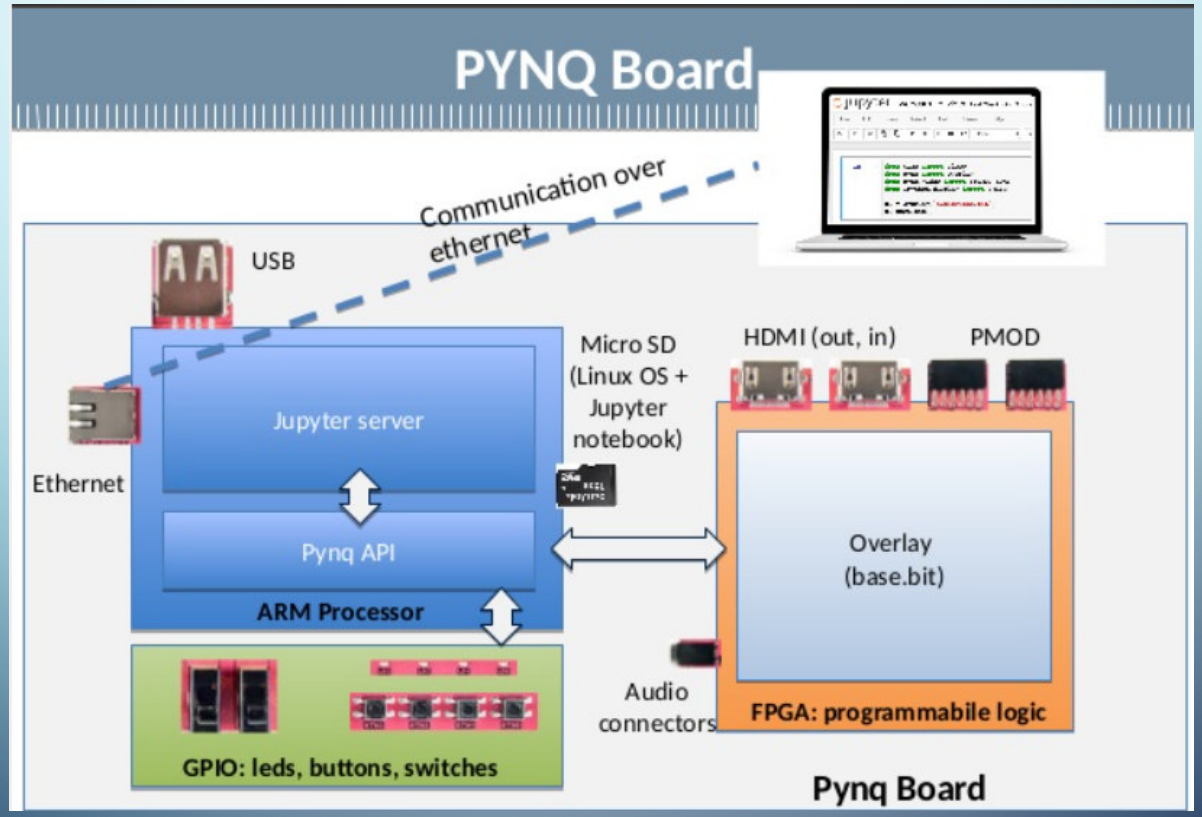
HIGH-LEVEL IMPLEMENTATION ON FPGA



PYTHON

FILES & FUNCTIONS

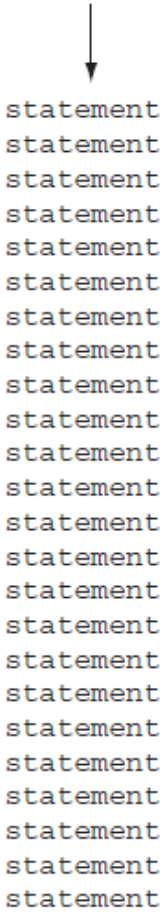
INTRODUCTION TO FUNCTIONS

- **Function: group of statements within a program that perform as specific task**
  - Usually one task of a large program
    - Functions can be executed in order to perform overall program task
  - Known as divide and conquer approach

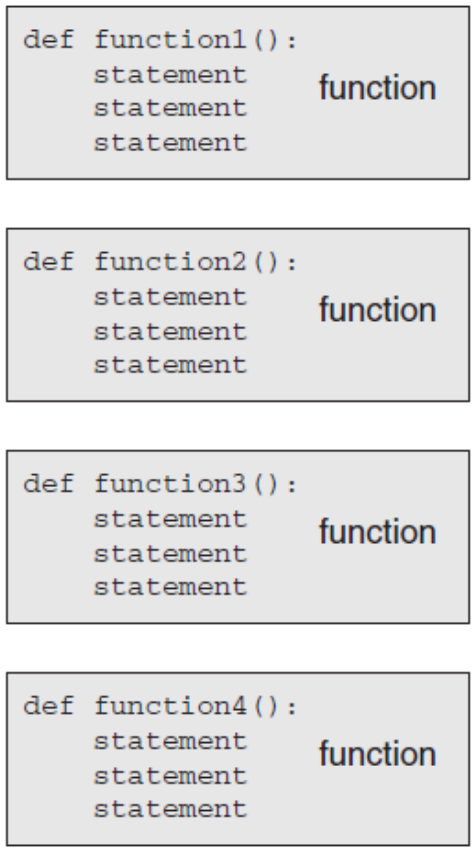
- **Modularized program: program where each task within the program is in its own function**

Figure 5-1 Using functions to divide and conquer a large task

This program is one long, complex sequence of statements.



In this program the task has been divided into smaller tasks, each of which is performed by a separate function.



BENEFITS OF MODULARIZING A PROGRAM WITH FUNCTIONS

- **The benefits of using functions include:**
  - Simpler code
  - Code reuse
    - Write the code once and call it multiple times
  - Better testing and debugging
    - Can test and debug each function individually
  - Faster development
  - Easier facilitation of teamwork
    - Different team members can write different functions

VOID FUNCTIONS AND VALUE– RETURNING FUNCTIONS

- **A void function:**
  - Simply executes the statements it contains and then terminates

- **A value-returning function:**
  - Executes the statements it contains, and then it returns a value back to the statement that called it.
    - The input, int, and float functions are examples of value-returning functions.

DEFINING AND CALLING A FUNCTION

- **Functions are given names**
  - Function naming rules:
    - Cannot use key words as a function name
    - Cannot contain spaces
    - First character must be a letter or underscore
    - All other characters must be a letter, number or underscore
    - Uppercase and lowercase characters are distinct

DEFINING AND CALLING A FUNCTION (CONT'D.)

- **Function definition: specifies what a function does**

```
def function_name() :
    statement
    statement
```

- **Function header: first line of function**
  - Includes keyword ***def*** and function name, followed by parentheses and colon
- **Block: set of statements that belong together as a group**
  - Example: the statements included in a function

DEFINING AND CALLING A FUNCTION (CONT'D.)

- **Call a function to execute it**
  - When a function is called:
    - Interpreter jumps to the function and executes statements in the block
    - Interpreter jumps back to part of program that called the function
      - Known as function return

- **main function: called when the program starts**
  - Calls other functions when they are needed
  - Defines the mainline logic of the program

INDENTATION IN PYTHON

- **Each block must be indented**
  - Lines in block must begin with the same number of spaces
    - Use tabs or spaces to indent lines in a block, but not both as this can confuse the Python interpreter
    - IDLE and SPYDER automatically indents the lines in a block
  - Blank lines that appear in a block are ignored

DESIGNING A PROGRAM TO USE FUNCTIONS

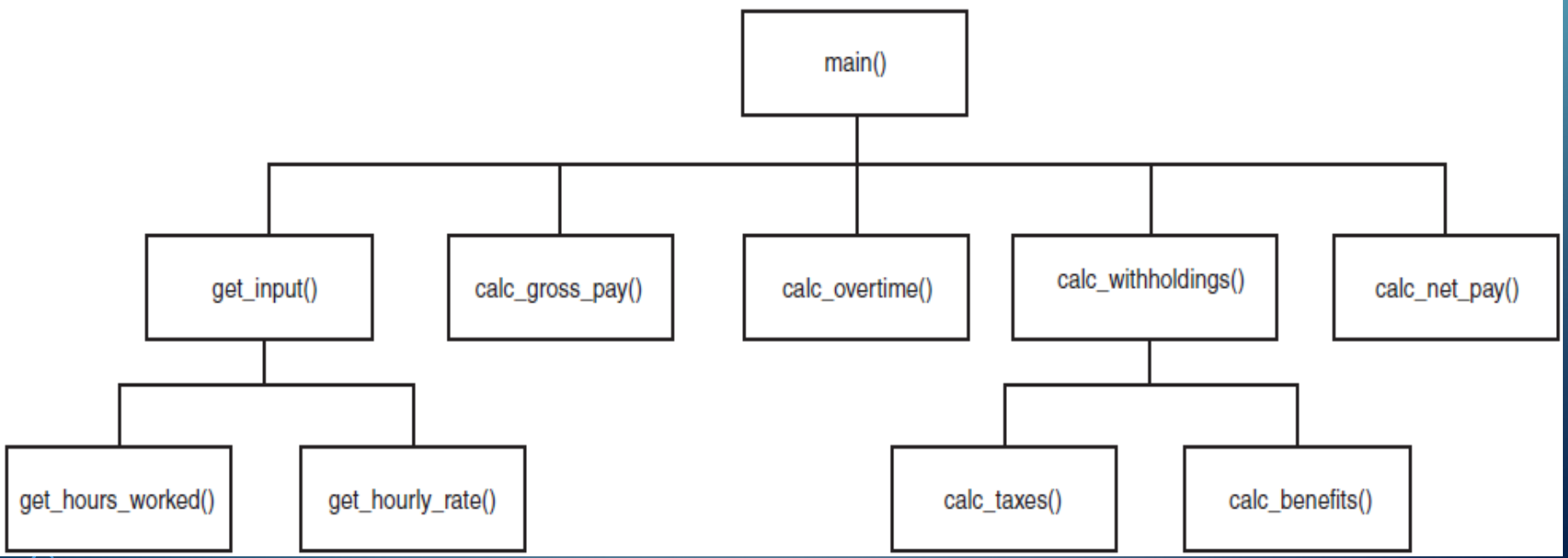
- **In a flowchart, function call shown as rectangle with vertical bars at each side**
  - ➢ Function name written in the symbol
  - ➢ Typically draw separate flow chart for each function in the program
    - ❖ End terminal symbol usually reads Return
- **Top-down design: technique for breaking algorithm into functions**

DESIGNING A PROGRAM TO USE FUNCTIONS (CONT'D.)

- **Hierarchy chart: depicts relationship between functions**
  - AKA structure chart
  - Box for each function in the program, Lines connecting boxes illustrate the functions called by each function
  - Does not show steps taken inside a function
- **Use input function to have program wait for user to press enter**

DESIGNING A PROGRAM TO USE FUNCTIONS (CONT'D.)

Figure 5-10 A hierarchy chart



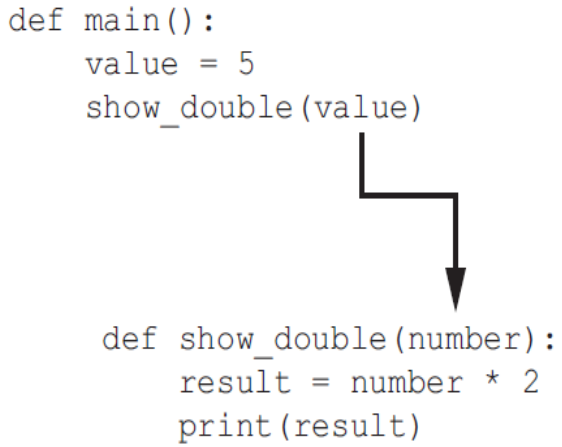
LOCAL VARIABLES

- **Local variables: variable that is assigned a value inside a function**
  - Belongs to the function in which it was created
    - Only statements inside that function can access it, error will occur if another function tried to access the variable
  - **Scope: the part of a program in which a variable may be accessed**
    - For local variable: function in which created
- **Local variable cannot be accessed by statements inside its function which precede its creation**
- **Different functions may have local variables with the same name**
  - Each function does not see the other function's local variables, so no confusion

PASSING ARGUMENTS TO FUNCTIONS

- **Argument: piece of data that is sent into a function**
  - Function can use argument in calculations
  - When calling the function, the argument is placed in parentheses following the function name

Figure 5-13 The `value` variable is passed as an argument



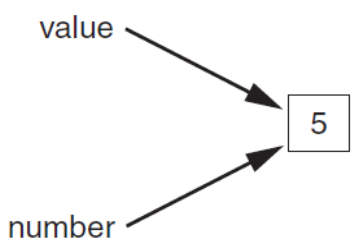
PASSING ARGUMENTS TO FUNCTIONS (CONT'D.)

- **Parameter variable: variable that is assigned the value of an argument when the function is called**
  - The parameter and the argument reference the same value
  - General format:
    - def function_name(parameter):
  - **Scope of a parameter**: the function in which the parameter is used

Figure 5-14 The `value` variable and the `number` parameter reference the same value

```
def main():
    value = 5
    show_double(value)


def show_double(number):
    result = number * 2
    print(result)
```



PASSING MULTIPLE ARGUMENTS

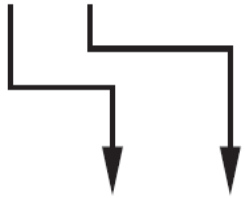
- **Python allows writing a function that accepts multiple arguments**
  - Parameters list replaces single parameter
    - Parameter list items separated by comma

- **Arguments are passed *by position* to corresponding parameters**
  - First parameter receives value of first argument, second parameter receives value of second argument, etc.

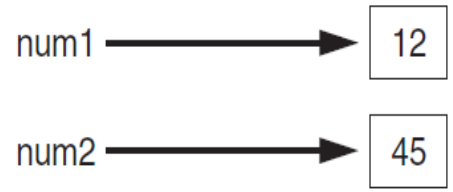
PASSING MULTIPLE ARGUMENTS (CONT'D.)

Figure 5-16 Two arguments passed to two parameters

```
def main():
    print('The sum of 12 and 45 is')
    show_sum(12, 45)
```



```
def show_sum(num1, num2):
    result = num1 + num2
    print(result)
```



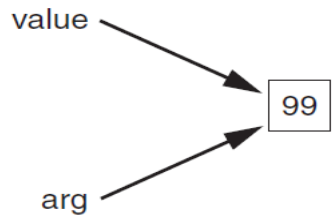
MAKING CHANGES TO PARAMETERS

- **Changes made to a parameter value within the function do not affect the argument**
  - ➢ Known as pass by value
  - ➢ Provides a way for unidirectional communication between one function and another function
    - ❖ Calling function can communicate with called function

Figure 5-17 The value variable is passed to the `change_me` function

```
def main():
    value = 99
    print('The value is', value)
    change_me(value)
    print('Back in main the value is', value)


def change_me(arg):
    print('I am changing the value.')
    arg = 0
    print('Now the value is', arg)
```



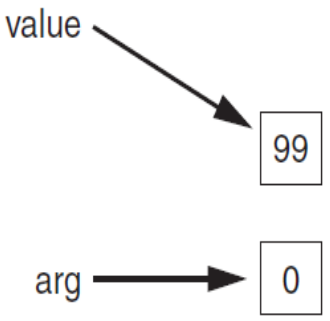
MAKING CHANGES TO PARAMETERS (CONT'D.)

- **Figure 5-18**
  - The value variable passed to the change_me function cannot be changed by it

Figure 5-18 The `value` variable is passed to the `change_me` function

```
def main():
    value = 99
    print('The value is', value)
    change_me(value)
    print('Back in main the value is', value)


def change_me(arg):
    print('I am changing the value.')
    arg = 0
    print('Now the value is', arg)
```



GLOBAL VARIABLES AND GLOBAL CONSTANTS

- **Global variable: created by assignment statement written outside all the functions**
  - Can be accessed by any statement in the program file, including from within a function
  - If a function needs to assign a value to the global variable, the global variable must be declared within the function
    - **General format:** global variable_name

GLOBAL VARIABLES AND GLOBAL CONSTANTS (CONT'D.)

- **Reasons to avoid using global variables:**
  - Global variables making debugging difficult
    - Many locations in the code could be causing a wrong variable value
  - Functions that use global variables are usually dependent on those variables
    - Makes function hard to transfer to another program
  - Global variables make a program hard to understand

GLOBAL CONSTANTS

- **Global constant: global name that references a value that cannot be changed**
  - Permissible to use global constants in a program
  - To simulate global constant in Python, create global variable and do not re-declare it within functions

STANDARD LIBRARY FUNCTIONS AND THE IMPORT STATEMENT

- **Standard library: library of pre-written functions that comes with Python**
  - Library functions performs tasks that programmers commonly need
    - Example: print, input, range
    - Viewed by programmers as a "black box"

- **Some library functions built into Python interpreter**
  - To use, just call the function

STANDARD LIBRARY FUNCTIONS AND THE IMPORT STATEMENT (CONT'D.)

- **Modules: files that stores functions of the standard library**
  - Help organize library functions not built into the interpreter
  - Copied to computer when you install Python

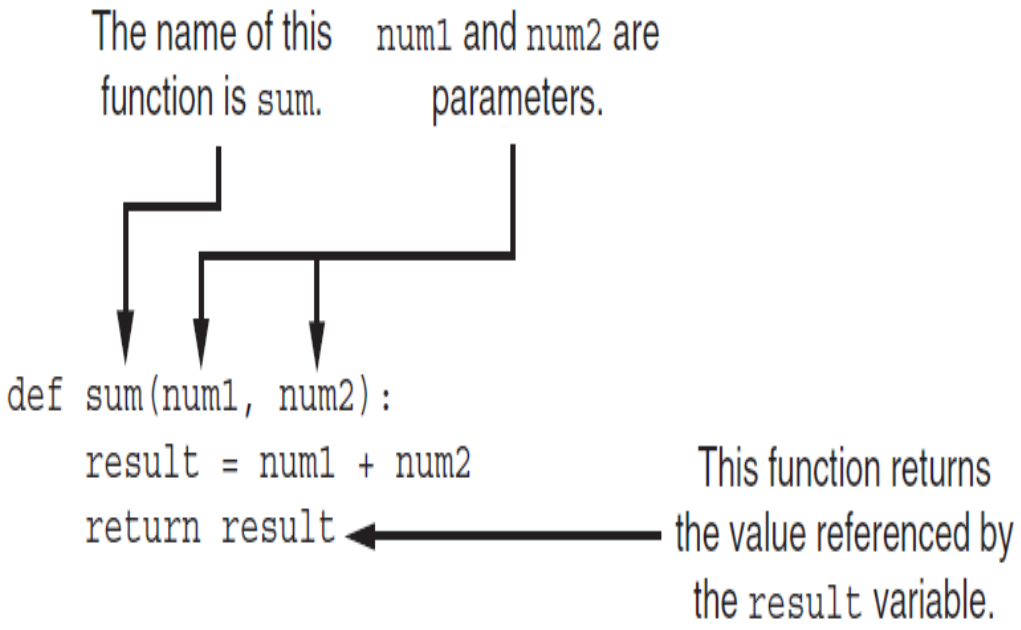
- **To call a function stored in a module, need to write an import statement**
  - Written at the top of the program
  - **Format:** import module_name

WRITING YOUR OWN VALUE-RETURNING FUNCTIONS

- **To write a value-returning function, you write a simple function and add one or more return statements**
  - **Format:** return *expression*
    - The value for *expression* will be returned to the part of the program that called the function
  - The expression in the return statement can be a complex expression, such as sum of two variables or the result of another value-returning function

WRITING YOUR OWN VALUE-RETURNING FUNCTIONS (CONT'D.)

Figure 5-23 Parts of the function



HOW TO USE VALUE-RETURNING FUNCTIONS

- **Value-returning function can be useful in specific situations**
  - Example: have function prompt user for input and return the user's input
  - Simplify mathematical expressions
  - Complex calculations that need to be repeated throughout the program

- **Use the returned value**
  - Assign it to a variable or use as an argument in another function

RETURNING STRINGS

- **You can write functions that return strings**
- **For example:**

```
def get_name():
        # Get the user's name
        name = input ("Enter your name: ")
        # Return the name
        return name
```

RETURNING BOOLEAN VALUES

- **Boolean function: returns either True or False**
  - Use to test a condition such as for decision and repetition structures
    - Common calculations, such as whether a number is even, can be easily repeated by calling a function
  - Use to simplify complex input validation code

RETURNING MULTIPLE VALUES

- **In Python, a function can return multiple values**
  - Specified after the return statement separated by commas
    - **Format:** return *expression1,*
      expression2, etc.
  - When you call such a function in an assignment statement, you need a separate variable on the left side of the = operator to receive each returned value

RETURNING MULTIPLE VALUES

Returning Multiple Values

first_name, last_name =get_name()

```
def  get_name():
    # Get the user's first and last name.
    first = raw_input('Enter your first name: ')
  last = raw_input('Enter your last name: ')

  # Return both names.
    return first, last
```

STORING FUNCTIONS IN MODULES

- **In large, complex programs, it is important to keep code organized**

- **Modularization: grouping related functions in modules**
  - Makes program easier to understand, test, and maintain
  - Make it easier to reuse code for multiple different programs
    - Import the module containing the required function to each program that needs it

STORING FUNCTIONS IN MODULES (CONT'D.)

- **Module is a file that contains Python code**
  - Contains function definition but does not contain calls to the functions
    - Importing programs will call the functions
- **Rules for module names:**
  - File name should end in .py
  - Cannot be the same as a Python keyword
- **Import module using import statement**

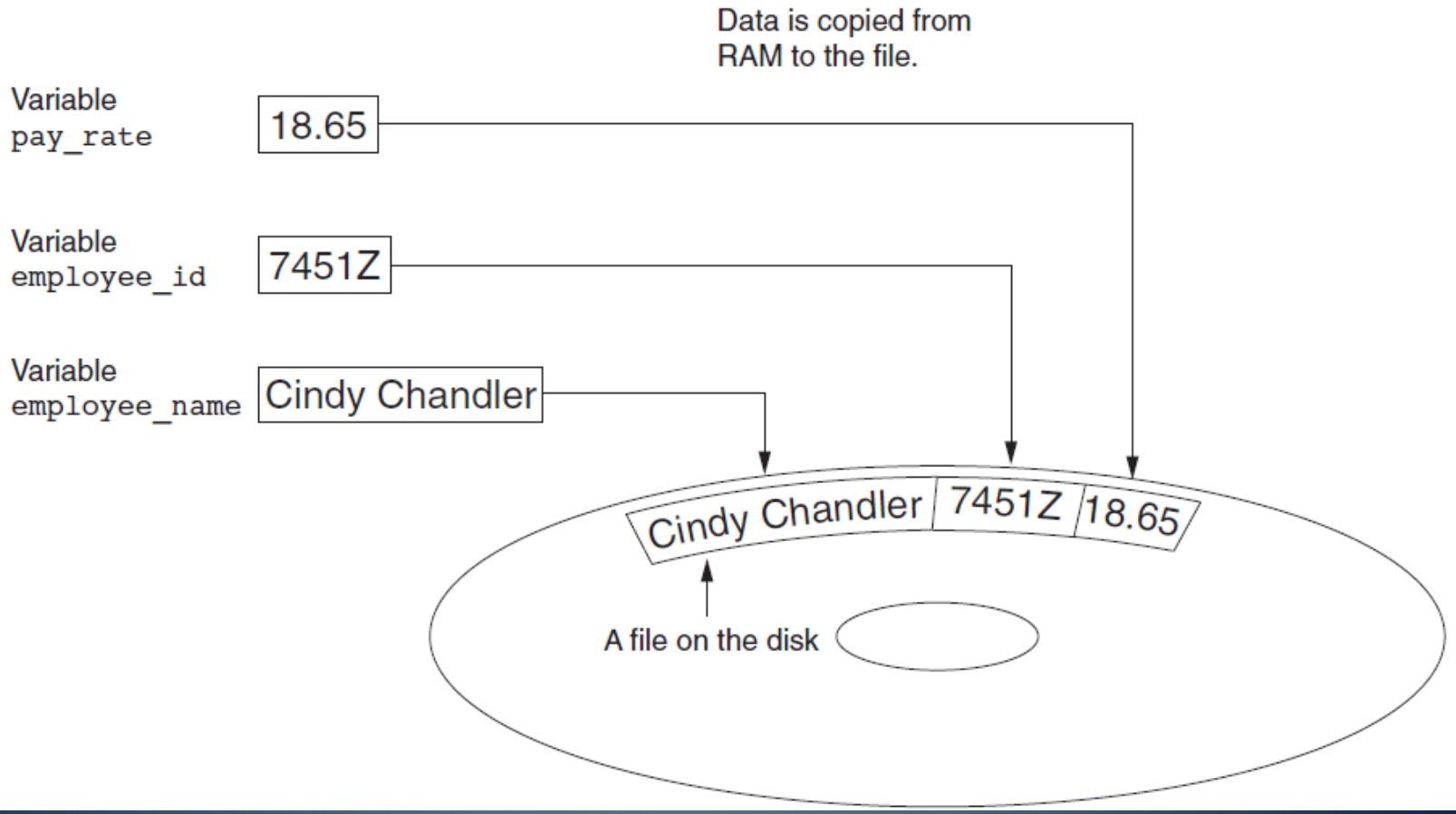
PYTHON

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**

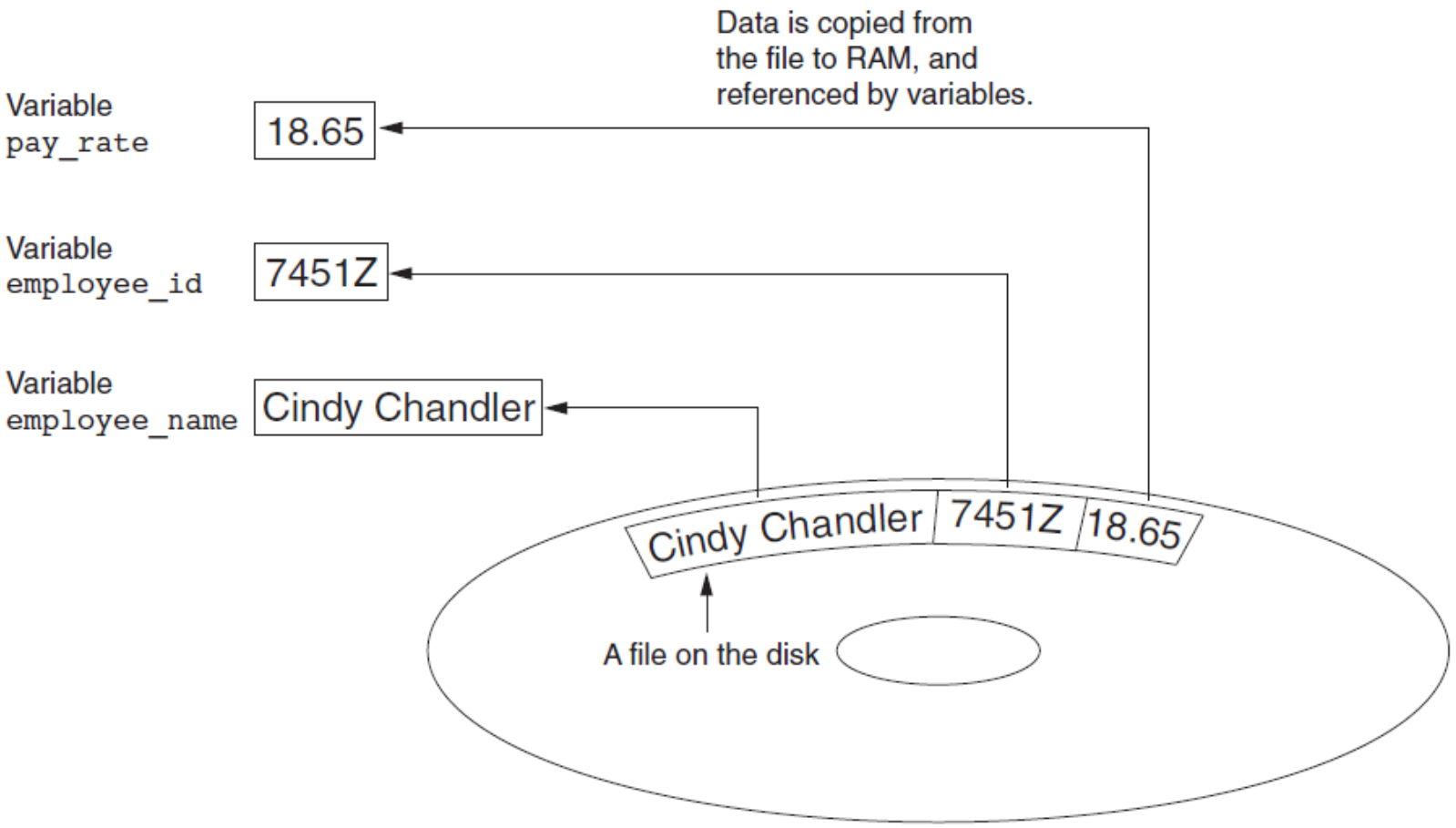
Figure 6-1 Writing data to a file



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

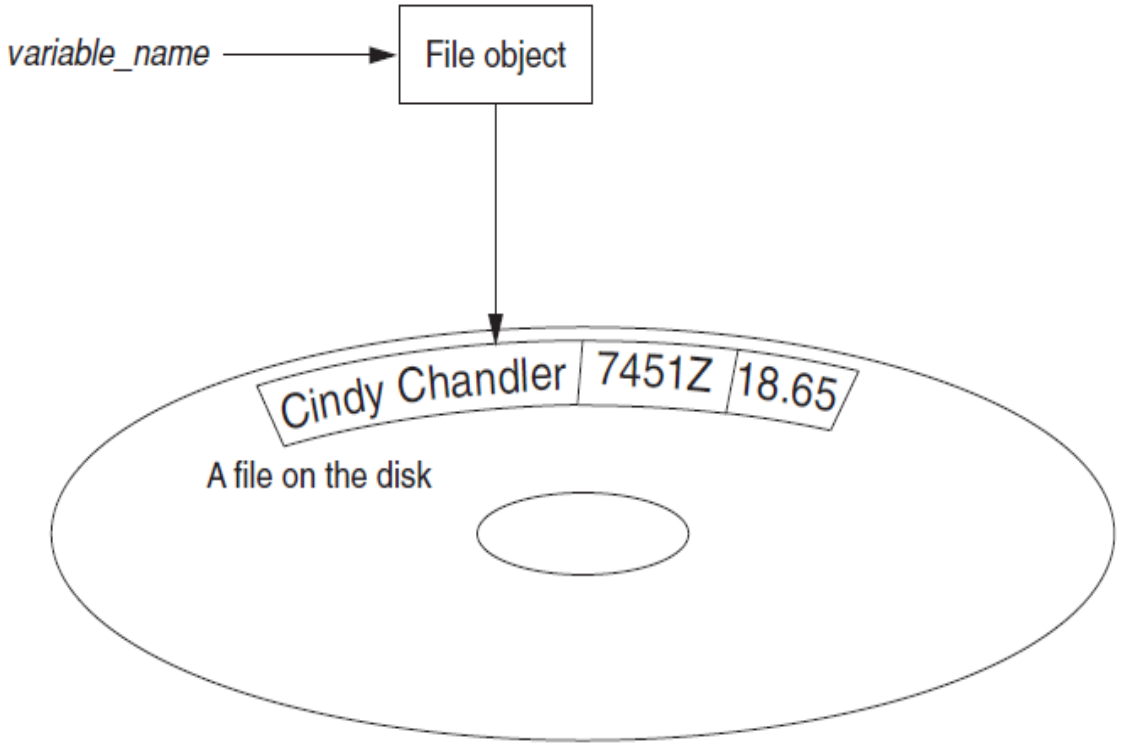
- **In general, two types of files**
  - Text file: contains data that has been encoded as text
  - Binary file: contains data that has not been converted to text
- **Two ways to access data stored in file**
  - Sequential access: file read sequentially from beginning to end, can't skip ahead
  - Direct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

WRITING DATA TO A FILE

Program 7-1 (file_write.py)

```
1  # This program writes three lines of data
2  # to a file.
3  def main():
4      # Open a file named philosophers.txt.
5      outfile = open('philosophers.txt', 'w')
6
7      # Write the names of three philosphers
8      # to the file.
9      outfile.write('John Locke\n')
10     outfile.write('David Hume\n')
11     outfile.write('Edmund Burke\n')
12
13     # Close the file.
14     outfile.close()
15
16 # Call the main function.
17 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**

READING DATA FROM A FILE

```
1  # This program reads and displays the contents
2  # of the philosophers.txt file.
3  def main():
4      # Open a file named philosophers.txt.
5      infile = open('philosophers.txt', 'r')
6
7      # Read the file's contents.
8      file_contents = infile.read()
9
10     # Close the file.
11     infile.close()
12
13     # Print the data that was read into
14     # memory.
15     print file_contents
16
17 # Call the main function.
18 main()
```

```
1  # This program reads the contents of the
2  # philosophers.txt file one line at a time.
3  def main():
4      # Open a file named philosophers.txt.
5      infile = open('philosophers.txt', 'r')
6
7      # Read three lines from the file.
8      line1 = infile.readline()
9      line2 = infile.readline()
10     line3 = infile.readline()
11
12     # Close the file.
13     infile.close()
14
15     # Print the data that was read into
16     # memory.
17     print line1
18     print line2
19     print line3
20
21 # Call the main function.
22 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
- **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

CONCATENATING A NEWLINE TO AND STRIPPING IT FROM A STRING

Concatenating a Newline to a String

```
myfile.write(name1 + '\n')
```

Reading a String and Stripping the Newline From It

```
line1 = infile.readline()

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

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**
- **Numbers 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

WRITING AND READING NUMERIC DATA

```
1  # This program demonstrates how numbers that are
2  # read from a file must be converted from strings
3  # before they are used in a math operation.
4
5  def main():
6      # Open a file for reading.
7      infile = open('numbers.txt', 'r')
8
9      # Read three numbers from the file.
10     num1 = int(infile.readline())
11     num2 = int(infile.readline())
12     num3 = int(infile.readline())
13
14     # Close the file.
15     infile.close()
16
17     # Add the three numbers.
18     total = num1 + num2 + num3
19
20     # Display the numbers and their total.
21     print 'The numbers are:', num1, num2, num3
22     print 'Their total is:', total
23
24 # Call the main function.
25 main()
```

USING LOOPS TO PROCESS FILES

- **Files typically used to hold large amounts of data**
  - Loop typically involved in reading from and writing to 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 != ‘ ’

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

USING PYTHON'S FOR LOOP TO READ LINES

```
1  # This program uses the for loop to read
2  # all of the values in the sales.txt file.
3
4  def main():
5      # Open the sales.txt file for reading.
6      sales_file = open('sales.txt', 'r')
7
8      # Read all the lines from the file.
9      for line in sales_file:
10         # Convert line to a float.
11         amount = float(line)
12         # Format and display the amount.
13         print '$%.2f' % amount
14
15     # Close the file.
16     sales_file.close()
17
18 # Call the main function.
19 main()
```

PYTHON

CLASSES AND OBJECT-ORIENTED PROGRAMMING

PROCEDURAL PROGRAMMING

- **Procedural programming: writing programs made of functions that perform specific tasks**
  - Procedures typically operate on data items that are separate from the procedures
  - Data items commonly passed from one procedure to another
  - Focus: to create procedures that operate on the program's data

OBJECT-ORIENTED PROGRAMMING

- **Object-oriented programming: focused on creating objects**
- **Object: entity that contains data and procedures**
  - Data is known as data attributes and procedures are known as methods
    - Methods perform operations on the data attributes
- **Encapsulation: combining data and code into a single object**

OBJECT-ORIENTED PROGRAMMING (CONT'D.)

- **Data hiding: object's data attributes are hidden from code outside the object**
  - Access restricted to the object's methods
    - Protects from accidental corruption
    - Outside code does not need to know internal structure of the object
- **Object reusability: the same object can be used in different programs**
  - **Example:** 3D image object can be used for architecture and game programming

AN EVERYDAY EXAMPLE OF AN OBJECT

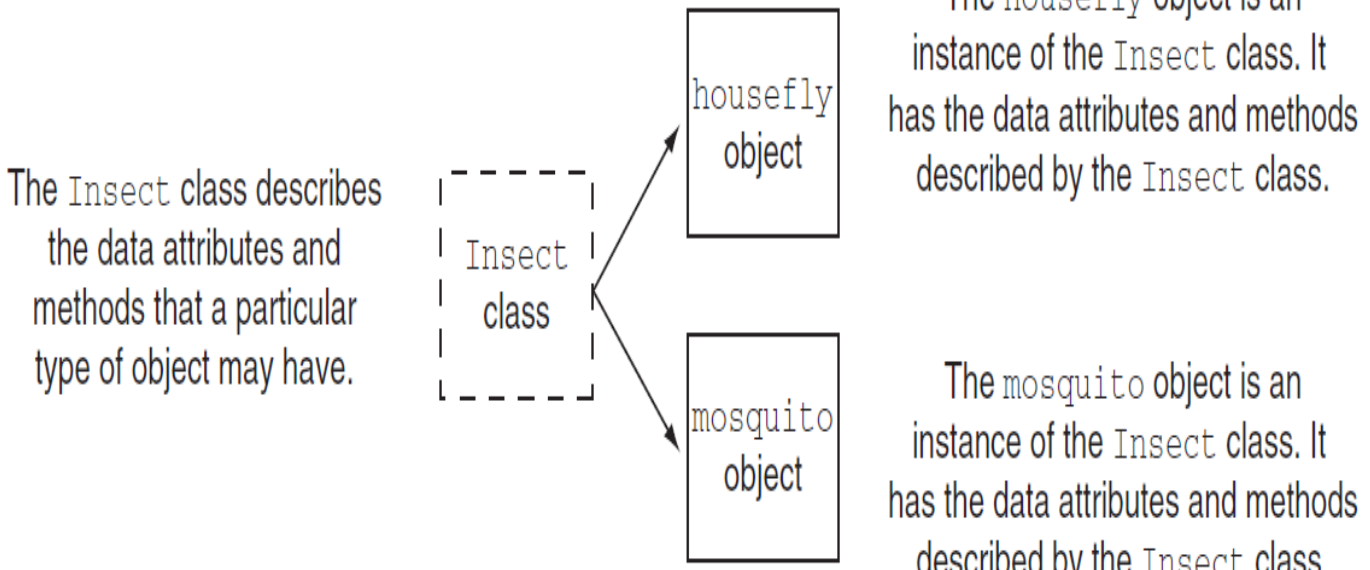
- **Data attributes: define the state of an object**
  - **Example:** clock object would have *second, minute,* and *hour* data attributes
- **Public methods: allow external code to manipulate the object**
  - **Example:** *set_time, set_alarm_time*
- **Private methods: used for objects inner working**

CLASSES

- **Class: code that specifies the data attributes and methods of a particular type of object**
  - Similar to a blueprint of a house or a cookie cutter
- **Instance: an object created from a class**
  - Similar to a specific house built according to the blueprint or a specific cookie
  - There can be many instances of one class

CLASSES (CONT'D.)

Figure 10-5 The `housefly` and `mosquito` objects are instances of the `Insect` class



CLASS DEFINITIONS

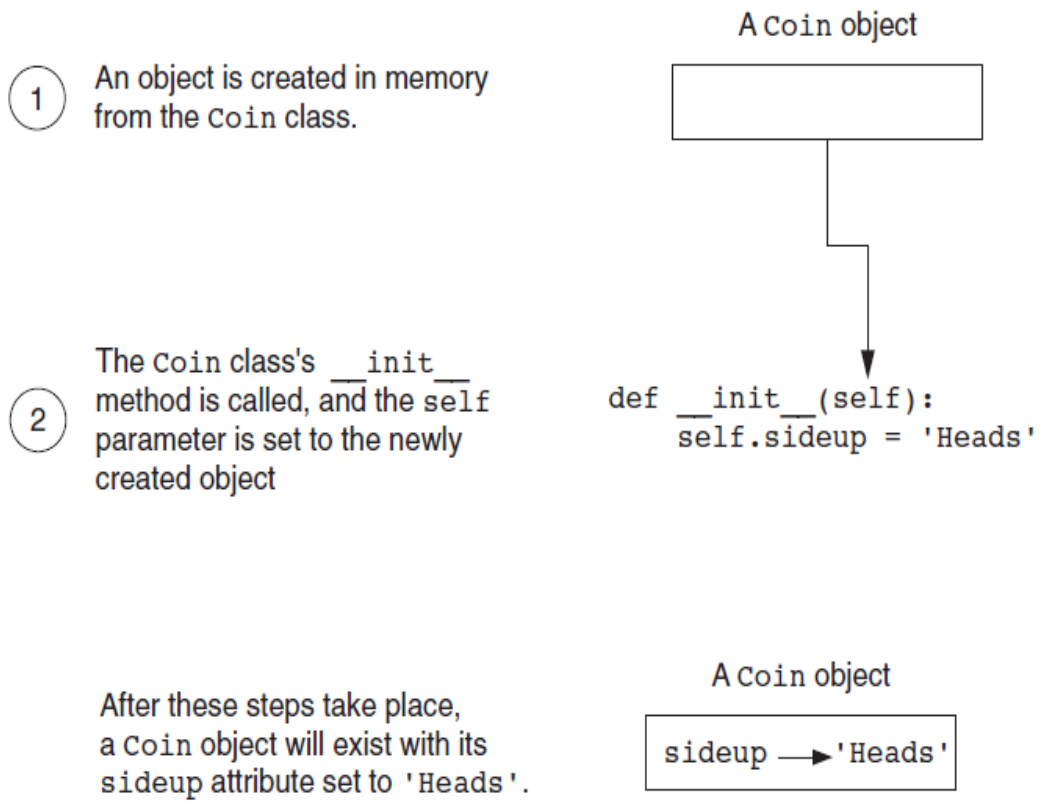
- **Class definition: set of statements that define a class's methods and data attributes**
  - **Format:** begin with class *Class_name:*
    - Class names often start with uppercase letter
  - Method definition like any other python function definition
    - *self* parameter: required in every method in the class – references the specific object that the method is working on

CLASS DEFINITIONS (CONT'D.)

- **Initializer method: automatically executed when an instance of the class is created**
  - Initializes object's data attributes and assigns *self* parameter to the object that was just created
  - **Format:** *def __init__ (self)*:
  - Usually the first method in a class definition

CLASS DEFINITIONS (CONT'D.)

Figure 10-6 Actions caused by the `Coin()` expression



CLASS DEFINITIONS (CONT'D.)

- **To create a new instance of a class call the initializer method**
  - **Format:** *My_instance = Class_Name()*
- **To call any of the class methods using the created instance, use dot notation**
  - **Format:** *My_instance.method()*
  - Because the *self* parameter references the specific instance of the object, the method will affect this instance
    - Reference to *self* is passed automatically

CLASS DEFINITIONS (CONT'D.)

```
1  import random
2
3  # The Coin class simulates a coin that can
4  # be flipped.
5
6  class Coin:
7
8      # The __init__ method initializes the
9      # sideup data attribute with 'Heads'.
10
11     def __init__(self):
12         self.sideup = 'Heads'
13
14     # The toss method generates a random number
15     # in the range of 0 through 1. If the number
16     # is 0, then sideup is set to 'Heads'.
17     # Otherwise, sideup is set to 'Tails'.
18
19     def toss(self):
20         if random.randint(0, 1) == 0:
21             self.sideup = 'Heads'
22         else:
23             self.sideup = 'Tails'
24
25     # The get_sideup method returns the value
26     # referenced by sideup.
27
28     def get_sideup(self):
29         return self.sideup
30
31 # The main function.
32 def main():
33     # Create an object from the Coin class.
34     my_coin = Coin()
35
36     # Display the side of the coin that is facing up.
37     print 'This side is up:', my_coin.get_sideup()
38
39     # Toss the coin.
40     print 'I am tossing the coin...'
41     my_coin.toss()
42
43     # Display the side of the coin that is facing up.
44     print 'This side is up:', my_coin.get_sideup()
45
46 # Call the main function.
47 main()
```

HIDING ATTRIBUTES AND STORING CLASSES IN MODULES

- **An object's data attributes should be private**
  - To make sure of this, place two underscores (__) in front of attribute name
    - **Example:** __current_minute
- **Classes can be stored in modules**
  - Filename for module must end in .py
  - Module can be imported to programs that use the class

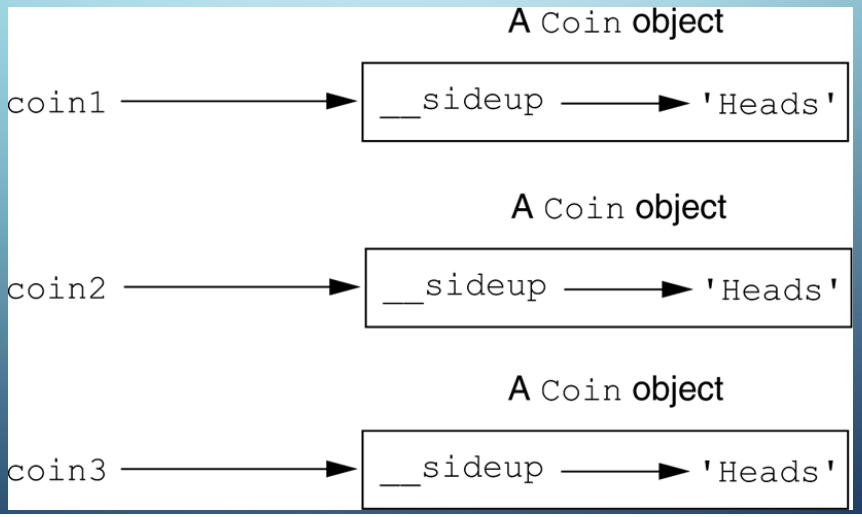
THE BANKACCOUNT CLASS - MORE ABOUT CLASSES

- **Class methods can have multiple parameters in addition to self**
  - For __init__, parameters needed to create an instance of the class
    - **Example:** a *BankAccount* object is created with a balance
      - ✓ When called, the initializer method receives a value to be assigned to a *__balance* attribute
    - For other methods, parameters needed to perform required task
      - ✓ **Example:** *deposit* method account to be deposited

WORKING WITH INSTANCES

- **Instance attribute: belongs to a specific instance of a class**
  - Created when a method uses the *self* parameter to create an attribute
- **If many instances of a class are created, each would have its own set of attributes**

Figure The `coin1`, `coin2`, and `coin3` variables reference three Coin objects



ACCESSOR AND MUTATOR METHODS

- **Typically, all of a class's data attributes are private and provide methods to access and change them**
- **Accessor methods: return a value from a class's attribute without changing it**
  - Safe way for code outside the class to retrieve the value of attributes
- **Mutator methods: store or change the value of a data attribute**

PASSING OBJECTS AS ARGUMENTS

- **Methods and functions often need to accept objects as arguments**
- **When you pass an object as an argument, you are actually passing a reference to the object**
  - The receiving method or function has access to the actual object
    - Methods of the object can be called within the receiving function or method, and data attributes may be changed using mutator methods

TECHNIQUES FOR DESIGNING CLASSES

- **UML diagram: standard diagrams for graphically depicting object-oriented systems**
  - Stands for Unified Modeling Language
- **General layout: box divided into three sections:**
  - Top section: name of the class
  - Middle section: list of data attributes
  - Bottom section: list of class methods

Figure 10-10 General layout of a UML diagram for a class

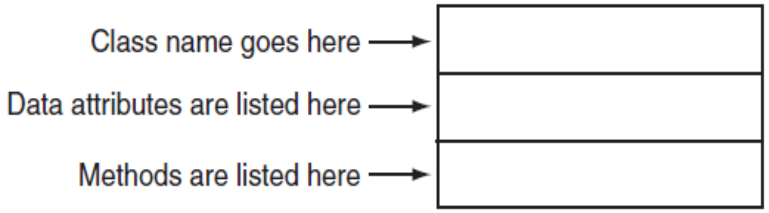
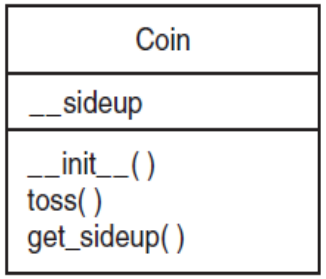


Figure 10-11 UML diagram for the `Coin` class



FINDING THE CLASSES IN A PROBLEM

- **When developing object oriented program, first goal is to identify classes**
  - Typically involves identifying the real – world objects that are in the problem
  - Technique for identifying classes:
    1. Get written description of the problem domain
    2. Identify all nouns in the description, each of which is a potential class
    3. Refine the list to include only classes that are relevant to the problem

FINDING THE CLASSES IN A PROBLEM (CONT'D.)

1. **Get written description of the problem domain**
   - May be written by you or by an expert
   - Should include any or all of the following:
     - Physical objects simulated by the program
     - The role played by a person
     - The result of a business event
     - Recordkeeping items
2. **Identify all nouns in the description, each of which is a potential class**
   - Should include noun phrases and pronouns
   - Some nouns may appear twice

FINDING THE CLASSES IN A PROBLEM (CONT'D.)

3. Refine the list to include only classes that are relevant to the problem

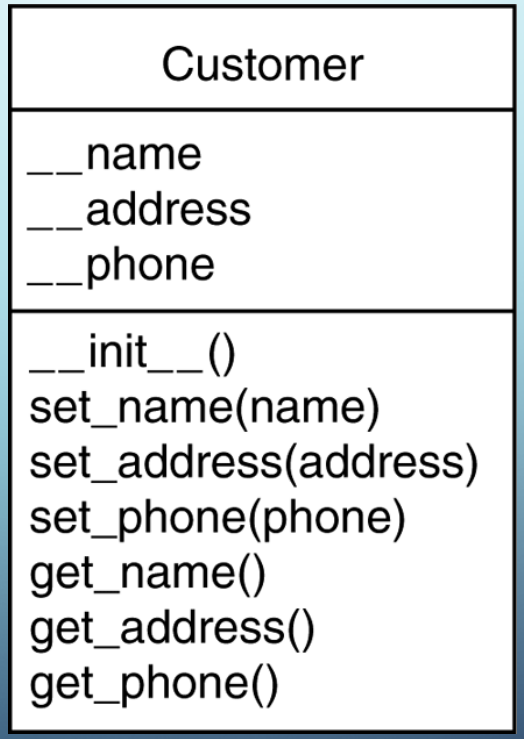
- Remove nouns that mean the same thing
- Remove nouns that represent items that the program does not need to be concerned with
- Remove nouns that represent objects, not classes
- Remove nouns that represent simple values that can be assigned to a variable

IDENTIFYING A CLASS'S RESPONSIBILITIES

- **A classes responsibilities are:**
  - The things the class is responsible for knowing
    - Identifying these helps identify the class's data attributes
  - The actions the class is responsible for doing
    - Identifying these helps identify the class's methods
- **To find out a class's responsibilities look at the problem domain**
  - Deduce required information and actions

IDENTIFYING A CLASS'S RESPONSIBILITIES

Figure UML diagram for the `Customer` class



PYTHON

INHERITANCE

INTRODUCTION TO INHERITANCE

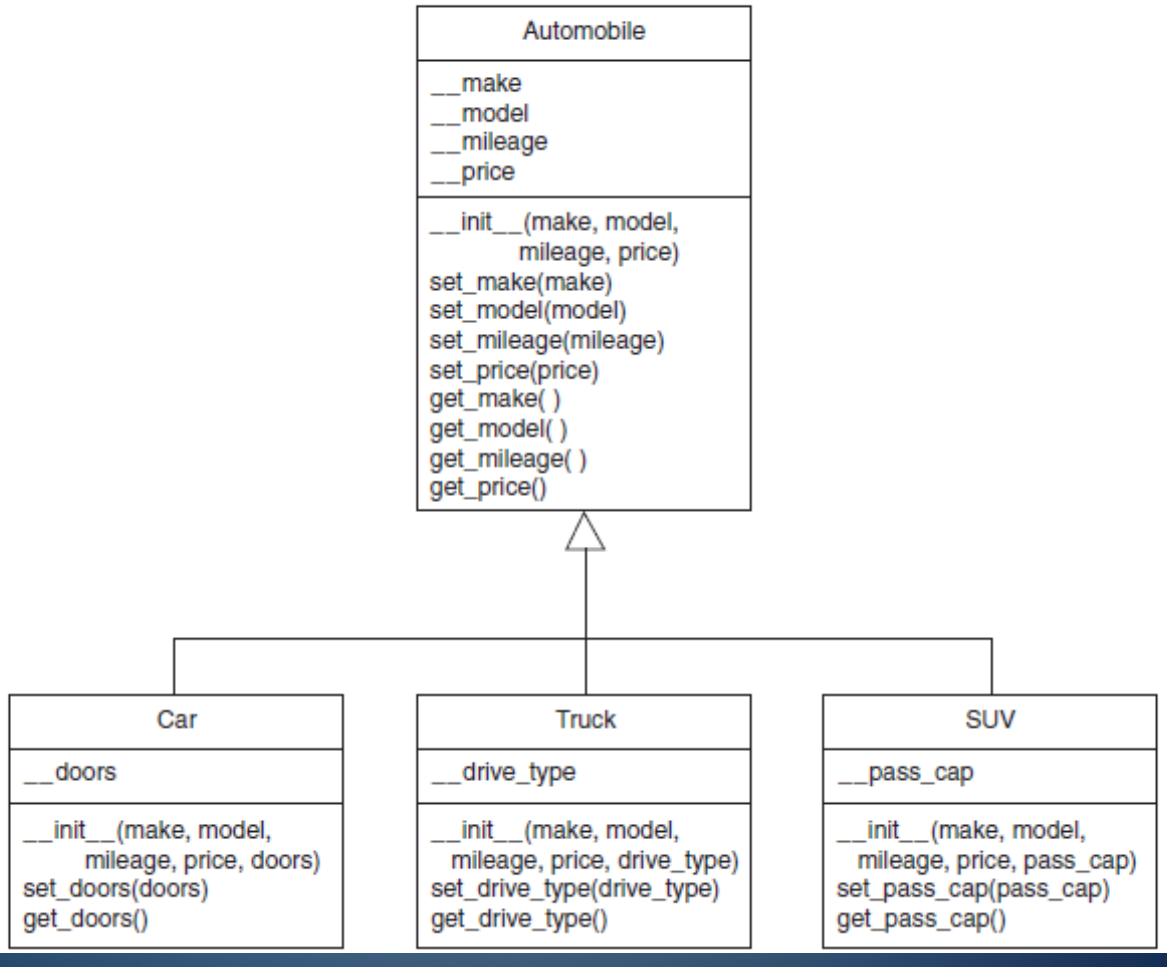
- **In the real world, many objects are a specialized version of more general objects**
  - Example: grasshoppers and bees are specialized types of insect
    - In addition to the general insect characteristics, they have unique characteristics:
      - ✓Grasshoppers can jump
      - ✓Bees can sting, make honey, and build hives
- **"Is a" relationship: exists when one object is a specialized version of another object**
  - Specialized object has all the characteristics of the general object plus unique characteristics
  - **Example:** Rectangle is a shape
    Daisy is a flower

INHERITANCE AND THE "IS A" RELATIONSHIP (CONT'D.)

- **Inheritance: used to create an "is a" relationship between classes**
- **Superclass (base class): a general class**
- **Subclass (derived class): a specialized class**
  - A extended version of the superclass
    - Inherits attributes and methods of the superclass
    - New attributes and methods can be added

INHERITANCE AND UML

Figure 11-2 UML diagram showing inheritance



POLYMORPHISM

- **Polymorphism: an object's ability to take different forms**
- **Essential ingredients of polymorphic behavior:**
  - Ability to define a method in a superclass and override it in a subclass
    - Subclass defines method with the same name
  - Ability to call the correct version of overridden method depending on the type if object that called for it
- **In previous inheritance examples showed how to override the __init__ method**
  - Called superclass __init__ method and then added onto that
- **The same can be done for any other method**
  - The method can call the superclass equivalent and add to it, or do something completely different