```
 4:00pm  2 workers came
         worker 1: whole time
         worker 2: whole time

 6:00pm  3 workers came (1 went home)
         worker 1: whole time
         worker 2: whole time

Saturday
-------
 8:00am  6 workers came (4 went home)
         worker 1: whole time
         worker 2: whole time

10:00am  3 workers came (1 went home)
         worker 1: 15 mins late
         worker 2: whole time

12:00pm  9 workers came (7 went home)
         worker 1: whole time
         worker 2: whole time

 2:00pm  1 worker came
         worker 1: whole time

 4:00pm  2 workers came
         worker 1: whole time
         worker 2: 30 mins late

 6:00pm  4 workers came (2 went home)
         worker 1: whole time
         worker 2: whole time
```

FIGURE 9-27 Unscheduled Worker Simulation

two workers show up. Thus, there are expected to be times when members who show up to work are not needed, and therefore need to return another time. In our simulation, this happened 77 times, which depending on other factors (for example, how long it takes to get to the co-op, how far out of the way is it from other locations that members would typically go to, etc.) is an additional burden on individual members.

Thus, it is understandable from the co-op's point of view that an unscheduled approach to worker "scheduling" is a better approach. However, *for individual members it adds an extra burden of time, and thus it may not be considered an improvement from their perspective.*

Number of Time Slots that:	Scheduled Approach	Unscheduled Approach
No One Showed Up	3	0
Only One Person Showed Up	14	3
Number of Times that Someone was 15 Minutes Late	6	6
Number of Times that Someone was 30 minutes Late	3	3
Number of Times that Someone was 45 Minutes Late	1	0
Number of Times that Someone Left 15 Minutes Early	3	4
Number of Times that Someone Left 30 Minutes Early	4	1
Number of People Turned Away Because had Enough Workers (unscheduled only)	-	77

FIGURE 9-28 Results of Simulation Runs of Scheduled vs. Unscheduled Approach

Chapter Exercises 379

CHAPTER SUMMARY General Topics Python-Specific Programming Topics

Associative Data Structures Set Operations
Hexadecimal Numbers Computer Simulation
Test Drivers/Test Stubs Dictionaries in Python
Set and Frozenset Types in Python

CHAPTER EXERCISES Section 9.1

1. Indicate whether an indexed data structure, an associative data structure, or a set type would be most appropriate for each of the following.
(a) The number of inches of rain for each day of the year in a given locality, used for computing the average yearly rainfall.

(b) The number of inches of rain, only for the days when there was rainfall in a given locality, used to retrieve the amount of rain for any given day as quickly as possible.
(c) Faculty members that belong to various committees, in which each faculty

member on the university senate must also be a member of at least one college committee, but may not also be a member of another university-level committee.
2. Create a dictionary named password_lookup that contains usernames as keys (as string types), and
passwords as associated string values. Make up data for five dictionary entries.
3. Give a program segment that creates an initially empty dictionary named password_lookup, prompting one-by-one for usernames and passwords (until a username of 'z' is read) entering each into the
dictionary.
4. Create a dictionary named password_hint that contains email addresses as keys, and associated values that contain both a user's "password security question," and the answer to the question. Make up data
for five dictionary entries.
5. Create a dictionary named member_table that contains users' email addresses as keys, and their current password as the values. Write a function that generates a temporary new password for a given user
and updates it in the table.

Section 9.2
6. Declare a set named vowels containing the strings 'a','e','i','o', and 'u'. Give a program segment that prompts the user for any English word, and displays how many vowels it contains.

7. Give a program segment that prompts the user for two English words, and displays which letters the two words have in common.
8. Give a program segment that prompts the user for two English words, and displays which letters are in the first word but not in the second.
9. Give a program segment that prompts the user for two English words, and displays which letters of the alphabet are in neither of the two words.
10. Give a program segment that prompts the user for two English words, and displays which letters are in either the first word or the second word, but not in both words.
11. Give a program segment that prompts the user for two English words, entered in no particular order, and determines if all the letters of first word are contained within the second.

PYTHON PROGRAMMING EXERCISES

P1. Write a Python function called addDailyTemp that is given a (possibly

empty) dictionary meant to hold the average daily temperature for each day of the week, the day, and the day's average temperature. The function should add the temperature to the dictionary only if does not already contain a temperature for that day. The function should return the resulting dictionary, whether or not it is updated.

P2. Write a Python function named moderateDays that is given a dictionary containing the average daily temperature for each day of a week, and returns a list of the days in which the average temperature was between 70 and 79 degrees.

P3. Write a Python function named getDailyTemps that prompts the user for the average temperature for each day of the week, and returns a dictionary containing the entered information.
P4. Write a Python function named getWeekendAvgTemp that is passed a dictionary of daily temperatures, and returns the average temperature over the weekend for the weekly temperatures given.
P5. Write a Python function named addVegetable that is passed a (possible empty) set of vegetable names, and raises a ValueError exception if the given vegetable is already in the set, otherwise, the new vegetable should be added and the new set returned.
P6. Write a Python function named numVowels that is passed a string containing letters, each of which may be in either uppercase or lowercase, and returns a tuple containing the number of vowels and the number of consonants the string contains.

PROGRAM MODIFICATION PROBLEMS

M1. Word Frequency Count Program: Making Use of Sets
Modify the Word Frequency Count Program in Chapter 8 making use of the set type where possible.
M2. Phone Number Spelling Program: Modifying for Australia
In Australia, phone numbers are of the form 0x-xxxx-xxxx. Modify the Phone Number Spelling program so that it prints out Australian phone numbers with spellings for the last four digits.
M3. Kitchen Tile Visualization Program: Color Name Entry
Modify the Kitchen Tile Visualization Program so that, instead of requiring the user to enter hexadecimal RGB color codes, they are given a subset of twelve color names from which to select (from the colors listed in Figure 9-11).

M4. Kitchen Tile Visualization Program: Randomly Generated Patterns
Modify the Kitchen Tile Visualization Program so that the user can enter up to three different tile colors, as well as the grout color. The program should then display a pattern of tiles that is randomly generated using the three colors specified.
M5. Kitchen Tile Visualization Program: Automatic Complementary Colors
The complementary color of a given color is the "opposite" color (directly opposite on the color wheel). Designers often use complementary colors because of their vibrant contrast. Modify the Kitchen Tile visualization program so that the user enters one tile color, with the complementary color automatically

generated as the secondary tile color. To calculate the complementary color of a color, each of the three color values is replaced by its arithmetic complement to 255,

FFFFFF (white) → 255 255 255 → 255-255 255-255 255-255 → 000000 (black)
00FF00 (green) → 0 255 0 → 255-0 255-255 255-0 → 255 0 255 → FF00FF (magenta)
87CEEB (sky blue) → 135 206 235 → 255-135 255-206 255-235 → 120 49 20 → 783114 (brownish red)

M6. Kitchen Tile Visualization Program: Muted and Tinted Colors
Sometimes two colors are too vibrant together. Colors can be muted, or made less vibrant and "toned down," by adding some of their complementary color. Modify the Kitchen Tile Visualization Program so that the user can mute either or both of the two displayed colors to varying degrees. (See problem M5 about complementary colors.)

M7. Food Co-op Worker Schedule Simulation Program: Adjusting the Model
The program developed for the food co-op simulation allows for easy adjustment of some of the parameters in the model. This includes the assumed number of co-op members, as well as the various assumed probabilities for members' behaviors. Since the number of members in the co-op only affects the unscheduled simulation results, run the unscheduled simulation for various number of co-op members and
(a) Plot a graph using the x-axis for the number of co-op members, and the y-axis for the number of

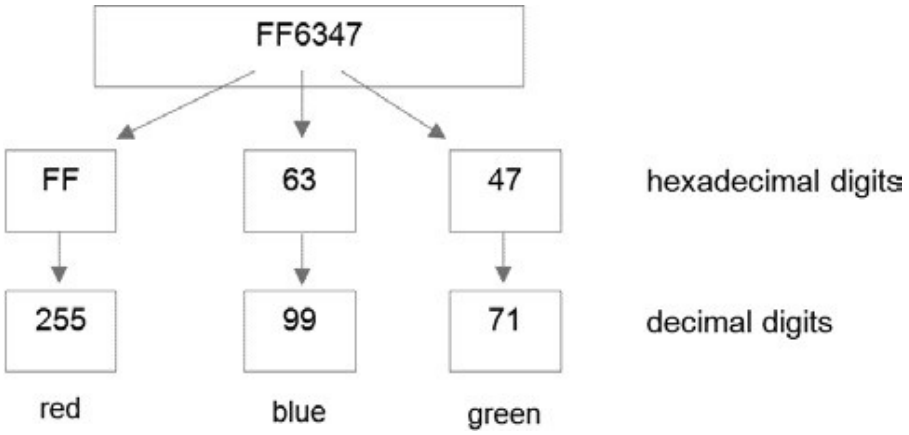
members turned away when showing up to work over the period of one week.
(b) Determine the optimal number of co-op members so that each time slot is covered, but with the least
number of members showing up to work turned away.
(c) Adjust the probabilities for the assumed behavior of co-op members to what you consider to be a
more accurate reflection of peoples' behavior and describe the results.
M8. Food Co-op Worker Schedule Simulation Program: Multiple Simulations
Modify the Food Co-op Worker Schedule Simulation program so that the user can select the number of multiple, sequential simulations. The program should output the average results over all the simulation runs as shown below:

♦ average number of workers that showed up to work for any given time slot ♦ average number of workers that worked the complete time slot
♦ average number of workers that were 15, 30, and 45 minutes late
♦ average number of workers that left 15 or 30 minutes early
♦ average number of workers turned away over the week because the co-op had enough

workers (for unscheduled approach)
PROGRAM DEVELOPMENT PROBLEMS

D1. Reverse Phone Spelling Program
Develop and test a program that allows the user to enter a spelled phone number for the last four digits (for example, 410-555-book) and generates the phone number that produces that spelling.

D2. Color Encoding Conversion Program
Develop and test a program that allows the user to enter six-digit hexadecimal RGB color codes and converts them to base 10. (You are *not* to use the '#' symbol for denoting hexadecimal values in Python in this program.) In this format, the first two hexadecimal digits represents the amount of red, the second two the amount of green, and the last two the amount of blue. Following is the hexadecimal value for the color "tomato,"



Hexadecimal numbers have place values that are powers of 16, in which the letters A–F have the values 10–15, respectively. Thus, FF in hexadecimal is equal to 15 * 16 1 15 * 1 5 240 1 15 5 255. Likewise, 63 in hexadecimal is equal to 6 * 16 1 3 * 1 5 96 1 3 5 99.

D3. Color Code Recording Program
Develop and test a program that allows the user to store color codes that they would like to save for future reference. The program should be designed so that they can enter a color code in either hexadecimal or decimal format (see problem D2). Whichever color code format is entered, it should be stored in both hexadecimal and decimal form. For example, if the user enters F0A514, it should be stored as both F0A514 and (240, 165, 20), and vice versa. The program should allow an annotation to be added for each color (for example, "Used this shade of green on my website"). The user should also be able to delete unwanted color entries.

Object-Oriented Programming CHAPTER 10

In Chapter 6, we learned what objects are and how they are used. We found that all values in Python are objects. Classes in object-oriented programming define objects. Thus, a class is a "cookie cutter" for creating any number of objects of that type. As with functions, there are predefined classes in Python, as well as the ability of programmers to define their own. In this chapter, we see how to define and use classes in Python.

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to ♦ Explain the fundamental concepts of object-oriented programming ♦ Explain the

concept of a class
♦ Define encapsulation, inheritance, and polymorphism ♦ Explain the use of subclasses as a subtype
♦ Explain the purpose of UML
♦ Explain the relationships of a UML class diagram
♦ Write simple UML class diagrams
♦ Define and use classes in Python
♦ Explain and use special methods in Python
♦ Effectively use inheritance and polymorphism in Python

CHAPTER CONTENTS

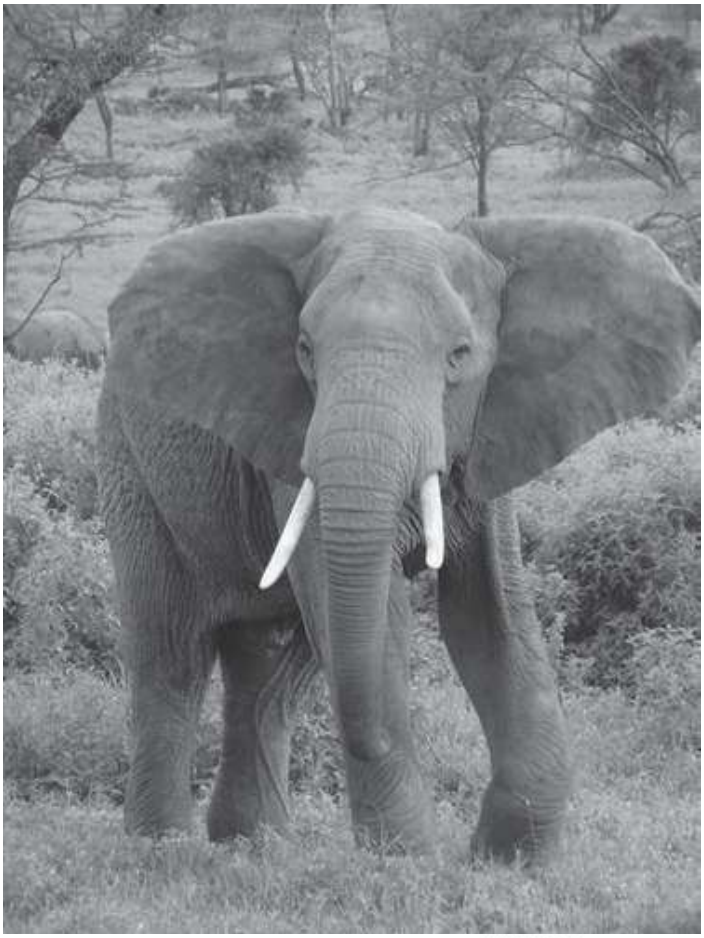
MOTIVATION

Classification can be described as the act of grouping entities into various categories that have something in common. In Biology, organisms are placed in a taxonomy based on their individual traits, for example. Libraries group books using various classification systems based on subject matter.

Most classification systems contain subcategories (and subcategories of subcategories, etc.), resulting in a hierarchy of types as shown in Figure 10.1. For example, chimpanzees are species in the Hominidae family, which is in the order Primate, which is in the mammal class, which is in the Animal Kingdom. The African elephant, on the other hand, is a species in the Elephantidae family, which is in the order Proboscidea, which is in the same class as chimpanzees, the Mammal class.

By this taxonomy, therefore, chimpanzees and African elephants have the same traits identified in the Mammal class, as well as traits of the Animal Kingdom.

Their differences, on the other hand, are identified in the Primate and Proboscidea orders, as well as the traits within the Hominidae and Elephantidae families, and the chimpanzee and African elephant species. In this chapter, we see how the organization of class type and subtypes are utilized in object-oriented programming.



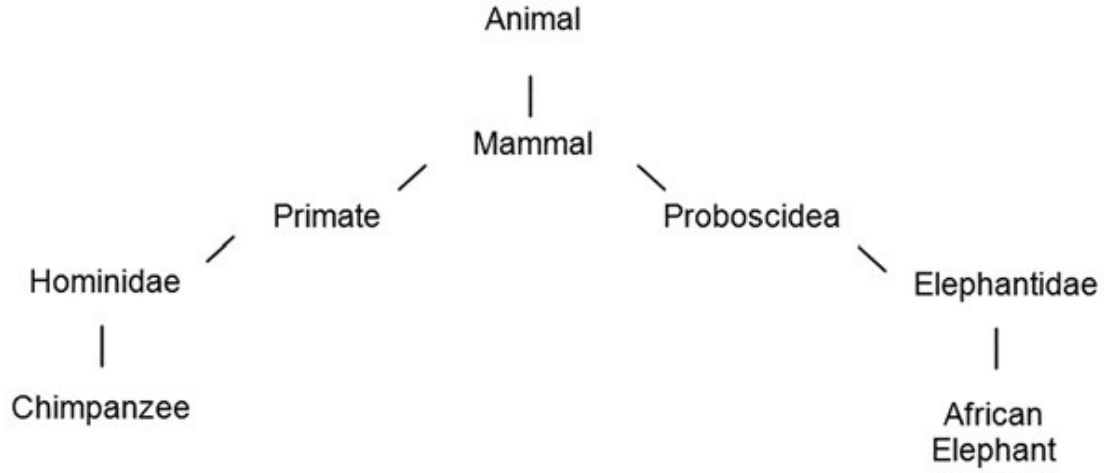


FIGURE 10-1 Taxonomy of Chimpanzees and African Elephants
FUNDAMENTAL CONCEPTS
10.1 What Is Object-Oriented Programming?

We have discussed and have been using objects in our programs. The use of objects in itself, however, does not constitute object-oriented programming. For that, we first need to introduce the concept of a class, which we discuss next.

10.1 What Is Object-Oriented Programming? 385
10.1.1 What Is a Class?

A **class**specifies the set of instance variables and methods that are "bundled together" for defining a type of object. A class, therefore, is a "cookie cutter" that can be used to make as many object instances of that type object as needed. For example, strings in Python are object instances of the built-in String class, depicted in Figure 10-2.

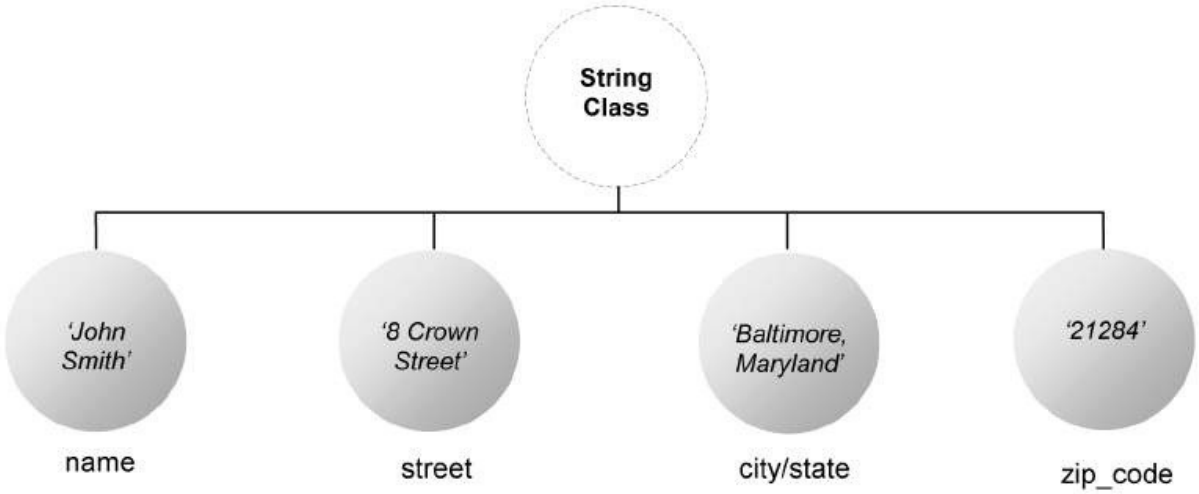


FIGURE 10-2 Object Instances of String Class
One method of the String class is isdigit. Thus, every string object has this method. The specific object whose isdigit method is called determines the specific string that it is applied to, name.isdigit() →False city_state.isdigit() → False address.isdigit() →False zip_code.isdigit() → True We next look at the three fundamental features of object-oriented programming.
A **class** specifies the set of instance variables and methods that are "bundled together" for defining a type of object.
10.1.2 Three Fundamental Features of Object-Oriented Programming

Object-oriented programming languages, such as Python, provide three

fundamental features that support object-oriented programming—*encapsulation*, *inheritance*, and *polymorphism*. These support a paradigm shift in software development from the focus on variables and passing of variables to functions in procedural programming, to the focus on objects and *message passing* between them.

Message passing occurs when a method of one object calls a method of another, as depicted in Figure 10-3. For example, if Object B were a list and B1 a sorting method, then a call to B1 is a message (or request) for it to become sorted. A message to one object can result in the propagation of messages among many other objects in order to accomplish a request. In the next sections we discuss these three features of object-oriented programming, as well as introduce a means of specifying an object-oriented design using UML.

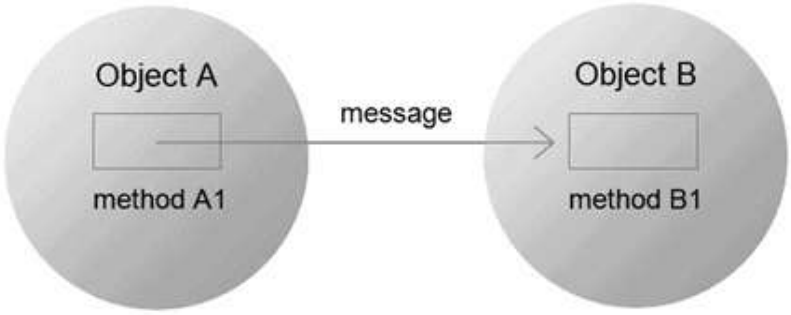


FIGURE 10-3 Message Passing in Object-Oriented Programming

Three fundamental features supporting the design of object-oriented programs are referred to as *encapsulation*, *inheritance*, and *polymorphism*.

10.2 Encapsulation

In this section we develop a Fraction class for demonstrating the notion of encapsulation. (Note that the Fraction class developed here is unrelated to the Fraction class of the Python Standard Library.)

10.2.1 What Is Encapsulation?

Encapsulation is a means of bundling together instance variables and methods to form a given type (class). Selected members of a class can be made inaccessible ("hidden") from its clients, referred to as *information hiding*. Information hiding is a form of abstraction. This is an important capability that object-oriented programming languages provide. As an example, we give a depiction of a Fraction object in Figure 10-4.

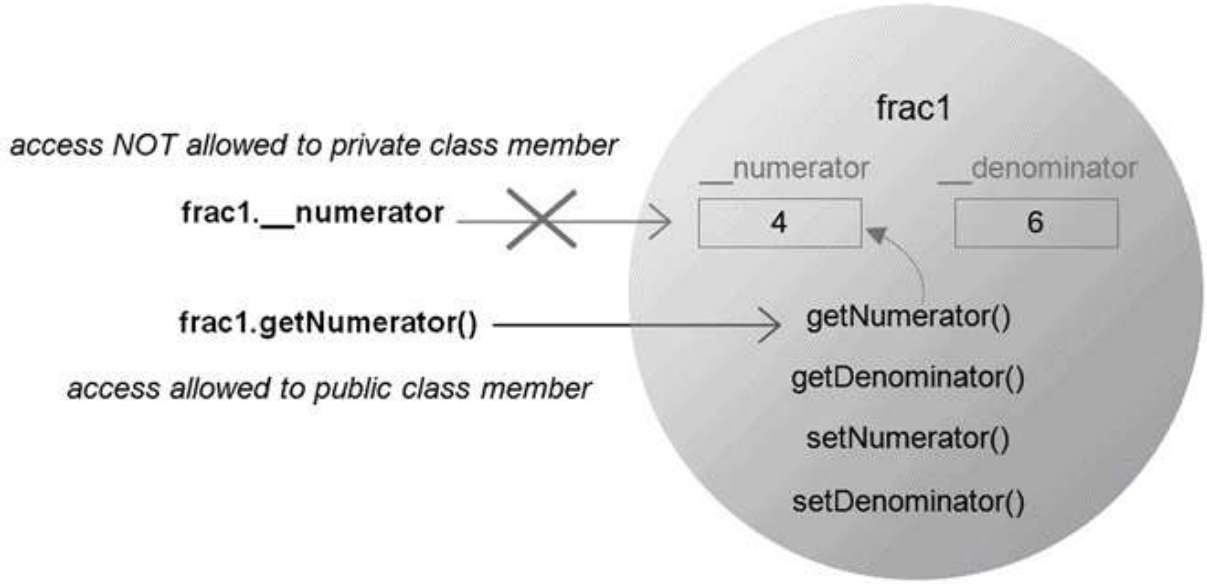


FIGURE 10-4 Fraction Object Access

The Fraction object shows private instance variables __numerator and __denominator, and four public methods. Private members of a class begin with two underscore characters, and cannot be directly accessed. For example, trying to access the instance variables of Fraction object frac1 is invalid,

frac1.__numerator 5 4 NOT ALLOWED frac1.__denominator 56 NOT ALLOWED
Public members of a class, on the other hand, are directly accessible. For example, the following are valid method calls,

frac1.getNumerator() ALLOWED frac1.getDenominator() ALLOWED
frac1.setNumerator(4) ALLOWED frac1.setDenominator(6) ALLOWED

These methods are referred to as **getters** and **setters** since their purpose is to get (return) and set (assign) private instance variables of a class. Restricting access to instance variables via getter and setter methods allows the methods to control what values are assigned (such as not allowing an assignment of 0 to the denominator), and how they are represented when retrieved. Thus, the instance variables of a class are generally made private, and the methods of the class generally made public.

Encapsulation is a means of bundling together instance variables and methods to form a given type, as well as a way of restricting access to certain class

members.
10.2.2 Defining Classes in Python In this section, we develop a Python Fraction class.
Defining a Fraction Class
The first stage in the development of a Fraction class is given in Figure 10-5.

The class keyword is used to define classes, much as def is used for defining functions. All lines following the class declaration line are indented. Instance variables are initialized in the __init__ special method. (We discuss special methods in the following.) Being private, instance variables __numerator and __denominator are not meant to be directly accessed.

... frac1.__numerator
AttributeError: 'Fraction' object has no attribute '__numerator'
In actuality, however, private members *are* accessible if written as follows:
... frac1._Fraction__numerator

```
class Fraction(object):
                                                    Special Methods
    def __init__(self, numerator, denominator):
        """Inits Fraction with values numerator and denominator."""

        self.__numerator = numerator
        self.__denominator = denominator
        self.reduce()
                                                    Getter and Setter Methods
    def getNumerator(self):
        """Returns the numerator of a Fraction."""

        return self.__numerator

    def getDenominator(self):
        """Returns the denominator of a Fraction."""

        return self.__denominator

    def setNumerator(self, value):
        """Sets the numerator of a Fraction to the provided value."""

        self.__numerator = value

    def setDenominator(self, value):
        """Sets the denominator of a Fraction to the provided value.

          Raises a ValueError exception if a value of zero provided.
        """

        if value == 0:
            raise ValueError('Divide by Zero Error')

        self.__denominator = value
```

FIGURE 10-5 Initial Fraction Class

To understand this, all private class members are automatically renamed to begin with a single underscore character followed by the class name. Such renaming of identifiers is called *name mangling*. Unless the variable or method is accessed with its complete (mangled) name, it will not be found. Name mangling prevents unintentional access of private members of a class, while still allowing access when needed.

The methods of a class, as we have seen, are essentially functions meant to operate on the instance variables of the class. In Python, functions serving as a method must have an extra first parameter, by convention named self. This parameter contains a reference to the object instance to which the method

belongs. When a method accesses any other member of the same class, the member name must be preceded by 'self' (self.__numerator). Getter and setter methods are also defined. Note that setDenominator raises an exception when passed a value of 0 to ensure that Fraction objects cannot be set to an invalid value. We further discuss special methods in Python next.
LET'S TRY IT

Enter and execute the following Python class. Then enter the given instructions within the Python shell and observe the results.

class SomeClass(object): ... obj 5 SomeClass() def __init__(self):··· obj.__n self.__n 5 0??? self.n2 5 0 ... obj._SomeClass__n

???
... obj.n2
???

The class keyword is used to define a class in Python. Class members beginning with two underscore characters are intended to be private members of a class. This is effectively accomplished in Python by the use of *name mangling*.

Special Methods in Python

Method names that begin and end with two underscore characters are called *special methods* in Python. Special methods are automatically called. For example, the __init__ method of the Fraction class developed is automatically called whenever a new Fraction object is created,

frac1 5Fraction(1,2) – creates new fraction with value 1/2 frac2 5Fraction(6,8) – creates new fraction with value 6/8

The values in parentheses are arguments to the __init__ method to initialize a new Fraction object to a specific value. Note that although there are three parameters defined (self, numerator, denominator), the first is always implied. Therefore, only the remaining arguments (numerator and denominator) are explicitly provided when creating a Fraction object.

Two other special methods of Python are __str__ and __repr__. These methods are used for representing the value of an object as a string. The __str__ method

is called when an object is displayed using print (and when the str conversion function is used.) The __repr__ function is called when the value of an object is displayed in the Python shell (when interactively using Python). This is demonstrated below.

class DemoStrRepr(): ... s 5 DemoStrRepr() def __repr__(self): ... print(s) return '__repr__ called' __str__ called def __str__(self): ... s return '__str__ called' __repr__ called

The difference in these special methods is that __str__ is for producing a string representation of an object's value that is most readable (for humans), and __repr__ is for producing a string representation that Python can evaluate. If special method __str__ is not implemented, then special method __repr__ is used in its place. An implementation of __repr__ for the Fraction class is given below.

def __repr__(self):
return str(self.__numerator) 1 '/' 1 str(self.__denominator) This, therefore, will display Fraction values as we normally write them:

... frac1 5 Fraction(3,4) ... print('Value of frac1 is', frac1) ... frac1 Value of frac1 is 3/4
3/4

We give special method __repr__ of the Fraction class (to also serve as the implementation of special method __str__) in Figure 10-6.

```
class Fraction(object):
```

Additional Special Methods

```
    def __repr__(self):
        """Returns Fraction value as x/y."""

        return str(self.__numerator) + '/' + str(self.__denominator)
```

FIGURE 10-6 Additional Special Method repr
We will look at more Python special methods in the next section.
LET'S TRY IT
Enter and save the following class definition in a Python file and execute it. Then enter the given instructions in the Python shell and observe the results.
class XYcoord(object):

def __init__(self, x, y): self.__x 5 x
self.__y 5 y
... coord 5 XYcoord(5, 2) ... print(coord)
???

... str(coord) ???
def __repr__(self):
return '(' 1 str(self.__x) 1 ',' \

1 str(self.__y) 1 ')' ... coord ???

Special methods in Python have names that begin and end with two underscore characters, and are automatically called in Python.
Adding Arithmetic Operators to the Fraction Class
Special methods used to provide arithmetic operators for class types are shown in Figure 10-7.

Operator	Example Use	Special Method
- (negation)	-frac1	__neg__
+ (addition)	frac1 + frac2	__add__
- (subtraction)	frac1 - frac2	__sub__
* (multiplication)	frac1 * frac2	__mul__

FIGURE 10-7 Arithmetic Operator Special Methods

The expression frac1 1frac2, for example, evaluates to the returned value of the __add__ method in the class, where the left operand (frac1) is the object on which the method call is made: frac1.__add__(frac2). Arithmetic methods for the Fraction class are given in Figure 10-8.

Arithmetic Operator Special Methods

```
def __neg__(self):
    """Returns a new Fraction equal to the negation of self."""

    return Fraction(-self.__numerator, self.__denominator)
```

```
def __add__(self, rfraction):
    """Returns a new reduced Fraction equal to self + rfraction."""

    numer = self.__numerator * rfraction.getDenominator() + \
            rfraction.getNumerator() * self.__denominator

    denom = self.__denominator * rfraction.getDenominator()

    resultFrac = Fraction(numer, denom)
    return resultFrac.reduce()
```

```
def __sub__(self, rfraction):
    """Returns a new reduced Fraction equal to self - rfraction."""

    return self + (-rfraction)
```

```
def __mul__(self, rfraction):
    """Returns a new reduced Fraction equal to self * rfraction."""

    numer = self.__numerator * rfraction.getNumerator()
    denom = self.__denominator * rfraction.getDenominator()

    resultFrac = Fraction(numer, denom)
    resultFrac.reduce()

    return resultFrac
```

FIGURE 10-8 Arithmetic Operators of the Fraction Class

Special method __add__ is implemented to add the numerator and denominator of each fraction based on a common denominator as shown below,

2/4 1 1/5 → (2 * 5)/(4 * 5) 1 (1 * 4)/(5 * 4) → 10/20 4/20
→ 14/20
→ 7/10 (after call to method reduce)

Note that a reduce method of the class (not given) is called on the resulting fraction to return the result in simplest form. Thus, rather than returning 14/20, the value 7/10 is returned.

The __sub__ special method has a very simple and elegant implementation. It simply adds the first fraction to the negation of the second fraction, relying on the implementation of the __add__ and __neg__ methods. Finally, the __mul__

special method multiplies the numerators of each of the fractions, as well as the denominators of each, building a new Fraction object from the results, and reducing the new fraction to simplest form by call to method reduce. (We omit the special method for the division operator here.)

Just as with the other arithmetic operators, a new (negated) Fraction object is returned by the __neg__ method, rather than the original fraction object being altered. To negate a fraction, reassignment would be used instead,

frac1 5 2frac1
LET'S TRY IT Enter and save the following class definition in a Python file, and execute. Then enter the given instructions in the Python shell and observe the results. class XYcoord(object):

def __init__(self, x, y): self.__x 5 x
self.__y 5 y

def __repr__(self) return '(' 1 str(self.__x) 1 ',' \ ???

1 str(self.__y) 1 ')' def __add__(self, rCoord):
new_x 5 self.__x 1 rCoord.__x
new_y 5 self.__y 1 rCoord.__y
return XYCoord(new_x, new_y)
... coord_1 5 XYcoord(4,2) ... coord_2 5 XYCoord(6,10) ... coord_1 1 coord_2
???

... coord 5 coord_1 1 coord_2 ... print(coord)
Adding Relational Operators to the Fraction Class
We have yet to add the capability of applying relational operators to Fraction objects. The set of relational operator special methods that can be implemented is shown in Figure 10-9.

Operator	Example Use	Special Method
< (less than)	frac1 < frac2	__lt__
<= (less than or equal to)	frac1 <= frac2	__le__
== (equal to)	frac1 == frac2	__eq__
!= (not equal to)	frac1 != frac2	__ne__
> (greater than)	frac1 > frac2	__gt__
>= (greater than or equal to)	frac1 >= frac2	__ge__

FIGURE 10-9 Relational Operator Special Methods

The special methods for providing the relational operators of the Fraction class are given in Figure 10-10.

Relational Operators

```
def __eq__(self, rfraction):
    """Returns True if self arithmetically equal to rfraction.
       Otherwise, returns False.
    """

    temp_frac1 = self.copy()
    temp_frac2 = rfraction.copy()
    temp_frac1.reduce()
    temp_frac2.reduce()

    return temp_frac1.getNumerator() == temp_frac2.getNumerator() and \
       temp_frac1.getDenominator() == temp_frac2.getDenominator()
```

```
def __neq__(self, rfraction):
    """Returns True if Fraction not arithmetically equal to rfraction.
       Otherwise, returns False.
    """

    return not self.__eq__(rfraction)
```

```
def __lt__(self, rfraction):
    """Returns True if self less than rfraction."""

    if self.getDenominator() == rfraction.getDenominator():
        return self.getNumerator() < rfraction.getNumerator()
    else:
        temp_frac1 = self.copy()
        temp_frac2 = rfraction.copy()

        saved_denom = temp_frac1.getDenominator()
        temp_frac1.__adjust(temp_frac2.getDenominator())
        temp_frac2.__adjust(saved_denom)

        return temp_frac1.getNumerator() < temp_frac2.getNumerator()
```

```
def __le__(self, rfraction):
    """Returns True if self less than or equal to rfraction."""

    return not (rfraction < self)
```

```
def __gt__(self, rfraction):
    """Returns True if self greater than rfraction."""

    return not(self <= rfraction)
```

```
def __ge__(self, rfraction):
    """Returns True if self greater than or equal to rfraction."""

    return not(self < rfraction)
```

FIGURE 10-10 Relational Operators of the Fraction Class

Each of the special methods for the relational operators is implemented. For example, frac1 , frac2 is determined by call to method __lt__ on the first object, frac1, with the second object, frac2, passed as an argument,

frac1.__lt__(frac2)

In order to compare two fractions, they must have common denominators. Therefore, method __ lt__ first checks if the denominators are equal. If so, then the result of self.__getNumerator() , rfraction.__getNumerator()is returned. Otherwise, since we do not want the two fractions to be altered as a result of the comparison, a copy of each is made, assigned to temp_frac1 and temp_frac2. In order to convert them common denominators, the numerator and denominator of each is multiplied by the denominator of the other. This is accomplished by call to private method __adjust. Then, the Boolean result temp_frac1.__getNumerator(),temp_frac2.__ getNumerator() is returned.

Most other relational operators are grounded in the implementation of the less than special method, __lt__. The implementation of special method __le__ (less than or equal to) is based on the fact that a ,5 b is the same as not (b , a). Special method __neq__ (not equal to) is simply implemented as not (a 5 b). Finally, special method __gt__ (greater than) is implemented as not a ,5 b, and special method __ge__ (greater than or equal to) is implemented as not (a , b). Finally, the implementation of methods copy, reduce, and __adjust are left as an exercise at the end of the chapter.

The Fraction type developed represents the power of abstraction, as implemented through the use of encapsulation. The Fraction class contains two integer instance variables and an associated set of methods. That is what it "really" is. Through the use of information hiding, however, the client is provided an abstract view, which, for all intents and purposes, *is*a Fraction type. The Fraction class can be defined within its own Python file, and thus serve as a module that can be easily imported and used by any program.

There exist special methods for the arithmetic and relational operators in Python that can be implemented to determine how these operators are evaluated for a given object type.

10.2.3 Let's Apply It—A Recipe Conversion Program

The following Python program (Figure 10-12) will convert the measured amount of ingredients of recipes, based on a provided conversion factor, to vary the number of servings. The program utilizes the following programming features:

➤ programmer-defined class

Example execution of the program is given in Figure 10-11. The program makes use of the fraction class module developed
in section 10.2.2, imported on **line 3**. Since there is only one item to
be imported (the class), the choice of using import Fraction vs.
from Fraction import * only affects whether Fraction objects
are created as frac1 5Fraction.Fraction(1,2) for the
first form of import, or frac1 5Fraction(1,2) for the second

Chocolate Chip Cookies Recipe

4 1/2 cups all-purpose flour
2 teaspoons baking soda
2 cups butter, softened
1 1/2 cups packed brown sugar
1/2 cup white sugar
2 packages instant vanilla pudding mix
4 eggs
2 teaspoons vanilla extract
4 cups semisweet chocolate chips
2 cups chopped walnuts

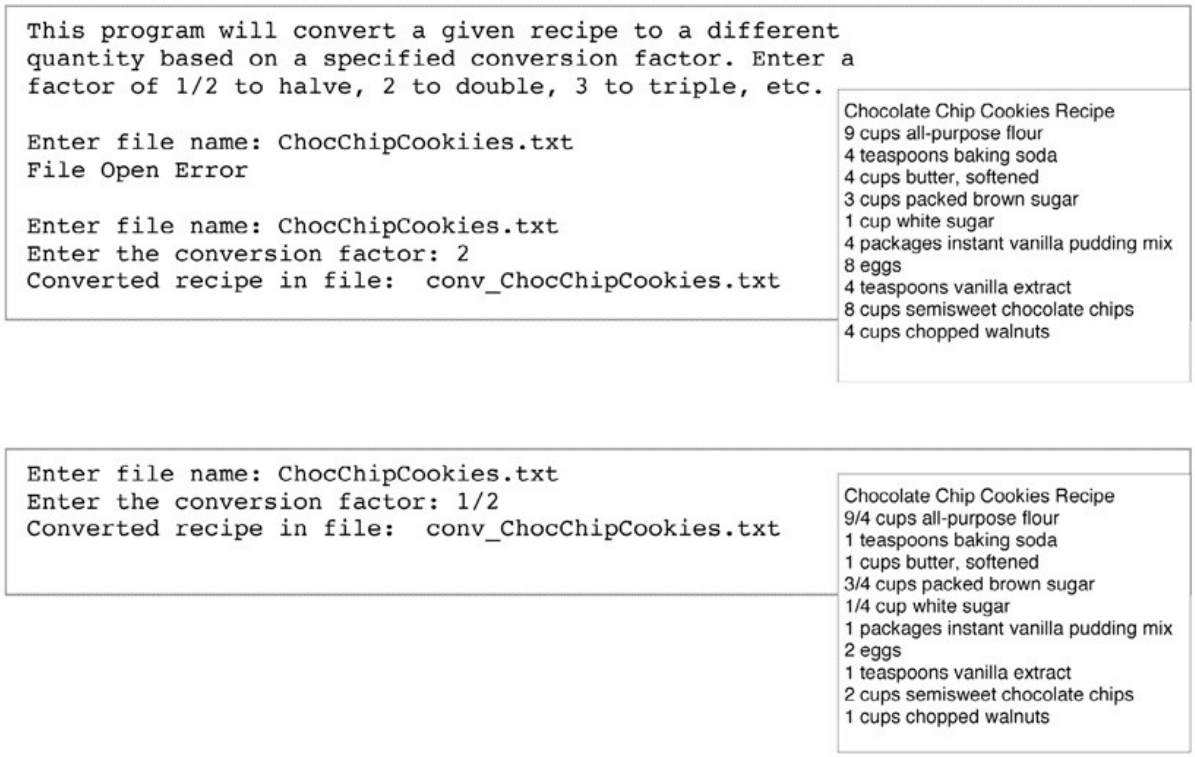


FIGURE 10-11 Execution of the Recipe Conversion Program

form. We therefore choose the from-import form of import. Note that the methods of a class are called the same way regardless of the form of import used to import the class.

The main section of the program is in **lines 106–143**. The program welcome is provided on **lines 109–111**. The rest of the code is encompassed within a try block for catching any IOError exceptions. There is one instance of an IOError exception that is raised by the program (in addition to those raised by the Python standard functions) in function getFile (**lines 5–28**).

The getFile function prompts for a file name to open, returning both the file name and associated file object as a tuple. If, after three attempts the file fails to open successfully, an IOError exception is raised containing the error message 'Exceeded number of open file attempts' (**line 26**). Thus, a try block is used to catch each IOError exception raised by the open function. When such an exception is caught, variable num_attempts is incremented (**line 22**) in the corresponding exception handler (**lines 21–23**). When an input file is successfully opened, the loop terminates and file_name and input_file are returned as a tuple.

Back in the main module of the program, the user is prompted for the conversion factor (**line 119**). Since all calculations in the program are executed as Fraction types, the conversion factor is scanned (read) by call to function scanAsFraction (**lines 45–84**). If a single integer value (read as a string) is entered, for example '2', then scanAsFraction returns the Fraction value 2/1. If a single fraction value is entered, such as '2/4', then the Fraction value (in reduced form) 1/2 is returned. If an integer and fraction are entered, such as '1 1/2', then a Fraction equal to the sum of both is returned, 3/2.

Function scanAsFraction returns, as a Fraction value, the total value of the initial part of the parameter string passed.
4 1/2 cups all-purpose flour

```
# Recipe Conversion Program

from fraction import *

def getFile():

    """Returns as a tuple the file name entered by the user and the
       open file object. If the file exceeds three attempts of opening
       successfully, an IOError exception is raised.
    """

    file_name = input('Enter file name: ')
    input_file_opened = False
    num_attempts = 1

    while not input_file_opened and num_attempts < 3:
        try:
            input_file = open(file_name, 'r')
            input_file_opened = True
        except IOError:
            print('File Open Error\n')
            num_attempts = num_attempts + 1
            file_name = input('Enter file name: ')

    if num_attempts == 3:
        raise IOError('Exceeded number of file open attempts')

    return (file_name, input_file)

def removeMeasure(line):

    """Returns provided line with any initial digits and fractions
       (and any surrounding blanks) removed.
    """

    k = 0
    blank_char = ' '

    while k < len(line) and (line[k].isdigit() or \
                             line[k] in ('/', blank_char)):
        k = k + 1

    return line[k:len(line)]

def scanAsFraction(line):

    """Scans all digits, including fractions, and returns as a
       Fraction object. For example, '1/2' would return as Fraction
       value 1/2, '2' would return as Fraction 2/1, and '2 1/2' would
       return as Fraction value 3/2.
    """

```

```
    completed_scan = False
    value_as_frac = Fraction(0,1)

    while not completed_scan:
        k = 0
        while k < len(line) and line[k].isdigit():
            k = k + 1

        numerator = int(line[0:k])

        if k < len(line) and line[k] == '/':
            k = k + 1
            start = k
            while k < len(line) and line[k].isdigit():
                k = k + 1

            denominator = int(line[start:k])
        else:
            denominator = 1

        value_as_frac = value_as_frac + Fraction(numerator,
                                                 denominator)

        if k == len(line):
            completed_scan = True
        else:
            line = line[k:len(line)].strip()

            if not line[0].isdigit():
                completed_scan = True

    return value_as_frac

def convertLine(line, factor):

    """If line begins with a digit, returns line with the value
       multiplied by factor. Otherwise, returns line unaltered
       (e.g., for a factor of 2, '1/4 cup' returns as '1/2 cup'.)
    """

    if line[0].isdigit():
        blank_char = ' '
        frac_meas = scanAsFraction(line) * factor

        if frac_meas.getDenominator() == 1:
            frac_meas = frac_meas.getNumerator()

        conv_line = str(frac_meas) + blank_char + removeMeasure(line)
    else:
        conv_line = line

    return conv_line

```

```
106 # ---- main
107
108 # display welcome
109 print('This program will convert a given recipe to a different')
110 print('quantity based on a specified conversion factor. Enter a')
111 print('factor of 1/2 to halve, 2 to double, 3 to triple, etc.\n')
112
113 try:
114
115     # get file name and open file
116     file_name, input_file = getFile()
117
118     # get conversion factor
119     conv_factor = input('Enter the conversion factor: ')
120     conv_factor = scanAsFraction(conv_factor)
121
122     # open output file named 'conv_' + file_name
123     output_file_name = 'conv_' + file_name
124     output_file = open(output_file_name, 'w')
125
126     # convert recipe
127     empty_str = ''
128     recipe_line = input_file.readline()
129
130     while recipe_line != empty_str:
131         recipe_line = convertLine(recipe_line, conv_factor)
132         output_file.write(recipe_line)
133         recipe_line = input_file.readline()
134
135     # close files
136     input_file.close()
137     output_file.close()
138
139     # display completion message to user
140     print('Converted recipe in file: ', output_file_name)
141
142 except IOError as err_mesg:  # catch file open error
143     print(err_mesg)
```

FIGURE 10-12 Recipe Conversion Program

First, variable completed_scan is initialized to False (**line 53**) and set to True when the initial digit (and '/') characters have been scanned. Then variable value_as_frac (**line 54**) is initialized to the Fraction value 0/1, so that it can be used to accumulate values (**line 73**) when both an integer and a fractional value are found, as shown. Within the while loop, index variable k is initialized to 0 (**line 57**). Following that, k is incremented to scan past each character in line that is a digit character (as long as k is less than the length of the line). This places k at the index location of the first non-digit. Then, the range of characters scanned so far, line[0:k], is converted to an integer type and assigned to numerator. The next character is then scanned for the ‘/’ character (indicating that a fraction

notation exists), as long as the end of line has not been reached (that is, k ,len(line)). If the slash character is found, then there is a denominator to go with the numerator value just scanned. Thus, the following characters are scanned until a non-digit is found (**lines 66–67**).

The current location of k is first stored in variable start (**line 65**) so that the beginning of the new substring of digits can be scanned. At that point, denominator is set to the integer value of the digits from index start to k21 (line[start:k]. Then, on **line 73**, variable value_ as_frac is set to the current value of value_as_frac plus the newly created Fraction object with the current values of numerator and denominator. If a slash character is not found, then denominator is set to one (**line 71**) so that the integer value scanned, for example 2, is returned as 2/1.

On **line 76** a check is made to determine if the end of the line has been reached. (This would not normally be true of a recipe line, but would be True when scanning a conversion value.) If the end has been reached, then completed_scan is set to True, which terminates the main while loop, causing the final return statement to return the scanned value. If, however, the end of the line has not been reached, the next character is checked to see if it is a digit. If it is, then the while loop at **line 56** iterates again to scan the following value, which is expected to be a fractional value. If the next character is not a digit, then completed_scan is set to True (**line 82**), causing the function to return the (non-fractional) value scanned.

In **lines 123–124** of the main module, output_file_name is assigned to the name of the input file name and prepended with the string 'conv_', and file object output_file is created (where the converted recipe is written). Following that, the process of converting each line of the recipe file begins. First, empty_str is initialized and the first line of the recipe file is read as recipe_line (**lines 127–128**). The while loop at **line 130** then converts the current recipe line by a call to convertLine. Function convertLine (**lines 86–104**) first checks that the first character of the line is a digit. If not, then the line is assigned unaltered to conv_line on **line 102** (since there is no initial numerical value to convert). If a digit is found, then blank_char is initialized (**line 94**), and frac_meas is set to the Fraction value returned by scanAsFunction (for the given conversion factor). If the denominator of frac_meas is found to be 1 (e.g., 2/1), then frac_meas is set to the value of the numerator (2), otherwise it is left as is. Variable line is set to the remaining part of the line by call to removeMeasure (**line 100**), and

conv_line is set to the string representation of frac_meas concatenated with a blank and the remaining part of the original line, and returned. Finally, function removeMeasure (**lines 30–43**) scans past the initial digit, blank, and slash characters, returning the remaining part of the line.

Self-Test Questions

1. Encapsulation bundles together instance variables and methods of a class, and allows certain members of the class to be made inaccessible by use of keyword private in Python. (TRUE/ FALSE)

2. Private members of a class are not meant to be directly accessed by methods of other classes. (TRUE/FALSE)
3. Methods of a class that provide access to privates members of the class are called _____ and _____.

4. Which of the following are special methods in Python?
(a) Getter and setters
(b) Method names that begin and end with two underscore characters
(c) Methods that are part of any Python built-in type

5. Which of the following is a special method for defining an operator in a class?
(a)__init__
(b)__add
(c)__add__

ANSWERS: 1. False, 2. True, 3. getters/setters, 4. (b), 5. (c)

10.3 Inheritance

The true capabilities of object-oriented programming present themselves when inheritance of classes is employed. In this section, we explore the use of inheritance in Python.

10.3.1 What Is Inheritance?

Inheritance , in object-oriented programming, is the ability of a class to inherit members of another class as part of its own definition. The inheriting class is called a **subclass** (also “derived class” or “child class”), and the class inherited from is called the **superclass** (also “base class” or “parent class”). Superclasses may themselves inherit from other classes, resulting in a hierarchy of classes as shown in Figure 10-13 (inherited class members are in gray).

Class A is the superclass of all the classes in the figure. Thus, subclasses B and E each inherit variable var1 and method method1 from Class A. In addition, Class B defines variable var2 and method method2, and Class E defines method5. Since Class C is a subclass of Class B, it inherits everything in Class A and Class B, adding var3 and method3 in its own definition. And since Class D is also a subclass of Class B, it inherits everything in Class A and Class B, also defining method4.

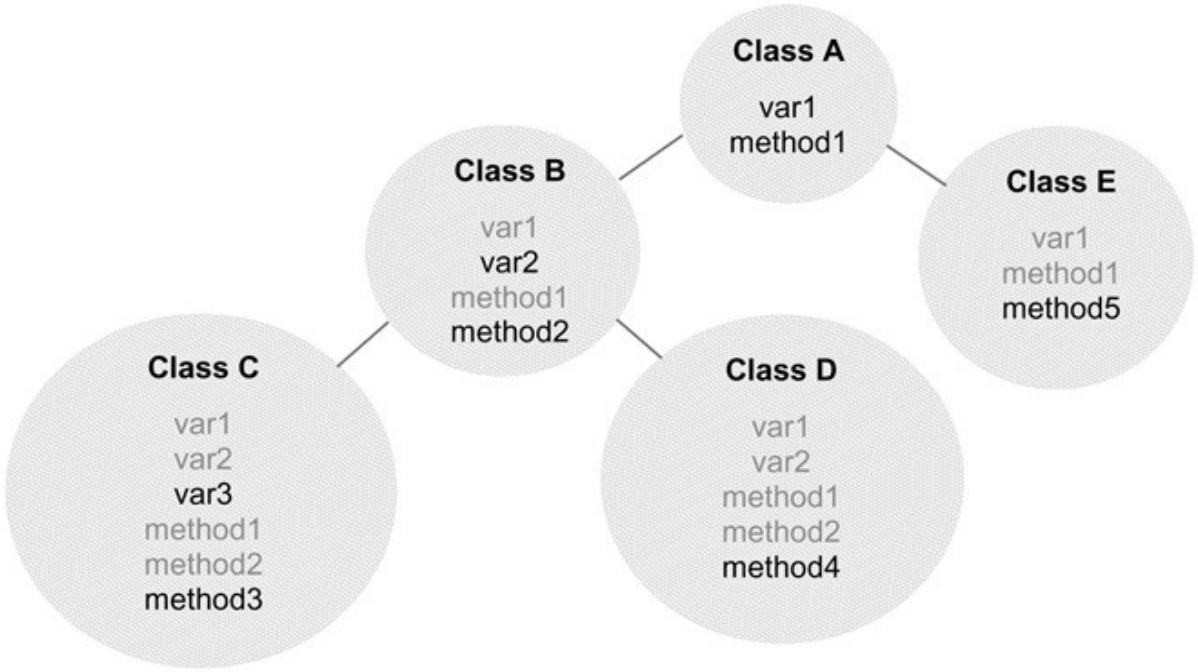


FIGURE 10-13 Class Hierarchy

Inheritance in object-oriented programming is the ability of a **subclass** (also “derived class” or “child class”) to inherit members of a **superclass** (also “base class” or “parent class”) as part of its own definition.

10.3.2 Subtypes

A **subtype** is something that can be substituted for and behave as its parent type (and its parent’s parent type, etc.). For example, consider the characteristic features within the Animal Kingdom in Figure 10-14.

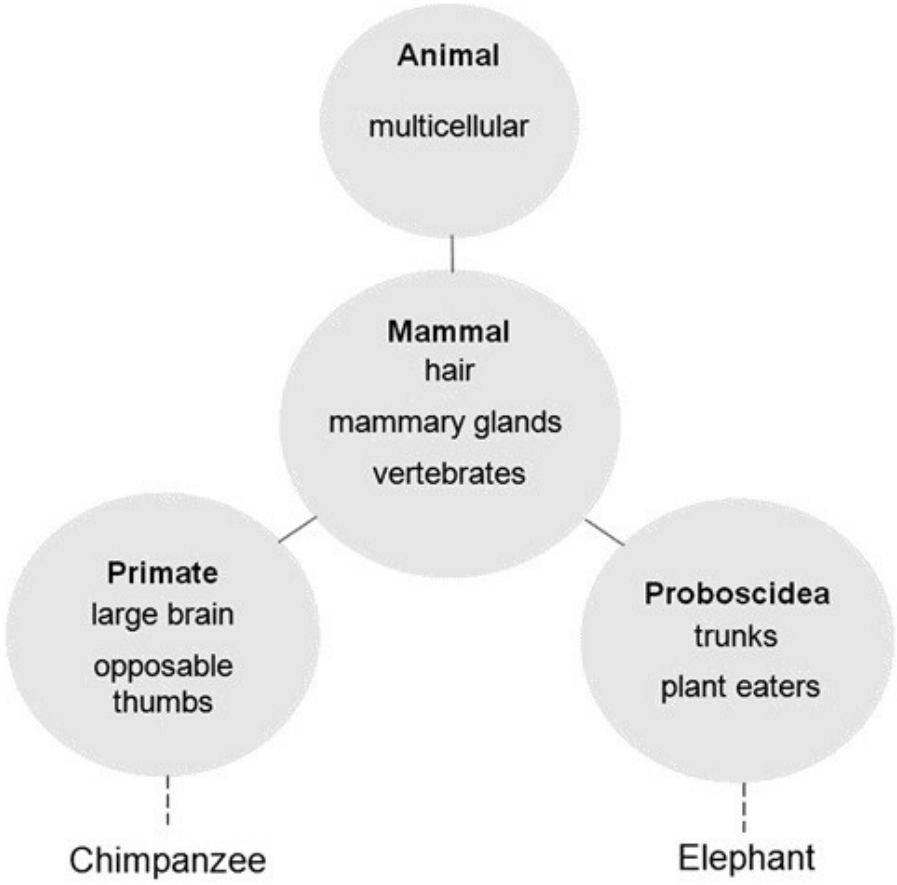


FIGURE 10-14
Hierarchy of Characteristics of Animals

Mammals are a type of animal. Therefore, they have the characteristic of being multicellular. In addition, they have the characteristics of having hair, mammary glands, and vertebrae. Primates (which include chimpanzees) are a type of mammal (and thus a type of animal). Therefore, they have the characteristics of mammals, as well as having large brains and opposable thumbs. Proboscidea (which includes elephants) are a type of mammal (and a type of animal) and thus have the characteristics of mammals, as well as having some form of trunk and being plant eaters. Thus, each classification describes a subtype of the classifications from which it derives. For example, consider the following simple story.

Andy was very interested in animals. He had many books about them, and went to see animals whenever he had the chance.
Because chimpanzees are a type of animal, an alternate version of the story can be generated by substituting “chimpanzee” for “animal,”
Andy was very interested in chimpanzees. He had many books about them, and

went to see chimpanzees whenever he had the chance.
Now consider the following story,
Andy was very interested in chimpanzees. He had many books about them, and loved to watch the chimpanzees swing from tree to tree.
In this case, if we substitute another animal, “elephant,” for “chimpanzee,” we get the following story,
Andy was very interested in elephants. He had many books about them, and loved to watch the elephants swing from tree to tree.

Because an elephant is not a chimpanzee, this version of the story does not make sense. Note, however, that since an elephant is a type of animal, “elephant” can be substituted in the original story just as chimpanzee was and make sense. We next look at how to create subclasses in Python.

A **subtype** is something that can be substituted for and behave as its parent type (and *its* parent type, etc.).
10.3.3 Defining Subclasses in Python

We now look at how to employ the object-oriented programming feature of inheritance in Python. We give an example of an “exploded” string class as a subclass of the built-in string class, and look at whether an exploded string can be substituted as a string type.

Class Names of the Built-In Types in Python

Recall that all values in Python are objects. Thus, there exists a class definition for each of the builtin types. To determine the type (class name) of a particular value (object) in Python, the built-in function type can be used.

... type(12) ... type(12.4) type('') ,class 'int' . ,class 'float' . ,class 'str' . ... type([]) ... type(()) type({}) ,class 'list' . ,class 'tuple' . ,class 'dict' .

The resulting expression ,class *classname*. gives the associated class name for any value. (We use some empty values here, such as the empty string, but the type of any value can be determined this way.) A detailed description of a built-in class can be displayed by use of the help function,

... help(int) ... help(str) Partial display of the str built-in type is given in Figure 10-15.

```
 >>> help(str)
Help on class str in module builtins:

class str(object)
 |  str(string[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given encoded string.
 |  encoding defaults to the current default string encoding.
 |  errors can be 'strict', 'replace' or 'ignore' and defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x
 |
 |  __eq__(...)
 |      x.__eq__(y) <==> x==y
 |
    .
    .

 |  find(...)
 |      S.find(sub[, start[, end]]) -> int
 |
 |      Return the lowest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Return -1 on failure.
    .
    .
```

FIGURE 10-15 Description of Built-in String Class

For programmer-defined classes, such as the Fraction class developed earlier, we use the name of the class (with any required arguments) for creating objects of that type,

frac1 5 Fraction(1,2)

For built-in types such as the str type, object instances are generally created using a more convenient syntax. For example, to create a new string value (object), we simply put quotes around the desired characters of the string,

name 5 'John Smith'

For creating a new list, the list elements are surrounded by square brackets,

nums 5 [10,20,30,40]

We have seen a similar means of creating tuples, dictionaries, and sets. Knowing

the class names of the built-in types, we can alternatively create object instances of each as follows,
... int(1) ... int('1') ... int(1.2) 1 1 1
... list([1, 2, 3]) ...list((1,2,3)) ... list('123') [1, 2, 3] [1, 2, 3] [1, 2, 3]

When using the class name of a built-in type to create a new object instance, arguments of various types can be used to initialize its value. For example, an integer can be created from a provided string or float value; a list can be created from a provided tuple or string. Finally, we note that the built-in class names in Python contain only lowercase letters. Programmer-defined classes, such as Fraction however, are by convention named with a beginning uppercase letter.

LET'S TRY IT
Enter the following in the Python shell and observe the results.

... type(1) ... type([]) ... help(int) ??? ??? ??? ... type(1.5) ... type([1,2,3]) ... help(float) ??? ??? ??? ... type('') ... type(()) ... help(list) ??? ??? ??? ... type('Hi') ... type((1,2,3)) ... help(tuple) ??? ???

Built-in function type can be used to determine the type (class name) of any value in Python. Built-in function help can be used to get the class description of a built-in type.
Defining an Exploded String Type

Given the built-in string class, we can easily create a new string type that is identical to the string class, and in addition provide the option of being "exploded." By an exploded string is meant a string with spaces (blank characters) between all characters, for example, 'H e l l o'. We call this new class ExplodedStr, and give its definition in Figure 10.16.

When defining a subclass, the name of the class is followed by the name of the parent class within parentheses,
class ExplodedStr(str):

```
# Exploded String Class

class ExplodedStr(str):
    def __init__(self, value = ''):

        # call to init of str class
        str.__init__(value)

    def explode(self):

        # empty str returned unaltered
        if len(self) == 0:
            return self
        else:
            # create exploded string
            empty_str = ''
            blank_char = ' '
            temp_str = empty_str

            for k in range(0, len(self) - 1):
                temp_str = temp_str + self[k] + blank_char

            # append last char without following blank
            temp_str = temp_str + self[len(self)- 1]

            # return exploded str by joining all chars in list
            return temp_str
```

FIGURE 10-16 ExplodedStr Type as a Subclass of the Built-in str Class
Our new exploded string type can be used as shown below.

... title 5 ExplodedStr('My Favorite Movies') ... print(title.explode())
M y F a v o r i t e M o v i e s

Let’s see how this works. The ExplodedStr class does not define any instance variables of its own. Thus, its __init__ method simply calls the __init__ method of the built-in str class to pass the value that the string is to be initialized to. If an initial value is not provided, the method has a default argument assigned to the empty string,

def __init__(self, value 5 "):
# call to init of str class
str.__init__(value)
Here str.__init__(value) is used to call the str class’s__init__ method.

Method explode returns the exploded version of the string. First, a check is made

to see if the string is the empty string. If so, then the reference self is returned, thus returning its unaltered value. Otherwise, a new string is created (temp_str), equal to the original string referenced by self, with a blank appended after every character except the last. A for loop is used for this construction,

for k in range(0, len(self) 2 1):
temp_str 5 temp_str 1 self[k] 1 blank_char

Since the range function is called with parameters 0 and len(self) – 1, all except the last character of the original string is appended by this loop (since the last character should not have a blank character appended after it). Finally, the newly constructed exploded string (temp_str) is returned.

We can place the ExplodedStr class in its own module called exploded_str and import it when needed. Testing this new string type, we see that it behaves as desired.

... reg_str 5 'Hello' defining a regular string ... ex_str 5 ExplodedStr('Hello') ...
reg_str
'Hello'
... ex_str
'Hello'
... ex_str.explode()
H e l l o
... reg_str 55 ex_str
True
...
defining an exploded string value of regular string

value of exploded string
call to explode method
comparing the strings
LET’S TRY IT For the following underlined string uStr class definition, enter and save it in a Python file, and execute. Then enter the associated instructions within the Python shell and observe the results.
class uStr(str): ...reg_str 5 'Hello' ...u_str 5 uStr('Hello') def __init__(self, u_str): str.__init__(u_str) ...reg_str

???

```
def underline(self): ...u_str
???
return str.__str__(self) 1 '\n' 1 \
format('', '2,' 1 str(len(self))) ...u_str.underline()
???
...reg_str 55 u_str
???
```

We see that exploded strings can be used as a regular string or in exploded form. Thus, it is only the added behavior of being able to be exploded that is different, not its value. As a result, the ExplodedStr subclass serves as a subtype of the built-in string class. We discuss the related issue of polymorphism next.

When defining a subclass, the name of the class is followed by the name of the parent class within parentheses.

10.3.4 Let's Apply It—A Mixed Fraction Class

The following MixedFraction class is implemented as a subclass (subtype) of the Fraction class developed earlier. The program utilizes the following programming features:

➤ Inheritance of classes

The Fraction class that we developed only represents values as common fractions, that is, with just a numerator and denominator. Thus, the value one and a half is represented as 3/2. Mixed (compound) fractions denote values with a separate whole value and (proper) fraction—3/2 is represented as 1 1/2. In certain applications, such as in the recipe conversion program developed earlier, the latter representation is preferable. An example of this is given below for a conversion of the chocolate chip cookie recipe. Example testing of the Fraction class is shown in Figure 10-17.

```
>>> frac1 = Fraction(5,4)
>>> frac1
5/4
>>> frac2 = Fraction(11,4)
>>> frac2
11/4
>>> -frac1
-5/4
>>> frac1 + frac2
4/1
>>> frac1 - frac2
-3/2
>>> frac1 * frac2
55/16
>>>
```

Use of `Fraction` Type

```
>>> frac1 = MixedFraction(5,4)
>>> frac1
1 1/4
>>> frac2 = MixedFraction(11,4)
>>> frac2
2 3/4
>>> -frac1
-1 1/4
>>> frac1 + frac2
4
>>> frac1 - frac2
-1 1/2
>>> frac1 * frac2
3 7/16
>>>
```

Use of `MixedFraction` Type

FIGURE 10-17 Interactive Testing of the MixedFraction Class

The implementation of the MixedFraction class is given in Figure 10-18. The Fraction class is imported on **line 3** using the from-import form of import. This class provides the (special) methods for performing arithmetic and relational operations on fractions. Therefore, the operations

```
# MixedFraction Class

from fraction import *

class MixedFraction(Fraction):

    def __init__(self, *args):                                    Special Methods
        if len(args) == 2:
            self.__whole_num = 0
            Fraction.__init__(self, args[0], args[1])
        elif len(args) == 3:
            self.__whole_num = args[0]
            Fraction.__init__(self, args[1], args[2])
        else:
            raise TypeError('MixedFraction takes 2 or 3 arguments ' + \
               '(' + str(len(args)) + ' given)')

    def __str__(self):
        empty_str = ''
        blank = ' '

        displayFrac = Fraction.copy(self)
        displayFrac.reduce()

        whole_num = 0
        numer = displayFrac.getNumerator()
        denom = displayFrac.getDenominator()

        if numer == 0:
            return '0'

        if denom == 1:
            return str(numer)

        if numer < 0:
            numer = abs(numer)
            sign = '-'
        else:
            sign = empty_str

        if abs(numer) > abs(denom):
            whole_num = abs(numer) // abs(denom)
            numer = abs(numer) % abs(denom)

        if whole_num == 0:
            return sign + str(numer) + '/' + str(denom)
        else:
            return sign + str(whole_num) + blank + \
                   str(numer) + '/' + str(denom)

    def __repr__(self):
        return self.__str__()

```

FIGURE 10-18 MixedFraction Class (*Continued*)

of addition, subtraction, multiplication, and comparison on mixed fractions can be accomplished by the inherited methods of the Fraction class.

The difference between the Fraction class and the MixedFraction class is in how fraction values are displayed. For example, a Fraction object with the value 3/2 is displayed by the

```
                                                    Getter and Setter Methods
def getWholeNum(self):
    return self.getNumerator() // self.getDenominator()

def setWholeNum(self, value):
    self.setNumerator(self.getNumerator() + \
                      value * self.getDenominator())

def set(self, whole_num, numer, denom):
    Fraction.set(self, numer + whole_num * denom, denom)

                                                Special Arithmetic Operator Methods
def __neg__(self):
    return MixedFraction(-Fraction.getNumerator(self),
                         Fraction.getDenominator(self))

def __sub__(self, rfraction):
    tempFrac = Fraction.__sub__(self, rfraction)

    return self.__createMixedFraction(tempFrac)

def __add__(self, rfraction):
    tempFrac = Fraction.__add__(self, rfraction)

    return self.__createMixedFraction(tempFrac)

def __mul__(self, rfraction):
    tempFrac = Fraction.__mul__(self, rfraction)

    return self.__createMixedFraction(tempFrac)

def __createMixedFraction(self, frac):                    Private Methods
    numer = frac.getNumerator()
    denom = frac.getDenominator()

    return MixedFraction(numer, denom)
```

FIGURE 10-18 MixedFraction Class

MixedFraction class as 1 1/2. For each object instance, however, the same values are stored— as an integer numerator and integer denominator value, and thus can be operated on the same way. How fractions are displayed is determined by the implementation of the __str__ special method. In the Fraction class, the numerator and denominator values are simply concatenated with a ‘/’ between

them. In the MixedFraction class, however, the fraction value is reconstructed in three parts, a whole number part (possibly 0), and a proper fraction part in which the numerator is less than the denominator. How the fraction is displayed depends on these three values. If, for example, the whole number part is 0, then only a proper fraction is displayed,

0 1 1/2 displayed as 1/2

If the denominator is 1, then only the numerator is displayed, 4/1 displayed as 4 Otherwise, both the whole number part and the associated proper fraction part are displayed, 5/4 displayed as 1 1/4

The implementation of special method __str__ in the MixedFraction class is on **lines 18– 49**. Variables empty_str and blank are initialized on **lines 19–20**. These are used in the construction of the mixed-fraction string. On **line 22** a copy is made of the object's value by use of the copy method of the Fraction class. This is so that the value of the fraction can be reduced to its simplest form, in preparation for the conversion of the fraction value to a whole number and proper fraction part. The reduction is performed by the reduce() method inherited from the Fraction class.

Once the (temporary) fraction object is reduced to simplest terms, it can be determined whether there is a whole number part to be displayed as part of the fraction value. This depends on whether the numerator is greater than the denominator. Thus, on **line 25**, variable whole_num is initialized to 0. The numerator and denominator values of the temporary displayFrac object are set to variables numer and denom, respectively (**lines 26–27**). What remains is to determine how the fraction value should be displayed based on these three values. On **line 29**, if variable numer is 0, then the fraction value is also 0; therefore '0' is returned (**line 30**). If the denominator in variable denom is 1, then the fraction value can be displayed as a single integer value (**lines 32–33**).

Next, special method __str__ handles the proper display of negative fractions. For example, 24/3 is stored as negative integer value 4 and positive integer value 3. When this value is displayed in mixed fraction form, the negative sign should appear as follows,

2 1 1/3

Thus, on **line 35**, a check is made to determine if the numerator value in numer

is negative. If so, then numer is set to its absolute value (**line 36**). This prevents a negative sign with the numerator from being displayed. Because a negative sign will need to appear with the whole number part of the displayed value, variable sign is set the dash character (**line 37**). If, on the other hand, the numerator is found to be nonnegative, then variable sign is set to the empty string (**line 39**).

In addition to checking the sign of the numerator, a check is made to see if the numerator is greater than the denominator (**line 41**). If so, then there needs to be a separate whole number value displayed along with the (proper) fraction value. Thus, if the numerator is found to be greater, then variable whole_num is set to the result of the integer division of the absolute value of the numerator by the absolute value of the denominator (**line 42**). Absolute values are used because the sign of the displayed result has already been determined (stored in variable sign). The numerator is then set to the remaining fractional part by use of the modulo operator (**line 43**). Thus, for 4/3, whole_ num is set to 1 (4 // 3), and numer is set to 1 (4 % 3). Finally, if the whole number part of the value is 0, then a string containing only the proper fraction is returned (**line 46**). If not 0, then a string of the form '1 1/3' is constructed and returned (**lines 48–49**).

The remaining part of the MixedFraction class provided the needed getters and setters for mixed fractions (**lines 56–65**). It also provides a set of arithmetic operators and arithmetic methods to replace those inherited from the Fraction class (**lines 69–87**). Finally, private method __createMixedFraction is defined as a private supporting method called by the arithmetic operators in the class.

Self-T est Questions

1. A class is made a subclass of another class by the use of _____.

2. Which of the following contains terms that mean the same thing?
(a) Parent class, derived class
(b) Parent class, base class
(c) Subclass, base class

3. All subclasses are a subtype in object-oriented programming. (TRUE/FALSE)
4. When defining a subclass in Python that is meant to serve as a subtype, the subtype Python keyword is used. (TRUE/FALSE)

5. Built-in function type can be used to,

(a) Determine the type of only the built-in types in Python
(b) Determine the type of only programmer-defined types (classes)
(c) Determine the type of all types

ANSWERS: 1. inheritance, 2. (b), 3. False, 4. False, 5. (c)

10.4 Polymorphism

Polymorphism is a powerful feature of object-oriented programming languages. It allows for the implementation of elegant software that is well designed and easily modified. We explore the use of polymorphism in this section.

10.4.1 What Is Polymorphism?

The word *polymorphism* derives from Greek meaning "something that takes many forms." In objectoriented programming, **polymorphism** allows objects of different types, each with their own specific behaviors, to be treated as the same general type. For example, consider the Shape class and its subclasses given in Figure 10-19.

All Shape objects have an x, y coordinate (with corresponding getter and setter methods). In addition, Shape objects can also calculate their areas. How a shape's area is computed, however,

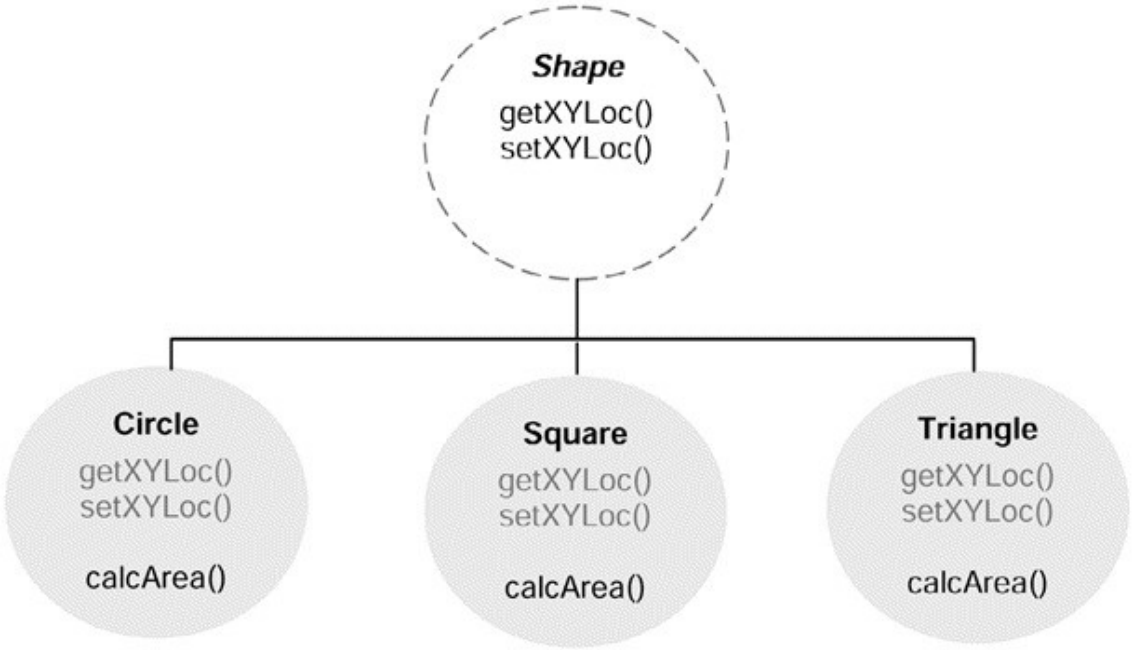


FIGURE 10-19 Polymorphic Shape Class

depends on what shape it is. Thus, it is not possible to define a calcArea method in the Shape class that serves the purposes of all types of shapes. On the other hand, we want all shape types to have a calcArea method. Therefore, we add an *unimplemented* version of the calcArea method to the Shape class as shown in Figure 10-20.

```
class Shape:

    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def getXYLoc(self):
        return (self.__x, self.__y)

    def setXYLoc(self, x, y):
        self.__x = x
        self.__y = y

    def calcArea(self):
        raise NotImplementedError("Method calcArea not implemented")
```

FIGURE 10-20 Abstract Shape Class

Subclasses of the Shape class must implement the calcArea method, otherwise a NotImplementedError exception is raised. A class in which one or more methods are unimplemented (or only implemented to raise an exception) is called an **abstract class**. Figure 10-21 gives Circle, Square, and Triangle subclasses of the Shape class.

```
class Circle(Shape):

    def __init__(self, x, y, r):
        Shape.__init__(self, x, y)
        self.__radius = r

    def calcArea(self):
        return math.pi * self.__radius ** 2
```

```
class Square(Shape):

    def __init__(self, x, y, s):
        Shape.__init__(self, x, y)
        self.__side = s

    def calcArea(self):
        return self.__side ** 2
```

```
class Triangle(Shape):

    def __init__(self, x, y, s):
        Shape.__init__(self, x, y)
        self.__side = s

    def calcArea(self):
        return (self.__side ** 2) * math.sqrt(3) / 4.0
```

FIGURE 10-21 Subclasses Circle, Square, and Triangle

Each of the subclasses contains an __init__ method, in which the first two arguments provide the x,y location of the shape (within a graphics window) and the third argument indicates its size. Each first calls the __init__ method of the Shape class with arguments x,y to set its location, since the x,y values are maintained by the Shape class. Note that methods getXYLoc and setXYLoc are not defined in the subclasses, as they are inherited from the Shape class.

The size of each shape is handled differently, however. In the Circle class size is stored as the radius, and in the Square and Triangle classes it is stored as the length of each side. (The Triangle class represents only equilateral triangles, those in which each side is of the same length.) Given these classes, we can now see how polymorphism works in Python.

Suppose that there was a list of Shape objects for which the total area of all shapes combined was to be calculated,
shapes_list 5 (circle1, circle2, square1, triangle1, triangle2) Because each

implements the methods of the Shape class, they are all of a common general type, and therefore can be treated in the same way,
total_area 5 0
for shape in shapes_list:
total_area 5 total_area 1 shape.calcArea()
LET’S TRY IT

For the following class definitions, enter and save them in a single Python file, and execute. Then enter the associated instructions within the Python shell and observe the results.
class Bird(object):

def __init__(self, w):
print('__init__ of Bird Class called') self.__weight 5 w
... b1 5 BlueJay(1) ???

... b2 5 Cardinal(1.4) ???
def getWeight(self):
return str(self.__weight) 1 'ounces' ... b3 5 BlackBird(3.5) ???
def getColor(self):
raise NotImplementedError(\
'Method color not implemented') ... b1.getWeight() ???
class BlueJay(Bird):
def __init__(self, w): Bird.__init__(self, w) def getColor(self):
return 'Blue'
class Cardinal(Bird):
def __init__(self, w): Bird.__init__(self, w) def getColor(self):
return 'Red'
class BlackBird(Bird):
def __init__(self, w):

Bird.__init__(self, w) ... b2.getWeight() ???

... b3.getWeight() ???
... b1.getColor() ???
... b2.getColor() ???
... b3.getColor() ???

def getColor(self): return 'Black' ... b4 5 Bird(2.0) ... b.getColor()

In Python, it is not because Circle, Square, and Triangle are subclasses of the Shape class that allows them to be treated in a similar way. It is because the classes are *subtypes* of a common parent class. (Actually, in Python, any set of classes with a common set of methods, even if not subclasses of a common type, can be treated similarly. This kind of typing is called **duck typing—** that is, “if it looks like a duck and quacks like a duck, then it’s a duck.”)

In object-oriented programming, **polymorphism** allows objects of different types, each with their own specific behaviors, to be treated as the same general type.
10.4.2 The Use of Polymorphism

To fully appreciate the benefits of polymorphism, let’s consider the development of a program for manipulating geometric shapes. We consider a graphical environment in which classes Circle, Square, and Triangle do not have a common set of methods, and therefore cannot be treated polymorphically.

We assume that we have a graphics program in which the user selects the geometric shape that they want (stored in variable selected_shape) for which the appropriate object type is created:

if selected_shape 55 1: cir 5 Circle(0, 0, 1)
elif selected_shape 55 2: sqr 5 Square(0, 0, 1)
elif selected_shape 55 3: tri 5 Triangle(0, 0, 1)

Next, the geometric object is displayed. Since each object has its own set of methods, the appropriate method must be called. This is determined by use of another if statement:

if selected_shape 55 1: cir.drawCircle()
elif selected_shape 55 2: sqr.drawSquare()
elif selected_shape 55 3: tri.drawTriangle()

The user may then request that the area of the graphic object be displayed. Since each of the Circle, Square, and Triangle classes have a different method for calculating their area, then an if statement must again be used:

if selected_shape 55 1:
area 5 cir.calcCircleArea()
elif selected_shape 55 2:

area 5 sqr.calcSquareArea()
elif selected_shape 55 3:
area 5 tri.calcTriangleArea()

And when the graphic object is repositioned, an if statement is again needed:

if selected_shape 55 1:
cir.moveCircle(x, y)
elif selected_shape 55 2:
sqr.moveSquare(x, y)
elif selected_shape 55 3:
tri.moveTriangle(x, y)

The design of this program becomes rather tedious and inelegant. If statements abound throughout the program. Let's now look at how the same program can be written with the use of polymorphism. First, we give a more complete Shape class in Figure 10-22.

```
class Shape(object):
    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def getXYLoc(self):
        return (self.__x, self.__y)

    def setXYLoc(self, x, y):
        self.__x = x
        self.__y = y

    def draw(self):
        raise NotImplementedError("Method draw not implemented")

    def calcArea(self):
        raise NotImplementedError("Method calcArea not implemented")

    def resize(self, amt):
        raise NotImplementedError("Method resize not implemented")
```

FIGURE 10-22 Methods of Circle, Square, and Triangle Classes Following Duck Typing

As before, the unimplemented methods of the class raise a NotImplementedError

exception. In this case, each of the Circle, Square, and Triangle classes are defined as subclasses of the Shape class. Therefore, each is required to have methods for the unimplemented methods of the Shape class in order to be completely defined.

We reconsider the implementation of a graphics program for manipulating geometric shapes. We give side-by-side listing of nonpolymorphic vs. polymorphic code in Figure 10-23.
One benefit of polymorphism, as apparent from the example, is that the program code is much more straightforward and concise. Without polymorphism, there is constant use of if statements that clutters up the code. Another benefit is that it allows the programmer to think at a more abstract level ("this section of code needs to draw *some kind of geometric shape*") without having to keep in mind specific entities such as circles, squares, and triangles ("this section of code needs to draw *either a circle, square, or triangle*").
The most significant benefit of polymorphism is that programs are much easier to maintain and update. Suppose, for example, that our program had to be updated to also handle rectangles.

Non-Polymorphic Code

```
# Create Appropriate Object

if selected_shape == 1:
    cir = Circle(0, 0, 1)
elif selected_shape == 2:
    sqr = Square(0, 0, 1)
elif selected_shape == 3:
    tri = Triangle(0, 0, 1)


# draw
if selected_shape == 1:
    cir.drawCircle()
elif selected_shape == 2:
    sqr.drawSquare()
elif selected_shape == 3:
    tri.drawTriangle()

# calc area
if selected_shape == 1:
    area = cir.calcCircleArea()
elif selected_shape == 2:
    area = sqr.calcSquareArea()
elif selected_shape == 3:
    area = tri.calcTriangleArea()

# resize
if selected_shape == 1:
    cir.resizeCircle(percentage)
elif selected_shape == 2:
    sqr.resizeSquare(percentage)
elif selected_shape == 3:
    tri.resizeTriangle(percentage)

# reposition
if selected_shape == 1:
    cir.setXY(x, y)
elif selected_shape == 2:
    sqr.setPosition(x, y)
elif selected_shape == 3:
    tri.moveTo(x, y)
```

Polymorphic Code

```
# Create Appropriate Object

if selected_shape == 1:
    fig = Circle(0, 0, 1)
elif selected_shape == 2:
    fig = Square(0, 0, 1)
elif selection_shape == 3:
    fig = Triangle(0, 0, 1)


# draw
fig.draw()

# calc area
area = fig.calcArea()

# resize
fig.resize(selected_percentage)

# reposition
fig.setXYLoc(x, y)
```

For the call to method `draw()`, the method defined in the specific subclass is the actual method called.

For the call to method `calcArea()`, the method defined in the specific subclass is the actual method called.

For the call to method `resize()`, the method defined in the specific subclass is the actual method called.

For the call to method `setXYLoc()`, there is no method defined in any of the subclasses. Therefore, for each particular shape, the method of the `Shape` class is the method called.

FIGURE 10-23 Nonpolymorphic vs. Polymorphic Code

How much of the program would need to be changed? Without polymorphism, every if statement for the selection of method calls would need to be updated. With polymorphism, however, only the first if statement that creates the appropriate type object would need to be changed to include a Rectangle type. The rest of the code would remain the same, as shown in Figure 10-24.

In the end, selection needs to occur somewhere—either within the program (in the nonpolymorphic approach), or by the programming language (with the use of polymorphism).
Polymorphism allows the programmer to think at a more abstract level during program development, and supports the development of programs that are easier to maintain and update.

```
# Create Appropriate Object

if selected_shape == 1:
    fig = Circle(0, 0, 1)
elif selected_shape == 2:
    fig = Square(0, 0, 1)
elif selected_shape == 3:
    fig = Triangle(0, 0, 1)
elif selected_shape == 4:
    fig = Rectangle(0, 0, 1)
```

FIGURE 10-24 Updated Polymorphic Code for Incorporating a New Rectangle Type

Self-Test Questions

1. The term *polymorphism* in object-oriented programming refers to
(a) The ability to treat various type objects in a similar way
(b) The ability to change an object from one type to another type
(c) The ability to have multiple objects of the same type treated as one

2. The use of duck typing in Python results in
(a) More restriction on the type values that can be passed to a given method **(b)** Less restriction on the type values that can be passed to a given method

3. The *biggest* reason for the use of polymorphism in a program is
(a) There is less program code to write
(b) The program will result in a more elegant design, and thus will be easier to maintain and

update
(c) It allows the programmer to think at a more abstract level
ANSWERS: 1. (a), 2. (b), 3. (b)

10.5 Object-Oriented Design Using UML

We have mainly focused on object-oriented *programming* (OOP) . The first step

in the development of any object-oriented program, however, is the development of an appropriate **object-oriented design (OOD)** . Next we discuss a specification language for denoting an object-oriented design, referred to as the *Unified Modeling Language*.

10.5.1 What Is UML?

The **Unified Modeling Language (UML)** is a standardized design specification (modeling) language for specifying an object-oriented design. The term "Unified" comes from the fact that the language is a unification of three earlier object-oriented design modeling languages.

UML is a language-independent, *graphical* specification language. It contains numerous types of graphical diagrams for expressing various aspects of an object-oriented design. One of the most widely used graphical diagrams is called a "class diagram." A class diagram specifies the classes and their relationships of a given object-oriented design. We look at class diagrams in UML next.

UML ("Unified Modeling Language") is a standardized language-independent, graphical modeling language for specifying an object-oriented design.

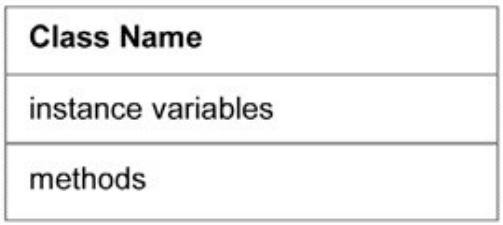
10.5.2 UML Class Diagrams

In UML, **class diagrams** are used to express the *static* aspects of a design, such as the instance variables and methods of individual classes, their visibility (i.e., public or private), and various relationships between classes. Other UML diagrams, called **interaction diagrams**, are used to represent the sequence of method calls between objects during program execution (the *dynamic* aspect of a design). We omit discussion of interaction diagrams and look at the UML notation for denoting the classes of an object-oriented design.

Class diagrams in UML are used to express the static aspects of an object-oriented design. Other diagrams, called **interaction diagrams**, are used to represent the sequence of method calls between objects during program execution.

The Representation of Classes

A class is denoted in UML in three parts: a class name, a set of class attributes (instance variables), and a set of methods, as given below.



We give the UML specification for the abstract Shape class and the (concrete) Circle class in Figure 10-25. The names of unimplemented (abstract) methods are denoted in italics.

Initialization methods like __init__ in Python are named create() in UML. The types of attributes (instance variables) and the return type of methods is indicated by : ,type name., for any given type (for example, :Integer). The 1 and 2 symbols are used to specify if a given member of a class has either public (1) or private (2) access.

A class is denoted in UML in three parts: a class name, a set of class attributes (instance variables), and a set of methods.

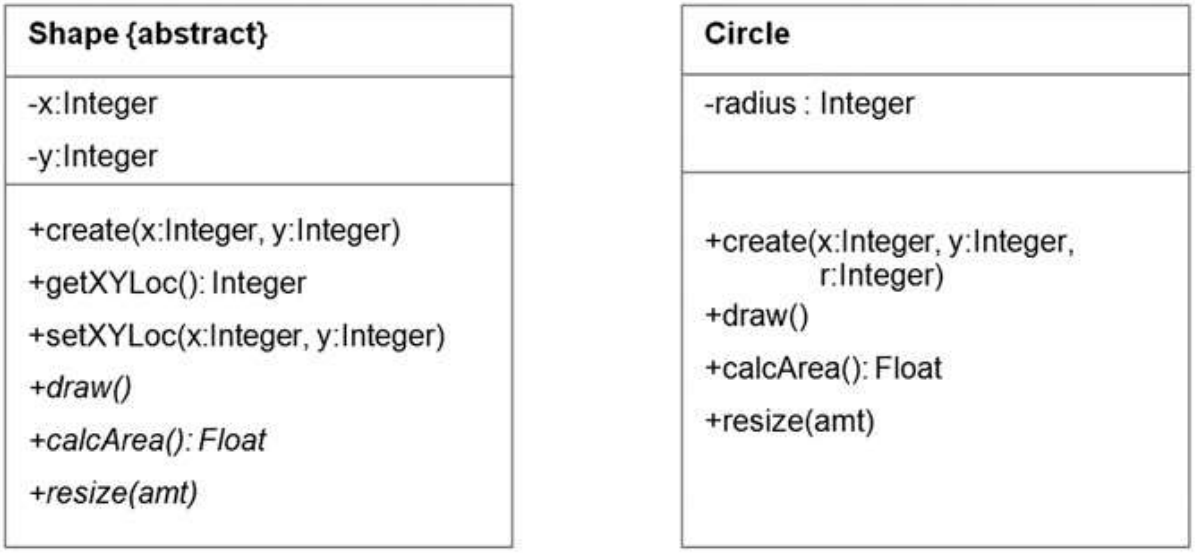


FIGURE 10-25 UML Class Diagrams for Shape and Circle Classes

Denoting Associations between Classes

Associations are the most common relationship in UML class diagrams. An **association** between two classes indicates that the methods of one class make calls to methods of the other, as shown in Figure 10-26.

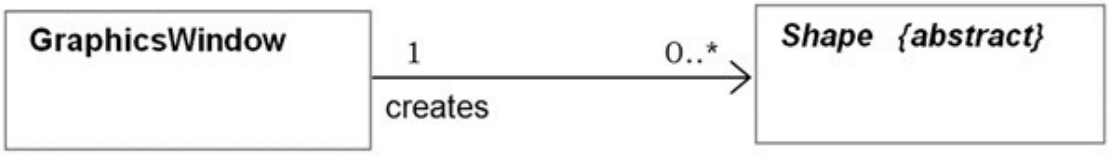


FIGURE 10-26 Association in UML

In this example, it is assumed that the classes are part of a graphical design package, in which the GraphicsWindow class creates and can manipulate a set of Shape objects. (We have left out the details of the GraphicsWindow and Shape classes in this diagram.)

The numbers above the association ends are referred to as **multiplicity**. The multiplicity of 1 at the GraphicsWindow end of the association and 0..* at the Shape end indicates that one GraphicsWindow object may be associated with any number of (zero or more) Shape objects. The “creates” label at the GraphicsWindow association end is referred to as a **role name**. Role names are used to describe an association between two classes.

Finally, the arrow denotes **navigability**. It indicates the direction of method calls made. In this example, it shows that a GraphicsWindow object makes method calls (sends messages) to Shape objects, and not the other way around.

An **association** between two classes, denoted by a connecting solid line (and a possible arrowhead) indicates that methods of one class call methods of the other.

Denoting Subclass Relationships

Subclasses are indicated in UML by use of a solid line with a closed arrow head from a subclass to its superclass, as shown in Figure 10-27.

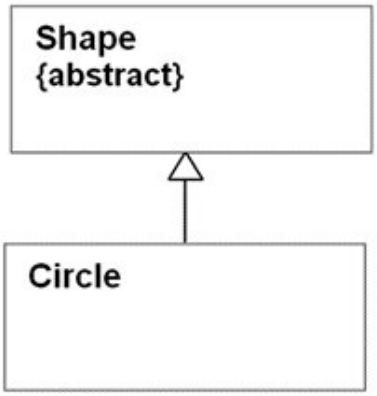
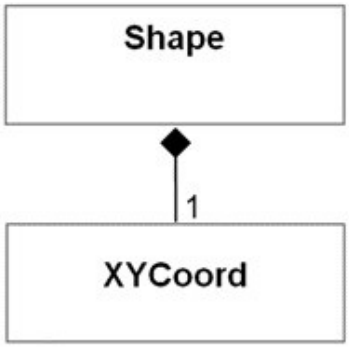


FIGURE 10-27 UML Subclass Notation

This diagram indicates that the Circle class is a subclass of the abstract Shape class. (We again have left out the details of both the Shape and Circle classes.) Note that multiplicity is not used in subclass relationships.

Subclass relationships in UML are indicated by use of a solid line with a closed arrow head from a subclass to a superclass.
Denoting Composition vs. Aggregation

Composition indicates a "part-of" relationship between classes (Figure 10-28). The containing object is viewed as "owning" the contained object, of which the contained object is an integral part. For example, the Shape class is comprised of two integers for holding the x,y location of any Shape type. Because these two integer values are used together, we could develop an XYCoord class, in which the Shape class contains an instance. This is an example of the use of composition.



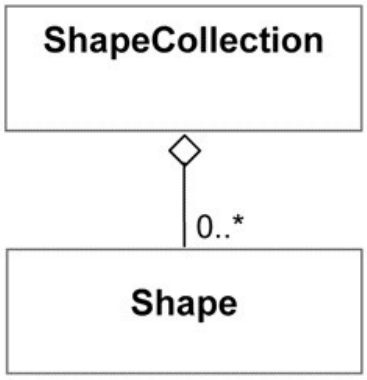

```
def class XYCoord:

    def __init__(self, x, y):
        self.__xCoord = x
        self.__yCoord = y

    def getXY(self):
       return (self.__x, self.__y)

    def setXY(self, x, y):
        self.__xCoord = x
        self.__yCoord = y
```

FIGURE 10-28 Composition of Classes in UML

When denoting composition, a filled diamond head is used at the end of the line connected to the containing class (the Shape class). With composition, it is implied that the containing class (Shape) makes calls to the member class (XYCoord), and thus composition also connotes the relationship of association.

We note that, since all values in Python are objects, *every* class with instance variables involves the use of composition, including the Shape class. However, the reason to indicate a non-built-in type as composition is that it provides a place to specify the details of the type. Instance variables of a built-in type can simply be included in the class as primitive types as was done for the x and y instances variables of the Shape class above.

Aggregation , in contrast to composition, is not a part-of relationship. It is used to denote a class that groups together, or aggregates, a set of objects that exists independently of the aggregating class. An example of aggregation is given in Figure 10-29.




```
def class ShapeCollection:

    def __init__(self):
        self.__collection = []

    def add(self, shape):
        self.__collection.append(shape)

    def remove(self, shape):
        self.__collection.remove(shape)
```

FIGURE 10-29 Aggregation of Classes in UML

Aggregation is denoted by an unfilled diamond head. Here, the ShapeCollection class contains references to an arbitrary number of Shape objects. This might be used, for example, when a graphics window allows the user to select a group of Shape objects on the screen and change an attribute of each, such as their size, all at once.

Composition is a “part of” relationship between classes denoted by a filled diamond head in UML. **Aggregation** is a “grouping” relationship, denoted by an unfilled diamond head.

An Example Class Diagram

Although UML is a specification language for modeling object-oriented software, it can be used to specify any set of entities and their relationships. Modeling everyday concepts and entities can be instructive in understanding UML. Figure 10-30 shows a UML class diagram modeling the concept of a car.

For every car, there is one engine, an integral part of a car. Thus, a composition relationship is denoted between Car and Engine, with a multiplicity of 1 on each end. As with engines, tires are an integral part of a car, so this is also indicated by composition, with a multiplicity of four tires for each passenger car.

A car is still a car with or without a driver. Therefore, a relationship of composition is not appropriate here. There is simply an association between Car

and Driver with multiplicity of 0..1 on the driver end of the association. Since a Driver is a Person, a subclass relationship is denoted between the two.

There is an association between Car and Person with a multiplicity of 0..*. Since the association denoted is not so apparent (owner? passenger?), we add the role name Passenger to the Person end of the association to be more explicit. Finally, any number of Drivers may belong to AAA (Automobile Association of America), denoted by the use of aggregation.

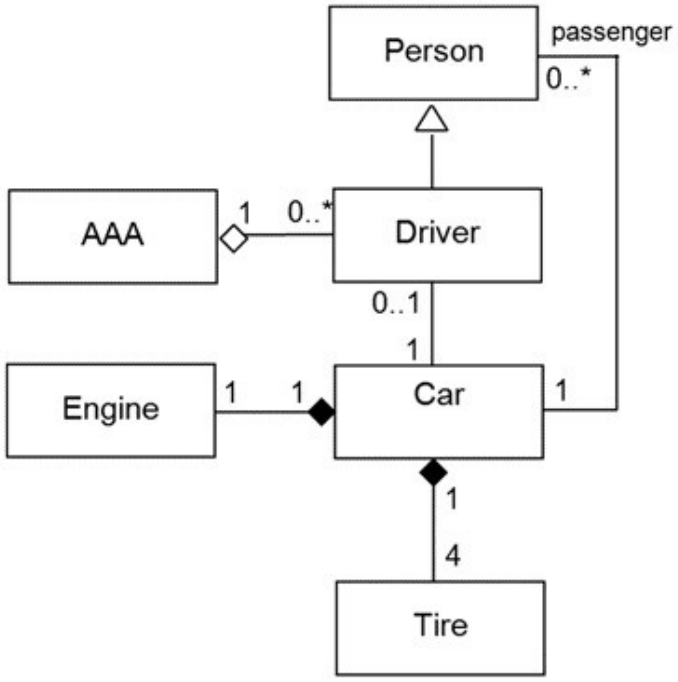


FIGURE 10-30 Passenger Car

UML
Class Diagram
Self-Test Questions

1. Which of the following is true of UML?
(a) UML is a specification language for designing Python programs
(**b)** UML is a specification language that can be used for designing programs in various programming languages

2. In UML, class diagrams are used to express the _____ aspects of a design, and _____ are used to denote the dynamic aspects

3. In UML, an association between two classes indicates that
(a) The two classes have a common superclass

(b) Objects of each of the two class types are created at the same time
(c) Methods of one of the classes make calls to methods of the other

4. Multiplicity in UML indicates
(a) How many objects of a given class type exist
(b) How many objects of one given class there are in relation to another
(**c)** How many subclasses of a given class there may be

5. Composition in UML indicates,
(a) A “part of” relationship
(b) A grouping of objects

6. Aggregation in UML indicates,
(a) A “part of” relationship
(b) A grouping of objects

ANSWERS: 1. (b), 2. static, interaction diagrams, 3. (c), 4. (b), 5. (a), 6. (b)

COMPUTATIONAL PROBLEM SOLVING 10.6 Vehicle Rental Agency Program

In this section, we design, implement, and test a program that will serve the needs of a vehicle rental agency.

10.6.1 The Problem

The problem is to develop an object-oriented design and imple mentation of a program capable of maintaining reservations for a vehicle rental agency. The agency rents out three types of vehicles—cars, vans, and moving trucks. The program should allow users to check for available vehicles, request rental charges by vehicle type, get the cost of renting a particular type vehicle for a specified period of time, and make/cancel reservations.

10.6.2 Problem Analysis

The program needs an appropriate set of objects for the vehicle rental agency domain. An obvious class to include is a Vehicle class. It can be implemented to maintain information common to all vehicle types. Subclasses of the Vehicle class can maintain information specific to each subtype.

For example, all vehicles have a miles-per-gallon rating and a vehicle identification number (VIN). Thus, this information can be maintained in the

Vehicle class. However, there are different make and model cars (with either two or four doors, that hold a specific number of passengers); different make and model vans (able to hold a specific number of passengers); and moving trucks of various lengths, each providing a certain amount of cargo space. Therefore, the Vehicle class is made a superclass of classes Car, Van, and Truck, in which each subclass contains information (instance variables and/or methods) specific to that vehicle type.

For each type vehicle, there is a rental charge based on daily, weekly, and weekend rental rates. There is also a mileage charge and some number of free miles (on select vehicles), plus the cost of optional insurance. Because these costs are associated with particular types, but cost is not inherently *part* of a vehicle's attribute, we include a separate VehicleCost class.

Finally, we incorporate a Reservation class that maintains the information for each reservation made. This will include the customer name, address, credit card number, and the VIN of the vehicle rented.

10.6.3 Program Design

Meeting the Program Requirements

The general requirements for this program are for users to be able to check for the availability of vehicles of a certain type (cars, vans, or trucks); request rental charges by vehicle type; determine the rental cost for a particular vehicle and rental period; and make and cancel reservations. The specific requirements of this program are given in Figure 10-31.

The program must maintain a group of specific model vehicles for the following vehicle categories: cars, vans, and (moving) trucks with the following characteristics:

Cars: make/model, miles-per-gallon, num of passengers, num of doors, VIN
Vans: make/model, miles-per-gallon, number of passengers, VIN
Trucks: miles-per-gallon, length, number of rooms, VIN

The program must be able to display the specific vehicles available for rent by vehicle type.

The program must display the cost associated with a given type vehicle including daily, weekend and weekly rate, insurance cost, mileage charge, and number of free miles. It must also allow the user to determine the cost of a particular vehicle, for a given period of time, an estimated number of miles, and the cost of optional insurance.

The program must be able to allow a particular vehicle to be reserved and cancelled.

FIGURE 10-31 Program Requirements for the Vehicle Rental Agency Program

The specific rental costs for each vehicle type are given in Figure 10-32.

	Daily Rate	Weekly Rate	Weekend Rate	Free Miles Per Day	Per Mile Charge
Car	$24.99	$180.00	$45.00	100	0.15
Van	$35.00	$220.00	$55.00	0	0.20
Truck	$34.95	$425.00	$110.00	25	0.25

FIGURE 10-32 Rental Costs by Vehicle Type

The specific vehicles in stock at the rental agency are shown in Figure 10-33.

Data Description

All the data is stored as string types, converted to a numeric type when needed in a computation (such as the cost of daily insurance).

Algorithmic Approach

The algorithmic methods of the program will consist of simple search (for finding and retrieving the requested vehicle information by the user), updating of information (for marking vehicles as reserved or unreserved), and direct calculation (for calculating the total cost of a rental).

Make/Model	Mileage	Num Passengers	Num Doors	Vehicle #
CARS				
Chevrolet Camaro	30 mpg	4	2	WG8JM5492DY
Chevrolet Camaro	30 mpg	4	2	KH4GM4564GD
Ford Fusion	34 mpg	5	4	AB4FG5689GM
Ford Fusion Hybrid	35 mpg	5	4	GH2KL4278TK
Ford Fusion Hybrid	32 mpg	5	4	KU4EG3245RW
Chevrolet Impala	36 mpg	6	4	QD4PK7394JI
Chevrolet Impala	30 mpg	6	4	RK3BM4256YH

Make/Model	Mileage	Num Passengers	Vehicle #
Vans			
Chrysler Town&Country	25 mpg	7	DK3KG8312UE
Chrysler Town&Country	25 mpg	7	VM9RE2645TD
Chrysler Town&Country	25 mpg	7	WK8BF4287DX
Dodge Caravan	25 mpg	7	QK3FL4278ME
Dodge Caravan	25 mpg	7	KY8EW2053XT
Ford Expedition	20 mpg	8	JK2RT8364HY
Ford Expedition	20 mpg	8	KH4ME4216XW

Make/Model	Mileage	Cargo Space	Vehicle #
Trucks			
Ten-Foot	12 mpg	1 bedroom	EJ5KU2435BC
Ten-Foot	12 mpg	1 bedroom	KF8JP7293EK
Seventeen-Foot	10 mpg	2 bedrooms	KG4DM5472RK
Seventeen-Foot	10 mpg	2 bedrooms	PR8JH4893WQ
Twenty-Four-Foot	8 mpg	4 bedrooms	EP2WR3182QB
Twenty-Four-Foot	8 mpg	4 bedrooms	TY3GH4290EK
Twenty-Four-Foot	8 mpg	4 bedrooms	KU9FL4235RH

FIGURE 10-33 Specific Vehicles of the Vehicle Rental Agency

Overall Program Steps

The overall steps in this program design are given in Figure 10-34.

UML Class Diagram

We give a UML class diagram for the program in Figure 10-35. In addition to the "domain objects" that we have decided on in our analysis, we add a text-based user interface, provided by the RentalAgencyUI class.

Three classes store the information in the system— Vehicle (and its subclasses), VehicleCost, and Reservation. For each of these classes there is a corresponding aggregator class— Vehicles, VehicleCosts, and Reservations—that maintains a collection of the corresponding object type. Each aggregator class has methods

for maintaining its collection of objects (for example, addVehicle in the Vehicles class and addVehicleCost in the VehicleCosts class).

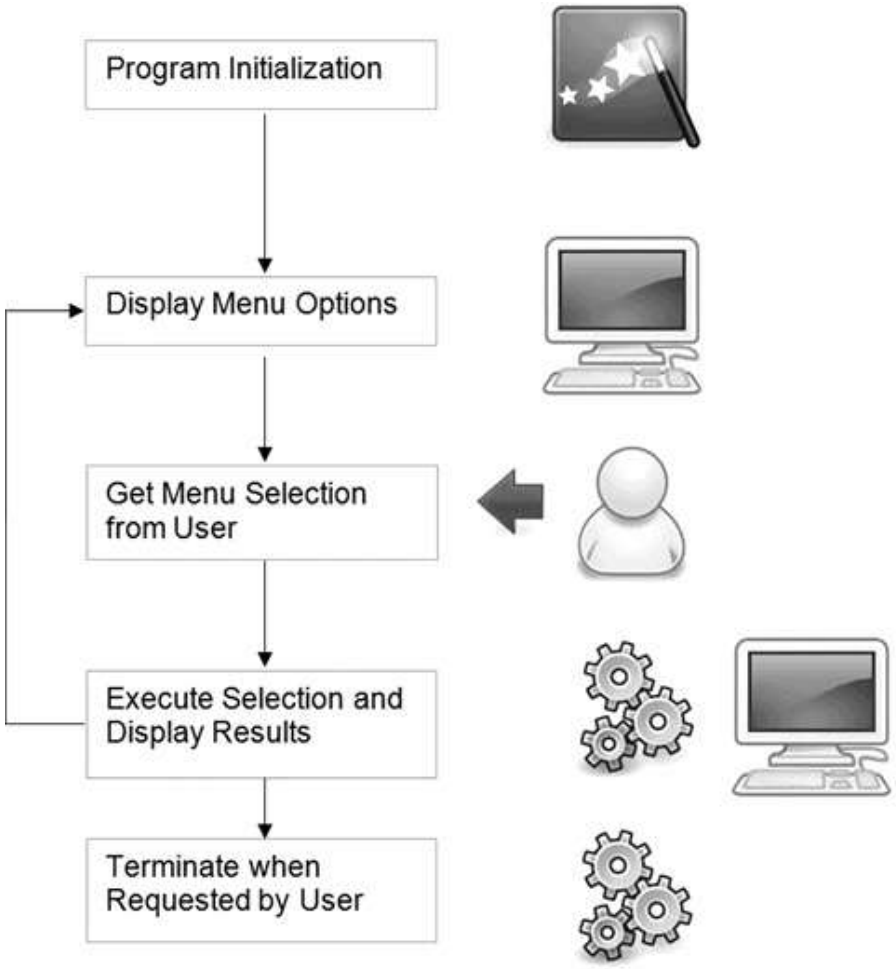


FIGURE 10-34
Overall Design of the Vehicle Rental Agency Program

The Vehicle class has three subclasses—Car, Van, and Truck. It is responsible for maintaining a vehicle's type, its VIN, and its reservation status. Method getDescription is provided in the Vehicle class to return the information common to all vehicles: miles per gallon, and a VIN. Each subclass builds on this inherited method to include the specific information for that vehicle type. The Car class stores the maximum number of passengers and number of doors, the Van class stores the maximum number of passengers, and the Truck class stores its length and the number of rooms of storage it can hold.

The VehicleCost class does not have any subclasses. Its create (__init__) method

is passed six arguments: the daily/weekly/weekend rates, the number of free miles, the per mile charge, and the daily insurance rate to initialize the object with. The getVehicleCost method of the VehicleCosts aggregating class returns the cost of a specified vehicle type as a single string for display. The Reservation and corresponding Reservations aggregator class are designed in a similar manner.

Finally , a SystemInterface class provides all the methods that any user interface would need for interacting with the system. Such a set of methods is referred to as an API— *Application Programming Interface*. Thus, the SystemInterface object is created first. It then reads all the vehicle rental agency data from text files VehiclesStock.txt and RentalCost.txt and populates the corresponding objects. Then, an instance of the RentalAgencyUI is created and initialized with a reference to the system interface. The only public method of the

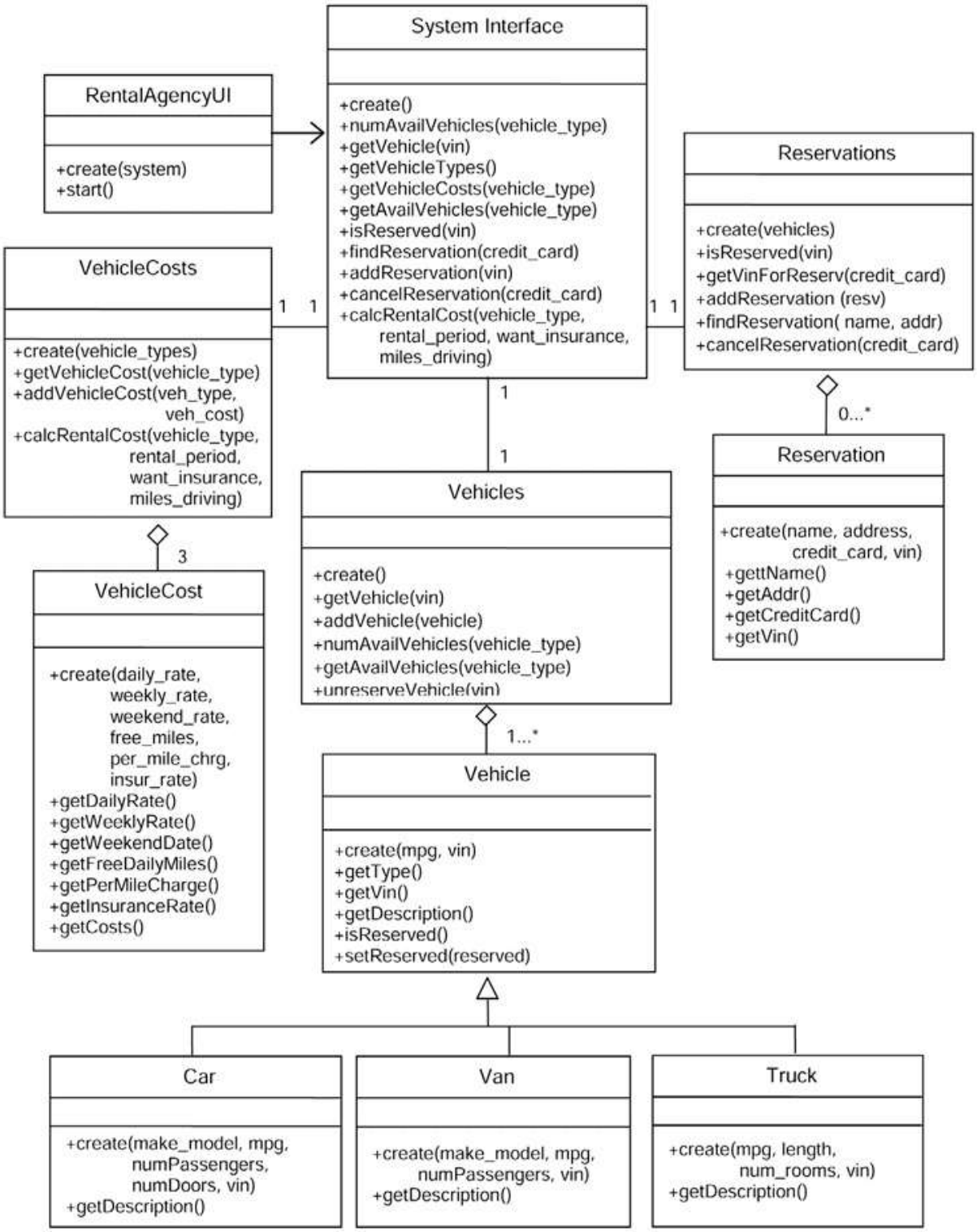


FIGURE 10-35 Class Diagram for Vehicle Rental Agency Program

RentalAgencyUI class, start, is called to start the console interaction. The main menu for the program is given in Figure 10-36.

Example use of the system is shown in Figure 10-37. (For the sake of space, the main menu is not repeatedly shown before each selection as in the actual program execution.)

```
<<< MAIN MENU >>>
1 - Display vehicle types
2 - Check rental costs
3 - Check available vehicles
4 - Get cost of specific rental
5 - Make a reservation
6 - Cancel a reservation
7 - Quit

Enter:
```

FIGURE 10-36 Text-Based (Console) Interface for the Vehicle Rental Agency Program

Display Vehicle Types

```
**************************************************
  * Welcome to the Friendly Vehicle Rental Agency *
**************************************************

<<< MAIN MENU >>>
1 - Display vehicle types
2 - Check rental costs
3 - Check available vehicles
4 - Get cost of specific rental
5 - Make a reservation
6 - Cancel a reservation
7 - Quit

Enter: 1
--------------- Types of Vehicles Available for Rent ---------------

1 - Car
2 - Van
3 - Truck
--------------------------------------------------------------------
```

Display Rental Fees for a Given Type Vehicle

```
Enter: 2
Enter type of vehicle
1 - Car
2 - Van
3 - Truck

Enter: 1
---------------------- Rental Charges for Cars ----------------------

                        Free   Per Mile  Daily
Daily  Weekly  Weekend  Miles  Charge    Insurance
24.99  180.00  45.00    100    .15       14.99
--------------------------------------------------------------------
```

Check for Available Vehicles

```
Enter: 3
Enter type of vehicle
1 - Car
2 - Van
3 - Truck

Enter: 1
-------------------------- Available Cars --------------------------
Chevrolet Camaro     passengers: 4   doors: 2    mpg: 30   vin: WG8JM5492DY
Chevrolet Camaro     passengers: 4   doors: 2    mpg: 30   vin: KH4GM4564GD
Ford Fusion          passengers: 5   doors: 4    mpg: 34   vin: AB4FG5689GM
Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: GH2KL4278TK
Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: KU4EG3245RW
Chevrolet Impala     passengers: 6   doors: 4    mpg: 30   vin: QD4PK7394J1
```

Get the Cost of a Particular Rental

```
Enter: 4
Enter type of vehicle
1 - Car
```

FIGURE 10-37 Example Program Execution (*Continued*)

```
2 - Van
3 - Truck
Enter: 1
Enter the rental period:
1 - Daily   2 - Weekly  3 - Weekend

Enter: 1
How many days do you need the vehicle? 3
Would you like the insurance? (y/n): n
Number of miles expect to drive?: 240

--------------------- ESTIMATED Car RENTAL COST ----------------------
* You have opted out of the daily insurance *

Daily rental for 3 days would be $ 74.97

Your cost with an estimated mileage of 240 would be 95.97
which includes 100 free miles and a charge of 0.15 per mile
----------------------------------------------------------------------
```

Reserve a Particular Vehicle

```
Enter: 5

Enter type of vehicle
1 - Car
2 - Van
3 - Truck

Enter: 1
-------------------------- Available Cars --------------------------
1-Chevrolet Camero     passengers: 4   doors: 2    mpg: 30   vin: WG8JM5492DY
2-Chevrolet Camero     passengers: 4   doors: 2    mpg: 30   vin: KH4GM4564GD
3-Ford Fusion          passengers: 5   doors: 4    mpg: 34   vin: AB4FG5689GM
4-Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: GH2KL4278TK
5-Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: KU4EG3245RW
6-Chevrolet Impala     passengers: 6   doors: 4    mpg: 30   vin: QD4PK7394J1
7-Chevrolet Impala     passengers: 6   doors: 4    mpg: 30   vin: RK3BM4256YH

Enter number of vehicle to reserve: 4
Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: GH2KL4278TK

Enter first and last name: John Smith
Enter address: 123 Main Street, Sometown 12345
Enter credit card number: 204881605
* Reservation Made *
```

Cancel a Reservation

```
Enter: 6
Please enter your credit card number: 204881605

RESERVATION INFORMATION
Name: John Smith 123 Main Street, Sometown 21252
Vehicle: Chevrolet Camaro     passengers: 4   doors: 2    mpg: 30   vin: KH4GM4564GD

Confirm Cancellation (y/n): y
* Reservation Cancelled *

Enter: 7
Thank you for using the Friendly Rental Agency
```

FIGURE 10-37 Example Program Execution

10.6.4 Program Implementation and Testing

From the UML specification, we implement and test the program. We start with the implementation of the Vehicle class, given in Figure 10-38. The classes will be placed in their own file, and imported into the program.

Development and Testing of the Vehicle, Car, Van, and Truck Classes

In the Vehicle class, the __init__ method (**lines 10–15**) defines the attributes common to each subclass—mpg, vin, and reserved. There is only one setter method, setReserved, since the other values are defined when the object is created. Other methods include getters getType (**line 17**), getVin (**line 22**), getDescription (**line 27**), and Boolean isReserved (**line 35**).

```
"""This module provides a Vehicle class. """

class Vehicle:
    """The Vehicle class holds the mpg, vin and reserved flag of a vehicle.

       Contains attributes common to all vehicles: mpg, vin, and reserved
       (Boolean). Provides polymorphic behavior for method getDescription.
    """

    def __init__(self, mpg, vin):
       """Initializes a Vehicle object with mpg and vin."""

        self.__mpg = mpg
        self.__vin = vin
        self.__reserved = False

    def getType(self):
        """Returns the type of vehicle (car, van, truck.)"""

        return type(self).__name__

    def getVin(self):
        """Returns the vin of vehicle."""

        return self.__vin

    def getDescription(self):
        """Returns general description of car, not specific to type."""

        descript = 'mpg:' + format(self.__mpg, '>3') + '   ' + \
                   'vin:' + format(self.__vin, '>12')

        return descript

    def isReserved(self):
        """Returns True if vehicle is reserved, otherwise returns False."""

        return self.__reserved

    def setReserved(self, reserved):
        """Sets reserved flag of vehicle to provided Boolean value."""

        self.__reserved = reserved
```

FIGURE 10-38 Vehicle Class

The getType methods returns the specific type of a Vehicle object (Car, Van, or Truck) by use of type(self).__name__ which returns its own type.

We can execute the file and interactively and unit test the class in the Python shell.

... v 5 Vehicle('32', 'ABC123') ... v.getType()
'Vehicle'
... v.getVin()
'ABC123'

— create instance
— get object type
— get vehicle identification number We test the setReserved and isReserved methods similarily. An implementation of the Car subclass is given in Figure 10-39.

```
1   """This module provides a Car class, a subtype of the Vehicle class."""
2
3   from vehicle import Vehicle
4
5   class Car(Vehicle):
6       """This class is a subtype of the Vehicle class.
7
8          Contains additional attributes of make and model, num of passengers, and
9          num of doors. Supports polymorphic behavior of method getDescription.
10      """
11
12      def __init__(self, make_model, mpg, num_passengers, num_doors, vin):
13          """Initialized with provided parameter values."""
14
15          super().__init__(mpg, vin)
16
17          self.__make_model = make_model
18          self.__num_passengers = num_passengers
19          self.__num_doors = num_doors
20
21      def getDescription(self):
22          """Returns description of car as a formatted string."""
23
24          spacing = '   '
25          descript = format(self.__make_model, '<18') + spacing + \
26                     'passengers: ' + self.__num_passengers + spacing + \
27                     'doors: ' + format(self.__num_doors, '<2') + spacing + \
28                     Vehicle.getDescription(self)
29
30          return descript
```

FIGURE 10-39 Car Subclass of the Vehicle Class

The Car class is defined as a subclass of the Vehicle class, imported on **line 3**. The methods inherited from the Vehicle class have already been tested. We therefore test method getDescription (**lines 21–30**) by executing the Car class file and performing the following.

... v 5 Car('Ford Fusion', '34', '5', '4', 'AB4FG5689GM') ... v.getDescription() 'Ford Fusion passengers: 5 doors: 4 mpg: 34 vin: AB4FG5689GM'

These results are correct. Figure 10-40 and Figure 10-41 show similar implementations of the Van and Truck classes.

The Van and Truck classes differ from the Car class in the particular instance

variables they contain, and the information returned by method getDescription. These differences support the polymorphic behavior of Vehicle types.

Development and Testing of the Vehicles Class

Whereas the Vehicle class represents the information of a single vehicle, the Vehicles class maintains a complete collection of Vehicle types. Figure 10-42 illustrates an implementation of the Vehicles class, maintaining the cost, availability, and reservation status of all vehicles of the rental agency.

```
"""This module provides a Van class, a subtype of the Vehicle class."""

from vehicle import Vehicle

class Van(Vehicle):

    """This class is a subtype of the Vehicle class.

       Contains additional attributes of make and mode, and num of passengers.
       Supports polymorphic behavior of method getDescription.
    """
    def __init__(self, make_model, mpg, num_passengers, vin):
        """ Initializes with make-model, mpg, num_passengers, vin."""

        super().__init__(mpg, vin)

        self.__make_model = make_model
        self.__num_passengers = num_passengers

    def getDescription(self):
        """Returns complete description of van"""

        spacing = '  '
        descript = format(self.__make_model, '<22') + spacing + \
                   'passengers:' + format(self.__num_passengers, '>2') + \
                   spacing + Vehicle.getDescription(self)

        return descript
```

FIGURE 10-40 Van Subclass of the Vehicle Class

```
"""This module provides a Truck class. a subtype of the Vehicle class."""

from vehicle import Vehicle

class Truck(Vehicle):

    """This class is a subtype of the Vehicle class.

       Contains additional attributes length and num of rooms storage capacity.
       Supports polymorphic behavior of method getDescription.
    """
    def __init__(self, mpg, length, num_rooms, vin):
        """ Initializes with mpg, length, num_rooms, vin."""

        super().__init__(mpg, vin)

        self.__length = length
        self.__num_rooms = num_rooms

    def getDescription(self):
        """Returns complete description of truck."""

        spacing = '   '
        descript = 'length(feet):' + format(self.__length, '>3') + spacing + \
                   'rooms:' + format(self.__num_rooms, '>2') + spacing + \
                   Vehicle.getDescription(self)

        return descript
```

FIGURE 10-41 Truck Subclass of the Vehicle Class

```
1  """Vehicles class module. Raises InvalidFileFormatError and InvalidVinError."""
2
3  from vehicle import Vehicle
4  from car import Car
5  from van import Van
6  from truck import Truck
7
8  class InvalidVinError(Exception):
9      """Exception indicating that a provided vin was not found."""
10     pass
11
12
13 class Vehicles:
14     """This class maintains a collection of Vehicle objects."""
15
16     def __init__(self):
17         """Initializes empty list of vehicles."""
18
19         self.__vehicles = []
20
21     def getVehicle(self, vin):
22         """Returns Vehicle for provided vin. Raises InvalidVinError."""
23
24         for vehicle in self.__vehicles:
25             if vehicle.getVin() == vin:
26                 return vehicle
27
28         raise InvalidVinError
29
30     def addVehicle(self, vehicle):
31         """Adds new vehicle to list of vehicles."""
32
33         self.__vehicles.append(vehicle)
34
35     def numAvailVehicles(self, vehicle_type):
36         """Returns number of available vehicles of vehicle_type."""
37
38         return len(self.getAvailVehicles(vehicle_type))
39
40     def getAvailVehicles(self, vehicle_type):
41         """Returns a list of unreserved Vehicles objects of vehicle_type."""
42
43         return [veh for veh in self.__vehicles \
44                 if veh.getType() == vehicle_type and not veh.isReserved()]
45
46     def unreserveVehicle(self, vin):
47         """Sets reservation status of vehicle with vin to unreserved."""
48
49         k = 0
50         found = False
51
52         while not found:
53             if self.__vehicles[k].getVin() == vin:
54                 self.__vehicles[k].setReserved(False)
55                 found = True
```

FIGURE 10-42 Vehicles Class

The Vehicles class imports classes Vehicle, Car, Van, and Truck (**lines 3–6**). An

InvalidVinError exception class is defined (**lines 8–10**), raised when method getVehicle is called for a nonexistent VIN. Method addVehicle (**lines 30–33**) adds a new vehicle to the collection. This is called when the Vehicles object is initially populated at the start of the program (in the SystemInterface class). Method getAvailVehicles (**lines 40–44**) returns a list of vehicles that are not currently reserved.

We can easily test these classes from the Python shell as given below.

First, the needed classes are imported: ... from vehicle import Vehicle ... from car import Car
... from van import Van
... from truck import Truck ... from vehicles import Vehicles

Then, an instance of each type Vehicle is created: ... veh_1 5 Car('Ford Fusion', '34', '5', '4', 'FG1000') ... veh_2 5 Van('Dodge Caravan', '25', '7', 'TF1000') ... veh_3 5 Truck('12', '10', '1', 'HG1000')

A Vehicles instance is created, and each Vehicle instance is added to the collection: ... v 5 Vehicles()
... v.addVehicle(veh_1)
... v.addVehicle(veh_2)
... v.addVehicle(veh_3)

Finally, the description of each vehicle is obtained: ... v.getAvailVehicles('Car')[0].getDescription()
'Ford Fusion passengers: 5 doors: 4 mpg: 34 vin: FG1000'

... v.getAvailVehicles('Van')[0].getDescription()
'Dodge Caravan passengers: 7 mpg: 25 vin: TF1000' ...
v.getAvailVehicles('Truck')[0].getDescription()
'length(feet): 10 rooms: 1 mpg: 12 vin: HG1000' ...

Method getAvailVehicles returns a list of available cars for the provided vehicle type. Since this returns a list of vehicles, the first vehicle in the list is selected (at index [0]), and the getDescription method is called to display the full description of the vehicle. We see that we get the correct results.

Implementation of the **VehicleCost** and **VehicleCosts** Classes The VehicleCosts class is responsible for maintaining and providing the costs for each vehicle

type. First, the VehicleCost class is given in Figure 10-43.

```
1  """This module provides a VehicleCost class."""
2
3  class VehicleCost:
4      """This class provides the methods for maintaining rental costs."""
5
6      def __init__(self, daily_rate, weekly_rate, weekend_rate,
7                   free_miles, per_mile_chrg, insur_rate):
8          """Initialzes rental rates, num free miles /mileage chrg/insur rate."""
9
10         self.__daily_rate = daily_rate
11         self.__weekly_rate = weekly_rate
12         self.__weekend_rate = weekend_rate
13         self.__free_miles = free_miles
14         self.__per_mile_chrg = per_mile_chrg
15         self.__insur_rate = insur_rate
16
17     def getDailyRate(self):
18         """Returns daily rental rate of vehicle."""
19
20         return float(self.__daily_rate)
21
22     def getWeeklyRate(self):
23         """Returns weekly rental rate of vehicle."""
24
25         return float(self.__weekly_rate)
26
27     def getWeekendRate(self):
28         """Returns weekend rental rate of vehicle."""
29
30         return float(self.__weekend_rate)
31
32     def getFreeMiles(self):
33         """Returns number of free miles for vehicle rental."""
34
35         return int(self.__free_miles)
36
37     def getPerMileCharge(self):
38         """Returns per mile charge for vehicle rental."""
39
40         return float(self.__per_mile_chrg)
41
42     def getInsuranceRate(self):
43         """Returns daily insurance rate for vehicle."""
44
45         return float(self.__insur_rate)
46
47     def getCosts(self):
48         """Returns a list containing all costs for vehicle."""
49
50         return [self.__daily_rate, self.__weekly_rate,
51                 self.__weekend_rate, self.__free_miles,
52                 self.__per_mile_chrg, self.__insur_rate]
```

FIGURE 10-43 VehicleCost Class

The VehicleCost class, like the Vehicle class, stores information accessed by the provided getter methods. The information to be stored is passed to the __init__ method (**lines 6–15**). The getCosts method (**lines 47–52**) returns all the

individual cost components as a list. This method is called when the vehicle costs need to be displayed, or the cost of a particular vehicle rental needs to be determined.
We can easily test the VehicleCost class from the Python shell as given below.

First, the VehicleCost class is imported: ... from vehicleCost import VehicleCost
Then, an instance of each type VehicleCost is created: ... vc 5
VehicleCost('24.99', '180.00', '45.00', '100', '.15', '14.99')
Finally, each getter method is tested:

```
... vc.getDailyRate()
24.99
... vc.getWeeklyRate()
180.0
... vc.getWeekendRate()
45.0
... vc.getFreeMiles()
100
... vc.getPerMileCharge()
0.15
... vc.getInsuranceRate()
14.99
... vc.getCosts()
['24.99', '180.00', '45.00', '100', '.15', '14.99'] ...
```

We see that we get the correct results.
In Figure 10-44, we give the VehicleCosts class that maintains the collection of VehicleCost objects.
The vehicleCosts class, as with the Vehicles class, maintains a collection of VehicleCost objects. The VehicleCost class is imported on **line 3**. On **lines 6–8** three symbolic
constants are defined, DAILY_RENTAL, WEEKLY_RENTAL, and WEEKEND_RENTAL. The use of
these constants (equal to integer values 1, 2, and 3, respectively) make the program more readable
than if the corresponding integer values were used.
The VehicleCosts class is defined on **lines 10–89**. The vehicle costs for the three types of vehicles (cars, vans, and trucks) are stored in a dictionary using the vehicle types as

key values. The __init__ method (**lines 13–16**), therefore, initializes instance variable
vehicle_costs to an empty dictionary. The vehicle costs are individually added to the collection of costs by method addVehicleCost (**lines 23–26**), called from the SystemInterface class when populating the rental costs from file. Method getVehicleCost
(**lines 18–21**) returns the costs as a list of individual costs. Finally, method calcRentalCost
(**lines 28–89**) calculates the cost of a particular rental. We test the VehicleCosts class from
the Python shell.

```
"""This module provides a VehicleCosts class"""

from vehicleCost import VehicleCost

# symbol constants
DAILY_RENTAL = 1
WEEKLY_RENTAL = 2
WEEKEND_RENTAL = 3

class VehicleCosts:
    """This class provides the methods for maintaining rental costs."""

    def __init__(self):
        """Initializes vehicle costs to empty."""

        self.__vehicle_costs = dict()

    def getVehicleCost(self, vehicle_type):
        """Returns VehicleCost object for the specified vehicle type."""

        return self.__vehicle_costs[vehicle_type]

    def addVehicleCost(self, veh_type, veh_cost):
        """Adds a vehicle cost object to dictionary with keyword veh_type."""

        self.__vehicle_costs[veh_type] = veh_cost

    def calcRentalCost(self, vehicle_type, rental_period,
                       want_insurance, miles_driving):
        """Returns estimate of rental cost for provided parameter values.

           Returns dictionary with key values: {'base_charges', 'insur_rate',
           'num_free_miles', 'per_mile_charge', 'estimated_mileage_charges'}
        """

        # get vehicle cost
        vehicle_cost = self.getVehicleCost(vehicle_type)

        # calc rental charges
        rental_time = rental_period[1]

        if rental_period[0] == DAILY_RENTAL:
            rental_rate = vehicle_cost.getDailyRate()
            rental_period_value = DAILY_RENTAL
            rental_period_str = 'daily rental'
            rental_days = rental_time
        elif rental_period[0] == WEEKLY_RENTAL:
            rental_rate = vehicle_cost.getWeeklyRate()
            rental_period_value = WEEKLY_RENTAL
            rental_period_str = 'weekly rental'
            rental_days = rental_time * 7
        elif rental_period[0] == WEEKEND_RENTAL:
            rental_rate = vehicle_cost.getWeekendRate()
            rental_period_value = WEEKEND_RENTAL
            rental_period_str = 'weekend rental'
            rental_days = 2

```

FIGURE 10-44 VehicleCosts Class (*Continued*)

```
58          elif rental_period[0] == WEEKEND_RENTAL:
59
60              rental_rate = vehicle_cost.getWeekendRate()
61              rental_period_value = WEEKEND_RENTAL
62              rental_period_str = 'weekend rental'
63              rental_days = 2
64
65          # get free miles, per mile charge and insurance rate
66          num_free_miles = vehicle_cost.getFreeMiles()
67          per_mile_charge = vehicle_cost.getPerMileCharge()
68          insurance_rate = vehicle_cost.getInsuranceRate()
69
70          # calc rental charge for selected rental period
71          if not want_insurance:
72              insurance_rate = 0
73
74          # calc base rental charges
75          base_rental_charges = rental_days * rental_rate + \
76                                rental_days * insurance_rate
77
78          miles_charged = miles_driving - num_free_miles
79
80          if miles_charged < 0:
81              miles_charged = 0
82
83          estimated_mileage_charges = miles_charged * per_mile_charge
84
85          return {'base_charges' : base_rental_charges,
86                  'insur_rate' : insurance_rate,
87                  'num_free_miles' : num_free_miles,
88                  'per_mile_charge' : per_mile_charge,
89                  'estimated_mileage_charges' : estimated_mileage_charges}
```

FIGURE 10-44 VehicleCosts Class

First, the VehicleCosts class is imported:
... from vehicleCosts import VehicleCosts
Then, instances of type VehicleCost are created:

... vc_1 5 VehicleCost('24.99', '180.00', '45.00', '100', '.15', '14.99') ... vc_2 5 VehicleCost('35.00', '220.00', '55.00', '0', '.20', '14.99') ... vc_3 5 VehicleCost('55.00', '425.00', '110.00', '25', '.25', '24.99')

Then a VehicleCosts instance is created, and each VehicleCost instance is added to the collection:

... vc.addVehicleCost('Car', vc_1)
... vc.addVehicleCost('Van', vc_2)
... vc.addVehicleCost('Truck', vc_3)

Then, each getter method is tested:

```
... vc.getVehicleCost('Car').getCosts()
['24.99', '180.00', '45.00', '100', '.15', '14.99']
... vc.getVehicleCost('Van').getCosts()
['35.00', '220.00', '55.00', '0', '.20', '14.99']
... vc.getVehicleCost(‘Truck').getCosts()
['55.00', '425.00', '110.00', '25', '.25', '24.99']
```

Finally, method calcRentalCost is tested:

```
... vc.calcRentalCost('Car', (1, 3), False, 150)
{'insur_rate': 0, 'base_charges': 74.97, 'num_free_miles': 100, 'per_mile_charge':
0.15, 'estimated_mileage_charges': 7.5}
```

In the call to calcRentalCost, recall that the first parameter is the vehicle type, the second is the rental period (1 for DAILY_RENTAL) and the number of days (3), the third indicates if the insurance is desired, and the fourth is the expected number of miles to be driven.

We see that we get the correct results. We next give the implementation of the Reservation and Reservations classes.

Implementation of the Reservation and Reservations Classes

The remaining aggregating class in the program is the Reservations class. We first give the Reservation class in Figure 10-45.

The Reservation class maintains the information for a given reservation. The __init__ method (**lines 6–14**) is provided a name, address, credit card number, and VIN when a new Reservation object is created, with a getter methods provided for each of these four values. We omit the testing of this class and next give the implementation of the corresponding Reservations class in Figure 10-46.

```
"""This module contains a Reservation class for storing details of a rental"""

class Reservation:
    """This class provides the methods for maintaining reservations info."""

    def __init__(self, name, address, credit_card, vin):
        """Initialzes a Reservation object with name, address, credit_card
           and vin.
        """

        self.__name = name
        self.__address = address
        self.__credit_card = credit_card
        self.__vin = vin

    def getName(self):
        """Returns first and last name for reservation."""

        return self.__name

    def getAddress(self):
        """Returns address for reservation."""

        return self.__address

    def getCreditCard(self):
        """Returns credit card on reservation."""

        return self.__credit_card

    def getVin(self):
        """Returns vehicle identification number on reservation."""

        return self.  vin
```

FIGURE 10-45 Reservation Class

```
1  """This module provides a Reservations class."""
2
3  from reservation import Reservation
4
5  class Reservations:
6      """This class provides the methods for maintaiing rental reservations."""
7
8      def __init__(self):
9          """Initializes empty collection of reservations."""
10
11         self.__reservations = dict()
12
13     def isReserved(self, vin):
14         """Returns True if reservation for vin, else returns False."""
15
16         return vin in self.__reservations
17
18     def getVinForReserv(self, credit_card):
19         """Returns vin of vehicles reserved with credit_card."""
20
21         return self.__reservations[credit_card].getVin()
22
23     def addReservation(self, resv):
24         """Adds new reservation."""
25
26         self.__reservations[resv.getCreditCard()] = resv
27
28     def findReservation(self, credit_card):
29         """Returns True if reservation for credit_card, else returns False."""
30
31         return credit_card in self.__reservations
32
33     def cancelReservation(self, credit_card):
34         """Deletes reservation matching provided credit card number."""
35
36         del(self.__reservations[credit_card])
```

FIGURE 10-46 Reservations Class

The Reservation class is imported on **line 3**. The __init__ method initializes an empty dictionary for storing the reservations, with credit card numbers serving as the key values. The remaining methods of the class include method isReserved (**lines 13–16**), getVinForReserv (**lines 18–21**), addReservation (**lines 23–26**), findReservation (**lines 28–31**), and CancelReservation (**lines 33–36**). We omit the testing of the Reservations class. We finally look at the implementation of the SystemInterface and RentalAgencyUI classes.

Implementation of the **SystemInterface** and **RentalAgencyUI** Classes An implementation of the SystemInterface class is given in Figure 10-47.

The docstring for the module specifies the formatting of the files storing the vehicle and vehicle rental cost information. There is one exception raised by the

module to inform the user interface when a file error has occurred. This could be due to either a file not being found, or an improperly formatted file.

The import statements on **lines 28–30** import most of the classes in the system. Symbolic constants VEHICLE_TYPES, VEHICLE_FILENAME, and VEHICLE_COSTS_FILENAME are defined (**lines 33–35**). Defining these constants makes the program more readable and modifiable. Exception class InvalidFileFormatError is defined (**lines 39–49**) and used within the SystemInterface class. Having this exception type allows typical I/O errors (such as a file not found error) to be reported as errors specific to the file formatting requirements of the program. The exception class defines special method __str__ to enable information to be displayed specific to the error (that is, which file header was expected and not found, in which file).

The __init__ method (**lines 55–77**) first creates instances of the three aggregator classes of the system—Vehicles, VehicleCosts, and Reservations. Each is initially empty when created. The rest of the __init__ method consists of a try block that attempts to open and read the files defined by VEHICLES_FILENAME and VEHICLE_COSTS_FILENAME. If opened successfully,

```
"""This module provides a SystemInterface class for the Vehicle Rental Program.

   Vehicles File Format
   --------------------
   The format for the vehicles file contains comma-separated values with the
   indicated header lines for cars, vans, and trucks.

     #CARS#
     make-model, mpg, num-passengers, vin
     .
     #VANS#
     make-model, mpg, num passengers, vin
     .
     #TRUCKS#
     mpg, length, num rooms, vin
     .


   Vehicle Cost File Format
   ------------------------
   The format for the rental costs file includes a header line, followed
   by three lines of comma-separated values for cars, vans and trucks.

     daily, weekly, weekend, free miles, mileage charge, insurance

   The following exceptions are raised: IOError.
"""
from vehicles import Vehicles, Car, Van, Truck
from vehicleCosts import VehicleCost, VehicleCosts
from reservations import Reservations

# symbolic constants
VEHICLE_TYPES = ('Car', 'Van', 'Truck')
VEHICLES_FILENAME = 'VehiclesStock.txt'
VEHICLE_COSTS_FILENAME = 'RentalCost.txt'

# exception class
class InvalidFileFormatError(Exception):
    """Exception indicating invalid file in file_name."""

    def __init__(self, header, file_name):
        self.__header = header
        self.__file_name = file_name

    def __str__(self):

        return 'FILE FORMAT ERROR: File header ' + self.__header + \
               ' expected in file ' + self.__file_name

```

FIGURE 10-47 SystemInterface Class (*Continued*)

```
51  class SystemInterface:
52      """This class provides the system interface of the vehicle rental
53         system.
54      """
55      def __init__(self):
56          """Populates vehicles and rental costs from file. Raises IOError."""
57
58          self.__vehicles = Vehicles()
59          self.__vehicle_costs = VehicleCosts()
60          self.__reservations = Reservations()
61          self.__vehicle_info_file = None
62
63          try:
64              self.__vehicle_info_file = open(VEHICLES_FILENAME, 'r')
65              self.__rental_cost_file = open(VEHICLE_COSTS_FILENAME, 'r')
66
67              self.__populateVehicles(self.__vehicle_info_file)
68              self.__populateCosts(self.__rental_cost_file)
69          except InvalidFileFormatError as e:
70              print(e)
71              raise IOError
72          except IOError:
73              if self.__vehicle_info_file == None:
74                  print('FILE NOT FOUND:', VEHICLES_FILENAME)
75              else:
76                  print('FILE NOT FOUND:', VEHICLE_COSTS_FILENAME)
77              raise IOError
78
79      def numAvailVehicles(self, vehicle_type):
80          """Returns the number of available vehiles. Returns 0 if no
81             no vehicles available.
82          """
83
84          return self.__vehicles.numAvailVehicles(vehicle_type)
85
86      def getVehicle(self, vin):
87          """Returns Vehicle type for given vin."""
88
89          return self.__vehicles.getVehicle(vin)
90
91      def getVehicleTypes(self):
92          """Returns all vehicle types as a tuple of strings."""
93
94          return VEHICLE_TYPES
95
96      def getVehicleCosts(self, vehicle_type):
97          """Returns vehicle costs for provided vehicle type as a list.
98
99             List of form [daily rate, weekly rate, weekend rate,
100                           num free miles, per mile charge, insur rate]
101         """
102
103         return self.__vehicle_costs.getVehicleCost(vehicle_type).getCosts()
104
```

FIGURE 10-47 SystemInterface Class (*Continued*)

then each is read to populate the corresponding object. The exception handler catches two types of exceptions, InvalidFileFormatError and IOERROR, as mentioned.

A set of getter methods is provided (**lines 86–103**) for retrieving vehicle, vehicle

type, and vehicle cost information. Method getAvailVehicles returns the list of vehicles that are not currently reserved. (Note that the list returned is constructed by the use of a list comprehension.) The list of unreserved vehicles is of a specified type. Four additional methods are provided (**lines 113–132**) for maintaining reservation information (isReserved, findReservation,

```
107     def getAvailVehicles(self, vehicle_type):
108         """Returns a list of descriptions of unreserved vehicles."""
109
110         avail_vehicles = self.__vehicles.getAvailVehicles(vehicle_type)
111         return [veh for veh in avail_vehicles]
112
113     def isReserved(self, vin):
114
115         return self.__reservations.isReserved(vin)
116
117     def findReservation(self, credit_card):
118
119         return self.__reservations.findReservation(credit_card)
120
121     def addReservation(self, resv):
122         """Creates reservation and marks vehicles as reserved."""
123
124         self.__reservations.addReservation(resv)
125
126     def cancelReservation(self, credit_card):
127         """Cancels reservation made with provided credit card."""
128
129         vin = self.__reservations.getVinForReserv(credit_card)
130
131         self.__vehicles.unreserveVehicle(vin)
132         self.__reservations.cancelReservation(credit_card)
133
134     def calcRentalCost(self, vehicle_type, rental_period,
135                        want_insurance, miles_driving):
136         """Returns estimate of rental cost for provided parameter values.
137
138            Returns dictionary with key values: {'base_charges', 'insur_rate',
139            'num_free_miles', 'per_mile_charge', 'estimated_mileage_charges'}
140         """
141
142         return self.__vehicle_costs.calcRentalCost(vehicle_type, rental_period,
143                                                    want_insurance, miles_driving)
144
145     # ---- Private Methods
146
147     def __populateVehicles(self, vehicle_file):
148         """Gets vehicles from vehicle_file. Raises InvalidFileFormatError."""
149
150         empty_str = ''
151
152         # init vehicle string file headers
153         vehicle_file_headers = ('#CARS#', '#VANS#', '#TRUCKS#')
154         vehicle_type_index = 0
155
156         # read first line of file (#CARS# expected)
157         vehicle_str = vehicle_file.readline()
158         vehicle_info = vehicle_str.rstrip().split(',')
159         file_header_found = vehicle_info[0]
160         expected_header = vehicle_file_headers[0]
161
```

FIGURE 10-47 SystemInterface Class (*Continued*)

addReservation , and cancelReservation). Finally, method calcRentalCost (**lines 134–143**) returns the calculated rental charges for a given vehicle type, rental period, insurance option, and expected miles driven.

The remainder of the class consists of private supporting methods. Private method populateVehicles reads the information given by VEHICLES_FILENAME and populates the Vehicles instance. Private method populateCosts reads the information given by VEHICLE_COSTS_FILENAME and populates the VehicleCosts instance.

```
        if file_header_found != expected_header:
            raise InvalidFileFormatError(expected_header, VEHICLES_FILENAME)
        else:
            # read next line of file after #CARS# header line
            vehicle_str = vehicle_file.readline()

            while vehicle_str != empty_str:

                # convert comma-separated string into list of strings
                vehicle_info = vehicle_str.rstrip().split(',')

                if vehicle_info[0][0] == '#':
                    vehicle_type_index = vehicle_type_index + 1
                    file_header_found = vehicle_info[0]
                    expected_header = vehicle_file_headers[vehicle_type_index]

                    if file_header_found !=  expected_header:
                        raise InvalidFileFormatError(expected_header,
                                                     VEHICLES_FILENAME)
                else:

                    # create new vehicle object of the proper type
                    if file_header_found == '#CARS#':
                        vehicle = Car(*vehicle_info)
                    elif file_header_found == '#VANS#':
                        vehicle = Van(*vehicle_info)
                    elif file_header_found == '#TRUCKS#':
                        vehicle = Truck(*vehicle_info)

                    # add new vehicle to vehicles list
                    self.__vehicles.addVehicle(vehicle)

                # read next line of vehicle information
                vehicle_str = vehicle_file.readline()

def __populateCosts(self, cost_file):
    """Populates RentalCost objects from provided file object."""

    # skip file header / read first line of file
    cost_file.readline()
    cost_str = cost_file.readline()

    for veh_type in VEHICLE_TYPES:
        # strip off newline (last) character and split into list
        cost_info = cost_str.rstrip().split(',')

        for cost_item in cost_info:
            cost_item = cost_item.strip()

        # add Vehicle Type/Rental Cost key/value to dictionary
        self.__vehicle_costs.addVehicleCost(veh_type,
                                            VehicleCost(*cost_info))

        # read next line of vehicle costs
        cost_str = cost_file.readline()
```

FIGURE 10-47 SystemInterface Class

We next look at the implementation of the final class, the RentalAgencyUI, given in Figure 10-48. Any user interface for the rental agency system must

access it through the system interface.
Thus, the __init__ method (**lines 14–16**) of the RentalAgencyUI class is passed a reference
to the system interface to store (in a private variable) providing it access to the system. The only

```
"""
    This module provides class RentalAgencyUI, a console user interface.

    Method start begins execution of the interface. Raises IOError exception.
"""
from systemInterface import VEHICLE_TYPES
from vehicles import InvalidVinError
from vehicleCosts import DAILY_RENTAL, WEEKLY_RENTAL, WEEKEND_RENTAL
from reservations import Reservation

class RentalAgencyUI:
    """This class provides a console interface for the rental agency system."""

    def __init__(self, sys):
        """Stores the provided reference to the vehicle rental system."""
        self.__sys = sys

    def start(self):
        """Begins the command loop."""

        self.__displayWelcomeScreen()
        self.__displayMenu()

        selection = self.__getSelection(7)

        while selection != 7:
            self.__executeCmd(selection)
            self.__displayMenu()
            selection = self.__getSelection(7)

        print('Thank you for using the Friendly Rental Agency')


    # Private Methods

    def __displayWelcomeScreen(self):
        """PRIVATE: Displays welcome message and general instructions."""

        print('*****************************************************')
        print('  * Welcome to the Friendly Vehicle Rental Agency *')
        print('*****************************************************')

    def __displayMenu(self):
        """PRIVATE: Displays a list of menu options on the screen."""

        print('\n<<< MAIN MENU >>>')
        print('1 - Display vehicle types')
        print('2 - Check rental costs')
        print('3 - Check available vehicles')
        print('4 - Get cost of specific rental')
        print('5 - Make a reservation')
        print('6 - Cancel a reservation')
        print('7 - Quit\n')

```

FIGURE 10-48 RentalAgencyUI Class (*Continued*)

other public method of the class is method start (**lines 18–31**). This method begins the command loop when called—that is, the repeated action of displaying the main menu, getting the user's selection, and executing the selected

command. The loop continues until a value of 7 (to quit the program) is entered. The rest of the methods of the class are private methods in support of the execution of commands in the start method.

```
55      def __getSelection(self, num_selections, prompt='Enter: '):
56          """PRIVATE: Returns user-entered value in range 1-num_selections."""
57
58          valid_input = False
59
60          selection = input(prompt)
61
62          while not valid_input:
63              try:
64                  selection = int(selection)
65
66                  if selection < 1 or selection > num_selections:
67                      print('* Invalid Entry*\n')
68                      selection = input(prompt)
69                  else:
70                      valid_input = True
71              except ValueError:
72                  selection = input(prompt)
73
74          return selection
75
76      def __displayDivLine(self, title = ''):
77          """PRIVATE: Displays line of dashes, with optional title."""
78
79          if len(title) != 0:
80              title = ' ' + title + ' '
81          print(title.center(70,'-'))
82
83      def __executeCmd(self, selection):
84          """PRIVATE: Executes command for provided menu selection."""
85
86          if selection == 1:
87              self.__CMD_DisplayVehicleTypes()
88          elif selection == 2:
89              self.__CMD_DisplayVehicleCosts()
90          elif selection == 3:
91              self.__CMD_PromptAndDisplayAvailVehicles()
92          elif selection == 4:
93              self.__CMD_DisplaySpecificRentalCost()
94          elif selection == 5:
95              self.__CMD_MakeReservation()
96          elif selection == 6:
97              self.__CMD_CancelReservation()
98
99      def __CMD_DisplayVehicleTypes(self):
100         """PRIVATE: Displays vehicle types (Cars, Vans, Trucks)."""
101
102         self.__displayDivLine('Types of Vehicles Available for Rent')
103         self.__displayVehicleTypes()
104         self.__displayDivLine()
105
```

FIGURE 10-48 RentalAgencyUI Class (*Continued*)

The methods provided by the system interface correspond to the user options given in the main menu in the user interface (as shown in Figure 10-49).

Pri vate method displayWelcomeScreen (**lines 36–41**) provides the welcome message of the program. Private method displayMenu (**lines 43–53**) is repeatedly called to redisplay the users' options before each next command.

Private method getSelection is called whenever the user is to enter a number within a provided range, such as in the main menu. Such selection occurs in other places in the program as well. Therefore, the method is designed to be passed an argument indicating the upper limit of the range of selections. (For example, for the selection of

```
106     def __CMD_DisplayVehicleCosts(self):
107         """PRIVATE: Displays rental costs for Cars, Vans, Trucks."""
108
109         empty_str = ''
110        blank_char = ' '
111
112        # get vehicle type from user
113        self.__displayVehicleTypes()
114        vehicle_type_num = self.__getSelection(len(VEHICLE_TYPES))
115        vehicle_type = VEHICLE_TYPES[vehicle_type_num - 1]
116
117        # set column headings
118        row1_colheadings = blank_char * 24 + format('Free', '7') + \
119                  format('Per Mile', '10') + format('Insurance', '17')
120
121        row2_colheadings = format('Daily', '7') + format('Weekly', '8') + \
122                  format('Weekend', '9') + format('Miles', '7') + \
123                  format('Charge', '10') + format('(per day)', '17')
124
125        # get vehicle costs
126        costs = self.__sys.getVehicleCosts(vehicle_type)
127
128        # build costs line to display
129        costs_str = empty_str
130
131        field_widths = ('7', '8', '9', '7', '10', '17')
132        for k in range(0, len(costs)):
133            costs_str = costs_str + format(costs[k],'<' + field_widths[k])
134
135        # display headings and costs line
136        self.__displayDivLine('Rental Charges for ' + vehicle_type + 's')
137        print()
138        print(row1_colheadings + '\n' + row2_colheadings + '\n' + costs_str)
139        self.__displayDivLine()
140
141    def __CMD_DisplayAvailVehicles(self, vehicle_type, numbered=False):
142        """PRIVATE: Displays unreserved vehicles of selected type.
143            When default argument is True, lists vehicles with sequential
144            numbers at the start of the line of each vehicle description.
145        """
146
147        avail_vehicles_list = self.__sys.getAvailVehicles(vehicle_type)
148        self.__displayDivLine('Available ' + vehicle_type + 's')
149
150        if avail_vehicles_list == []:
151            print('* No Vehicles Available of this Type *')
152        elif numbered:
153            k = 1
154            for veh in avail_vehicles_list:
155                print(str(k) + '-' + veh.getDescription())
156                k = k + 1
157        else:
158            for veh in avail_vehicles_list:
159                print(veh.getDescription())
160
```

FIGURE 10-48 RentalAgencyUI Class (*Continued*)

items from the main menu, getSelection would be passed the value 7.) Two types of errors are checked by the method. One is if a value is outside of the allowable

range. The other checks for invalid type of input (such as entering a letter instead of a number). The method requires that an appropriate value be entered by the user before returning.

```
161     def __CMD_PromptAndDisplayAvailVehicles(self):
162        """PRIVATE: Prompts user for vehicle type, and displays all
163            available vehicles of that type.
164        """
165
166        self.__displayVehicleTypes()
167        vehicle_type_num = self.__getSelection(len(VEHICLE_TYPES))
168        vehicle_type = VEHICLE_TYPES[vehicle_type_num - 1]
169        self.__CMD_DisplayAvailVehicles(vehicle_type)
170
171     def __CMD_DisplaySpecificRentalCost(self):
172        """PRIVATE: Prompts user for selections and displays rental cost."""
173
174        # get vehicle type and rental period from user
175        self.__displayVehicleTypes()
176        vehicle_type_num = self.__getSelection(len(VEHICLE_TYPES))
177        vehicle_type = VEHICLE_TYPES[vehicle_type_num - 1]
178
179        # assign tuple, e.g. ('DAILY_RENTAL', '4')
180        rental_period = self.__getRentalPeriod()
181
182        # prompt user for optional insurance
183        want_insurance = input('Would you like the insurance? (y/n): ')
184
185        while want_insurance not in ('y', 'Y', 'n', 'N'):
186            want_insurance = input('Would you like the insurance? (y/n): ')
187
188        # convert to Boolean value
189        want_insurance = want_insurance in ('y', 'Y')
190
191        # prompt user for num of miles expected to drive
192        print('\nNumber of miles expect to drive?')
193        num_miles = self.__getSelection(1000, prompt = 'Miles: ')
194
195        # calc base rental cost
196        rental_cost = self.__sys.calcRentalCost(vehicle_type, rental_period,
197                                                want_insurance, num_miles)
198        # display estimated rental cost
199        print()
200        self.__displayDivLine('ESTIMATED ' + vehicle_type + ' RENTAL COST')
201
202        if want_insurance:
203            print('Insurance rate of', rental_cost['insur_rate'], 'per day')
204        else:
205            print('* You have opted out of insurance coverage *')
206
207        if rental_period[0] == DAILY_RENTAL:
208            print('\nDaily rental for', rental_period[1],
209                  'days would be $', format(rental_cost['base_charges'], '.2f'))
210        elif rental_period[0] == WEEKLY_RENTAL:
211            print('\nWeekly rental for', rental_period[1],
212                  'weeks would be $', format(rental_cost['base_charges'], '.2f'))
213        elif rental_period[0] == WEEKEND_RENTAL:
214            print('\nWeekend rental is $', rental_cost['base_charges'])
215
216        est_total_cost = rental_cost['base_charges'] + \
217                         rental_cost['estimated_mileage_charges']
218
219        print('\nYour cost with an estimated mileage of',
220              num_miles, 'miles would be', format(est_total_cost, '.2f'))
221
222        print('which includes', rental_cost['num_free_miles'],
223              'free miles and a charge of',
224              format(rental_cost['per_mile_charge'], '.2f' ), 'per mile')
225
226        self.__displayDivLine()
```

FIGURE 10-48 RentalAgencyUI Class (*Continued*)

```
227 def __CMD_MakeReservation(self):
228     """PRIVATE: Prompts user for vehicle vin and reserves vehicle."""
229
230     # get vehicle type
231     self.__displayVehicleTypes()
232     vehicle_type_num = self.__getSelection(len(VEHICLE_TYPES))
233     vehicle_type = VEHICLE_TYPES[vehicle_type_num - 1]
234
235     if self.__sys.numAvailVehicles(vehicle_type) == 0:
236         print('Sorry - No available', \
237               VEHICLE_TYPES[vehicle_type_num - 1] + 's', 'at the moment')
238     else:
239         self.__CMD_DisplayAvailVehicles(vehicle_type, numbered=True)
240         avail_vehicles = self.__sys.getAvailVehicles(vehicle_type)
241
242         valid_input = False
243
244         while not valid_input:
245             selected = input('\nEnter number of vehicle to reserve: ')
246
247             if not selected.isdigit():
248                 print('Please enter the number preceeding the vehicle')
249             elif int(selected) < 1 or \
250                   int(selected) > self.__sys.numAvailVehicles(vehicle_type):
251                 print('\nINVALID SELECTION - Please re-enter: ')
252             else:
253                 valid_input = True
254
255         vin = avail_vehicles[int(selected) - 1].getVin()
256         vehicle = self.__sys.getVehicle(vin)
257         print(vehicle.getDescription())
258
259         vehicle.setReserved(True)
260
261         name = input('\nEnter first and last name: ')
262         addr = input('Enter address: ')
263         credit_card = input('Enter credit card number: ')
264
265         reserv = Reservation(name, addr, credit_card, vin)
266         self.__sys.addReservation(reserv)
267         print('* Reservation Made *')
268
269 def __CMD_CancelReservation(self):
270     """PRIVATE: Prompts user for credit card and cancels reservation."""
271
272     credit_card = input('Please enter your credit card number: ')
273
274     if self.__sys.findReservation(credit_card):
275         print('Calling cancelReservation of sys')  # **************
276         self.__sys.cancelReservation(credit_card)
277         print('** RESERVATION CANCELLED **')
278     else:
279         print('* Reservation not Found - Invalid Credit Card Entered *')
280
```

FIGURE 10-48 RentalAgencyUI Class (*Continued*)

Method displayDivLine is called to display a row of dashes. It has a default parameter title. Thus, the method may be called with or without a supplied argument. If an argument is not supplied, then a complete row of dashes is displayed. If a title is displayed, it is centered within the displayed line of dashes as given below:

----------Types of Vehicles Available for Rent----------

```
281  def __displayVehicleTypes(self):
282     """PRIVATE: Displays the numbered selection of vehicle types."""
283
284     print('\nEnter type of vehicle')
285     vehicle_types = self.__sys.getVehicleTypes()
286
287     k = 1
288
289     for veh_type in vehicle_types:
290         print(k, '-', veh_type)
291         k = k + 1
292
293     print()
294
295  def __getRentalPeriod(self):
296     """PRIVATE: Prompts user for desired rental period.
297
298        Returns tuple of the form (x, n), where x is one of DAILY_RENTAL,
299        WEEKLY_RENTAL or WEEKEND_RENTAL, and n is the number of those units,
300        e.g. (DAILY_RENTAL, 3) three days of a daily rental.
301     """
302     print('\nEnter the rental period:')
303     print(DAILY_RENTAL, '- Daily ', WEEKLY_RENTAL, '- Weekly ',
304           WEEKEND_RENTAL, '- Weekend\n')
305
306     valid_input = False
307
308     while not valid_input:
309         try:
310             selection = int(input('Enter: '))
311
312             while selection not in (DAILY_RENTAL, WEEKLY_RENTAL,
313                                     WEEKEND_RENTAL):
314
315                 print('* Incorrect entry. Please Re-enter *\n')
316                 selection = int(input('Enter: '))
317
318             if selection == DAILY_RENTAL:
319
320                 print('How many days do you need the vehicle?', end = '')
321                    num_days = self.__getSelection(14, prompt = ' ')
322
323                 if num_days > 6:
324                     print("Why don't you consider a weekly rental?\n")
325
326                 return (DAILY_RENTAL, num_days)
327             elif selection == WEEKLY_RENTAL:
328                 print('How many weeks do you need the vehicle? (1-3): ')
329                 num_weeks = self.__getSelection(3, prompt = 'Weeks: ')
330
331                 return (selection, num_weeks)
332
333             elif selection == WEEKEND_RENTAL:
334                 return (selection, 1)
335
336         except ValueError:
337             print('* Invalid Input - Number Expected *\n')
```

FIGURE 10-48 RentalAgencyUI Class

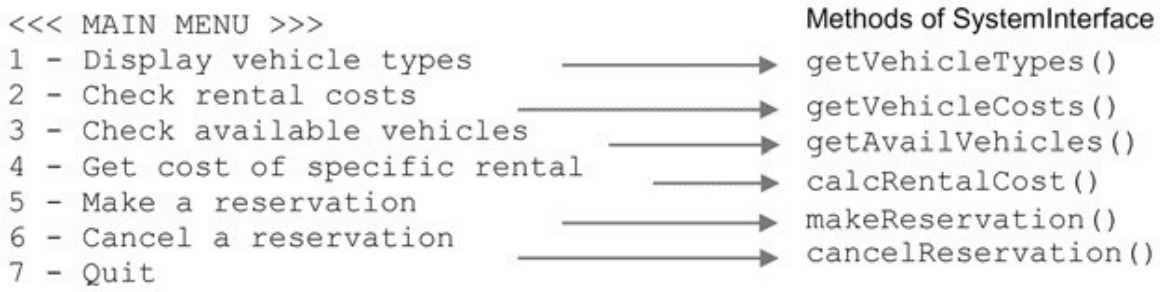


FIGURE 10-49 Correspondence of Menu Selections and Methods of System Interface

Method executeCmd (**lines 83–97**) is called to execute each command of the main menu by selection number. Rather than place the code for each command within this method, a specific command method is called for each command. Private method CMD_DisplayVehicleTypes (**lines 99– 104**) displays the three types of vehicles (Cars, Vans, and Trucks). It in turn makes use of private method displayVehicleTypes (**lines 281–293**). Within this method, getVehicleTypes of the systemInterface is called to retrieve the types, since that is where they are defined.

Private method CMD_DisplayVehicleCosts (**lines 106–139**) displays the rental costs for a specific vehicle type:

```
Enter vehicle type: 1

------------ Following are the Rental Charges for Cars -------------

                        Free   Per Mile  Daily
Daily  Weekly  Weekend  Miles  Charge    Insurance
24.99  180.00  45.00    100    .15       14.99
--------------------------------------------------------------------
```

It displays a selection of vehicle types (**line 113**) for the user to choose from, constructs two rows of column headings (**lines 118–123**), and gets the vehicle costs for the selected vehicle type by call to method getVehicleCosts of the system interface (**line 126**). The rest of the method displays the information using formatted strings to align under the column headings.

Private method CMD_DisplayAvailVehicles displays all vehicles of a given vehicle type that are not currently reserved:

```
Enter vehicle type: 1
------------------------- Available Cars --------------------------
Chevrolet Camaro     passengers: 4   doors: 2    mpg: 30   vin: WG8JM5492DY
Chevrolet Camaro     passengers: 4   doors: 2    mpg: 30   vin: KH4GM4564GD
Ford Fusion          passengers: 5   doors: 4    mpg: 34   vin: AB4FG5689GM
Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: GH2KL4278TK
Ford Fusion Hybrid   passengers: 5   doors: 4    mpg: 36   vin: KU4EG3245RW
Chevrolet Impala     passengers: 6   doors: 4    mpg: 30   vin: QD4PK7394J1
Chevrolet Impala     passengers: 6   doors: 4    mpg: 30   vin: RK3BM4256YH
```

It displays a selection of vehicle types (**lines 158–159**) for the user to select from, and gets the available vehicles for the selected type from the system interface by call to method getAvailVehicles of the system interface (**line 147**). The rest of the method simply displays, row by row, the information for each vehicle.

Pri vate method CMD__PromptAndDisplayAvailVehicles (**lines 161–169**) prompts the user for a vehicle type, and then displays the information of vehicles of that type not currently reserved. Private method CMD_DisplaySpecificRentalCost (**lines 171–226**) displays the costs of a particular vehicle rental based on the vehicle type, the time period rented, whether insurance is opted for, and the estimated number of miles driven:

```
--------------------- ESTIMATED Car RENTAL COST ----------------------
* You have opted out of the daily insurance *

Your cost of a daily rental would be 74.97

Your cost with an estimated mileage of 240 would be 95.97
which includes 100 free miles and a charge of 0.15 per mile
----------------------------------------------------------------------
```

First, it displays a selection of vehicle types for the user to choose from (**line 176**). It then requests the desired rental period by a call to private method getRentalPeriod (**line 180**). A tuple is returned containing constant value DAILY_RENTAL, WEEKLY_RENTAL, or WEEKEND_RENTAL, and the number of days or weeks rental period (with weekends defaulting to 1). The user is then asked if they want the optional insurance (**line 186**) and the number of miles expected to drive (**line 193**). The estimated rental cost is then calculated by a call to method calcRentalCost of the system interface. The rest of the method has to do with formatting and displaying the output.

Pri vate method makeReservation allows the user to reserve a vehicle of a given type (**line 227**). The method first checks if there are vehicles available (**line**

235). It then calls CMD__DisplayAvailVehicles (**line 239**) with the default argument set to True. This causes the vehicle descriptions to be listed as a number list so that the desired vehicle can be selected. On **lines 242–253**, the vehicle number selected is read, checking for any invalid selections. On **line 255**, the VIN is retrieved for the selected vehicle. The VIN is needed by method getVehicle (of the SystemInterface class) to retrieve a given Vehicle object. Having the object, it then calls method getDescription (**line 257**) to display the description of the vehicle to the user. It also sets the reservation status to True by a call to setReserved (**line 259**).

The user's name, address, and credit card number are requested (**lines 261–263**) to make the reservation. It does this by creating a new Reservation object constructed with the provided information (**line 265**) and adds it to the collection of reservations (**line 266**). Confirmation of the reservation is displayed (**line 267**). Private method CMD_CancelReservation (**line 269–279**) cancels reservations by the credit card number used for making the reservation. Finally, private methods displayVehicleTypes and getRentalPeriod, supporting methods of the command methods, are given on **lines 281–337**.

Putting it All Together

Figure 10-50 shows the main module of the Vehicle Rental Agency Program that puts everything together. It first creates the vehicle rental agency system (with a system interface) and then creates a user interface containing a reference to the system, providing the user with a natural means of accessing the system.

```
1  # Rental Agency Program
2
3  """ This program performs the tasks of a vehicle rental agency, including the
4      display of vehicle information and the ability to make/cancel reservations.
5  """
6  from systemInterface import SystemInterface
7  from rentalAgencyUI import RentalAgencyUI
8
9  try:
10     # create system with populated data from file
11     sys = SystemInterface()
12
13     # associate user interface with system
14     ui = RentalAgencyUI(sys)
15
16     # start the user interface
17     ui.start()
18
19 except IOError:
20     print('** PROGRAM TERMINATION (IO Error) **')
```

FIGURE 10-50 Rental Agency Program Main Module

The main module for the program imports the SystemInterface class (**line 6**) and the RentalAgencyUI class (**line 7**). First, an instance of the SytemInterface class is created (**line 11**). The __init__ method of the class is implemented to open the vehicle information file, as well as the vehicle cost file. The information in these files is used to populate the vehicle and vehicle cost objects in the system. If any file errors occur during this process, an IOError exception is raised by the system interface and caught (**line 19**) to terminate the program.

Once the files are successfully opened and read, an instance of the RentalAgencyUI class is created (**line 14**). It is passed the needed reference to the system interface. Once the user interface is created, the start method of the user interface is called to begin the command loop and receive and execute commands from the user.

CHAPTER SUMMARY General Topics Python-Specific Programming Topics

Classes
Encapsulation, Inheritance, and Polymorphism Public vs. Private Class Members
Getters and Setters
Name Mangling
Superclass (Base Class/Parent Class)
Subclass (Derived Class/Child Class)
Subclass vs. Subtype

UML (Unified Modeling Language)
Class Diagrams in UML
Association, Multiplicity, and Role Names in UML Composition vs. Aggregation
Defining Classes in Python
Denoting Public and Private Class Members in

Python
The Use of self in Python
Special Method __init__ in Python Arithmetic and Relational Special Methods in

Python
Inheritance and Polymorphism in Python Duck Typing in Python

CHAPTER EXERCISES
Section 10.1
1. What are the two kinds of entities “bundled” in a class?
2. What kind of entity can there be any number of instances created for a given class? **3.** What are the three fundamental features that object-oriented programming languages have in support of object-oriented programming?

Section 10.2

4. What else does encapsulation provide other than the ability to bundle together instance variables and methods?
5. Describe what it means for a member of a class to be private.
6. Explain the purpose of getters and setters.
7. Explain what the special identifier self is used for in Python.
8. Explain the use of name mangling in Python.
9. Explain when special methods __str__ and __repr__ are each used in Python.
10. Give an implementation of special method __str__ for a Range class (representing a range of integers) that contains integer instance variables, __start and __end, so that the value of Range objects are displayed as follows: '10 . . . 16', when output with print.
11. Give an implementation of special method __lt__ for the Range class of exercise 10 so that range1 , range2 evaluates to True if all the values in range1 are less than all the values in range2, and returns False otherwise.

Section 10.3
12. Give the one-line class definition header for a class named MySubclass that is a subclass of the class

MySuperclass .
13. Explain when a subclass can serve as a subtype.
14. For an object obj, show how in Python the type of the object may be determined from within a program or the Python shell.
15. Show how in the Python shell information about one of the built-in types of Python can be displayed.

Section 10.4
16. Explain the concept of polymorphism in object-oriented programming.
17. Explain the advantages of using polymorphism in program design.
18. What is meant by "duck typing" in Python?
Section 10.5
19. What does the name UML stand for?

20. What are the two types of diagrams in UML mentioned in the chapter, and what aspects of a program design does each represent?
21. Give a class diagram for the XYCoord class in the Let's Try It box of section 10.2.2.
22. Give a class diagram that includes the partial description of the built-in str type in Figure 10-15, and the ExplodedStr subclass in Figure 10-16.
23. Give a class diagram for the Fraction class developed in section 10.2. Include the MixedFraction subclass in the diagram.

Python Programming Exercises 455
PYTHON PROGRAMMING EXERCISES

P1. Give a UML class diagram for a library. Include as entities tangible objects (such as books), persons (such as borrowers and librarians), and status (such as whether a book is checked out or not). Use multiplicity, navigation, and role names where appropriate.

P2. Design and implement a Money class that stores monetary values in dollars and cents. Special method __init__ should have the following function header, def __init__(self, dollars, cents)

Include special method __repr__ (__str__) for displaying values in dollars and

cents: $ 0.45, $ 1.00, $ 1.25. Also include special method __add__, and three getter methods that each provide the monetary value in another currency. Choose any three currencies to convert to.

P3. Implement a class named AvgList as a subclass of the built-in list class in Python, able to compute the average of a list of numeric values. If the list contains any nonnumeric types, a ValueError exception should be raised.

P4. Design and implement a FootMeasure class that stores a linear measurement of feet and inches. Your class should have the following function header for special method __init__,
def __init__(self, feet 50, inches 50)
Thus, the class should be able to create a FootMeasure object in various ways by use of optional keyword arguments,

meas 5 FootMeasure()
meas 5 FootMeasure(feet 5 5)
meas 5 FootMeasure(feet 5 5, inches 58) meas 5 FootMeasure(inches 5 68)

Implement special method __repr__ in the class so that measurements are displayed as follows,
5 ft. NOT 5 ft. 0 in. 5 ft. 8 in. NOT 68 in.

When the measurement is 0, it should be displayed as, 0 ft. 0. ins. Include special method add() for adding FootMeasure values. Also include all the special methods for implementing the relational operators.

P5. Develop an abstract class named Temperature that stores a single temperature. The class should have the following function header for special method __init__,
def __init__(self, temperature)
The abstract class should contain the following methods:
__str__ — should return a string of the form “75 degrees Fahrenheit”

aboveFreezing() — returns True if temperature above the freezing point convertToFahren — returns a new Temperature object converted to degrees Fahrenheit convertToCelsius — returns a new Temperature object converted to degrees Celsius convertToKelvin — returns a new Temperature object converted to degrees Kelvin Develop the subclasses Fahrenheit, Celsius and Kelvin to appropriately implement each of the methods in the abstract Temperature class.

(Note that when a meaningless conversion method is applied, for example, temp1.convertToFahrenheit() where temp1 is an object of type Fahrenheit, then a copy of the Temperature object should be returned.)

Demonstrate the correctness of your classes by doing the following:
Temperature objects of a mix of Temperature types ♦ Create a list of

♦ Print out the value of each temperature in the list, and add “above freezing” if the temperature is above freezing (for the specific temperature scale).
♦ Create a new list of temperatures containing each temperature of the original list converted to a common temperature scale (Fahrenheit, Celsius, or Kelvin).
♦ For each temperature object in the new list, print out its temperature value, and if it is above the freezing point.

PROGRAM MODIFICATION PROBLEMS

M1. Fraction Class—Adding a Supporting Reduce Method
Complete the Fraction class in section 10.2.2 by implementing the missing methods using the specifications below,

```
def copy(self):
"""Creates a copy of a given Fraction."""

def reduce(self):
"""Reduces self to simplest terms. Also removes the signs if both numerator and
denominator are negative."""

def __adjust(self, factor):
"""Multiplies numerator and denominator by factor."""
```

M2. Vehicle Rental Agency Program—View Reservations
Modify the Vehicle Rental Agency program in section 10.6 to add the capability of viewing all current reservations.

M3. Division Operator for Fraction Class
Modify the Fraction class developed in section 10.2.2 to include a division operator by implementing special method __truediv__.

M4. Screen Display in Recipe Conversion Program
Modify the Recipe Conversion Program in section 10.2.3 so that the original

recipe ingredient measurements and the converted measurements are both displayed on the screen (in addition to having the converted recipe written to a file).

M5. Adjusted Measurements in the Recipe Conversion Program
Modify the Recipe Conversion Program in section 10.2.3 so that the units of measure for teaspoons and tablespoons in a converted recipe are displayed more appropriately. For example, if a recipe that calls for 2 teaspoons of baking soda is tripled, 6 teaspoons of baking soda are currently displayed in the converted recipe. Since there are 3 teaspoons to a tablespoon, the converted measurement would be more appropriately displayed as 2 tablespoons. Similarly, since there are 16 tablespoons in a cup, any number of tablespoons over 16 would be more appropriately displayed as one cup, plus some number of tablespoons.

M6. Revised Estimated Rental Cost in Vehicle Rental Agency Program
The rental cost for a particular type vehicle computed in the Vehicle Rental Agency Program is based on the number of days (or weeks) rented, the cost of optional insurance, and the estimated number of miles that the vehicle is expected to be driven. However, the estimate does not include the cost of gas that the customer must also pay. Since the miles per gallon (mpg) is available for all vehicles, modify the program so that this additional expense is added to the calculation of the estimated cost. The cost that is displayed should identify this additional expense. Use an instance variable in the VehicleCosts class to store the current price of gas per gallon.

M7. Additional Vehicle Type in Vehicle Rental Agency Program
Modify the Vehicle Rental Agency Program to include a fourth SUV vehicle type, in addition to cars, vans, and trucks. The attributes stored for an SUV object should include make and model, miles per gallon (mpg), number of passengers, whether or not it has automatic sliding doors, and the vehicle identification number (VIN). Make *all* changes in the program to incorporate this new vehicle type.

M8. Polymorphic Behavior of Vehicle Cost in Vehicle Rental Agency Program
Modify the Vehicle Rental Agency Program to include three subclasses of the VehicleCost class, CarCost, VanCost, and TruckCost so that VehicleCost can behave polymorphically. Make all changes to the program to incorporate these new vehicle cost types.

PROGRAM DEVELOPMENT PROBLEMS

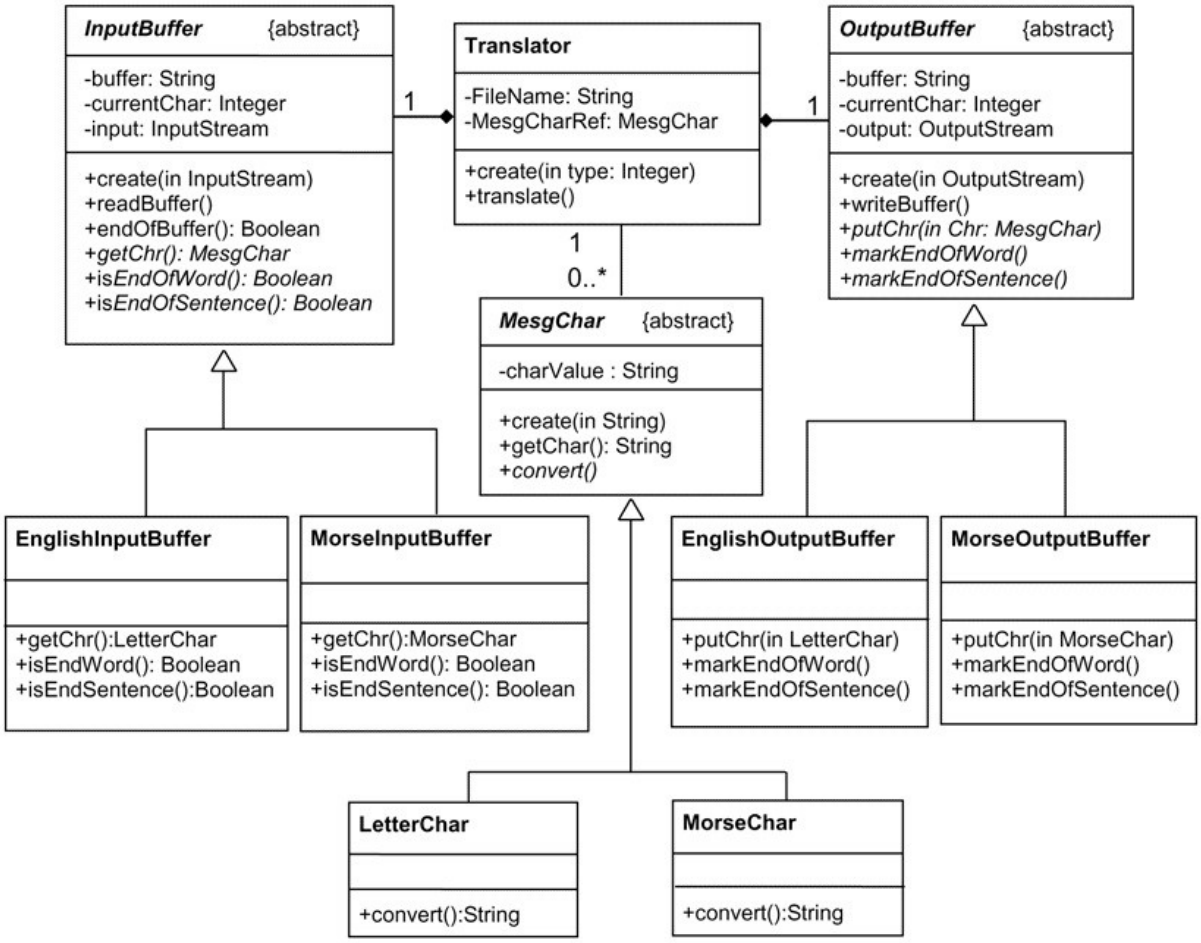
D1. RGB/Hexadecimal Color Code Manipulation Program
Design an abstract class named ColorCode, and two subclass named RGBColorCode and HexColorCode. RGB color codes (as discussed in Chapter 9) are of the form (125, 80, 210), indicating the amount of red, green, and blue, respectively, for a given color. Each color value is on a scale of 0–255. Hexadecimal color codes are of the form F4F060, in which the first two hexadecimal digits indicate the amount of red, the middle two digits the amount of green, and the last two digits the amount of blue. Each pair of hexadecimal digits represent values in the range 0–255. Thus, these color encodings are just different representations of the same range of colors. Include in abstract class ColorCode methods for displaying a color code, for adding and reducing the amount of red, green, or blue by a given percentage, and for producing the complement of a given color. The complement is determined by subtracting each color value from 255. Develop a program in which the user can enter a series of colors in both RGB and hexadecimal form, and have them altered in the ways given above.

D2. Morse Code Translation Program
Implement a Morse code translator program from the UML design given below for translating English message files into Morse code, and Morse code files into English. The requirements for this program are as follows:

♦ English message files will be stored as text files, one sentence per line.

♦ Morse code message files will be stored as text files, one encoded character per line, where end of words are indicated by a blank line, and end of sentences by two blank lines.
♦ English message files will contain file extension .eng, and Morse code files file extension .mor. The assumptions for this program are as follows:

♦ Messages will only contain lowercase letters, the digits 0–9, periods, commas, and question marks.
Following is the UML diagram for the program.



The Translator class coordinates the translation of messages (English to Morse code, and Morse code to English). It contains two class members through composition: InputBuffer and OutputBuffer. Thus the Translator class is responsible for their construction.

The InputBuffer contains the current message line read from the file. If the file contains an English message, then an EnglishInputBuffer is used; if the files contains a Morse-coded message, then a MorseInputBuffer is used. As the Translator reads the current MesgChar from the InputBuffer, it requests the MesgChar to translate itself, sending it to the OutputBuffer. When isEndOfWord or isEndOfSentence is found true, it calls the corresponding MarkEndOfWord or MarkEndOfSentence method of the OuputBuffer. If the output is in English, a space character for each end of word and a period for each end of sentence is added to the buffer; if the output is in Morse code, then a blank line for each end of word, and two blank lines for each end of sentence is added to the buffer. This is repeated until the end of file has been reached.

Note that what a given MesgChar is depends on the type of message file reading. If the message being read is an English message, then a MesgChar is a single character (letter, digit, etc.). If, however, the message being read is a Morse-code message, then a MesgChar is a MorseChar (i.e., a string containing up to six dot and dash characters). Thus, when getting and putting MesgChars of a given InputBuffer and OutputBuffer, the number of characters actually retrieved or placed in the buffer depends on what type of buffer is involved. The Morse Code table for this assignment is given below.

Morse Code Table

A	.-	N	-.	1	.----	.	.-.-.-
B	-...	O	---	2	..---	,	--..--
C	-.-.	P	.--.	3	...--	?	..--..
D	-..	Q	--.-	4	....-	(	-.--.
E	.	R	.-.	5	.....	)	-.--.-
F	..-.	S	...	6	-....	-	-....-
G	--.	T	-	7	--...	"	.-..-.
H	....	U	..-	8	---..	_	..--.-
I	..	V	...-	9	----.	'	.----.
J	.---	W	.--	0	-----		
K	-.-	X	-..-				
L	.-..	Y	-.--				
M	--	Z	--..				

CHAPTER 11 Recursion

In Chapter 3, we covered the fundamental types of control—sequential, selection, and iterative—used to affect the control flow of programs. There is one final form of control that we have yet to cover, referred to as recursion. We look at the use of recursion and recursive problem solving in this chapter.

OBJECTIVES
After reading this chapter and completing the exercises, you will be able to: ♦ Describe the design of recursive functions
♦ Define infinite recursion
♦ Apply recursive program solving
♦ Explain the appropriate use of iteration vs. recursion ♦ Develop recursive functions in Python

CHAPTER CONTENTS

Motivation
Fundamental Concepts
11.1 Recursive Functions
11.2 Recursive Problem Solving 11.3 Iteration vs. Recursion Computational Problem Solving 11.4 Towers of Hanoi

MOTIVATION

Almost all computation involves the repetition of steps. Iterative control statements, such as the for and while statements, provide one means of controlling the repeated execution of instructions. Another way is by the use of *recursion*.

In *recursive problem solving*, a problem is repeatedly broken down into similar subproblems, until the subproblems can be directly solved without further breakdown. For example, consider the method of searching for a name in a sorted list of names below.

1. If the list contains only one name, then if the name found is the name you are looking for, then terminate with “name found,” otherwise terminate with “name not found.”

2. Otherwise, look at the middle item in the list. If that is the name you are looking for, then terminate with “name found.”

3. Otherwise, continue by searching in a similar manner either the top half of the list, if the name you are looking for is alphabetically before the middle name of the list, or the bottom half of list, if the name you are looking for is alphabetically after.

The first two steps of this method are straightforward. The detail comes when the list needs to be continually broken down into sublists, and the appropriate sublists are searched. The beauty of recursive problem solving, however, is that the details of how to solve (smaller) subproblems do not need to be specified—the same steps that were used on the original list still apply. Although it is natural to try to think through all the resulting steps that are taken to recursively

solve a problem, the power of “recursive thinking” is to understand that doing so is unnecessary. In this chapter, we demonstrate the power of recursive problem solving by looking at some classic examples that highlight its effectiveness.

FUNDAMENTAL CONCEPTS

11.1 Recursive Functions

Computational problem solving via the use of *recursion* is a powerful problem-solving approach. The development and use of recursive functions, however, requires a different perspective on computation than we have had so far. We discuss the design and use of recursive functions in this section.

11.1.1 What Is a Recursive Function?

A **recursive function** is often defined as “a function that calls itself.” While this is an accepted definition, it is not necessarily the most appropriate explanation, for it plants in one’s mind the image given in Figure 11-1.

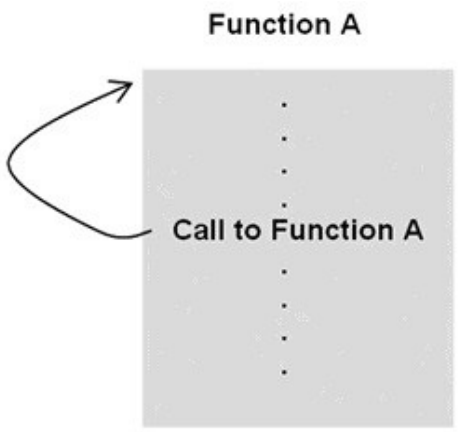


FIGURE 11-1 Recursive Function Definition

The illustration in the figure depicts a function, A, that is defined at some point to call function A (itself). The notion of a self-referential function is inherently confusing. There are two types of entities related to any function however—the *function definition*, and any current *execution instances*.

What is meant by the phrase “a function that calls itself ” is a function *execution instance* that calls another *execution instance* of the same function. A function definition is a “cookie cutter” from which any number of execution instances can be created. Every time a call to a function is made, another execution instance of the function is created. Thus, while there is only one definition for any function,

there can be any number of execution instances. In order to fully understand the mechanism of recursive function calls, we first consider the general mechanism of non-recursive function calls as depicted in Figure 11-2.

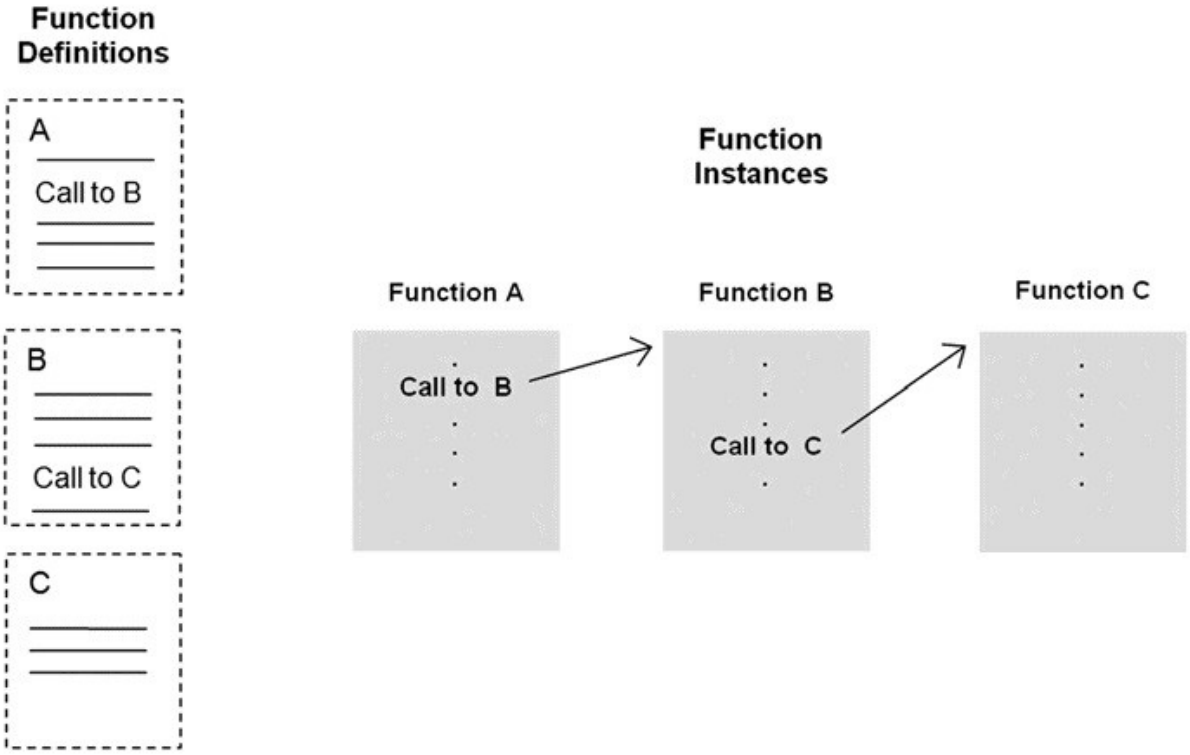


FIGURE 11-2 General Function Calls

In the function calls in the figure, there is no trouble visualizing the sequence of events that occur. First, an execution instance from the definition of function A is created and begins executing. When the call to function B is reached, the execution instance of function A is suspended while an execution instance of function B is created and begins executing. In turn, when the function call to function C is reached, function B suspends execution while an execution instance of function C is created and begins executing.

This calling and suspending of executing function instances could (theoretically) continue indefinitely. However, in this case, function C does not make a call to any other function. Thus, it simply executes until termination, returning control to the function that called it, function B. Function B then continues its execution until terminating, returning control to the function that called it, function A. Finally, function A completes and terminates, returning control to wherever it was called from.

Now, let's consider the situation when the original function, function A, is a recursive function—that is, its definition includes a call to function A (itself). As depicted in Figure 11-3, each current execution instance of function A will spawn a *new* execution instance of function A.

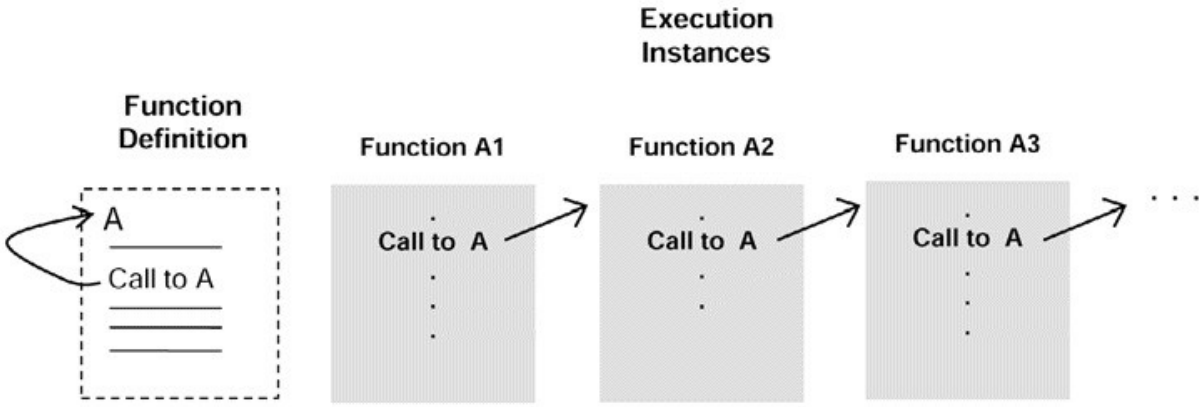


FIGURE 11-3 Recursive Function Execution Instances

Note that the execution of a series of recursive function instances is similar to the execution of a series of non-recursive instances, except that the execution instances are "clones" of each other (that is, of the same function definition). Thus, since all instances are identical, the function calls occur in exactly the same place in each.

Clearly, if the definition of a recursive function were written so that the function calls itself unconditionally, then *every* execution instance would unconditionally call another execution instance, ad infinitum. Such a nonterminating sequence of calls is referred to as **infinite recursion**, similar to the notion of an infinite loop. Therefore, *properly designed recursive functions always conditionally call another execution instance* so that eventually the chain of function calls terminates.

Now that we have better understanding of recursive functions, we can use the description of a recursive function as "a function that calls itself," understanding that this means that the function *definition* is self-referential, while the function execution instances are not. Next we will look at a classic example of a recursive function, computing the factorial of a given number.

LET'S TRY IT From the Python Shell, enter the following and observe the results.

... def rfunc(n): ... def rfunc(n): print(n) if n 55 1: if n .0: return 1

rfunc(n 21) else: return n 1 rfunc(n 2 1) ... rfunc(4) ... rfunc(1)
??? ???
... rfunc(0) ... rfunc(3)
??? ???
... rfunc(100) ... rfunc(100)
??? ???

A **recursive function** is a function (definition) that conditionally calls itself.
11.1.2 The Factorial Function

We now look at a particular mathematical function, factorial, which is often defined by a recursive definition. We then look at how such a recursive definition can be written as a recursively defined program function.

The Recursive Definition of the Factorial Function
The factorial function is an often-used example of the use of recursion. The computation of the factorial of 4 is given as,
factorial(4) 5 4 ? 3 ? 2 ? 1 5 24
In general, the computation of the factorial of any (positive, nonzero) integer n is,
factorial(n) 5 n ? (n 2 1) ? (n 2 2) ? . . . 1

The one exception is the factorial of 0, defined to be 1. Note that if we apply this definition to the factorial of n 2 1, we get factorial(n 2 1) 5 (n 2 1)(n 2 2) . . . 1. Therefore, the factorial of n can be defined as n times the factorial of n 2 1,

factorial(n) 5 n ? (n 2 1) ? (n 2 2) ? . . . 1 factorial(n 2 1)
Thus, the complete definition of the factorial function is,
factorial(n) 51, if n 0 5 n ? factorial(n 2 1), otherwise

This definition of the factorial function is clearly defined in terms of itself, referred to as a **recursive definition**. The part of the definition “factorial(n) 5 1, if n 5 0” is referred to as the **base case**. The base case of a recursive definition is what terminates the repeated application of the definition (and thus the repeated function calls when executed).

Consider what would happen if the base case for the factorial function were not part of the definition,

factorial(n) 5 n ? factorial(n 2 1), for all n
Applying this definition, the computation of the factorial of 4 would be,
factorial(4) 5 4 ? 3 ? 2 ? 1 ? 0 ? 21 ? 22 ? 23 ? 24 ? . . .

The factorial of any (positive) number would be 0, since 0 would always be part of the product of values. Thus, not only is this an incorrect definition of the factorial function, if we implemented this definition as a recursive function, it would never terminate (and thus never produce a result).

Suppose, on the other hand, the base case for the definition of the factorial function is given,
but that n * factorial(n 2 1) were given as n * factorial(n 1 1) instead,
factorial(n) 51, if n 0 5 n ? factorial(n 1 1), otherwise
Applying this definition, the computation of the factorial of 4 would be,
factorial(4) 5 4 ? 5 ? 6 ? 7 ? 8 ? 9 ? 10 ? 11 ? 12 ? . . .

If we implemented this (incorrect) version of factorial as a recursive function, the function would also never terminate. The problem, however, it is not because a base case is not included. It is because *the problem is not being broken down into subproblems in which the base case can be applied*.

This highlights three important characteristics of any recursive function, given in Figure 11-4.

1. There must be at least one base case (a problem instance whose solution is known without further recursive breakdown).
2. Problems that are not a base case are broken down into subproblems that are a similar kind of problem as the original problem and work towards a base case.
3. There is a way to derive the solution of the original problem from the solutions of the recursively solved subproblems.

FIGURE 11-4 Requirements of a Properly Designed Recursive Function
Going back to the original (correct) definition of the factorial function therefore,
factorial(n) 5 51, if n 0
5 n ? factorial(n 2 1), otherwise

we see that the first condition holds since the base case, factorial(0) 51, can be applied without any future recursive breakdown of the problem. It follows the second condition since the problem is broken down into a subproblem that is a smaller instance of the original. Finally, it meets that third condition since the results of the original problem can be determined by multiplying the solution of

each subproblem. Thus, this is a properly defined recursive function. We next look at an actual implementation of a recursive factorial function.

Every properly defined recursive function must have at least one base case, and must redefine the problem into subproblems that work towards a base case such that the solution of the original problem can be derived from the solutions of the recursively solved subproblems.

A Recursive Factorial Function Implementation
Given a recursive definition of the factorial function, we can simply write it as Python program code. This is given in Figure 11-5.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

FIGURE 11-5 Recursive Factorial Function
Implementation

Examination of this function reveals that the recursive function call is conditionally made. That is, only if n is not equal to zero is another execution instance created, otherwise the current execution instance terminates and returns a value of 1. Termination is guaranteed since the initial value of parameter n is required to be greater than or equal to 0, and each next function call operates on smaller values (i.e., n 2 1). Finally, the solutions of all of the subproblems provide a solution to the original one. The sequence of function execution instances generated for the factorial of 4 is given in Figure 11-6.

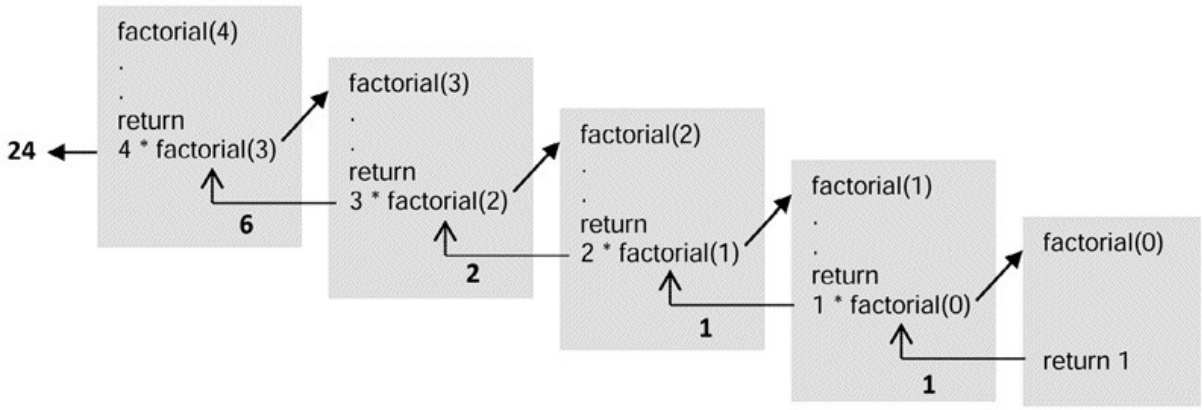


FIGURE 11-6 Factorial Recursive Instance Calls

Each execution instance of function factorial is suspended while the evaluation of expression n * factorial(n 21) is completed. When factorial is finally called with the value 0, five execution instances of the function exist, the first four suspended until instance factorial(0) completes. When factorial(0) returns the value 1, the evaluation of expression 1 * factorial(0) can be completed, returning 1 as the value of factorial(1), and so on, until 4 * factorial(3) is evaluated and 24 is returned as the value of the original function call.

Finally, we point out that although the factorial function serves as a good example, in practice, recursion is an inappropriate choice for implementing this function. This is because an iterative version can be easily implemented, providing a much more efficient computation. (We discuss the appropriate use of recursion in section 11.3.)

LET'S TRY IT From the Python Shell, enter the following and observe the results. ... def factorial(n):
if n 55 0:
return 1
return n * factorial(n 2 1)

... factorial(4) ???
... factorial(0) ???
... factorial(100) ???
... factorial(10000) ???
... def ifactorial(n):

result 5 1
if n 55 0:
return result

for k in range(n, 0, 21): result 5 result * k
return result
... ifactorial(0)
???
... ifactorial(100)
???
... ifactorial(10000)
???

Although the factorial function is an often-used example of a recursive function, the function can be executed more efficiently when implemented to use iteration.

11.1.3 Let's Apply It—Fractals (Sierpinski Triangle)

Figure 11-7 illustrates a well-known recursive set of images called the Sierpinski triangle. The Sierpinski triangle is an example of a *fractal*. A fractal is a shape that contains parts that are similar to the whole shape, thus having the property of self-similarity (Figure 11-8). The turtle graphics program in Figure 11-9 generates Sierpinski triangles at various levels of repetition. This program utilizes the following programming features:

recursive functions➤

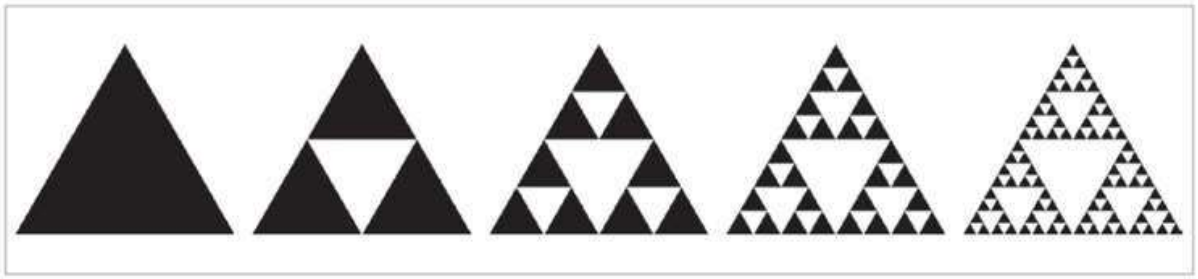


FIGURE 11-7 Sierpinski Triangle (Fractal Image)

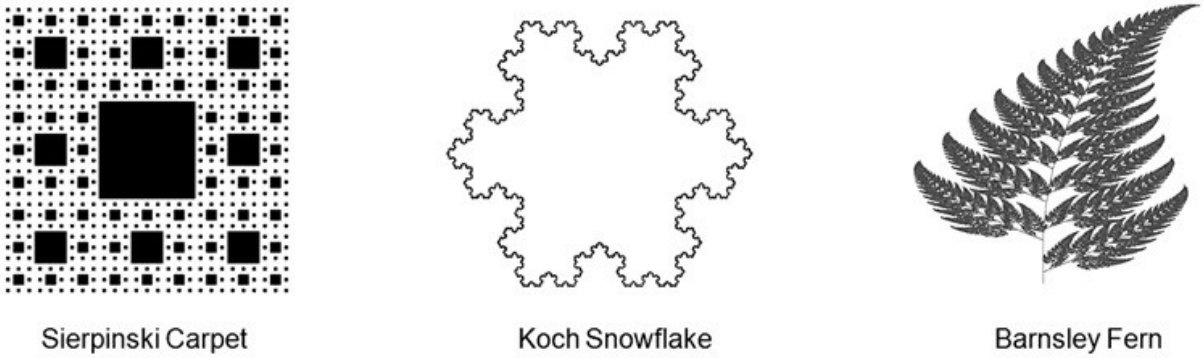


FIGURE 11-8 Other Fractal Images

In addition to the turtle module, the program also imports the math module, needed for the square root function in the calculation of the height of a triangle (in function triangleHeight on **lines 21–25**). In the Sierpinski triangle, each next level in the pattern replaces each triangle with three smaller triangles. In order for the position of each next triangle to be determined (by functions getLeftTrianglePosition, getRightTrianglePosition, and getTopTrianglePosition at **lines 27**, **38**, and **49**, respectively) both the length of the sides of the triangle, as well as its height are needed. This is depicted in Figure 11-10.

The position of turtle shapes in turtle graphics is relative to the center of the shape. Each triangle is positioned relative to its center point (shown by the dot in the figure). Thus, method getLeftTrianglePosition calculates the location of the bottom left triangle as,

[position [0] 2 side / 4, position [1] 2 triangleHeight(side) / 4]

```
# Sierpinski Triangle Program

import turtle
import math

def createTriangleShape(coords):

    """Creates turtle shape from coords. Registers as 'my_triangle'. """

    turtle.penup()
    turtle.begin_poly()
    turtle.setposition(coords[0])
    turtle.setposition(coords[1])
    turtle.setposition(coords[2])
    turtle.setposition(coords[0])
    turtle.end_poly()

    tri_shape = turtle.get_poly()
    turtle.register_shape('my_triangle', tri_shape)

def triangleHeight(side):

    """Returns height of equilateral triangle with length side."""

    return math.sqrt(3) / 2 * side

def getLeftTrianglePosition(position, side):

    """Returns position of bottom left triangle in larger triangle.

       Returns (x,y) position for provided position and side length of
       larger triangle to be placed within.
    """

    return (position[0] - side / 4,
            position[1] - triangleHeight(side) / 4)

def getRightTrianglePosition(position, side):

    """Returns position of bottom right triangle in larger triangle.

       Returns (x,y) position for provided position and side length of
       larger triangle to be placed within.
    """

    return (position[0] + side / 4,  \
            position[1] - triangleHeight(side) / 4)

```

FIGURE 11-9 Sierpinski Triangle Program (*Continued*)

As a result, the x (horizontal) position of the lower left triangle is one quarter of the length of the side less than the larger triangle, therefore positioned to the left of the larger triangle's location. The y (vertical) position of the smaller triangle is one quarter of the height of the triangle less than the larger triangle, thus placed below the position of the original triangle. Positioning of the lower right triangle

(by method getRightTrianglePosition) is similarly determined (positioned to the right of and below the location of the original triangle). Finally, method getTopTrianglePosition determines the position of the smaller top triangle to be at the same x location as the original triangle, and one quarter of the height of the triangle higher.

```
49 def getTopTrianglePosition(position, side):
50
51     """Returns x,y position of top triangle within larger one.
52
53        For triangle at position, with length side.
54     """
55
56     return (position[0], position[1] + triangleHeight(side) / 4)
57
58 def drawSierpinskiTriangle(t, len_side, levels):
59
60     """Recursive function that draws a Sierpinski triangle.
61
62        Draws the number of levels of triangle given in levels.
63     """
64
65     if levels == 0:
66         t.color('black')  # display triangle
67         t.showturtle()
68         t.stamp()
69
70         return
71
72     # resize triangle to half its size
73     stretch_width, stretch_length, outline = t.turtlesize()
74     t.turtlesize(0.5 * stretch_width, 0.5 * stretch_length, outline)
75
76     # determine positions for each of the three embedded triangles
77     left_triangle_position = getLeftTrianglePosition(t.position(),
78                                                      len_side)
79     right_triangle_position = getRightTrianglePosition(t.position(),
80                                                        len_side)
81     top_triangle_position = getTopTrianglePosition(t.position(),
82                                                    len_side)
83
84     # recursively display left triangle
85     t.setposition(left_triangle_position)
86     drawSierpinskiTriangle(t, len_side / 2, levels - 1)
87     t.turtlesize(0.5 * stretch_width, 0.5 * stretch_length, outline)
88
89     # recursively display right triangle
90     t.setposition(right_triangle_position)
91     drawSierpinskiTriangle(t, len_side / 2, levels - 1)
92     t.turtlesize(0.5 * stretch_width, 0.5 * stretch_length, outline)
93
94     # recursively display top triangle
95     t.setposition(top_triangle_position)
96     drawSierpinskiTriangle(t, len_side / 2, levels - 1)
97     t.turtlesize(0.5 * stretch_width, 0.5 * stretch_length, outline)
98
```

FIGURE 11-9 Sierpinski Triangle Program (*Continued*)

Once the proper positioning of the smaller triangles is determined, the rest of the program is based on the use of recursion to repeatedly apply this division of triangles until a given number of levels have been drawn. Function createTriangleShape (**lines 6–19**) creates an equilateral triangle by positioning

the turtle to each of the screen locations provided in parameter coords. Because this movement occurs within the begin_poly and end_poly instructions, the polynomial

```
 99 # ---- main
100
101 # set window size
102 turtle.setup(800, 600)
103
104 # get turtle
105 the_turtle = turtle.getturtle()
106
107 # init turtle
108 the_turtle.penup()
109 the_turtle.hideturtle()
110
111 # set the number of levels
112 num_levels = 3
113
114 # create triangle shape
115 coords = ((-240, -150), (240, -150), (0, 266))
116 createTriangleShape(coords)
117 len_side = 480
118
119 # create first triangle
120 the_turtle.shape('my_triangle')
121 the_turtle.setposition(0, -50)
122 the_turtle.setheading(90)
123
124 # call recursive function
125 drawSierpinskiTriangle(the_turtle, len_side, num_levels)
126 the_turtle.hideturtle()
127
128 # terminate program when close window
129 turtle.exitonclick()
```

FIGURE 11-9 Sierpinski Triangle Program

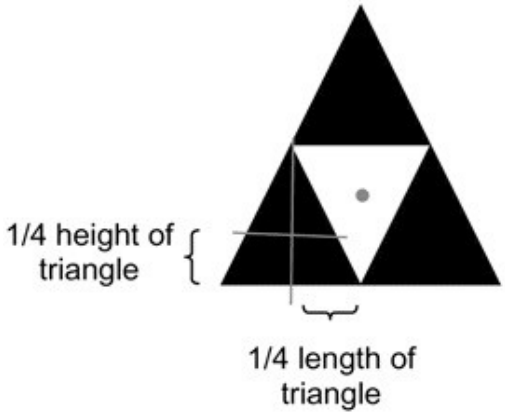


FIGURE 11-10 Relative Placement of Inner Triangles

drawn can be retrieved (**line 18**) and registered as a new turtle shape that can be

used within the program (**line 19**).

The main section of the program (**lines 101–129**) does the needed preparation before drawing can begin. On **line 102** the size of the turtle window is set. On **line 105** the (default) turtle is retrieved and named the_turtle. The turtle is then initialized so that the drawing capability is off (penup) and it is hidden. This is done since the only graphics to be produced by the turtle is when its (triangle) shape is stamped. Thus, the turtle is moved and resized to create all the stamped triangle images needed for creating a given Sierpinski triangle. On **line 112** the number of levels of the triangle is set. Thus, by changing this value, a Sierpinski triangle of various levels can be created.

On **line 115** a tuple of coordinates is defined that creates an equilateral triangle. (The absolute positions of these coordinates are not relevant, only their relative positions are used for defining the shape.) The length of the triangle is specified on **line 117**, matching the length of the triangle given by the specified coordinates. The turtle specified by the_turtle is set to shape 'my_triangle'. It is then positioned at location (0, 250) of the screen (a little below the center) and the heading is set to 90 degrees (to ensure that the triangle is pointing up).

Recursi ve function drawSierpinskiTriangle (**lines 58–97**) is passed three arguments: the turtle (in parameter t), the length of the sides of the overall triangle, and the number of levels of the Sierpinski triangle pattern to draw (**line 125**). As a recursive function, there must be a base case in which the function no longer calls itself. Since the number of levels starts at some nonzero value, the base case is reached when the value of parameter levels is 0 (**line 65**). At that point, the turtle size and location is specified for the smallest of the embedded triangles. Therefore, since no further breakdown of the triangles is needed, these lowest-level triangles are simply displayed.

When function drawSierpinskiTriangle is called with levels not equal to zero, the size of the current triangle shape of the turtle is cut in half (**lines 73–74**). The positions of each of the three smaller triangle to fit in the area of the current turtle shape are determined, and three recursive calls are made—one for each of the smaller triangles (**lines 85–97**). For each recursive call, the turtles are first positioned (**lines 85**, **90**, and **95**), and the recursive function calls made (**lines 86**, **91** and **95**). Because each recursive call resizes the turtle shape, on **lines 87**, **92**, and **97** the turtle shape is reset to what it was before each such call.

Self-Test Questions

1. A recursive function is best thought of as
(a) A function that calls itself
(b) A function execution instance that calls another execution instance of the same function

2. A recursive function call that never terminates is called . **3.** What does the following recursive function return as a value, for any given positive integer argument n?
def rfunc(n):
if n 55 0:
return 0
else:
return rfunc(n 2 1) 1 2
4. How many times does the factorial function in the chapter get called when computing the factorial of 4?
5. Iteration and recursion are two means of repeating a set of instructions. (TRUE/FALSE)
ANSWERS: 1. b, 2. infinite recursion, 3. 2n, 4. n 1 1, 5. True

11.2 Recursive Problem Solving
11.2.1 Thinking Recursively

We commonly solve problems by breaking a problem into subproblems, and separately solving each subproblem. Recursive problem solving is the same, except that a problem is broken down into subproblems that are another instance of the original type problem. This was true of the factorial function in section 11.1. The problem of solving the factorial of n was broken down into two subproblems—the multiplication by n as one subproblem, and the determination of the factorial of n 2 1 as the other, as shown in Figure 11-11.

The factorial function is an “easy” example of recursion, because the function itself is defined recursively. Thus, the recursive function developed is just an implementation of the mathematical definition. The use of recursion is most interesting when applied to problems that are not recursively defined. To do this, we need to meet the three requirements of recursive functions given earlier in Figure 11-4.

The power of recursion is that it provides a conceptually elegant means of

problem solving. The “elegance” derives from the fact that there is no need to specify or even think through all the steps that are taken to solve the problem. Since the recursive subproblems are the same kind of problem as the original, specifying the solution of the overall problem provides sufficient detail for solving each of the similar subproblems. To illustrate this, we look at some of the most well-known problems submitting to a recursive solution. We begin with an efficient means of sorting called MergeSort.

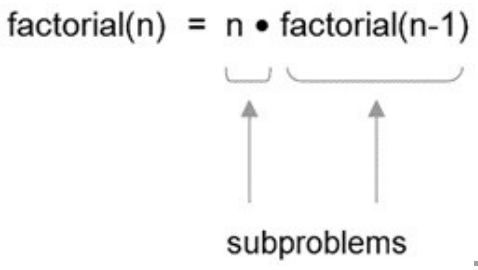


FIGURE 11-11
Subproblems of Factorial Function

The **power of recursion** is that it provides a conceptually elegant means of problem solving.

11.2.2 MergeSort Recursive Algorithm

In this section we look at how we can apply recursive problem solving to the problem of sorting lists. In order to solve this problem recursively, we have to imagine how the problem of sorting can be broken into subproblems, so that one or more of the subproblems is also the problem of sorting a (smaller) list.

When breaking a problem down, it is often most effective to break the problem into equal size subproblems. We consider, therefore, breaking down the problem of sorting n elements, into two subproblems of sorting lists of size n/2, as depicted in Figure 11-12.

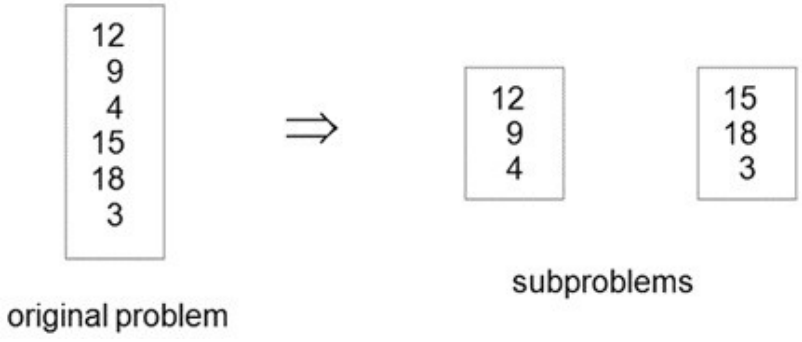


FIGURE 11-12
Breakdown of the Problem of Sorting a List

At this point, the power of recursive thinking comes into play. Since we are developing a method for sorting lists of size n, *we can assume that any list of size less than n can be sorted, without needing to determine in detail how that is done*. (This is closely related to proof by induction in mathematics.) Thus, we can continue to develop our method for sorting lists of size n based on that fact.

The next step is to identify a base case that does not need to be broken down any further in order to be solved. The obvious base case is lists of size 1, since by definition they are sorted. Since we are dividing each list into two sublists at each step of the recursion, eventually each sublist will be of size 1. Thus, the breakdown into subproblems lead towards the base case, as required.

The last step is to determine how two (recursively solved) sorted sublists can be combined into one complete sorted list as depicted in Figure 11-13.

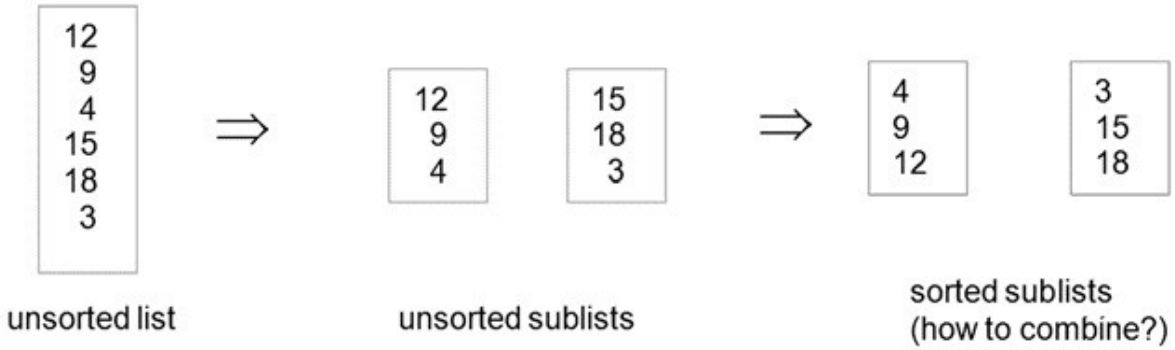


FIGURE 11-13 After Sorting Each of the Two Sublists

We cannot simply concatenate one sublist with the other. We need to somehow *merge* the values into one properly sorted list. This can be done as follows. Compare the smallest (top) item in each of the two lists. Whichever is the smaller value, move that as the first item of a new list. Cross off the item moved. Continue in the same manner until all the elements have been moved to the new list. The first steps of this process are shown in Figure 11-14.

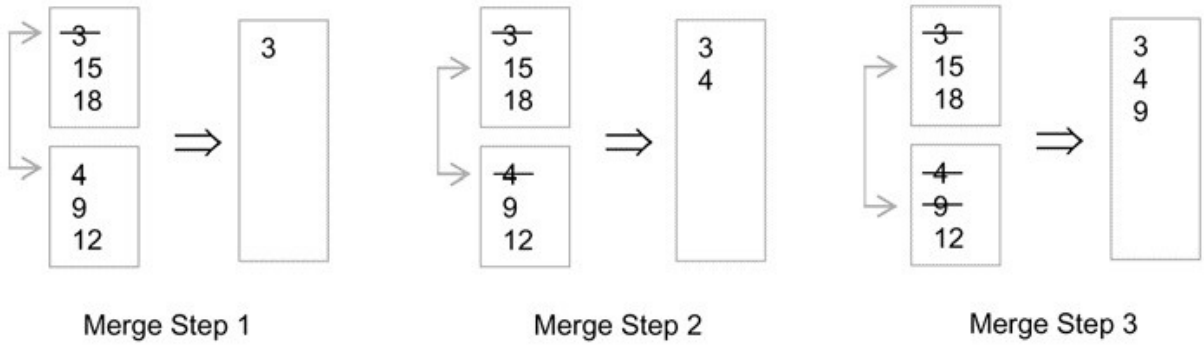


FIGURE 11-14 The Process of Merging Two Sorted Lists

Now that we have determined how to break down the problem into (smaller) subproblems, and how to combine the solutions of the subproblem into a solution of the larger problem, we can specify the complete steps of this “merge sort” algorithm.

1. Split the list into two sublists.
2. Sort each sublist.
3. Merge the sorted sublists into one sorted list.

Figure 11-15 shows how this recursive method works on a particular list.

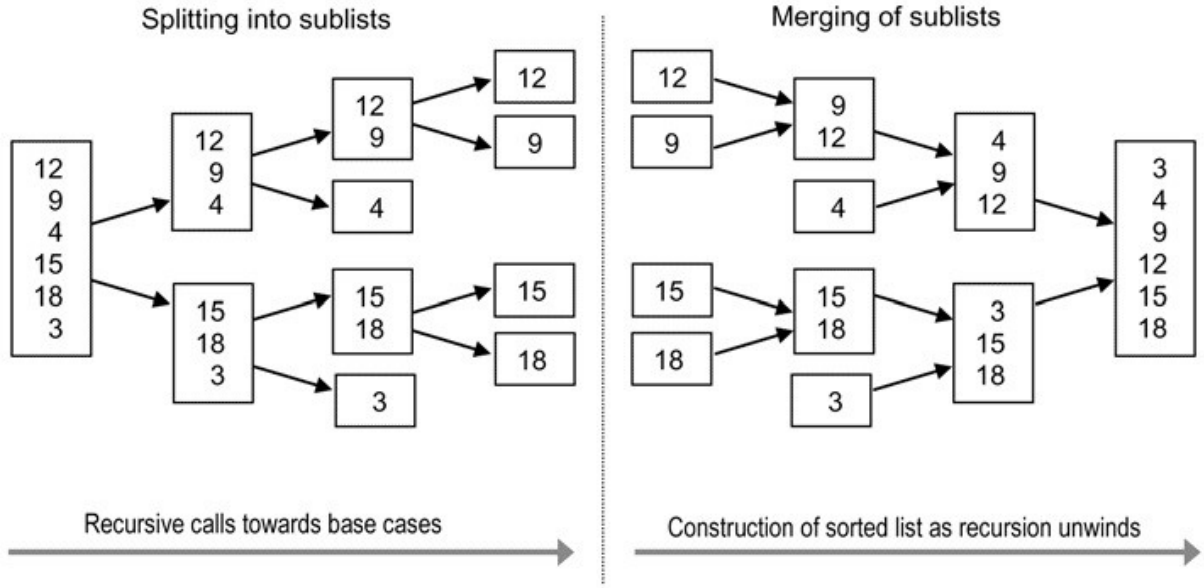


FIGURE 11-15 Recursive Problem-Solving Example (MergeSort)

The initial problem is to sort the list 12, 9, 4, 15, 18, 3. This is broken down into the problem of sorting the sublists 12, 9, 4 and 15, 18, 3. Since each sublist will be sorted in a similar manner, let’s follow how the sublist 12, 9, 4 is sorted. It is

broken down into the two sublists 12, 9 and 4. Since the sublist containing only the value 4 is by definition sorted (as a base case) that subproblem does not need to be broken down any more—it is solved. So the sublist 12, 9 is broken down into the list containing 12 and another containing 9. These are now also solved since they are each a base case. The diagram indicates how the original sublist 15, 18, 3 is similarly broken down.

Now that since all base cases have been reached, the process of merging sublists begins as shown on the right side of the figure. Thus, to produce the sorted list 9, 12 the two base cases 9 and 12 are merged. We have shown how this merging can be systematically done. To produce the sublist 4, 9, 12, the *sorted* sublist 9, 12 is merged with the *sorted* sublist 4. The sorted sublist 3, 15, 18 is similarly produced. Thus, the final step is to merge the sorted versions of the original two subproblems, 4, 9, 12 and 3, 15, 18. We next give a Python implementation of this algorithm.

11.2.3 Let's Apply It—MergeSort Implementation

An implementation of the MergeSort algorithm is given in Figure 11-16. This program utilizes the following programming features.

➤ recursive functions

Function mergesort implements the recursive algorithm given above. When mergesort receives a list of length 1 (the base case), it simply returns the list (**lines 3–4**). Otherwise, the list is broken into two sublists, sublist1 and sublist2 (**lines 6–7**). Note that integer division is used, len(lst)//2, so that we don't compute non-integer lengths. Thus, when lst contains an uneven number of elements, for example 5, sublist1 and sublist2 will be of lengths 2 and 3, respectively, which makes no difference in the algorithm.

Once the list is divided into sublists, each of the sublists is sorted by a recursive call to mergesort (**lines 9–10**). Utilizing the power of recursive thinking, *we can assume that these recursive calls work without having to think through all the recursive steps*. Attempting to think

```
1  def mergesort(lst):
2
3      if len(lst) == 1:
4          return lst
5
6      sublist1 = lst[0:len(lst) // 2]
7      sublist2 = lst[len(lst) // 2:len(lst)]
8
9      sorted_sublist1 = mergesort(sublist1)
10     sorted_sublist2 = mergesort(sublist2)
11
12     return merge(sorted_sublist1, sorted_sublist2)
13
14
15 def merge(lst1, lst2):
16
17     merged_list = []
18
19     i = 0
20     k = 0
21
22     while i != len(lst1) and k != len(lst2):
23         if lst1[i] < lst2[k]:
24             merged_list.append(lst1[i])
25             i = i + 1
26         else:
27             merged_list.append(lst2[k])
28             k = k + 1
29
30     if i < len(lst1):
31         for loc in range(i, len(lst1)):
32             merged_list.append(lst1[loc])
33     elif k < len(lst2):
34         for loc in range(k, len(lst2)):
35             merged_list.append(lst2[loc])
36
37     return merged_list
```

FIGURE 11-16 Recursive Function mergesort

through the steps at each recursive level is tedious, confusing and unnecessary. Finally, the two sorted sublists are merged and returned as the final sorted list (**line 12**).

Note that supporting function merge is longer and more detailed than function mergesort. This should not be surprising, since the real work lies in the merging of sublists. In the function, the new merged list to be returned is constructed in variable merged_list. Thus, merged_list is initialized as an empty list (**line 17**). Variables i and k are used to keep track of the position of the current elements being compared in each of sublist1 and sublist2; thus, each is initialized to 0 (**lines 19–20**).

The while loop in **lines 22–28** merges one more element into merged_list for each iteration of the loop. If the current item in lst1 (determined by the current value of i) is less than the current item in lst2 (determined by the current value of k), then the current item of lst1, lst1[i], is appended to merged_list. Otherwise, the current item of lst2, lst2[k], is appended. Once the while loop terminates, all of the elements of lst1 and/or lst2 have been merged. What is left to do is append any remaining items of either sublist whose elements have not yet been merged (when the two sublists are not of equal length). This is taken care of in **lines 30–35**. Finally, on **line 37**, the newly created merged list is returned.

We note that the MergeSort function makes two recursive calls. Such recursive functions are called *doubly recursive*. Double recursion makes the conversion of a recursive solution to an iterative one more difficult than a "singly recursive" algorithm. We shall see another example of a doubly recursive function in the Computational Problem Solving section of the chapter.

Self-T est Questions
1. Only problems that are recursively defined can be solved using recursion. (TRUE/FALSE)
2. The power of recursion is the execution speed over iteration. (TRUE/FALSE)

3. The base case in the MergeSort algorithm is,
(a) an empty list **(b)** a list of length one **(c)** a list of length two **4.** The MergeSort algorithm only works on lists of length two or more. (TRUE/FALSE) **5.** The MergeSort algorithm divides a list into equal, or nearly equal, sublists in each recursive level (except for the base case). (TRUE/FALSE)
ANSWERS: 1. False, 2. False, 3. (b), 4. False, 5. True

11.3 Iteration vs. Recursion

Recursion is fundamentally a means of repeatedly executing a set of instructions. The set of instructions are those of a function, and the repetition comes from the fact that the function is repeatedly executed, once for each recursive function call. Thus, recursion and iteration are two means of accomplishing the same result. Whatever can be computed using recursion can also be computed using iteration, and vice versa.

Since iteration and recursion are equivalent in terms of what can be computed, a natural question is "When should I use recursion, and when should I use iteration?" There is no clear-cut answer to this question, but there are some guidelines. Generally, a recursive function generally takes more time to execute

than an equivalent iterative approach. This is because the multiple function calls are relatively time-consuming. In contrast, while and for loops execute very efficiently. Thus, *when a problem can be solved both recursively and iteratively with similar programming effort, it is generally best to use an iterative approach*. Recall that the Factorial function was an example of such a situation. The iterative version of computing factorial is simple to implement, and executes much more efficiently.

On the other hand, some problems are very difficult to solve iteratively, and almost trivial to solve recursively. This is when recursion is most effectively used. A classic example of such a problem is the Towers of Hanoi, discussed next.

Whatever can be computed using recursion can also be computed using iteration, and vice versa.

COMPUTATIONAL PROBLEM SOLVING

11.4 Towers of Hanoi

In this section we look at a classic example of recursive problem solving in computer science, the Towers of Hanoi.

11.4.1 The Problem

The Towers of Hanoi problem (Figure 11-17) is based on a legend of unknown origin. According to the legend, there is a Vietnamese temple with a large room containing three pegs and 64 golden disks. Each disk has a hole in it so that it can be slipped onto any of the pegs. In addition, each disk is of different size. The 64 disks are moved by priests from one peg to another, with the following conditions,

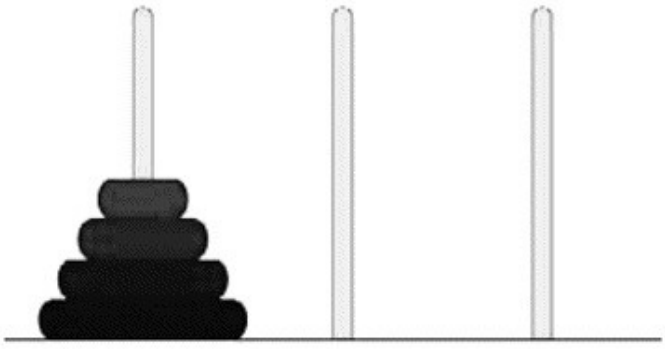


FIGURE 11-17 Towers of Hanoi

♦ Only one disk can be moved at a time. ♦ At no time can a larger disk be placed on top of a smaller one.

When the complete pile of 64 disks have been moved, the world is to end. There is not much need for concern, however—assuming that the priests moved one disk per second, it would take roughly 585 billion years to finish!

11.4.2 Problem Analysis

We will first attempt to solve this problem for three disks to gain some insight into the problem, and then develop a general solution for any number of disks. Thus, we will solve the simple problem of moving three disks from peg A to peg C as shown in Figure 11-18.

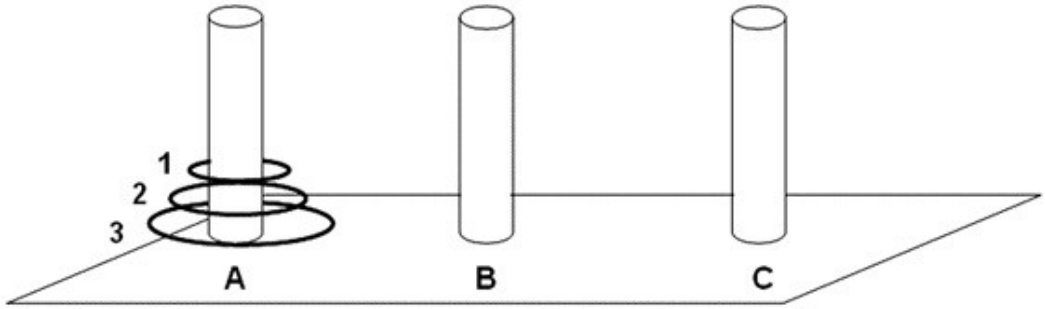


FIGURE 11-18 Towers of Hanoi Problem for Three Disks

First, let's solve this the hard way, by considering every step that must be taken. The obvious first move is to remove the smallest disk, and place it on either peg B or peg C. If we place the smallest disk on peg C, then we must place the next smallest disk on peg B, resulting in the configuration in Figure 11-19.

We then consider our third move. We can place the disk currently on peg B back on peg A, but that will be undoing what we just did in the last step. So in order to make progress, we move the smallest disk on peg C somewhere. We can move it on either peg A or peg B, since in each case it would be placed on a larger disk (thus not violating the problem's conditions). Let's assume that we

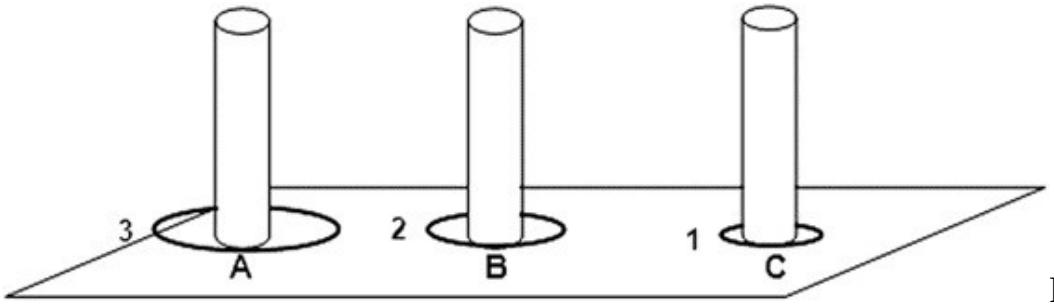


FIGURE 11-19 Towers of Hanoi after Two Moves

move the smallest disk currently on peg C on top of the second smallest disk on peg B. This move results in the configuration shown in Figure 11-20.

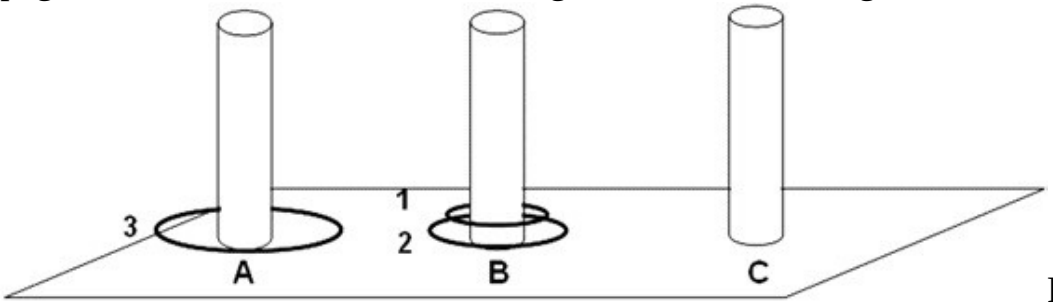


FIGURE 11-20 Towers of Hanoi after Three Moves

Now, we can move the largest disk currently on peg A to peg C (since peg C is currently empty). Then we can move the smallest disk from peg B to peg A, then move the second smallest disk from peg B to peg C, and finally move the smallest disk from peg A to peg C, thereby solving the problem as shown in Figure 11-21.

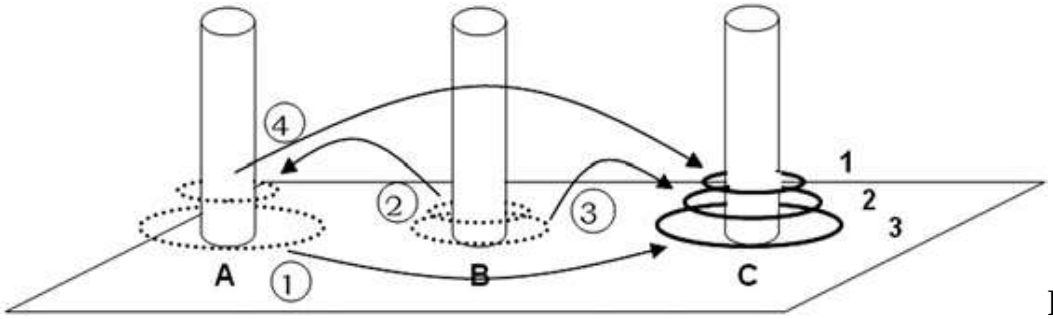


FIGURE 11-21 Towers of Hanoi Final Moves

The question is, what have we learned by thinking through the solution of this problem for three disks? What insight have we gained into the problem involving four disks? Five disks? Attempting to think through the individual steps for a larger and larger number of disks quickly becomes overwhelming.

As you should expect, there is a much more elegant solution for this problem involving recursion. The fundamental steps of a recursive solution for the Towers of Hanoi problem is given below:

Step 1 : View the stack as two stacks, one on top of the other. Call the top stack Stack1, and the bottom stack Stack2.
Step 2: Recursively move Stack1 from peg A to peg B.
Step 3: Recursively move (the exposed) Stack2 from peg A to peg C.

Step 4: Recursively move Stack1 from peg B to peg C.

As we have said, in recursive problem solving, we do not need to specify the detailed steps solving any subproblem that is a smaller, similar problem to the original problem. Here, steps 2, 3, and 4 have to do with solving subproblems that are the same type of problem as the original—moving a stack of disks from one peg to another. It does not matter that the stacks being moved are different, or are being moved between different pegs. Each of the subproblems can be recursively solved in a similar way, and thus we can assume that they can be solved without explicitly specifying how. Therefore, in order to effectively solve the general problem, we only need to decide and specify the details of step 1, how to break down a stack of disks into two separate (sub)stacks. Let's consider different ways of breaking down this problem.

Suppose we start with the obvious breakdown, that is, dividing the stack of disks (six in this example) into two equal (or close to equal) size stacks. If we choose that approach, then after (recursively) moving the top half of the original stack to peg B and (recursively) moving the bottom half of the stack to peg C, we have the configuration given in Figure 11-22.

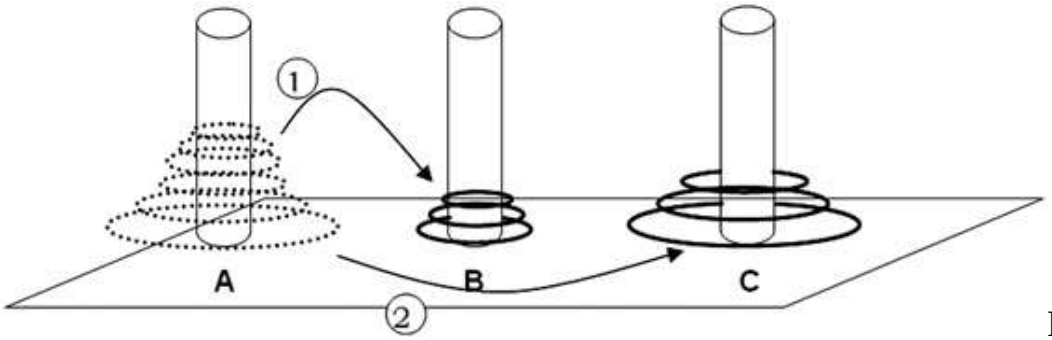


FIGURE 11-22 One Possible Breakdown of the Towers of Hanoi

Here is where we can benefit from recursive thinking. From this configuration, we need to (recursively) move the stack of disks on peg B to peg C to complete the problem solution. However, we are not allowed to move the whole stack at once, but instead must move each disk one at a time. This is accomplished by solving this problem in an identical way to the solution of the larger problem involving all disks. Without thinking through all of the details of each step, we know that the top disk on peg B must be moved at this point (since only top disks can be moved, and we do not want to move the disks on peg C, since they are currently in the correct position). So we can move the top disk from peg B to

either currently empty peg A, or on top of the disks on peg C.

Note that this is the smallest disk of all disks. *Therefore, once we place this smallest disk on either peg, that peg cannot be used to place any other disks on top of it*. But the one insight we should have gained by working through the detailed solution of three disks is that there is always the need for a spare peg when moving a stack of disks from one peg to another. Thus, once the smallest disk is placed on a peg, that peg becomes unusable for placing any other disks on it (and thus unusable as a spare peg), as shown in Figure 11-23 for peg A.

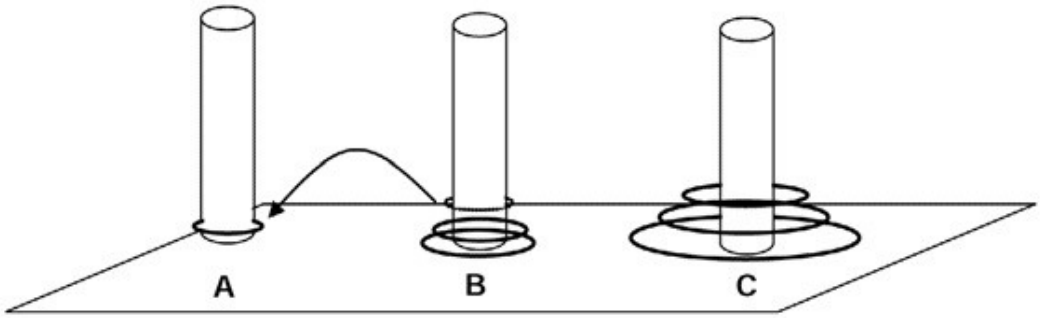


FIGURE 11-23 Inappropriate Breakdown of the Towers of Hanoi

Since we have determined that dividing the stack of disks into two (essentially) equal-size stacks will not work, we need to consider alternate methods of dividing the stack of disks. Suppose we take the opposite approach, and divide the stack into very *unequal*-sized stacks? That is, let's consider dividing the stack so that the top stack consists of the top (smallest) disk only, and the bottom stack consists of the rest of the disks. Let's see how this plays out.

Moving the top stack from peg A to peg B means moving the one smallest disk to peg B, as shown in Figure 11-24.

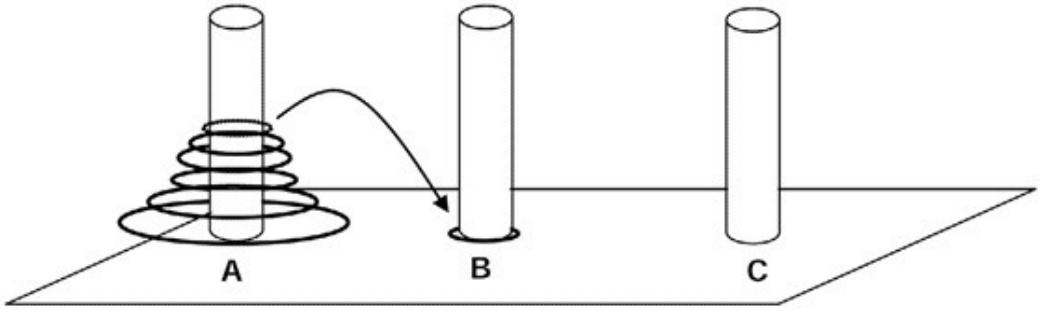


FIGURE 11-24 Another Possible Breakdown of the Towers of Hanoi

That is easy, since it is a stack of only one! Now, we must (recursively) move the

bottom stack (with all the remaining disks) from peg A to peg C. We know that this cannot be done in one step, and must be done by moving one disk at a time. We also know that in order to move such a stack of disks from any peg to any other peg, we must have a spare peg. However, since the smallest disk has been placed on peg B (our spare peg), that is unavailable for use! Again, then, this approach will not work!

Finally, we consider the correct approach. We again break down the stack of disks into very unequal-sized stacks. However, this time, we break it down so that the top stack consists of *all disks except the bottom (largest) disk.* Therefore, the correct approach would be to recursively move the top stack (of n 2 1 disks) from peg A to peg B. Then, simply move the largest (remaining) disk from peg A to peg C. Then recursively move the stack of disks from peg B to peg C (on top of the largest disk), as shown in Figure 11-25.

1_3
2
AB C FIGURE 11-25 Correct Breakdown of the Towers of Hanoi

Note that in this case, when moving the stack of disks from peg B to peg C, we have the needed spare peg, since peg A is currently empty, and peg C has on it the largest disk, making it possible to use either as a spare peg. Thus, our recursive solution is:

Step 1: Recursively move the top n 2 1 disks from peg A to peg B. Step 2: Move the one (remaining) largest disk from peg A to peg C. Step 3: Recursively move the n 2 1 disks from peg B to peg C.

As with MergeSort, this is a doubly recursive algorithm. This is a well-designed recursive solution because:
♦ *There is a base case*. When the stack of disks consists of only one disk, such a "stack" can be moved without further breakdown.
♦ *The recursive steps work towards the base case*. The subproblem involves the solution of smaller and smaller sized stacks of disks.
♦ *The solutions of the subproblems provide a solution for the original problem*. Solutions of subproblems eventually result in all the disks being on the destination peg in the required order.
11.4.3 Program Design and Implementation
Now that we have defined a recursive solution to this problem, implementation

of a program that solves it is surprisingly simple, given in Figure 11-26.

```
def towers(start_peg, dest_peg, spare_peg, num_disks):

    if num_disks == 1:
        print 'move disk from peg', start_peg, 'to peg', dest_peg
    else:
        towers(start_peg, spare_peg, dest_peg, num_disks - 1)
        print 'move disk from peg', start_peg, 'to', dest_peg
        towers(spare_peg, dest_peg, start_peg, num_disks - 1)
```

FIGURE 11-26 Recursive Function Implementation of the Towers of Hanoi

Hopefully you have gained an appreciation for the power and elegance of recursion through the examples given, and that you will be able to begin to apply “recursive thinking” in your own problem solving. We give a program utilizing turtle graphics that produces an animation of the solution of the Towers of Hanoi for a given number of disks in Figure 11-27.

In addition to the turtle module, the program also imports the time module (**lines 3 and 4**). The time module contains a sleep function used to control the speed at which the animation proceeds (**line 217**), pausing 0.5 seconds between the movement of disks.

The main module is in **lines 229–277**. On **line 230** a welcome message is displayed indicating what the program does. The call to setup on **line 233** establishes the turtle screen size (in pixels). The three pegs are positioned at the bottom center of the turtle screen (**line 236**) and the vertical separation of the disks on each peg is set to 24 pixels (**line 237**). (The separation between disks on a peg will be proportionally reduced in function calcDiskLocations so that smaller disks at the top of a stack of disks are closer together than larger disks.)

V ariables peg_A, peg_B, and peg_C are initialized to integer values 0, 1, and 2, respectively. Thus, these values can serve as index values for lists holding both the disk objects on each peg, and a list storing all disk locations. The number of disks to solve the problem for is retrieved from the user by a call to getNumDisks and assigned to variable num_disks (**line 245**). Function displayPegs is then called to display the three pegs (**line 249**) for the locations in variable peg_locations. In **lines 252–253** the image file names the required number of disks (specified in num_disks) are assigned to variable disk_images by a call to function getDisk Images. The image file names are then passed to

function registerDiskImages to be registered in turtle graphics as shapes to which turtle objects can be assigned.

The locations on each peg for where all the disks are to be placed is calculated by function calcDiskLocations (**lines 98–126**), called on **line 256**. The required number of disks

```
1  # Towers of Hanoi Program
2
3  from turtle import *
4  import time
5
6   def getNumDisks():
7
8       """Returns number of disks requested by user."""
9       num_disks = int(input('Enter the number of disks(2-8): '))
10
11      while num_disks < 2 or num_disks > 8:
12          print('Must select between 2 and 8 disks')
13          num_disks = int(input('Enter the number of disks (2-8): '))
14
15      return num_disks
16
17  def displayPegs(peg_locs):
18
19      """Displays peg images at peg_locs."""
20      # retrieve and register peg image
21      peg_image = 'Peg.gif'
22      register_shape(peg_image)
23
24      # set vertical offset
25      offset = 65
26
27      # create temp peg-shaped turtle
28      temp = Turtle()
29      temp.penup()
30      temp.shape(peg_image)
31
32      # stamp three images of peg-shaped turtle
33      for loc in peg_locs:
34          temp.setposition(loc[0], loc[1] + offset)
35          temp.stamp()
36
37  def getDiskImages(num):
38
39      """Returns a list of disk image file names for num requested.
40
41         Disk images in order from largest to smallest image.
42      """
43      # init empty images list
44      images = []
45
46      # init first image file name
47      disk_image = 'Disk-1.gif'
48
49      # create all disk images
50      for k in range(0,num):
51          images = ['Disk-' + str(k+1) + '.gif'] + images
52
53      return images
54
```

FIGURE 11-27 Towers of Hanoi Program (*Continued*)

(turtle objects) are created by function createDisks (**lines 80–96**) called on

line 261. All disks are initially placed on the start peg (peg A) by function placeDisksOnStartPeg (**lines 128– 132**), called on **line 264**. After being properly placed, the disks are made visible by function makeDisksVisible (**lines 134–138**) called on **line 267**. Once the disks are placed, there is a

```
55  def registerDiskImages(images):
56
57      """Registers disk image file names provided in images."""
58      for image in images:
59          register_shape(image)
60
61  def newPeg(image_file):
62
63      """Returns a new turtle with peg shape in image_file."""
64      peg = Turtle()
65      peg.hideturtle()
66      peg.shape(image_file)
67
68      return peg
69
70  def newDisk(image_file):
71
72      """Returns a new turtle with disk shape in image_file."""
73      disk = Turtle()
74      disk.penup()
75      disk.hideturtle()
76      disk.shape(image_file)
77
78      return disk
79
80  def createDisks(images):
81
82      """Returns a list of disk shape turtles for each image in images.
83
84         Disks ordered by image size from largest to smallest.
85      """
86      # init empty list of disks for all three pegs
87      disks = [[], [], []]
88
89      # create disks with all disks on first peg
90      for k in range(0, len(images)):
91
92          disks[0].append(newDisk(images[k]))
93          disks[1].append(None)
94          disks[2].append(None)
95
96      return disks
97
98  def calcDiskLocations(num_disks, peg_locs, separation):
99
100     """Returns the calculated locations of all disks for all three pegs.
101
102        Locations in order from largest (bottom) disk to smallest, with
103        decreasing separation for smaller disks.
104     """
```

FIGURE 11-27 Towers of Hanoi Program (*Continued*)

two-second delay by a call to the sleep function of the time module (**line 270**), followed by a call to the recursive function towers to begin the problem-solving process (**line 273**).

Function towers (**lines 211–225**) implements the recursive algorithm for solving

the problem. When called, it is passed the peg to be treated as the start peg (from which disks are to be moved), a destination peg (where the disks are to be moved), and a spare peg (used as temporary storage while moving disks), and the number of disks to move. The base case is when there is only

```
105     # init
106     saved_separation = separation
107     all_disk_locations = []
108
109     # caculate locations of disks for all three pegs
111     for peg in range(0, 3):
112
113         disk_locs = []
114
115         for k in range(0, num_disks):
116
117             xloc = peg_locs[peg][0]
118             yloc = int(peg_locs[peg][1] + k * separation)
119
120             disk_locs.append((xloc, yloc))
121             separation = separation * 0.95
122
123         all_disk_locations.append(disk_locs)
124         separation = saved_separation
125
126     return all_disk_locations
127
128 def placeDisksOnStartPeg(start_peg, disks, locations):
129
130     """Places and displays disks on start peg in turtle graphics."""
131     for k in (range(0, len(disks[start_peg]))):
132         disks[start_peg][k].setposition(locations[start_peg][k])
133
134 def makeDisksVisible(start_peg, disks):
135
136     """Makes visible all disk-shaped turtles on start_peg."""
137     for k in range(0,len(disks[start_peg])):
138         disks[start_peg][k].showturtle()
139
140 def removeTopDisk(peg, disks):
141
142     """Returns the top disk of disks currently on peg."""
143     # init k to top-most position of disks on peg
144     k = len(disks[peg]) - 1
145     found = False
146
147     # find top-most position with disk in place
148     while k >= 0 and not found:
149
150         if disks[peg][k] != None:
151             disk = disks[peg][k]
152             disks[peg][k] = None
153             found = True
154         else:
155             k = k - 1
156
157     return disk
158
```

FIGURE 11-27 Towers of Hanoi Program (*Continued*)

one disk to move (checked for on **line 215**). In this case, the disk is simply moved from the start peg to the destination peg by call to moveDisk. Otherwise,

the top num_disks – 1 are (recursively) moved from the start peg to the spare peg, the single remaining (bottom) disk is then moved from the start peg to the destination peg, and the num_disks 21 disks on the spare peg are (recursively) moved to the destination peg.

```
159 def placeTopDisk(disk, peg, disks, disk_locations):
160
161     """Places disk on top of disks currently on peg."""
162     # init k to bottom-most position of disks on peg
163     k = 0
164     found = False
165
166     # assign disk to bottom-most position of peg with no disk
167     while k < len(disk_locations[peg]) and not found:
168
169         if disks[peg][k] == None:
170
171             disks[peg][k] = disk
172             found = True
173         else:
174             k = k + 1
175
176     return disk
177
178 def getNextAvailDiskLoc(peg, disks, disk_locations):
179
180     """Returns location for next disk on peg."""
181     # init k to bottom-most position of disks on peg
182     k = 0
183     found = False
184
185     # get bottom-most location of peg with no disk
186     while k < len(disks[peg]) and not found:
187
188         if disks[peg][k] == None:
189             loc = disk_locations[peg][k]
190             found = True
191         else:
192             k = k + 1
193
194     return loc
195
196 def moveDisk(from_peg, to_peg, disks, disk_locations):
197
198     """Moves disk on top of from_peg to top of to_peg."""
199     # get disk on top of from_peg
200     disk = removeTopDisk(from_peg, disks)
201
202     # get available loc for new disk on to_peg
202     new_loc = getNextAvailDiskLoc(to_peg, disks, disk_locations)
204
205     # move disk image
206     disk.goto(new_loc)
207
208     # add disk to to_peg list of disks
209     placeTopDisk(disk, to_peg, disks, disk_locations)
210
```

```
211 def towers(start_peg, dest_peg, spare_peg, num_disks, disks,
212             disk_locations):
213
214     """Begins the problem solving process."""
215     if num_disks == 1:
216         moveDisk(start_peg, dest_peg, disks, disk_locations)
217         time.sleep(.5)
218
219     else:
220         towers(start_peg, spare_peg, dest_peg, num_disks - 1,
221                disks, disk_locations)
222
223         moveDisk(start_peg, dest_peg, disks, disk_locations)
224         towers(spare_peg, dest_peg, start_peg, num_disks - 1,
225                disks, disk_locations)
226
227 # ---- main
228
229 # program welcome
230 print('This program solves the Towers of Hanoi for up to eight disks')
231
232 # init turtle screen
233 setup(width=800, height=600)
234
235 # init display parameters
236 peg_locations = ((-150, -50), (0,-50), (150,-50))
237 disk_separation = 24
238
239 # initpeg numbers
240 peg_A = 0
241 peg_B = 1
242 peg_C = 2
243
244 # get number of disks
245 num_disks = getNumDisks()
246 print()
247
248 # display pegs
249 displayPegs(peg_locations)
250
251 # create and register disk images
252 disk_images = getDiskImages(num_disks)
253 registerDiskImages(disk_images)
254
255 # calc locations for all disks
256 disk_locs = calcDiskLocations(num_disks, peg_locations,
257 disk_separation)
258
259
```

```
260  # create and position number of disks requested
261  disks = createDisks(disk_images)
262
263  # place all disks on start peg
264  placeDisksOnStartPeg(peg_A,  disks, disk_locs)
265
266  # make all disks visible
267  makeDisksVisible(peg_A,  disks)
268
269  # delay
270  time.sleep(2)
271
272  # start movement of disks
273  towers(peg_A, peg_C, peg_B, num_disks, disks, disk_locs)
274
275  # exit turtle screen
276  print('\nDone - Click screen to quit')
277  exitonclick()
```

FIGURE 11-27 Towers of Hanoi Program

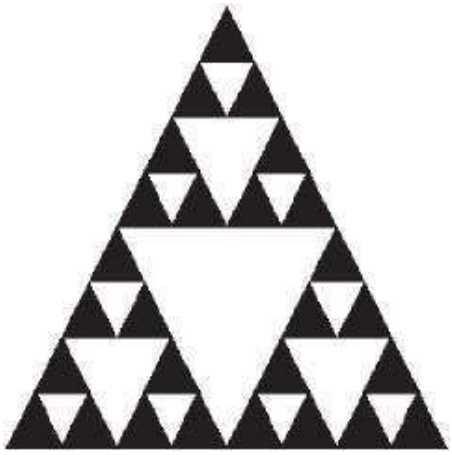
CHAPTER SUMMARY General Topics Python-Specific Programming Topics

Recursi ve Functions
Infinite Recursion
Factorial Function
Recursive Problem Solving MergeSort
Iteration vs. Recursion Fractals
Towers of Hanoi

CHAPTER EXERCISES Section 11.1
Recursive Functions in Python

1. For the following recursive functions, indicate which of the requirements of a properly designed recursive function each violates.
(a) def rfunc1(n):

return n 1 rfunc(n 2 1)
(b) def rfunc2(n):
if n 55 0:
return 1
return n 1 rfunc2(n 1 1)
2. For the following Sierpinski triangle, indicate how many times function drawSierpinskiTriangle is called in the Sierpinski triangle program of Figure 11-9.



Section 11.2

3. Give all the steps for MergeSort in sorting the following list.
11 4 16 9 5 12 2 14
4. Indicate the results produced by the recursive function MergeSort in Figure 11-16 if line 3 was replaced with, if len(lst) 55 0:

PYTHON PROGRAMMING EXERCISES
P1. Write both a nonrecursive and recursive function that determines if a given number is even or not.

P2. Write both a nonrecursive and recursive function that determines how many times a given letter occurs in a provided string.
P3. Write both a nonrecursive and recursive function that displays a provided string backwards.
P4. Write both a nonrecursive and recursive function that converts numbers to base 2.
P5. Write both a nonrecursive and recursive function that calculates the Fibonacci number for any positive integer, defined as follows,
fib(0) 5 0,
fib(1) 5 1,
fib(n) 5 fib(n 2 1) 1 fib(n 2 2)

P6. Write both a nonrecursive and recursive function that displays the rows of asterisks given below, **********
****
****
******

****
**

P7. Write both a nonrecursive and recursive function that displays the rows of asterisks given below, **
****
******
****
****
****

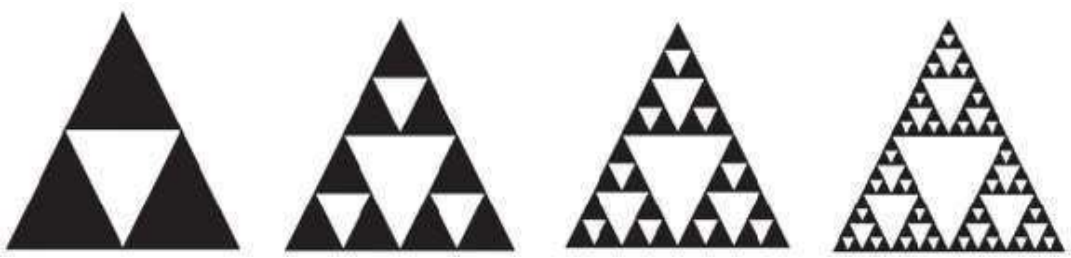
PROGRAM MODIFICATION PROBLEMS

M1. Towers of Hanoi: Reversal of Pegs
Modify the Towers of Hanoi program in section 11.4.3 so that the initial stack of disks is placed on peg C (as the start peg), and the final stack of disks is placed on peg A (the destination peg).

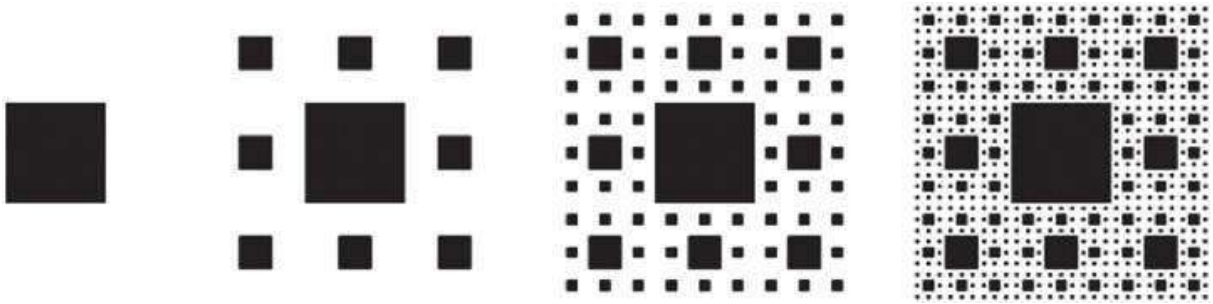
M2. Towers of Hanoi: Displayed List of Moves
Modify the Towers of Hanoi program in section 11.4.3 so that in addition to the movement of disks on the screen, the list of moves is displayed in the Python shell as given below,

Move disk from peg A to peg C
Move disk from peg A to peg B
Move disk from Peg C to peg A
etc.

M3. Sierpinski Triangle Program: Multiple Levels of Fractal Displayed
Modify the Sierpinski Triangle Program in section 11.1.3 so that it displays fours levels of the fractal as given below on the screen at once.

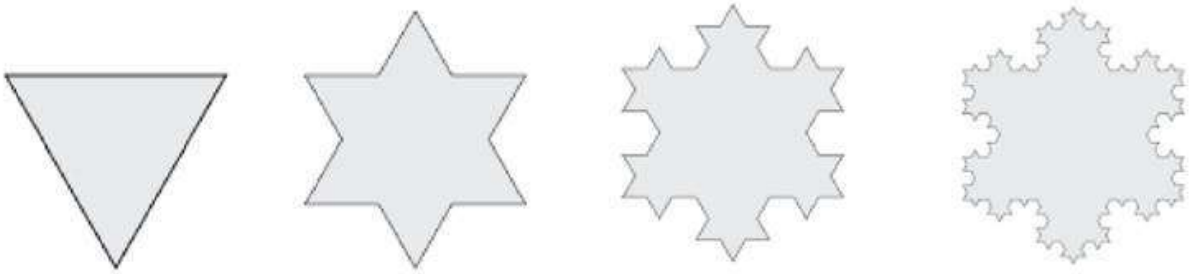


M4. Sierpinski Triangle Program: Modified for Creating Sierpinki Carpet
Modify the Sierpinski Triangle Program in section 11.1.3 so that it instead displays a Sierpinski carpet as the repeated pattern of a solid square surrounded by eight smaller squares as depicted below.



M5. Sierpinski Triangle Program: Modified for a Koch Snowflake
Modify the Sierpinski Triangle Program in section 11.1.3 so that a Koch snowflake fractal is generated instead. The progressive levels of a Koch snowflake fractal, like the Sierpinski triangle, are based on the repeated use of an equilateral triangle as given below,
1. Divide each side of the equilateral triangle into three segments.
2. Draw an equilateral triangle (of smaller size) with sides the length of the line segments above, with the

base placed at the middle line segment, pointing outwards.
3. Remove the line segment forming the base of the newly placed triangle.



PROGRAM DEVELOPMENT PROBLEMS

D1. Coin Change Program (Revisited)
Consider the Coin Change program in section 3.4.6 (for children learning to count change) in which the user had to enter a combination of coins that added up to a specified amount between 1 and 99 cents. Develop and test a program based on the use of recursion that “turns the tables” in which the user enters an amount between 1 and 99 cents, and the program must determine a set of coins

that adds up to that amount using the least number of coins.

D2. Palindrome Checker (Revisited)
In Chapter 7 (section 7.3.7) a Palindrome Checker program was given based on the use of a stack that was also developed. Develop and test a Palindrome Checker program based on the use of recursion, instead of a stack.

D3. Phone Number Spelling Program (Revisited)
In Chapter 9 (section 9.1.2) a Phone Number Spelling program was given based on the use of deeply nested for loops. Develop and test a Phone Number Spelling program based on the use of recursion instead.

D4. Creative Designs
Using your imagination, develop and test a program using recursion that generates a fractal of your own design. As with the example fractals in the chapter, start with a basic geometric shape, and define the next level of the fractal pattern containing two or more smaller versions of the original-size shape.

D5. Develop a program that uses recursion to solve the Man, Cabbage, Goat, Wolf problem from Chapter 1.

CHAPTER 12 Computing and Its Developments

In this final chapter, we look at the many developments and innovations that have led to present-day computing. This includes advancements in computer hardware, computer software, and computer networks, as well as theoretical developments that underlie our understanding and effective utilization of computing.

OBJECTIVES
After reading this chapter you will:
♦ Be knowledgeable about many of the developments in computing ♦ Become familiar with some of the most notable individuals in the field ♦ Be able to describe the four generations of computer technology

CHAPTER CONTENTS

12.2 Developments Leading to Electronic Computing

First-Generation Computers (1940s to mid-1950s)
12.3 The Early Groundbreakers
12.4 The First Commercially Available Computers Second-Generation Computers (mid-1950s to mid-1960s)
12.5 Transistorized Computers
12.6 The Development of High-Level Programming Languages Third-Generation Computers (mid-1960s to early 1970s)
12.7 The Development of the Integrated Circuit (1958)
12.8 Mainframes, Minicomputers, and Supercomputers

491

Fourth-Generation Computers (early 1970s to the Present)
12.9 The Rise of the Microprocessor
12.10 The Dawn of Personal Computing
The Development of Computer Networks
12.11 The Development of Wide Area Networks
12.12 The Development of Local Area Networks (LANs)
12.13 The Development of the Internet and World Wide Web

CONTRIBUTIONS TO THE MODERN COMPUTER
12.1 The Concept of a Programmable Computer
12.1.1 “Father of the Modern Computer”—Charles Babbage (1800s)

FIGURE 12-1 Charles Babbage

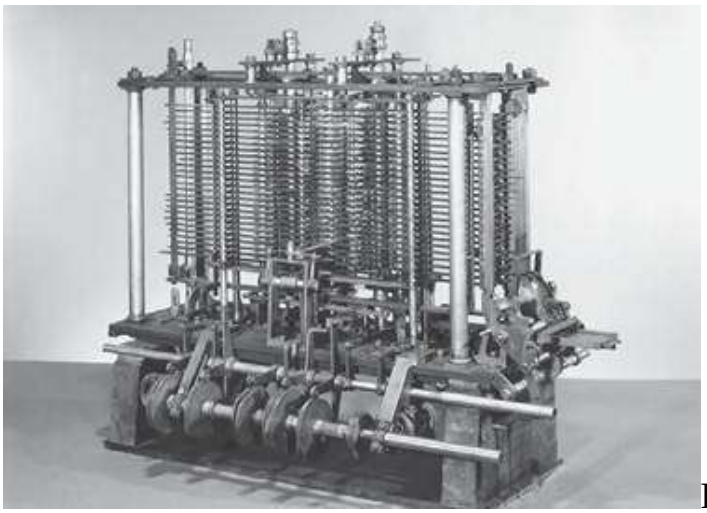
In London, England, during the 1800s, **Charles Babbage** (Figure 12-1) worked on a couple of different designs for a calculating machine. The first, called the "Difference Engine," was designed to perform only certain calculations. It was to be powered by steam with a built-in "printer" that would punch out calculation tables on metal plates, important for British navigation. A partial prototype was finished in 1822. Because of technical, financial, and other problems however, his continued attempts over the next nearly twenty years to finish its construction ended in failure. If completed, it would have measured ten-feet by tenfeet by five-feet, and would have weighted two tons.

Another machine that Babbage envisioned, however, would lead to his historical imminence—the **Analytical Engine** (Figure 12-2). He completed the first workable prototype in 1837,

working on its completion until his death in 1871. A major conceptual breakthrough of the Analytical Engine was that the calculations it performed were based on a set of instructions fed to it. Therefore, it could be "programmed" to solve *any* mathematical problem, not just certain problems as with the Difference Engine. As designed, it would have been fifteen feet tall and twenty-five feet long; about the size of a small locomotive train.

The Analytical Engine was never completed due to the limited technology of the time. Its design, however, is considered one of the greatest intellectual achievements of the nineteenth century. It contained the two fundamental components of current-day computers: a *mill* (a kind of central processing unit for executing instructions, fed as punched cards), and a *store* (a kind of memory or storage area). Because of Babbage's foresight, he has been given the de

served name "Father of the Modern Computer."

 FIGURE 12-2 The Analytical Engine

Charles Babbage is considered the “Father of the Modern Computer” for his work on the Analytical Engine in the 1800s.

12.1.2 “The First Computer Programmer”—Ada Lovelace (1800s)

Augusta Ada Byron (Figure 12-3), Countess of Lovelace (**“Ada Lovelace”**) and daughter of the poet Lord Byron, was a talented mathematician, who first met Charles Babbage in 1833. She was intrigued with his work, and inspired Babbage to continue with his Analytical Engine. She helped write widely published scientific articles on the machine and Babbage’s ideas. She showed her insightfulness by predicting that such machines were capable of processing not only numbers, but any encoded information. For example, she gave predictions that someday these machines would be able to produce graphics, compose music, and have scientific use (beyond mere scientific calculation). Because of her foresight, she is credited with being the “First Computer Programmer.” The **Ada programming language** is named in her honor.


FIGURE 12-3 Ada Lovelace

Ada Lovelace is considered the “First Computer Programmer” for her work, insight and writings on the Analytical Engine.

12.2 Developments Leading to Electronic Computing 12.2.1 The Development of Boolean Algebra (mid-1800s)

We have already mentioned George Boole (Figure 12-4) in Chapter 3 in our discussions of Boolean algebra. His development of what we now call Boolean algebra, and the corresponding Boolean operators AND, OR, and NOT was conceived as a means of mathematically proving the answers to any true/false question. Later work showed that there are always some questions that cannot be answered mathematically. However, Boolean algebra became greatly applicable one hundred years later. It provides the logical foundation for the design of digital logic circuits of modern electronic computers.

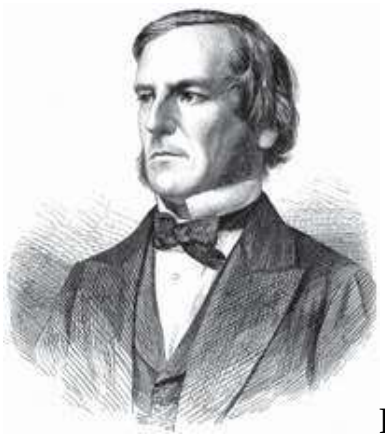
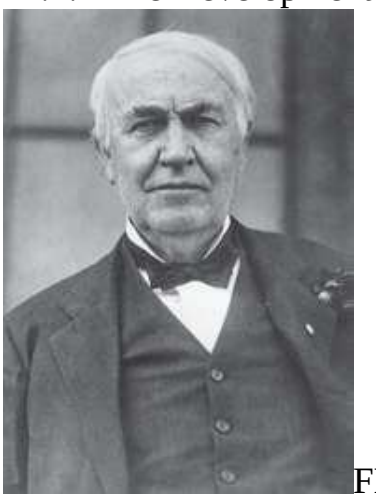


FIGURE 12-4 George Boole

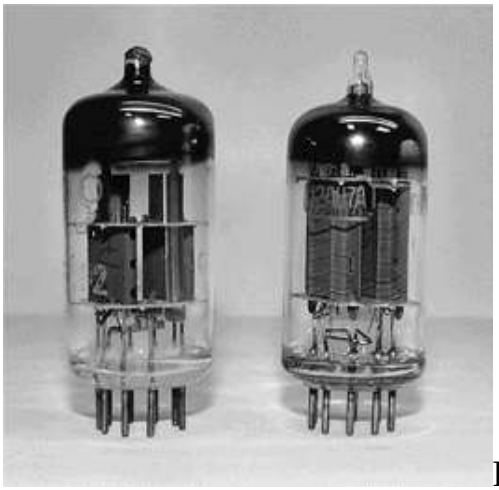
The development by **George Boole**, of what is now called Boolean algebra,

provided the logical foundation for the development of digital circuits.

12.2.2 The Development of the Vacuum Tube (1883)

 FIGURE 12-5
Thomas Edison

In 1883, **Thomas Edison** (Figure 12-5) invented the **vacuum tube** (Figure 12-6), an electronic device containing electrodes that electrons can travel between. The vacuum tube became the building block for the entire electronics industry at the time. A version of the device can be used as an electronic switch, which became a critical component in the development of the first modern electronic computers.

 FIGURE 12-6 Vacuum Tubes
Thomas Edison invented the vacuum tube, which became the first form of electronic switch in electronic computers.

12.2.3 The Development of Digital Electronic Logic Gates (1903)

Serbian-born **Nikola Tesla** (Figure 12-7), who once worked for Thomas Edison,

patented in 1903 the **electronic logic gate**. Such gates, composed of a small number of vacuum tubes as electronic switches, could electronically execute the logical operators of Boolean algebra—AND, OR, and NOT. Two different electrical signals representing either true or false were input to the gate, with an output electrical signal representing the true/false result, as shown in Figure 12-8. Logic gates would become fundamental to all electronic digital computers to come, the idea to be rediscovered decades later.

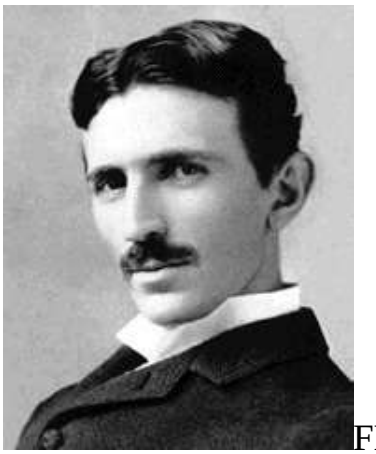


FIGURE 12-7 Nikola Tesla

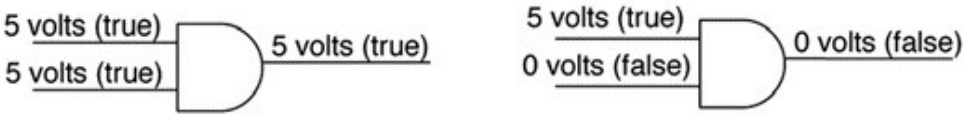


FIGURE 12-8 AND Logic Gate

Nikola Tesla patented the electronic logic gate, which would become a fundamental component of all electronic computers.

12.2.4 The Development of Memory Electronic Circuits (1919)

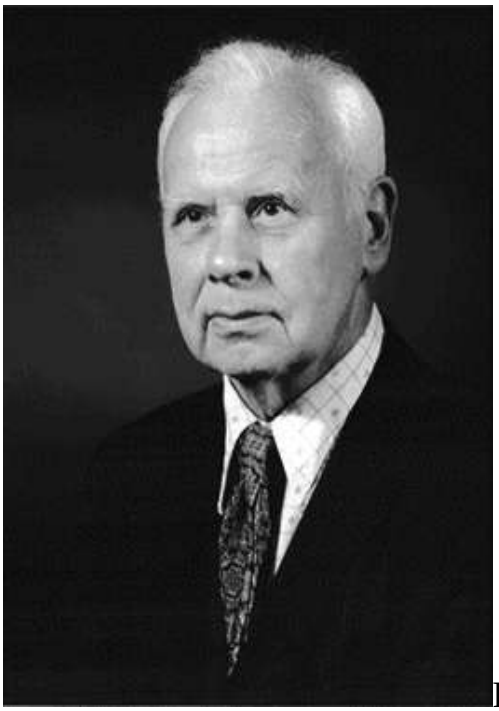
American physicists **W. H. Eccles** (Figure 12-9) and **R. W. Jordan** in 1919 invented the **flip-flop** electronic switching circuit. These devices, based on use of vacuum tubes, were able to “flip and flop” from one stable state to another. Such a device is always in one of two states, thus provides a form of electronic storage, or “memory.” Flip-flop electronic components, like Tesla’s electronic logic gates, would become key components in the development of future digital electronic computers.

 FIGURE 12-9
W. H. Eccles
W. H. Eccles and **R. W. Jordan** invented the flip-flop electronic switching circuit to become a fundamental component of all electronic computers.

12.2.5 The Development of Electronic Digital Logic Circuits (1937)

In 1937, **George Stibitz** (Figure 12-10) of Bell Laboratories, working on his kitchen table one night created the first digital electronic logic circuit. A digital electronic circuit is a circuit built out of electronic logic gates that can execute arithmetic or logical operations. Because such circuits are built out of combinations of logic gates, they are referred to as **combinatorial circuits**. Boolean algebra provides a mathematical basis for denoting, analyzing, and understanding such circuits.

The digital logic circuit that Stibitz built was for adding two binary digits, called a (full) **binary adder**. Such an adder is a fundamental component of all modern computers. Stibitz's work demonstrated the feasibility of building electronic logical and arithmetic circuits from the basic building blocks of electronic logic gates that Tesla earlier developed. This development, along with the flip-flop electronic circuit of Eccles and Jordan for the electronic storage of information, completed the technology needed for the design and construction of a fully electronic computer.

 FIGURE 12-10 George Stibitz

George Stibitz created the first digital electronic logic circuit, constructed out of a combination of electronic logic gates.

12.2.6 “The Father of Information Theory”—Claude Shannon (1948)

 FIGURE 12-11 Claude Shannon

While many of his colleagues were working on the hardware design of modern computers, **Claude Shannon** (Figure 12-11) at Bell Laboratories, worked on a theory of information to be processed by such machines. In 1937, at the age of 21, Shannon wrote a master’s thesis that showed how Boolean algebra can be simulated electronically by simple electronic switches to perform both logical and numerical calculations—the fundamental concept of digital computing

today. Some considered Shannon's result possibly the most important master's thesis of the twentieth century.

Later, in 1948, Shannon published a paper that laid the foundation for what would become the field of information theory. Shannon's *Fundamental Theorem of Information Science* states that *all* information can be represented by use of only two symbols, "0" and "1," which he called **bits** (b inary dig its). It has been said that the digital revolution began with Shannon's work.

It was therefore clear that electronic switches could both be used for the design of digital logic circuits (e.g., a circuit for adding two numbers), and for the storage of information of any kind, not only numerical values (as Ada Lovelace foresaw). Given Shannon's demonstration of the sufficiency of binary encoding for encoding all information, the stage was set for the development of electronic binary digital computing as we know it today. For his work, Shannon is known as the "Father of Information Theory."

Such binary coding of information was already in practical use for the transmittance of messages as audible electronic signals over wire. Earlier, in 1837, **Samuel Morse** (Figure 12-12) patented his design of a telegraph, which could electronically send messages (text) in the form of two signals, dots (e.g. short beep) and dashes (long beep), called **Morse Code**. Since only two sig

nals were used, this represented a form of binary encoding.

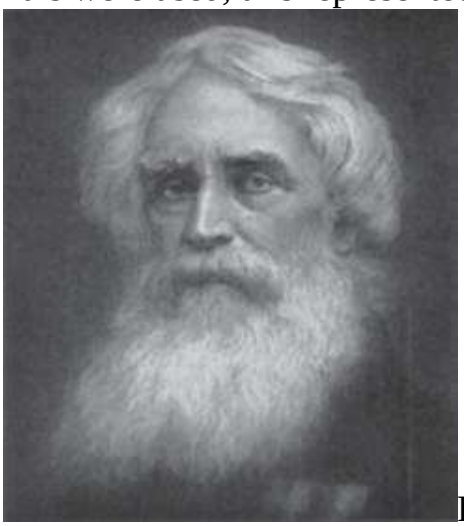


FIGURE 12-12 Samuel Morse

Claude Shannon developed the *Fundamental Theorem of Information Science,* stating that *all* information can be represented by only two symbols, "0" and "1,"

which he called bits (binary digits).

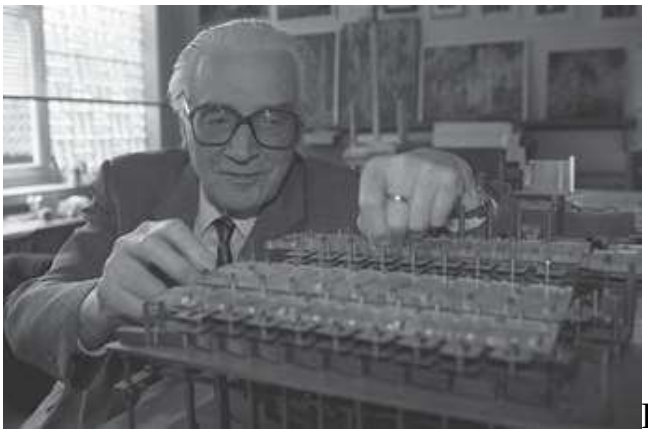
FIRST-GENERATION COMPUTERS (1940s–mid-1950s)
12.3 The Early Groundbreakers

12.3.1 The Z3—The First Programmable Computer (1941)

Konrad Zuse (Figure 12-13) built a number of binary computing devices starting in 1935, at a time when other computers were being built based on the decimal system. His initial machines were the **Z1**, the first mechanical binary digital computing device, and the **Z2**, the first fully functioning electromechanical computing device, completed in 1939 as World War II was approaching. Each of these devices had limited programmability, however.

While his design of binary computers with the Z1 and Z2 was significant, his most notable accomplishment was the development of the **Z3**, the first working electromechanical *programmable* computer, completed in 1941. The Z3, like the Z2, was an electromechanical device. After serving in the German army, he realized the computing potential of a fully electronic version of his computing devices. He submitted a proposal to the German army for funding for his next generation of machine, a fully electronic calculator, the **Z4**.

A fully electronic machine would have been thousands of times faster than his electromechanical versions. It would therefore have significant military use, such as for aircraft design and breaking coded messages. (We shall see that the Allies had crucial success here.) He estimated the project would take two years. His proposal was flatly turned down by the German Army Command since it was believed that Germany would soon win the war (which, of course, they lost a few years later in 1945). He went on to build the Z4 after the war, completing an electromechanical version in 1949 to become the world’s first commercially available digital computer. He even wrote his own programming language for the Z4, called *Plankalkul*, with features of modern-day languages that was definitely ahead of its time.

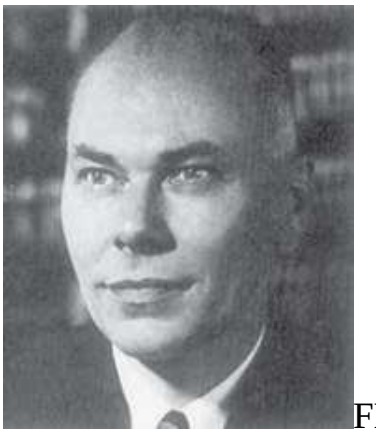
 FIGURE 12-13 Konrad Zuse

Konrad Zuse developed the first working programmable computing device, the **Z3**, an electromechanical binary digital computer, as well as the first commercially available electronic computer, the **Z4**.

12.3.2 The Mark I—First Computer Project in the United States (1937–1943)

Howard Aiken (Figure 12-14), a mathematics instructor at Harvard University, began a project in 1937 to build a general-purpose (i.e., programmable) electromechanical calculating machine. It was the first project for the development of a modern computer in the United States (developed without knowledge of the work of Konrad Zuse). Aiken, as opposed to most others in the early computing field, knew of Babbage's work, and therefore saw himself as following through with Babbage's unfulfilled dream. Now, however, Aiken had available current-day technology that Babbage did not.

The technology available at the time was electromechanical, making use of relay switches that mechanically switched on and off by control of electrical signals. The machine, therefore, was not a fully electronic computer, and thus limited in speed by the speed of the mechanical switching of the relay switches. IBM agreed to fund the project and supplied an engineering team to build the machine following Aiken's design. When the machine was finally finished in 1943, it was eight feet tall, fifty-one feet long, two feet thick, and weighed

 FIGURE 12-14 Howard Aiken

five tons (Figure 12-15). It read instructions from (punched) paper tape, and data from punched cards.

Ironically, true to technological developments of today, the Mark I was obsolete the moment it was completed, since fully electronic computers were just on the horizon. It was, however, very newsworthy at the time, and a big press release was given to announce its completion. The Mark I turned out to be very reliable compared to other electronic computers. It was able to run 24 hours a day, seven days a week, and therefore was very productive. It was used over a period of 16 years. Its development has been called the “real dawn of the computer age” (in the United States).

 FIGURE 12-15 Harvard Mark I Electromechanical Computer

The **Mark I** computer, designed by **Howard Aiken** at Harvard, was the first project for the development of a modern computer in the United States. Other, more advanced computers, however, were built before the completion of the Mark I.

12.3.3 The ABC—The First Fully Electronic Computing Device (1942)

 FIGURE 12-16 John Atanasoff

J **ohn V. Atanasoff** (Figure 12-16), a physicist at Iowa State University, completed a design of a calculating device between 1939 and 1942 with graduate student **Clifford Berry**, called the **ABC** (Atanasoff-Berry Computer). It was the first to use vacuum tubes both for calculations (for the logic circuits) and for memory storage. Therefore, it was the first fully electronic computing device. However, since it was designed to solve only particular types of mathematical problems and could not be programmed, it was not a general- purpose computer.

The ABC was a binary computer, designed to store numerical values in base 2 (i.e., binary notation), and not base 10 as had all other machines previously built, with the exception of the computing devices developed

by Konrad Zuse in Germany. Because of the war, neither Atanasoff nor Zuse knew of the others' work. Therefore each independently hit upon the novel idea of a binary computer at essentially the same time.

A functioning prototype was finished in October 1939 that could do simple addition and subtraction in binary of eight digit decimal values. In 1942, the final version was completed and tested (Figure 12-17). It was the size of a desk, weighed 700 pounds, had over 300 vacuum tubes, and contained a mile of wire.

FIGURE 12-17 The ABC Was the First Fully Electronic Computing Device

After testing of the machine was completed, Atanasoff worked on assignments for the war effort, and no further development or use of the machine followed. Given that the ABC was not a programmable device, it lacked a key feature of other machines at the time, but is credited as the first fully electronic (binary) computing device.

John V. Atanasoff and **Clifford Berry** developed the **ABC** (Atanasoff-Berry Computer), a fully electronic binary computer. It was not, however, a programmable device.

12.3.4 Colossus—A Special-Purpose Electronic Computer (1943)

FIGURE 12-18 The Enigma (During World War II)

At the start of World War II, in 1939, Great Britain had decided that it would call upon the greatest mathematical and scientific minds available to break the ciphers that Germany was using for top-secret communications. The Enigma (Figure 12-18), developed at the end of World War I, generated complex coding

of messages based on constantly changing alphabetic substitutions. It was believe by the Germans to be an unbreakable code.

A secret project was begun in Great Britain to build a machine capable of breaking the Enigma code. The electromechanical machine was called **Bombe** (Figure 12-19).

This allowed, in particular, the breaking of German naval codes, and resulted in the significant reduction of shipping losses across the Atlantic to the United States, essential for the allied support. Later, however, a second generation of encryption was developed by the Germans that was significantly more complex, using a new encryption called Lorenz code. A second new, fully electronic machine was designed and built for breaking this new code.

The machine was called the **Colossus**, a fully electronic binary machine, completed at the end of 1943. A number of machines of this design were built and used (Figure 12-20). A second version, the Colossus II, was finished on June 1, 1944. A few days after its completion, the Allies were able to intercept and decode a message from the German High Command. The message indicated that the Germans had fallen for misinformation put out by the Allies that their landing point was an area called Calais, while in fact, their planned landing location was the beaches of Normandy. As a result, commander Dwight D. Eisenhower decided to go ahead with the Normandy invasion—the infamous D-Day—on June 6, 1944 (less than a week after the Colossus II was finished). It is believed by many that the ability to break German codes brought the turning point of the war in the Allies favor.

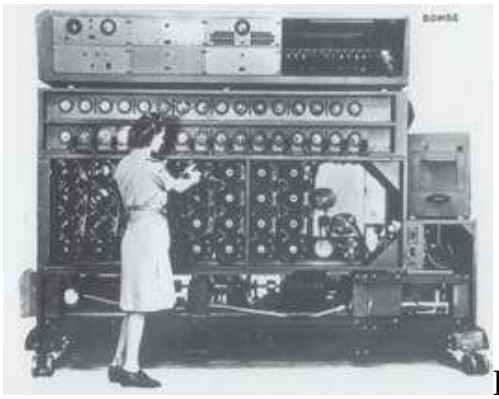
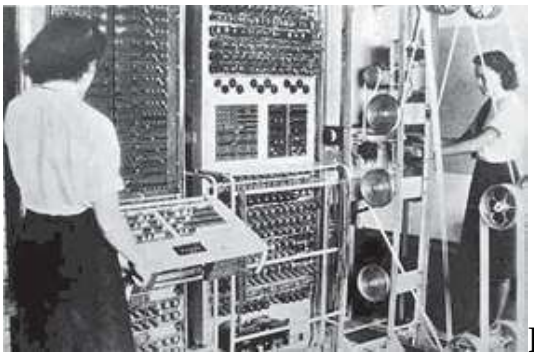


FIGURE 12-19 Bombe

 FIGURE 12-20 The Colossus
The **Colossus**, built in Britain to break German codes during World War II, is believed by many to have turned the war in the Allies, favor.

12.3.5 ENIAC—The First Fully Electronic Programmable Computer The U.S. Army's Need for a Fully Electronic Computer (1943)

When the United States entered World War II in 1941, there was a sudden increased need of people who calculated ballistic firing tables, so-called "human computers" (Figure 12-21). In order to hit their target, gunners needed to aim their weapon at the proper angle and direction based on a number of factors including the distance of the target, the wind speed, wind direction, and air temperature. Gunners depended on such firing tables to find the correct angle of fire for all the current conditions for all the different ballistic missiles.

At the U.S. Army's Ballistic Research Laboratory at Aberdeen Proving Ground in Maryland, hundreds of human computers (mostly female mathematicians) were employed to calculate such tables. Two to four thousand possible trajectories had to be calculated for each pair of projectile and gun. Each human computer, using an electromechanical desk calculator, took almost three days to compute a single trajectory! By 1943, the Army was not able to keep up with the calculation of all the firing tables needed, and was desperately in need of a solution. Thus, the Army was willing to fund the construction of a fully electronic computer capable of solving this problem.

 FIGURE 12-21 “Computers” During World War II

In 1943, the U.S. army was in desperate need of a means of computing the large number of firing tables needed for various ballistic missiles.

The Development of the ENIAC at the University of Pennsylvania (1945)

In August 1942, an assistant professor at The Moore School of Electrical Engineering at the University of Pennsylvania, **John W. Mauchly** (Figure 12-22) had written a paper on “The Use of High Speed Vacuum Tube Devices for Calculating.” While his idea was realized by others, he understood the kinds of computing speeds that such an electronic switching device could produce over electromechanical machines—tens of thousands of operations per second. He, along with a young electronics engineer named **Presper Eckert**, was awarded a contract by the U.S. Army in June 1943 to build their electronic calculating machine, with an agreement to build a duplicate of the machine at Aberdeen Proving Grounds. (The size of computers then did not allow them to be moved around— they were permanently located!)

FIGURE 12-22 John Mauchly and Presper Eckert

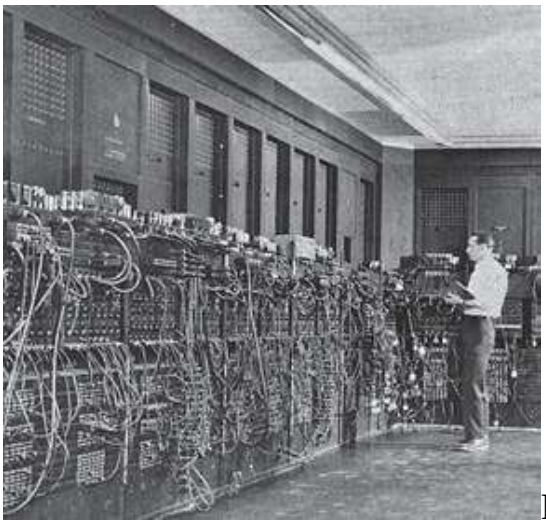
The **ENIAC** (for “Electronic Numerical Integrator and Computer”), shown in Figure 12-23, was finished in November 1945 (too late for its originally intended purpose) about three months after the end of the war.

It was eight feet high, eighty feet long, and weighed 30 tons. It used decimal notation, and thus was not a binary computer. When finished, however, it was roughly a thousand times faster than any other computer at the time. Whereas existing specialized calculators took fifteen to thirty minutes to compute the trajectory of a ballistic missile, the ENIAC took only twenty seconds, capable of 5,000 ten-digit additions per second.

As a demonstration of ENIAC’s general-purpose functionality, the ENIAC was reprogrammed to do crucial calculations on the development of the first hydrogen bomb (an advanced weapon of the atomic bomb was used in World War II). Programming on the ENIAC, however, did not mean sitting down and typing lines at a keyboard as is done today. It had to be physically reprogrammed by changing switches and reconnecting a patchwork of wires.

The data for the currently configured program was submitted on punch cards. Its biggest problem was the unreliability of the almost 18,000 vacuum tubes it contained. In fact, the reason that base ten representation was chosen over binary was that the binary approach would have taken many more such vacuum tubes. Although ENIAC proved itself to be extremely fast at the time, the unreliability of the vacuum tube was to remain one of its shortcomings.

Because of intellectual property rights disputes in 1946 with the University of Pennsylvania, John Mauchly and Presper Eckert decided to leave the university to form the Eckert-Mauchly Computer Corporation. They believed in the potential of commercial computers, whereas others did not see the need for more than one or two computers in the whole country. They went on to design and build one the first commercial computer in the United States, the UNIVAC I (shown in Figure 12-22).

 FIGURE 12-23 The ENIAC Computer
John Mauchly and **Presper Eckert** developed the ENIAC, the first fully-electronic programmable computer in the U.S.

12.3.6 EDVAC/ACE—The First Stored Program Computers (1950)

Even before Eckert and Mauchly had completed the design of the ENIAC, they conceived of the idea of a stored-program computer. In the ENIAC, only the data was stored in memory (via punched cards). To execute diffferent programs, the ENIAC had to be rewired, and later was reprogrammed through a series of switches. In a stored-program computer, the program need only be entered once and stored in memory. This concept was an important next step since high-speed computers like the ENIAC were slowed by the need to be reprogrammed this way.

In 1944, Eckert and Mauchly received another contract from the army's Ballistic Research Laboratory, this time for the development of a stored-program computer to be called EDVAC. A distinguished mathematician from Princeton named **John von Neumann** (Figure 12-24) joined the project. Even though Eckert and Mauchly

 FIGURE 12-24 John von Neumann

had earlier conceived of the idea of a stored- program computer, von Neumann developed a method of how it could work, and thus is generally cited as the originator of the idea. The *von Neumann machine* remains the predominant means of stored program execution today.

John von Neumann is credited with the concept of a stored-program computer. The first such computer built was called **EDVAC**. Its method of executing stored programs is utilized in essentially all computers today, called a **von Neumann machine**.

The British Stored-Program Computer—ACE (1950)

British scientists visited the Moore School of Electrical Engineering at the University of Pennsylvania soon after the end of World War II. On return to Britain, they worked on their own stored program computer named **ACE** (for "Automatic Computing Engine"). The lead designer of the project was one of the most influential and significant individuals in the history of computing, **Alan Turing**, who also played a key role in breaking German codes of World War II (Figure 12-25). A scaled-down version of the storedprogram machine was completed in 1950 (before the completion of the EDVAC).

 FIGURE 12-25 Alan Turing

After the development of the EDVAC computer, Great Britain began worked on their own storedprogram computer named **ACE** (for “Automatic Computing Engine”). The lead designer of the project was one of the most influential and significant individuals in the history of computing, **Alan Turing**.

12.3.7 Whirlwind—The First Real-Time Computer (1951)

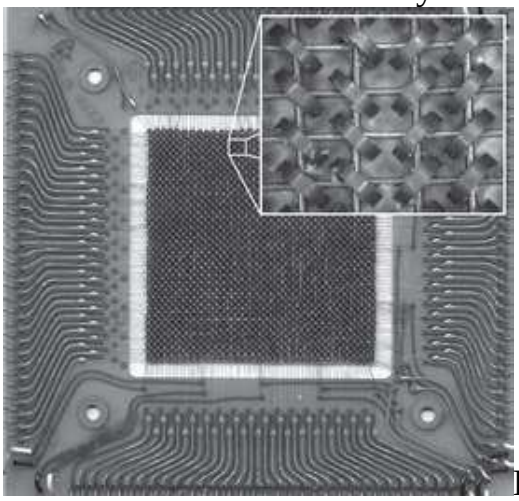
In December 1944, the U.S. Navy asked MIT to do a feasibility study of the development of a specialpurpose flight trainer to train pilots. This required the machine to be capable of real-time processing— that is, to compute fast enough to instantaneously respond to the actions of training pilots. A young gifted engineering graduate student, **Jay W. Forrester**, was offered the project. He changed the goal of the project from the development of a special-purpose device, to the design and development of a general electronic digital stored-program computer that could operate in real time.

The computer that Forrester developed was called **Whirlwind** (Figure 12-26). Although it operated in real time, it had one serious problem. The method of memory storage, consisting of thirty-two cathode ray tubes that rarely lasted more than a month, was unreliable, often putting the machine out of service.

 FIGURE 12-26 The Whirlwind Computer

Forrester revolutionized the technology of memory storage by using magnetically charged, doughnut-shaped ceramic ferrite “cores” that could be electronically magnetized in a clockwise or counterclockwise direction (to represent “1” and “0”), shown in Figure 12-27. The cores were configured in a grid form, threaded with cross sections of wires so that any given core could be read or written by selection of the proper “vertical and horizontal” wires. The memory board in the figure is 6.5 inches square, and contained a storage capacity of 1024 bits. The term *core* is still used today—mainly in the term *core dump* referring to the raw display, or “dump” of the contents of main memory—even though computers no longer use magnetic core memory.

Besides the speed of this form of memory, another important feature was that each bit (or core) of memory could be read and written in the same amount of time. This was called *random access memory* (RAM), an essential characteristic of computer memory today. Core memory technology remained the main memory technology for almost thirty years.

 FIGURE 12-27 One Plane of Magnetic Core

Memory
The Whirlwind, developed by **Jay W. Forrester**, was the first real-time computer, also introducing the use of core (random access) memory.
12.4 The First Commercially Available Computers

A new stage of computer history began when companies started manufacturing commercial computers. This was the first time that computers were not thought of as unique, one-of-a-kind machines, but as particular *model* computers. A new industry was beginning that many people did not see coming. This is reflected in a quote by Thomas Watson, president of IBM in 1943: “I think there is a market for maybe five computers in the world.” Two of the more notable commercially available computers first appeared at essentially the same time—the UNIVAC (mentioned earlier) in the United States, and the LEO, in Great Britain (discussed in the following).

12.4.1 The Struggles of the Eckert-Mauchly Computer Corporation (1950)

Eckert and Mauchly needed financial backing to produce their computer, the **UNIVAC I**. The Census Bureau was very interested in the development of this machine, and in 1946 gave the company their first contract to build a UNIVAC. However, the amount agreed on was not enough to cover all the development cost. By 1948, Eckert and Mauchly had five UNIVAC contracts but no completed machine, selling out to the Remington Rand Corporation in early 1950. Finally, in March 1951, the first UNIVAC (Figure 12-28) was built and delivered to the U.S. Census Bureau. Eventually, forty-six UNIVAC I computers were built and sold. The UNIVAC became very publicly known, appearing in cartoons and movies of the time. One notable event was when it was used on live TV to predict the U.S. presidential election

 FIGURE 12-28 The UNIVAC I

between Dwight D. Eisenhower and Adlai Stevenson in the 1952 election (Figure 12-29).

Most pollsters had predicted a close election race. However, UNIVAC, using data from past elections, predicted a landslide win for Eisenhower. Given the unbelievability of the results, they did not announce this prediction to the TV viewers. As it turned out, Eisenhower won by one of the biggest landslides in history. Afterwards, later that evening, news commentator Walter Cronkite had to admit that UNIVAC had earlier made the right predication— “UNIVAC was right, we were wrong!”

FIGURE 12-29 UNIVAC Predicting the 1952 Presidential Election

The UNIVAC I was the first commercially available computer in the U.S.

12.4.2 The LEO Computer of the J. Lyons and Company (1951)

 FIGURE 12-30 The J. Lyons and Company

The J. Lyons and Company of London, England, was one of the largest catering and food manufacturing companies in the world (Figure 12-30). After a trip to the United States in 1947 by two managers of the company, they realized that electronic computers would hold the key to improved efficiency of clerical procedures. Given that there was an ongoing computer project at Cambridge University, they offered some funding to the university for the project in return for advice on how to build their own computer. A recent Ph.D. graduate of Cambridge who had worked on the computer project there joined the Lyons company, and a team of technical employees was hired.

Work started on the construction of the computer, named **LEO** (for “Lyons Electronic Office”), started in January 1949 (Figure 12-31). The computer became operational in September 1951, just six months after the construction of the first UNIVAC computer in the United States. However, it took a couple more years before the machine was reliable enough

to market. It became, however, the “world’s first business computer.”

Soon, many companies heard of the machine, and it became a big success. At first, Lyons began “renting out” time on their machine. Eventually, Lyons started building machines for sale to others, and thus became, in addition to a food service company, a computer manufacturer. Various models of the LEO

computer sold moderately well until the 1960s, when American-built computers began dominating the UK computer market. Great Britain was soon after that out of the computer manufacturing business.

 FIGURE 12-31 The LEO (“Lyons Electronic Office”) I Computer

12.5 Transistorized Computers 505

The **LEO** (“Lyons Electronic Office”) was the first commercially available computer in the United Kingdom, with its production beginning in 1951.

SECOND-GENERATION COMPUTERS (mid-1950s to mid-1960s)

12.5 Transistorized Computers

12.5.1 The Development of the Transistor (1947)

The **transistor**, developed by **William B. Shockley**, **John Bardeen**, and **Walter H. Brattain** at Bell Telephone Laboratories in December 1947, is a solid-state, semiconductor device that enables the switching of electrical circuits “on” (e.g., “1”) and “off (“0”). Thus in combination, transistors can be used to create logic circuits (such as the addition of two numbers). A transistor is referred to as a **solid-state device** because it is composed of solid material, in contrast to the previous switching technology of the vacuum tube. Vacuum tubes are similar to light bulbs in size, generate significant heat, and eventually “burn out,” needing replacement. Solid-state devices do not burn out, do not generate significant heat, can be made arbitrarily small, and do not draw much power. A comparison of these two devices is shown in Figure 12-32.

Transistors are referred to as **semiconductors** because their electrical conductivity lies between that of insulators (like rubber) and conductors (like copper). The degree of conductivity can be ectronically altered, which is what makes them a suitable electronic switching device, as shown in Figure 12-33.

FIGURE 12-32 The Vacuum Tube vs. the Transistor

ECEC

current flows through positive current applied[B]
to base lead

negative current B barrier prohibits applied to base lead[current flow]

Transistor Switched On
(E – emitter C – collector B – base lead) FIGURE 12-33 Transistor Switching Devices

Transistor Switched Off
(E – emitter C – collector B – base lead)

Transistors operate through three connections, referred to as the *emitter*, the *collector*, and the *base lead*. The flow of current from emitter to collector is strictly determined by whether there is currently a positive voltage or negative voltage applied to the base lead. When a positive voltage is applied to the base lead, the material between the emitter and the collector becomes conductive, and electrons are free to flow, thus turning the connection “on.” Alternatively, when a negative voltage is applied to the base lead, material between the emitter and collector becomes nonconductive, preventing electrons from freely flowing, thus turning the connection “off.” These simple switching devices can be used in arbitrary combinations in order to produce any given digital logic combinatorial circuit desired. William B. Shockley, John Bardeen, and Walter H. Brattain (Figure 12-34) received the Nobel Prize in Physics in 1956 for their discovery of the transistor effect.

FIGURE 12-34 Bardeen, Brattain, and Shockley (Nobel Prize in Physics 1956)

The transistor is a solid state electronic switching device developed by **William B**. **Shockley**, **John Bardeen**, and **Walter H. Brattain**.

12.5.2 The First Transistor Computer (1953)

The world's first transistorized computer was completed in 1953 at the University of Manchester in the United Kingdom (Figure 12-35). A computer using all transistors as electronic switching devices rather than the vacuum tube standard at the time had a number of advantages, as mentioned above transistors do not burn out like they took much less space, and did not generated significant heat as compared to vacuum tube technology. In 1962, a transistorized computer that was developed by this research group was the fastest

in the world.

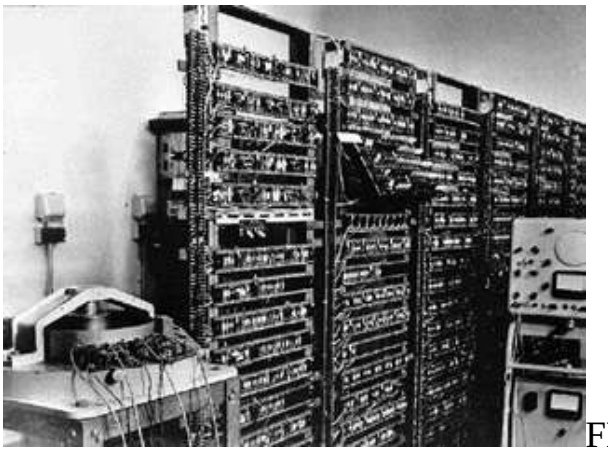


FIGURE 12-35 The First Transistorized Computer

The first fully-transistorized computer was completed at the University of

Manchester in 1953.

12.6 The Development of High-Level Programming Languages 12.6.1 The Development of Assembly Language (early 1950s)

In the mid-1950s, all of the computers at the time were extremely tedious to program. They needed to be programmed in either *machine code* or *assembly language*. **Machine code** (or **machine language**) is a numerical code and is the "native" language of the machine. For example, the numerical code "1001" might represent an add operation. **Assembly language** is a symbolic notation using

what are called *mnemonics* in place of the numerical codes of machine language —"add" might be used in place of "1001".

Although assembly language is somewhat better than machine code, it is still a low-level language. *Low level* means that each instruction performs a very simple task. For example, to accomplish the assignment A 5B 1C takes a number of assembly language instructions to accomplish. Assembly language represents some improvement over numerical machine code notation. However, programming in assembly language is still very tedious. Thus, *high-level* programming languages were developed, discussed next.

Assembly language is a symbolic notation using what are called *mnemonics* in place of the numerical codes of machine language.

12.6.2 The First High-Level Programming Languages (mid-1950s)

The use of assembly language, although an improvement over machine code, was only of minor help. Therefore, better programming languages needed to be designed. One of the most influential individuals in the history of computer science was **Grace Murray Hopper** (Figure 12-36). In 1951, working at Remington Rand (maker of the UNIVAC computer), she conceived a new type of programming language that could help "automate" the task of programming. The language was called a **high-level language** since the programmer could write instructions in a more natural form. For example, a programmer could write a single instruction such as A 5 B 1 C, without needing to break it down into the more primitive set of machine/ assembly language instructions.

Since computers are designed to execute only machine code, in order to execute

programs written in a high-level programming language, they need to first be translated into machine code. Such a translator is called a **compiler**. Whereas each line of an assembly language program is translated to one line of machine code, and thus a "one-to-one" translation, a compiler is capable of a "one-to-many" translation. Thus, one line of a program written in a high-level language is translated into many machine code instructions that accomplish it. The idea of a compiler that Grace Murray Hopper put forth was one of the most important in the development of modern computers.

IBM came out with the first programming language for commercially available computers in 1957 called **FORTRAN** ("FORmula TRANslation"). FORTRAN was a language suited for scientific programming. Other languages of note that were developed around the same time were **COBOL** ("Common Business Oriented Language") based largely on the work that Hopper did at Remington Rand, meant for business processing needs; **ALGOL** ("ALGOrithmic Language"), for general computing; **LISP** ("List Processing Language"), well suited for developing artificial intelligence programs; and **BASIC** ("Beginners All-Purpose Symbolic Instruction Code"), meant as an easy-tolearn language to make computer programming accessible to all college students, developed by **John Kemeny** at Dartmouth College in 1963. BASIC became the first programming language available on the earliest personal computers, and by the end of the 1980s, millions of school children had learned to use it.


FIGURE 12-36 Navy Admiral Grace Murray Hopper

Grace Murray Hopper, in 1951, conceived of a new type of programming

language that could help “automate” the task of programming, referred to as *high-level languages*.

12.6.3 The First “Program Bug” (1947)

American engineers have been calling small flaws in machines *bugs* for over a century. Thomas Edison talked about bugs in electrical circuits in the 1870s. When the first computers were built during the early 1940s, people working on them similarly referred to “bugs” in both the hardware of the machines and in the programs that ran them.

In 1947, engineers working on the Mark II computer at Harvard University found a moth stuck in one of the components, causing the machine to malfunction. They taped the insect in their logbook and labeled it “first actual case of bug being found (Figure 12-37).” It has become a standard part of the language of computer programmers. The log book, complete with the attached bug, is on display at the Smithsonian Institution in Washington, D.C.

FIGURE 12-37 The First “Computer Bug”

In 1947, engineers working on the Mark II computer at Harvard University found a moth stuck in one of the components, causing the machine to malfunction. They taped the insect in their logbook and labeled it “first actual case of bug being found.”

THIRD-GENERATION COMPUTERS (mid-1960s to early 1970s) 12.7 The Development of the Integrated Circuit (1958)

While transistors had the aforementioned advantages over vacuum tube

technology, there remained the problem of how to wire together the increasing number of transistors needed for the increasingly powerful computers being designed, known as the “tyranny of numbers.” This was evident with the transistorized computers. The solution to this problem was the development of the integrated circuit (or semiconductor chip), in which such wiring of components became unnecessary.

Two different individuals, **Jack Kilby** at Texas Instruments and **Robert Noyce** at Fairchild Semiconductor, were both working hard on a solution to the problem. The elegant, practical solution that they eventually found was to replace the method of *wiring* together components on digital circuit boards with a method of *printing* the “wiring” onto a thin wafer of semiconductor material (as shown in Figure 12-38). Thus, transistors and circuits were “integrated” together in the manufacturing. This device, therefore, was called an **integrated circuit (IC)** (or semiconductor chip). This not only

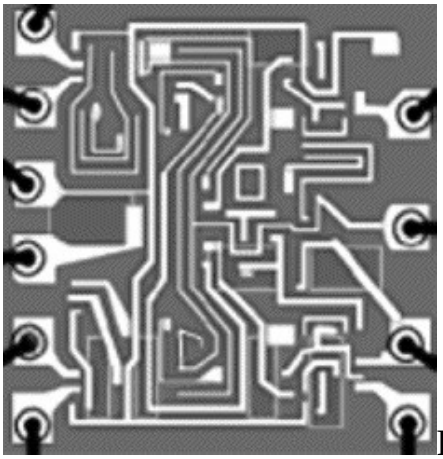


FIGURE 12-38 Integrated (Printed) Circuit

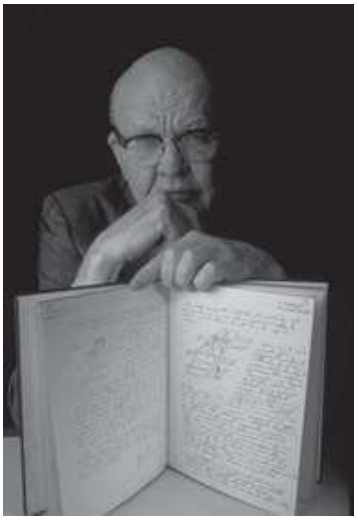


FIGURE 12-39 Jack Kilby—Inventor of the Integrated Circuit

made circuits smaller, but much cheaper to mass-produce. By 1961, both companies were producing commercially available integrated circuits.

It is generally agreed that Jack Kilby was first to conceive of and develop the integrated circuit in 1958, and Robert Noyce is credited with developing practical manufacturing principles that allowed for its commercialization. Jack Kilby went on to invent the portable calculator in 1967. Robert Noyce (with Gordon E. Moore, who was a cofounder of Fairchild) started in 1968 a new company named Intel (for “Integrated Electronics”).

In the year 2000, Jack Kilby (Figure 12-39) was awarded the Nobel Prize in Physics for the invention of the integrated circuit. The importance of the development of the integrated circuit cannot be overstated. It forms the basis of all modern computing today, and is said to have caused a “second industrial

revolution.” It stands historically as one of the most important inventions of mankind.

“What we didn’t realize then was that the integrated circuit would reduce the cost of electronic functions by a factor of a million to one, nothing had ever done that for anything before.”—Jack Kilby

An example packaged integrated circuit (“chip”) is shown in Figure 12-40.

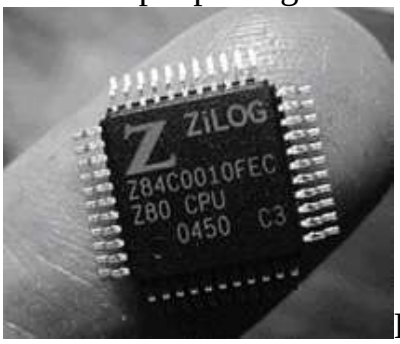


FIGURE 12-40 A Modern Integrated Circuit
The integrated circuit, invented and developed by **Jack Kilby** and **Robert Noyce**, stands as one of the most important inventions of mankind.

12.7.1 The Catalyst for Integrated Circuit Advancements (1960s)

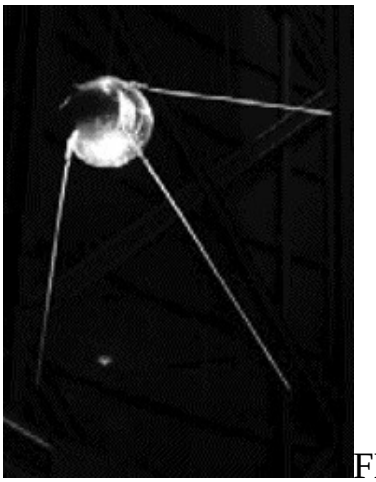
While integrated circuits had been commercially available since 1961, computer manufacturers did not jump on the new technology because it meant completely redesigning their current transistorbased machines. However, there was a

politically motivated, impossible-seeming challenge that would be put forth to Americans that would speed up the development of this technology. America and the Soviet Union were about to enter a “space race.”

The Beginning of the Space Race

On October 4, 1957, the Soviet Union (now Russia) launched the first artificial satellite, named “Sputnik” (Figure 12-41), sending shock waves throughout the United States. In April 1961, the Soviets had their first successful manned flight, by Cosmonaut Yuri Gagarin, with many successful manned flights following. The United States had no space program.

In response, in October 1958, **NASA** (National Aeronautics and Space Administration) was created, exactly one year after Sputnik was launched. The first U.S. manned space flight occurred on May 5, 1961. Alan B. Shepard Jr. took a 15-minute suborbital

 FIGURE 12-41 Sputnik Satellite

spaceflight that made him the first American in space. Less than a year later, on February 20, 1962, John H. Glenn Jr. became the first American to orbit the Earth. These were the first steps on a race to the moon.

On October 4, 1957, the Soviet Union (now Russia) launched the first artificial satellite, named “Sputnik” sending shock waves through the United States. President Kennedy’s Challenge of Going to the Moon (1961)

In 1961, **John F. Kennedy** was inaugurated as president. It was the height of the cold war with the Soviet Union. Although Allies in World War II, the United

States and Soviet Union came out of the war distrustful of each other. The biggest arms race in history had begun.

President Kennedy, concerned about the lead that the Soviets had in space, decided to raise the goal. In a speech to a special joint session of congress on May 25, 1961 (Figure 12-42), 20 days after Alan Shepard's flight, Kennedy proposed:


FIGURE 12-42 President Kennedy's challenge to go to the Moon

"I believe that this nation should commit itself to achieving the goal, before this decade is out, of landing a man on the moon and returning him safely to the earth. No single space project in this period will be more impressive to mankind, or more important for the long-range exploration of space; and none will be so difficult or expensive to accomplish. . . . in a very real sense, it will not be one man going to the moon . . . it will be an entire nation. For all of us must work to put him there."—President John F. Kennedy

Thus, President Kennedy had in clear terms put out a challenge to the Soviets, a "space race," to see who could reach the moon first. It was on July 22, 1969, that Neil Armstrong, Michael Collins, and Edwin Aldren safely reached and returned from the moon (Figure 12-43)—just over five months from the end of the decade, within the timeframe that President Kennedy had proposed. What technological devel opments had to be made to make this happen in a little over eight years time? Because of the limited space and constrained environment of space flight,

the integrated circuit was the technology needed.

FIGURE 12-43 The Apollo 11 Astronauts—Armstrong, Collins, and Aldren

President Kennedy had in clear terms put out a challenge to the Soviets, a “space race,” to see who could reach the moon first. Because of the limited space and constrained environment of space flight, the integrated circuit was the technology needed.

The Crucially Needed Advancements in Integrated Circuits (early 1960s)

FIGURE 12-44 The Apollo Guidance Computer (AGC)

Integrated circuits were obviously the way to go to achieve Kennedy’s goal, given their reliability and reduced weight and size over current technology. The problem was that, in 1961, integrated circuits were in their primitive stages of development, and extremely costly, at about $1,000 per chip.

The project for the development of what became called the **Apollo Guidance Computer** (AGC) (Figure 12-44) was given to MIT. The integrated circuits that were needed pushed electrical engineers to work on the development of better and more cheaply produced chips. By

1964, the cost of each chip had dropped to $25. When it was finished for the first Apollo missions using the Saturn V rocket (Figure 12-45) the AGC weight only 70 pounds, contained 4,000 integrated circuits, measuring only two feet by one foot by six inches! It was the only computer in existence completely designed using integrated circuits. The requirements of the AGC drove technological developments in the design and production of integrated circuits. By 1973, computers built completely out of integrated circuits were commonplace. By the end of the decade, integrated circuits were developed containing tens of thou

sands of transistors on a chip.

 FIGURE 12-45 The Saturn V Rocket

The challenge by **President John F. Kennedy** of putting a man on the moon was a catalyst for the advancements made in integrated circuits technology.

12.7.2 The Development of the Microprocessor (1971)

While integrated circuits were in use in computer systems, each chip was a special-purpose component used together in "chip sets." A different set of chip sets was needed for each particular computing device, and the cost of designing chips was expensive.

By 1969, integrated circuits were becoming more and more advanced in capability. An engineer working at Intel, named **Marcian E. ("Ted") Hoff**, hit upon a brilliant idea. Instead of Intel manufacturing numerous *special-purpose* chips for specific devices (such as handheld calculators or digital alarm clocks), why not build a *general-purpose* logic chip that could be programmed to perform any logical task. In this way, a single chip can satisfy the needs of any device. This was an advantageous approach for Intel, since they could strictly focus on the production of such general logic chips. Those incorporating them

into their products would be responsible for writing the programs.

This general logic chip was equivalent to the central processing unit (CPU) in mainframe computers. In mainframes computers, however, the CPU was

FIGURE 12-46 Marcian “Ted” Hoff (center), Inventor of the Microprocessor with co-developers Federico Faggin and Stanley Mazor

composed of a number of different circuit boards. A general logic chip compressed all those circuits onto one integrated circuit. Since this logic chip contained all the functionality of a central processing unit (the “heart” of any computer system), it became known as a **microprocessor**.

The first microprocessor was produced and made available on the open market in November 1971. It was developed with co-developers Federico Faggin and Stanley Mazor (Figure 12-46), as well as Masatoshi Shima (not shown). It was a four-bit processor called the 4004, and the first commercially available integrated circuit that could be programmed for different tasks. It contained all the necessary components of a central processing unit squeezed onto a chip one-sixth of an inch long and one-eighth of an inch wide, containing 2,250 transistors. (The term “microprocessor” was used to emphasize a complete CPU on a chip. While most CPUs are microprocessors, they are now simply referred to as a “processor.”) The 4004 was used in the first scientific handheld calculators produced by Hewlett-Packard in 1972.

Marcian E. (“Ted”) Hoff invented the microprocessor—a complete central processing unit on a chip.

As computing became more and more of a commercial item, and used by various types of companies and organizations, the computing needs likewise became varied. Some required and could afford large computer systems, while others needed something less. We will see next how this diversity of needs was satisfied though the development of a more diverse set of commercial computer systems.

12.8.1 The Establishment of the Mainframe Computer (1962)


FIGURE 12-47 IBM 7090 Mainframe at NASA

As commercial computing developed, large computers became known as **mainframe computers** (or simply **mainframes**). Such system could consist of multiple cabinets, storing the central processing unit (before the development of the microprocessor), memory, tape drives for storage, and so forth, taking up a whole room. An example of an early mainframe was the IBM 7090, shown at NASA in 1962 in Figure 12-47.

In 1964, IBM came out with a system called the IBM 360 (Figure 12-48). It is considered by some to be one of the most successful computers in history. Various models of the 360 were developed until 1978. Its design influenced many other computer systems that followed.

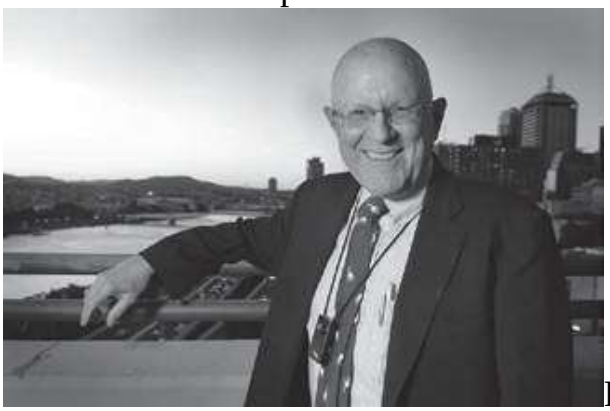
The IBM 360 was actually a series of models that were compatible enough so that companies could “trade up” for a larger system without the need for significant reprogramming. It was the first series of computers of

fering a large range of computing needs. Customers had

 FIGURE 12-48 IBM 360

the option of purchasing systems with various processor speeds and a range of main memory size, among other options.

12.8.2 The Development of the Minicomputer (1963)

 FIGURE 12-49 Gordon Bell—"Father of the Minicomputer"

Until the early 1960s, all computers were large and expensive mainframes, costing millions of dollars, only affordable by universities and large companies. An MIT graduate, named **Kenneth Olsen**, who had worked in the research labs there, came to a realization that smaller, less expensive computers had a place in the world, so-called **minicomputers**. He formed his own company in the early 1960s, called Digital Equipment Corporation (DEC). Chief Engineer **Gordon Bell** (Figure 12-49) at DEC designed a series of computers

that were small and inexpensive, transistor-based single-user systems—the PDP series.

In, 1965, DEC produced a model called the PDP-8 (Figure 12-50). This was the first widely successful minicomputer, and is considered as ushering in the minicomputer age (and eventually, personal computers). The PDP-8 was an 11-

bit computer. Following the PDP-8 was the 16-bit PDP-11, and the 32-bit Vax-11. This series of computers proved to be very successful and longlasting. The last VAX system was manufactured in 2005.

A new dimension was added to the classification of systems—by size and cost. Prior to their development, companies and other organizations that were not able to afford their own computer had to purchase time-sharing from an available mainframe over very slow phone line connections. Now a new alternative was available.

In 2010, Bell received an honorary Doctor of Science and Technology degree from Carnegie Mellon University. The university referred to him as the “Father of the Minicomputer.”

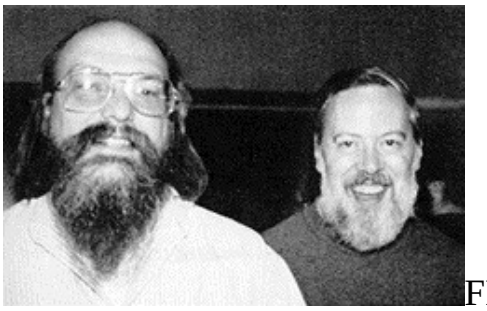

FIGURE 12-50 PDP-8 Minicomputer

Gordon Bell, in the early 1960s, designed the first minicomputers, and thus is called the “**Father of the Minicomputer**.”

12.8.3 The Development of the UNIX Operating System (1969)

In the early 1960s a joint project was formed between General Electric, MIT, and Bell Labs to develop a time-shared computer called MULTICS. The project, however, ended without being as successful as hoped.

Work on the MULTICS project, however, motivated those involved from Bell Labs to continue towards the goal of interactive computing. Thus, they began work on their own time-sharing operating

 FIGURE 12-51 Kenneth Thompson and Dennis Ritchie

system in 1969, to be known as **UNIX** (see www.bell-labs. com/history/unix/). The two individuals that developed UNIX were **Kenneth Thompson** and **Dennis Ritchie** (Figure 12-51).

During the development of UNIX, a new programming language named “C” was developed by Dennis Ritchie, evolved from an earlier language named “B” that Ken Thompson had developed. Eventually, the UNIX operating system was rewritten in the C language, which along with UNIX, became widely used.

The developments of UNIX and C, and their adoption by industry and academia, is one of the biggest success stories in computer science. Thompson and Ritchie received numerous prestigious awards, including the A.M. Turing Award from the Association for Computing Machinery (ACM) in 1983, the U.S. National Medal of Technology from President Bill Clinton (Figure 12-52) in 1999, and the Japan Prize for Information and Technology (2011).

UNIX and C (and its object-oriented successor **C 11**) are in much use today, including a personal computer version of UNIX called **Linux**.

FIGURE 12-52 Thompson and Ritchie Receiving the National Medal of Technology