close()

    # Print the data that was read into
    # memory.
    print(file_contents)

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

```
# This program reads the contents of the
# philosophers.txt file one line at a time.
def main():
    # Open a file named philosophers.txt.
    infile = open('philosophers.txt', 'r')

    # Read three lines from the file
    line1 = infile.readline()          # read line by line
    line2 = infile.readline()
    line3 = infile.readline()

    # Close the file.
    infile.close()

    # Print the data that was read into
    # memory.
    print(line1)
    print(line2)
    print(line3)

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

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

```
# This program gets three names from the user
# and writes them to a file.

def main():
    # Get three names.
    print('Enter the names of three friends.')
    name1 = input('Friend #1: ')
    name2 = input('Friend #2: ')
    name3 = input('Friend #3: ')

    # Open a file named friends.txt.
    myfile = open('friends.txt', 'w')

    # Write the names to the file.
    myfile.write(name1 + '\n')
    myfile.write(name2 + '\n')
    myfile.write(name3 + '\n')

    # Close the file.
    myfile.close()
    print('The names were written to friends.txt.')

main()
```

Concatenating a Newline to and Stripping it From a String

- **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

```
# This program reads the contents of the
# philosophers.txt file one line at a time.
def main():
    # Open a file named philosophers.txt.
    infile = open('philosophers.txt', 'r')

    # Read three lines from the file
    line1 = infile.readline()
    line2 = infile.readline()

    # Strip the \n from each string.
    line1 = line1.rstrip('\n')
    line2 = line2.rstrip('\n')

    # Close the file.
    infile.close()

    # Print the data that was read into
    # memory.
    print(line1)
    print(line2)

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

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

```
# This program demonstrates how numbers
# must be converted to strings before they
# are written to a text file.

def main():
    # Open a file for writing.
    outfile = open('numbers.txt', 'w')

    # Get three numbers from the user.
    num1 = int(input('Enter a number: '))
    num2 = int(input('Enter another number: '))
    num3 = int(input('Enter another number: '))

    # Write the numbers to the file.

outfile.write("This is num1: {}\n".format(num1))

    # Close the file.
    outfile.close()
    print('Data written to numbers.txt')

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

```
def main():
    # Open a file for reading.
    infile = open('numbers.txt', 'r')

    # Read three numbers from the file.
    num1 = int(infile.readline())
    num2 = int(infile.readline())
    num3 = int(infile.readline())

    # Close the file.
    infile.close()

    # Add the three numbers.
    total = num1 + num2 + num3

    # Display the numbers and their total.
    print('The numbers are:', num1, num2, num3)
    print('Their total is:', total)

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

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

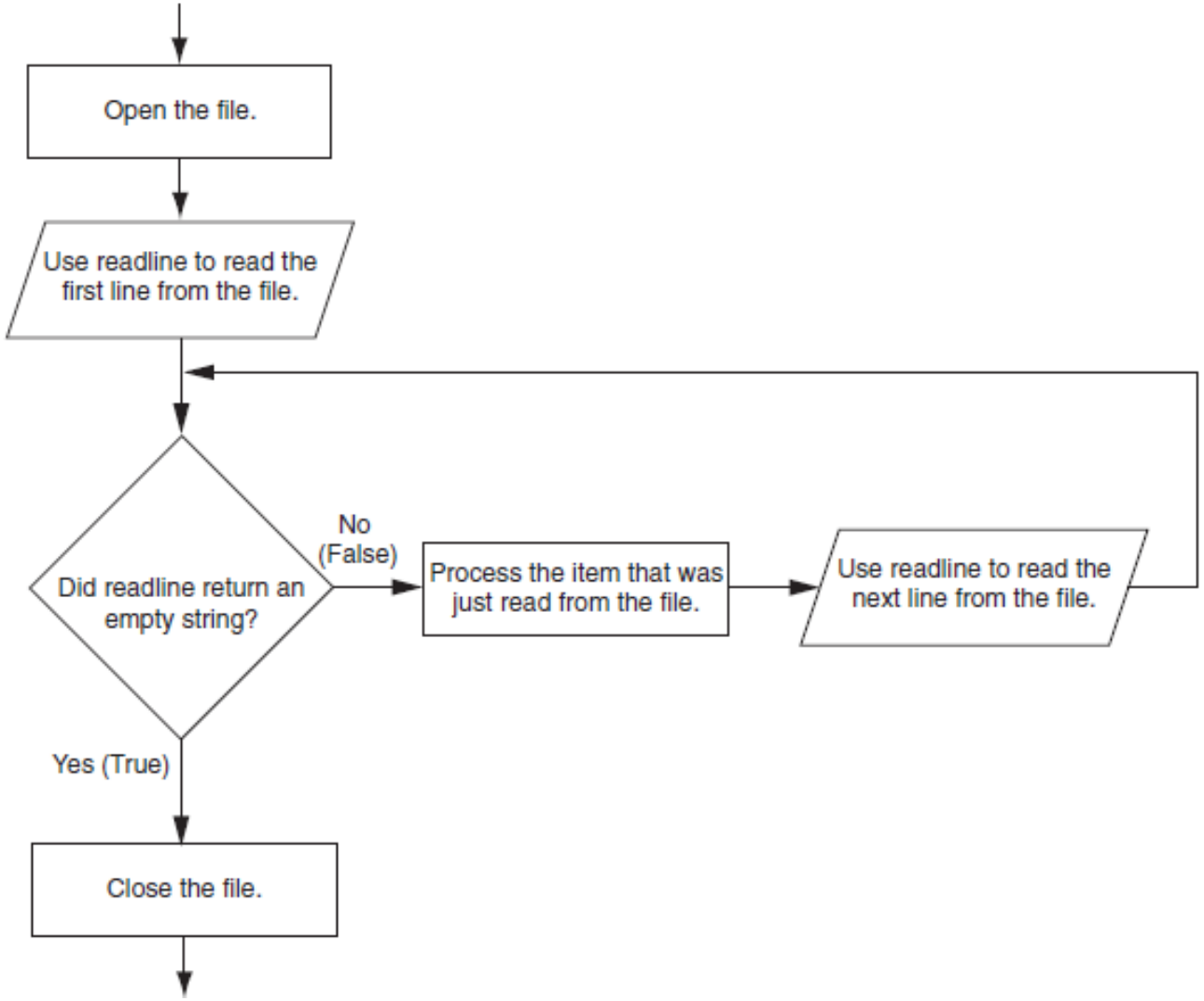
```
def main():
    # Get the number of days.
    num_days = int(input('For how many days do ' + \
                         'you have sales? '))

    # Open a new file named sales.txt.
    sales_file = open('sales.txt', 'w')

    # Get the amount of sales for each day and write
    # it to the file.
    for count in range(1, num_days + 1):
        # Get the sales for a day.
        sales = float(input('Enter the sales for day #' + \
                            str(count) + ': '))

        # Write the sales amount to the file.
        sales_file.write(str(sales) + '\n')

    # Close the file.
    sales_file.close()
    print('Data written to sales.txt.')

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

Using Loops to Process Files

- **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



```
def main():
    # Open the sales.txt file for reading.
    sales_file = open('sales.txt', 'r')

    # don't convert to a number yet. We still
    # need to test for an empty string.
    line = sales_file.readline()

    while line != '':
        # Convert line to a float.
        amount = float(line)

        # Format and display the amount.
        print(format(amount, '.2f'))

        # Read the next line.
        line = sales_file.readline()

    # Close the file.
    sales_file.close()

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

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

```
# This program uses the for loop to read
# all of the values in the sales.txt file.

def main():
    # Open the sales.txt file for reading.
    sales_file = open('sales.txt', 'r')

    # Read all the lines from the file.
    for line in sales_file:
        # Convert line to a float.
        amount = float(line)
        # Format and display the amount.
        print(format(amount, '.2f'))

    # Close the file.
    sales_file.close()

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

Processing Records

- **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**

```
def main():
    # Get the number of employee records to create.
    num_emps = int(input('How many employee records ' + \
                         'do you want to create? '))
    # Open a file for writing.
    emp_file = open('employees.txt', 'w')

    # Get each employee's data and write it to the file.
    for count in range(1, num_emps + 1):
        # Get the data for an employee.
        print('Enter data for employee #', count, sep='')
        name = input('Name: ')
        id_num = input('ID number: ')
        dept = input('Department: ')

        # Write the data as a record to the file.
        emp_file.write(name + '\n')
        emp_file.write(id_num + '\n')
        emp_file.write(dept + '\n')

        # Display a blank line.
        print()

    # Close the file.
    emp_file.close()
    print('Employee records written to employees.txt.')
main()
```

```
def main():
    # Open the employees.txt file.
    emp_file = open('employees.txt', 'r')
    name = emp_file.readline()

    # If a field was read, continue processing.
    while name != '':
        id_num = emp_file.readline()
        dept = emp_file.readline()

        # Strip the newlines from the fields.
        name = name.rstrip('\n')
        id_num = id_num.rstrip('\n')
        dept = dept.rstrip('\n')

        # Display the record.
        print('Name:', name)
        print('ID:', id_num)
        print('Dept:', dept)
        print()
        name = emp_file.readline()

    emp_file.close()

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

Exceptions

- **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

```
# This program divides a number by another number.

def main():
    # Get two numbers.
    num1 = int(input('Enter a number: '))
    num2 = int(input('Enter another number: '))

    # Divide num1 by num2 and display the result.
    result = num1 / num2
    print(num1, 'divided by', num2, 'is', result)

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

```
Enter a number: 10
Enter another number: 0
Traceback (most recent call last):
  File "ch6-division.py", line 13, in <module>
    main()
  File "ch6-division.py", line 9, in main
    result = num1 / num2
ZeroDivisionError: division by zero
```

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.)

- **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

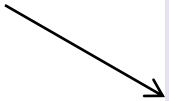
Exceptions (cont'd.)

- **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**

```
# This program calculates gross pay.

def main():
    # Get the number of hours worked.
    hours = int(input('How many hours did you work? '))

    # Get the hourly pay rate.
    pay_rate = float(input('Enter your hourly pay rate: '))

    # Calculate the gross pay.
    gross_pay = hours * pay_rate

    # Display the gross pay.
    print('Gross pay: $', format(gross_pay, ',.2f'), sep='')

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

```
How many hours did you work? kk
Traceback (most recent call last):
  File "ch6-gross_pay1.py", line 17, in <module>
    main()
  File "ch6-gross_pay1.py", line 5, in main
    hours = int(input('How many hours did you work? '))
ValueError: invalid literal for int() with base 10: 'kk'
```

Exception name



```
# This program calculates gross pay.

def main():
    try:
        # Get the number of hours worked.
        hours = int(input('How many hours did you work? '))

        # Get the hourly pay rate.
        pay_rate = float(input('Enter your hourly pay rate: '))

        # Calculate the gross pay.
        gross_pay = hours * pay_rate

        # Display the gross pay.
        print('Gross pay: $', format(gross_pay, ',.2f'), sep='')

    except ValueError:
        print('ERROR: Hours worked and hourly pay rate must')
        print('be valid integers.')

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

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

```
def main():
    # Initialize an accumulator.
    total = 0.0

    try:
        # Open the sales_data.txt file.
        infile = open('sales_data.txt', 'r')

        # Read the values from the file and
        # accumulate them.
        for line in infile:
            amount = float(line)
            total += amount

        # Close the file.
        infile.close()

        # Print the total.
        print(format(total, ',.2f'))

    except IOError:
        print('An error occured trying to read the file.')

    except ValueError:
        print('Non-numeric data found in the file.')

    except:
        print('An error occured.')

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

Handling Multiple Exceptions

- **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

```
# This program displays the total of the
# amounts in the sales_data.txt file.

def main():
    # Initialize an accumulator.
    total = 0.0

    try:
        # Open the sales_data.txt file.
        infile = open('sales_data.txt', 'r')

        # Read the values from the file and
        # accumulate them.
        for line in infile:
            amount = float(line)
            total += amount

        # Close the file.
        infile.close()

        # Print the total.
        print(format(total, ',.2f'))
    except:
        print('An error occurred.')

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

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 Exception as e:`
  - Can pass exception object variable to `print` function to display the default error message

```
# This program calculates gross pay.

def main():
    try:
        # Get the number of hours worked.
        hours = int(input('How many hours did you work? '))

        # Get the hourly pay rate.
        pay_rate = float(input('Enter your hourly pay rate: '))

        # Calculate the gross pay.
        gross_pay = hours * pay_rate

        # Display the gross pay.
        print('Gross pay: $', format(gross_pay, ',.2f'), sep='')
    except Exception as e:
        print("The exception {} occurs.".format(e))

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

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

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

```
# This program calculates gross pay.

def main():
    success = True
    try:
        hours = int(input('How many hours did you work? '))
        pay_rate = float(input('Enter your hourly pay rate: '))
    except Exception as e:
        print("The exception {} occurs.".format(e))
        success = False
    else:
        gross_pay = hours * pay_rate
        print('Gross pay: $', format(gross_pay, ',.2f'), sep='')
    finally:
        print("Execution result is {}".format(success))

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

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

If you think you can ignore an exception

- **pass it**

```
# This program calculates gross pay.

def main():
    success = True
    try:
        hours = int(input('How many hours did you work? '))
        pay_rate = float(input('Enter your hourly pay rate: '))
    except Exception as e:
        pass
```

Open file by with

```
try:
    f = open('/path/', 'r')
    print(f.read())
finally:
    if f:
        f.close()
```

You need a complex logic to check if the file was opened

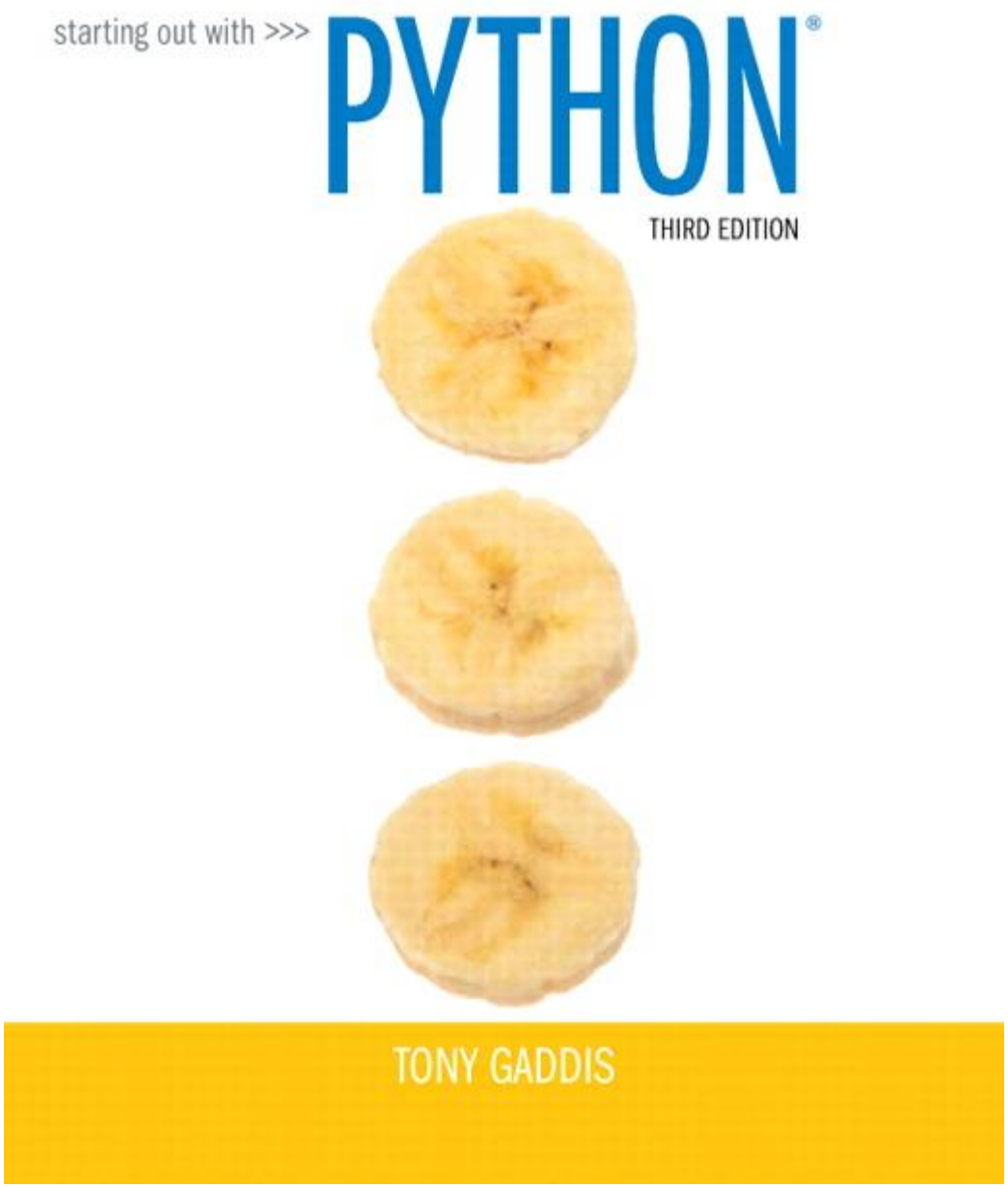
If opened, you need to call the f.close()

```
with open('/path/to/file', 'r') as f:
    print(f.read())
```

By the “with”, the python will call the close() automatically

CHAPTER 7

List and Tuple



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

- `Numbers = [5, 10, 15, 20]`
- `print(Numbers)`

Introduction to Lists

- **`list()` function can convert certain types of objects to lists**
  - *`Numbers = list(range(5))`*
  - *`Numbers = list(range(1,10,2))`*

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`*
    - *`Numbers=[1,2,3]*3`*
- **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.
  - An `IndexError` exception is raised if an invalid index is used

The `len` function

- **`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

```
NUM_DAYS = 5
def main():
    # Create a list to hold the sales for each day.
    sales = [0] * NUM_DAYS

    # Create a variable to hold an index.
    index = 0

    print('Enter the sales for each day.')

    # Get the sales for each day.
    while index < NUM_DAYS:
        print('Day #', index + 1, ': ', sep='', end='')
        sales[index] = float(input())
        index += 1

    # Display the values entered.
    print('Here are the values you entered:')
    for value in sales:
        print(value)

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

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**

```
List1=[1,2,3]
List2=[4,5,6]
List3 = List1 + List2     (or  List1 += List2)
```

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

List Slicing

```
Numbers = list(range(1,11))
print(Numbers[1:3]) →[2, 3]
print(Numbers[:3]) →[1, 2, 3]
print(Numbers[2:]) →[3, 4, 5, 6, 7, 8, 9, 10]
print(Numbers[1:8:2]) →[2, 4, 6, 8]
```

My solution of pre-test

```python
plate = input("")

try:
    temp = int(plate[:4])
    plate = plate[4:] + plate[:4]
except:
    plate = plate[2:] + plate[:2]

print(plate)
```

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**

```
# This program demonstrates the in operator
# used with a list.

def main():
    # Create a list of product numbers.
    prod_nums = ['V475', 'F987', 'Q143', 'R688']

    # Get a product number to search for.
    search = input('Enter a product number: ')

    # Determine whether the product number is in the list.
    if search in prod_nums:
        print(search, 'was found in the list.')
    else:
        print(search, 'was not found in the list.')

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

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

```
def main():
    name_list = []
    again = 'Y'

    while again.upper() == 'Y':
        # Get a name from the user.
        name = input('Enter a name: ')

        # Append the name to the list.
        name_list.append(name)

        # Add another one?
        print('Do you want to add another name?')
        again = input('y = yes, anything else = no: ')

    # Display the names that were entered.
    print('Here are the names you entered.')

    for name in name_list:
        print(name)

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

```
def main():
    # Create a list with some items.
    food = ['Pizza', 'Burgers', 'Chips']

# Get the item to change.
    item = input('Which item should I change? ')

    try:
        # Get the item's index in the list.
        item_index = food.index(item)

        # Get the value to replace it with.
        new_item = input('Enter the new value: ')

        # Replace the old item with the new item.
        food[item_index] = new_item

        # Display the list.
        print('Here is the revised list:')
        print(food)
    except ValueError:
        print('That item was not found in the list.')

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

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**
  - `mylist.sort()`
- **`remove(item)`: removes the first occurrence of *`item`* in the list**
- **`reverse()`: reverses the order of the elements in the list**
  - `mylist.reverse()`

```
# This program demonstrates the insert method.

def main():
    # Create a list with some names.
    names = ['James', 'Kathryn', 'Bill']

    # Display the list.
    print('The list before the insert:')
    print(names)

    # Insert a new name at element 0.
    names.insert(0, 'Joe')

    # Display the list again.
    print('The list after the insert:')
    print(names)

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

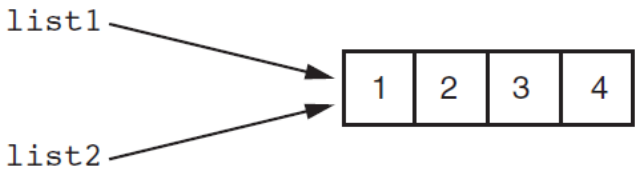
```
def main():
    # Create a list with some items.
    food = ['Pizza', 'Burgers', 'Chips']

    # Display the list.
    print('Here are the items in the food list:’.format(food))

    # Get the item to change.
    item = input('Which item should I remove? ')

    try:
        # Remove the item.
        food.remove(item)

        # Display the list.
        print('Here is the revised list:’.format(food))

    except ValueError:
        print('That item was not found in the list.')

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

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
  - `Mylist=[5, 4, 3, 2, 50, 40, 30]`
  - `max(Mylist)`
  - `min(Mylist)`

Copying Lists

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




```
>>> list1=[1,2,3,4]
>>> list2=list1
>>> print(list1)
[1, 2, 3, 4]
>>> print(list2)
[1, 2, 3, 4]
>>> list1[0]=99
>>> print(list1)
[99, 2, 3, 4]
>>> print(list2)
[99, 2, 3, 4]
```

List1 and list2 share the same memory

Like the pointer concept

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

```
>>> list1=[1,2,3,4]
>>> list2=[]
>>> for item in list1:
...     list2.append(item)
```

```
>>> list1=[1,2,3,4]
>>> list3=[]+list1
>>> print(list3)
[1, 2, 3, 4]
```

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**

```
# This program calculates the total of the values
# in a list.

def main():
    # Create a list.
    numbers = [2, 4, 6, 8, 10]

    # Create a variable to use as an accumulator.
    total = 0

    # Calculate the total of the list elements.
    for value in numbers:
        total += value

    # Display the total of the list elements.
    print('The total of the elements is', total)

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

```
# This program calculates the average of the values
# in a list.

def main():
    # Create a list.
    scores = [2.5, 7.3, 6.5, 4.0, 5.2]

    # Create a variable to use as an accumulator.
    total = 0.0

    # Calculate the total of the list elements.
    for value in scores:
        total += value

    # Calculate the average of the elements.
    average = total / len(scores)

    # Display the total of the list elements.
    print('The average of the elements is', average)

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

```
# This program uses a function to calculate the
# total of the values in a list.

def main():
    # Create a list.
    numbers = [2, 4, 6, 8, 10]

    # Display the total of the list elements.
    print('The total is', get_total(numbers))

def get_total(value_list):
    # Create a variable to use as an accumulator.
    total = 0

    # Calculate the total of the list elements.
    for num in value_list:
        total += num

    # Return the total.
    return total

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

Processing Lists (cont'd.)

- **A function can return a reference to a list**

```
>>> def array():
...     a = [1, 2, 3, 4]
...     return a
...
>>> b_array = array()
>>> b_array
 [1, 2, 3, 4]
>>> b_array[3] = 5
>>> b_array
 [1, 2, 3, 5]
```

```
# This program reads a file's contents into a list.

def main():
    # Open a file for reading.
    infile = open('cities.txt', 'r')

    # Read the contents of the file into a list.
    cities = infile.readlines()

    # Close the file.
    infile.close()

    # Strip the \n from each element.
    index = 0
    while index < len(cities):
        cities[index] = cities[index].rstrip('\n')
        index += 1

    # Print the contents of the list.
    print(cities)

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

```
# This program saves a list of numbers to a file.

def main():
    # Create a list of numbers.
    numbers = [1, 2, 3, 4, 5, 6, 7]

    # Open a file for writing.
    outfile = open('numberlist.txt', 'w')

    # Write the list to the file.
    for item in numbers:
        outfile.write(str(item) + '\n')

    # Close the file.
    outfile.close()

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

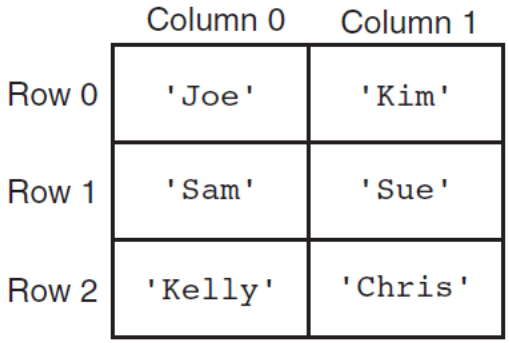
```
# This program reads numbers from a file into a list.

def main():
    # Open a file for reading.
    infile = open('numberlist.txt', 'r')

    # Read the contents of the file into a list.
    numbers = infile.readlines()

    # Close the file.
    infile.close()

    # Convert each element to an int.
    index = 0
    while index < len(numbers):
        numbers[index] = int(numbers[index])
        index += 1

    # Print the contents of the list.
    print(numbers)

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

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]

```
# This program assigns random numbers to
# a two-dimensional list.
import random

# Constants for rows and columns
ROWS = 3
COLS = 4

def main():
    # Create a two-dimensional list.
    values = [[0, 0, 0, 0],
              [0, 0, 0, 0],
              [0, 0, 0, 0]]

    # Fill the list with random numbers.
    for r in range(ROWS):
        for c in range(COLS):
            values[r][c] = random.randint(1, 100)

    # Display the random numbers.
    print(values)

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

Real list in python

- In actually, no dimension concept in python
- You can put everything and anything to the list

```
>>> b_array
[1, 2, 3, 5]
>>> a = ["test", 1]
>>> a.append(b_array)
>>> a
['test', 1, [1, 2, 3, 5]]
```

```
>>> a[2][1] = "change"
>>> a
['test', 1, [1, 'change', 3, 5]]
```

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**

Week 03 on-site assignment

- **Write a program to store student's name and score**
- **Support add, delete, modify functionalities**
- **Restore the execution results to a file**